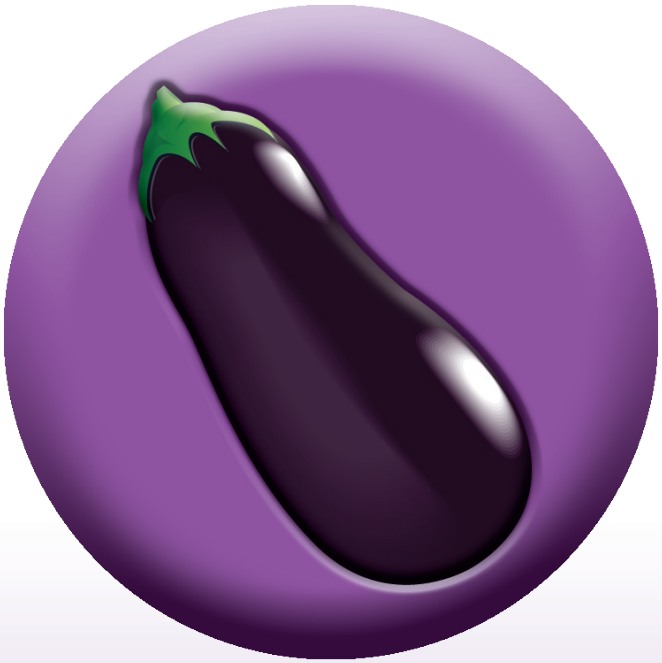
eggPlant

v11.0 MAC OS X

SenseTalk Reference



TestPlant

SenseTalk Reference Manual

Trademarks

Contents

About This Manual

What This Manual Contains

This manual is organized in 4 sections: Overview, which introduces the language; The Basics, covering language fundamentals; Objects and Messages, describing these topics in detail; and Commands and Functions, which describes the facilities available for performing a wide variety of tasks.

Overview

SenseTalk™ is a language for controlling your computer. It's an English-like language that is easy for people to read, write, and understand. People who use SenseTalk may find themselves trying something without knowing whether it will do what they want and are often pleasantly surprised when it just works!

Computers, of course, are not intelligent. For the computer to correctly understand your instructions, they must be stated in a clear and unambiguous way. For this reason, SenseTalk is a more structured language than English, with rules that must be followed.

Introducing SenseTalk **–** provides a general introduction to SenseTalk, offering a high-level overview of the language.

SenseTalk In A Nutshell **–** is a high-speed tour of SenseTalk. Experienced programmers and newcomers alike will find this section a valuable first stop to gain a quick grasp of the essential nuts and bolts of the language.

The Basics

To use SenseTalk effectively, there are a few basic concepts you will need to understand: values, containers, expressions, and control structures. These are not difficult concepts, but they are important for you to understand in order to take full advantage of the power of SenseTalk and your computer. The next few sections explain these concepts in detail:

Values **–** describes the different kinds of values that you can work with in SenseTalk, including numbers, text, dates, and so forth.

Containers **–** describes the different types of containers that can hold values, including local and global variables. It also describes the put and set commands, which are used to store values in containers, and the ability to store references to containers.

Expressions **–** introduces expressions and the various operators that can be used to combine and manipulate values in a variety of ways.

Chunk Expressions **–** describes SenseTalk's powerful text chunk expressions, which make working with the characters, words, lines, and items within text values incredibly easy and natural.

Script Structure **–** explains the structure of a script and the control structures that allow your scripts to perform complex and repetitive tasks.

Lists and Property Lists **–** takes an in-depth look at these important multi-value collections and how to work with them.

Ranges, Iterators, and Each Expressions **–** describes these valuable tools for generating and manipulating many values at once.

Objects and Messages

The material covered up to this point is enough to allow you to write scripts and accomplish many tasks. To fully understand SenseTalk and leverage its power, there are a few more concepts to master: messages and handlers, and objects and their helpers.

Objects, Messages, Handlers and Helpers **–** introduces the powerful concept of Objects, and describes how to create them, access their properties, and send messages to them. Object Helpers, which allow objects to "help" others, are also described.

Working with Messages **–** describes the nuts and bolts of the commands and constructs that deal with sending and handling messages.

Commands and Functions

Having mastered the concepts and structure of the SenseTalk language, the final sections describe the commands and functions you will need to accomplish specific types of tasks.

Working with Text – details the text functions and commands which are available for manipulating strings of text.

Working with Numbers – documents the mathematical functions and commands which are available for performing various numeric calculations in SenseTalk.

Working with Dates and Times – describes how SenseTalk scripts can use and manipulate values representing dates and times.

Working with Files and File Systems – explains the extensive facilities available in SenseTalk for reading and writing data in files, and for working with files and folders in the file system.

Working with URLs and the Internet – describes how SenseTalk can be used to access resources on the Internet and to manipulate URL strings.

Working with Trees and XML – describes the SenseTalk tree structure and how it can be used to read, write, and manipulate XML data.

Working with Color – describes facilities provided by the STColor XModule to enable your scripts to work with values representing colors.

Working with Binary Data – explains mechanisms for working with binary (non-textual) data in your scripts.

Other Commands and Functions – describes commands and functions for interacting with the user and with the system.

Appendices

Appendix A – Restricted Words **–** provides lists of the words whose use is restricted (either they can't be used as command names, or as variable names) in order for SenseTalk to be able to understand your scripts.

Appendix B – All About "It" **–** describes the important role played by it in SenseTalk and lists the commands that manipulate it.

Conventions Used in This Manual

The following visual cues are used in this manual to identify different types of information:

This manual uses `Courier` type to represent SenseTalk scripts or script fragments. Many of the script examples are colorized in ways that indicate the role played by each word or other element of the script.

Note

A note like this contains information that is interesting but not essential for an understanding of the main text.

◊ Specific Language Elements

A section that describes a specific language element such as a command, function, operator, or property is marked with a lozenge in the margin (as shown here) to make it easy to locate.

Advanced Topics

A section placed in a purple box denotes a topic that is mostly of interest to more advanced scripters. Beginning users may want to skip these topics until after they are comfortable with the basics.

Syntax Definitions

Syntax definitions for language elements use **boldface** to indicate words that must be typed exactly, *italics* to represent expressions or other variable elements, and curly braces { } to indicate optional elements. Elements may be enclosed in square brackets [ ] separated by vertical bars | to indicate alternative options (where one or the other may be used, but not both).

So, for example, the following partial syntax definition:

```
{in} [ascending | descending] {order}
```

indicates that you may optionally use the word “in” followed by either the word “ascending” or the word “descending”, followed optionally by the word “order”. In other words, all of the following would be valid (as well as several other variations):

```
ascending
descending
ascending order
in descending
in ascending order
```

Overview

Introducing SenseTalk – provides an introduction to SenseTalk, offering a high-level overview of the language.

SenseTalk In A Nutshell – is a high-speed tour of SenseTalk. Experienced programmers and newcomers alike will find this section a valuable first stop to gain a quick grasp of the essential nuts and bolts of the language.

Introducing SenseTalk™

SenseTalk™ is a remarkably English-like language that enables you to communicate with your computer to harness its power and abilities in a refreshingly easy and understandable manner. SenseTalk provides access to the full range of capabilities inherent in your computer while remaining easy to learn and understand. It achieves this by leveraging the powerful concepts behind many words that have become buzzwords in the computer industry.

In today's technical jargon:

> *SenseTalk is a very high level, object oriented, interpreted scripting language, based on a modular, extensible architecture.*

Now, in plain English, what does that really mean?

SenseTalk is "object oriented" because thinking about objects is a natural and understandable way to describe and deal with potentially complex systems. We deal with "things" daily in our lives, whether they are tangible things like a telephone or a glass of water, or intangible things like our bank account or our child's soccer schedule. With SenseTalk you can create "objects" in your computer which represent each of these things.

SenseTalk is considered a "very high level" language because it can do a lot with very few words. High level commands make your job easier. Suppose you have a list of names that you would like to sort into alphabetical order by last name. In a lower level programming language you would need to write instructions to find the last name by locating the last space in each name, and then write many more instructions to carefully rearrange the names alphabetically. In SenseTalk you could simply give the command "`sort the lines of nameList by the last word of each`" and be done with it.

As an "interpreted scripting language", SenseTalk is very responsive, providing immediate feedback as you learn the language and try out new commands. You can type commands or partial scripts and have the computer carry out your instructions at once.

And SenseTalk's "modular, extensible architecture" allows you to add new commands and functions to extend the range of what SenseTalk can do. The language is not cast in stone, but can grow and evolve according to your needs. The underlying structure has been crafted to support new and changing requirements as computer capabilities advance and your knowledge and understanding grows.

To put it simply, *SenseTalk is an English-like language that lets you tell your computer what you want it to do.* You do this by creating software "objects" that organize the information you want to work with in understandable ways, and by writing "scripts" that describe how you want each object to behave.

You could think of it as being a little bit like writing a play. You create a cast of characters, and provide each of them with their own script which defines their role, and how they should respond to various messages which might be sent to them. Unlike a play, though, the action doesn't have to follow the same sequence every time the script is run. The characters (objects) interact with one another and with the user of the system (you) by sending and receiving messages.

A message may tell an object what to do, or provide it with information. When it receives a message, it may respond in different ways at different times depending on what else is going on in the system at that time. It all depends on what the object's script tells it to do. Each object has its own script — its own set of instructions to follow — and the instructions are written using the SenseTalk language.

SenseTalk was originally created as the scripting language of *HyperSense*™, Thoughtful Software's powerful authoring system. While some examples and descriptions in this manual may refer to HyperSense objects or other features of that system, SenseTalk also stands alone as an independent scripting language. This manual aims to describe the SenseTalk language separately from any particular environment or host application in which it might be used.

Why SenseTalk?

About now you may be thinking: Okay, that all sounds nice enough, but there are plenty of scripting languages out there already. Does the world really need another language? What makes SenseTalk different?

That's an excellent question, and there isn't necessarily one answer for everyone. Every language has its own strengths and peculiarities, and different characteristics of a language will appeal to different people. Probably the single thing that distinguishes SenseTalk the most from other languages – the theme that provides its unique flavor – was stated in the first sentence of this section: SenseTalk is "remarkably English-like".

Now, some people may not see being "English-like" as a benefit for a programming language. English can be wordy, and may at times be ambiguous in its meaning. Such people may prefer a more concise and precise language with stricter rules of syntax. So be it.

On the other hand, an English-like language offers some special benefits: it can be easier for a beginner to learn; easier for an experienced user to remember; and much easier for everyone to read and understand. This feature – readability – can make scripts much easier to maintain and modify.

Let's take a look at SenseTalk's readability by comparing it to some other popular languages. We'll start with a comparison to Perl and Python, two of the most widely-used scripting languages in the world today.

Perl is described on Wikipedia as a language designed to be "practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)" and is famous for being rather cryptic and unreadable, so it serves to illuminate SenseTalk through extreme contrast. Here is an example Perl script that is taken almost verbatim from a popular book written by Perl experts (the “Perl Cookbook”, p. 247). We’ll explain what it does in a moment.

```
sysopen(FH, "numfile", O_RDWR|O_CREAT)or die "can’t open numfile: $!";
$num = <FH> || 0;        # DO NOT USE "or" HERE!!
seek(FH, 0, 0)           or die "can’t rewind numfile: $!";
truncate(FH, 0)          or die "can’t truncate numfile: $!";
print FH $num+1, "\n"    or die "can’t write numfile: $!";
close(FH) § or die "can’t close numfile: $!";
```

Python, on the other hand, is another popular scripting language that is known for being easy to read. Here is the equivalent script written in Python:

```
file = open("numfile", "r+")
num = int(file.read())
file.seek(0,0)
file.truncate(0)
file.write(str(num+1) + "\n")
file.close()
```

Finally, here is a SenseTalk script which accomplishes the same thing (if it's not clear yet what these scripts do, maybe this will help):

```
add 1 to file "numfile"
```

Okay, is it becoming clearer now? The purpose of all three scripts is to add one to a counter which is kept in a file called “numfile”. The purpose of this comparison is not to bash Perl or Python, which are very powerful languages used extensively by many people, but to point out the advantage that SenseTalk offers for scripters whose needs are perhaps less rigorous but who would like to get something done quickly and easily.

To be fair, there are ways the Perl and Python scripts could be shortened somewhat, and SenseTalk also offers longer ways of doing the same thing which might be more comparable. Here is another script in SenseTalk which performs the same task, following more closely the approach used in the other scripts:

```
open file "numfile"
```

```
read from file "numfile" until end
put the first word of it into num
seek in file "numfile" to the beginning
write num+1 & return to file "numfile"
close file "numfile"
```

If you eliminate all of the “or die” parts of the Perl script (which are just there to be very careful about reporting mostly unlikely errors) you’ll see that this SenseTalk script is now slightly longer than either the Perl or Python version. But you’ll probably agree that the SenseTalk script is also much clearer, easier to read and understand, and also easier to learn and remember. Because it is so much more readable, it is also easier to find and correct mistakes, or to make changes to a script you wrote several months earlier if you decide it should do something slightly different.

Here's one more example, in several languages this time, showing how to perform a fairly common operation: deleting the last character of a text string that is stored in a variable called var:

PHP: $var = substr($var, 0, strlen($var)-1)

Ruby: var.chop

Perl: chop($var)

Java: var = var.substring(0, var.length()-1);

Python: var = var[0:-1] // (or, commonly: var = var[:-1])

JavaScript: var = var.slice(0, -1)

Excel VBA macro: ActiveCell.Value = Left(ActiveCell.Value, Len(ActiveCell.Value) - 1)

SenseTalk: `delete the last` character of **`var`**

Here, the Ruby and Perl versions are the shortest, and are very easy to read once you know what the "chop" function does. Would it be clear to someone new to the language what any of these examples do?

Now, suppose you need to modify your script to *delete the first character of var* instead of the last? Or to *delete the last two characters of var*? Would it be obvious how to do that? (Hint: In case you didn't guess already, the actual SenseTalk commands to do those operations are embedded in italics in the questions!)

Perhaps that’s enough of an introduction to give you some idea of what SenseTalk is about and what we’re trying to achieve. Now it’s time to dig in and learn something about how this language works and how to use it.

SenseTalk in a Nutshell

A One Hour Introduction for Experienced Programmers and Inquiring Minds.

SenseTalk is a powerful, high level, and easy-to-read language. It is designed to be similar to English, avoiding cryptic symbols and rigid syntax when possible, in order to be both easy to read and to write. This section briefly describes many aspects of the language.

The information presented here is intended to provide a quick overview of most of the key elements of SenseTalk. Experienced scripters and programmers may find this is enough to get them started using SenseTalk. Other users may want to skim over this section to get a quick sense of what's ahead before moving on to the more detailed explanations in following sections.

You may also find this section to be a convenient place to turn later when all you need is a quick reminder of how things work. For each topic, references are given to the section where more complete information may be found.

Key Elements of the Language

Scripts

A SenseTalk **script** is a series of command statements. When the script is **run**, each statement is executed in turn. Commands usually begin with a verb, and are each written on a separate line:

```
put 7 into days
multiply days by 4
put days -- 28
```

A script is often stored as a text file on your computer.

SenseTalk is not case-sensitive: commands can be written in uppercase, lowercase, or a mixture without changing the meaning:

```
Put 7 into DAYS
```

(See Script Structure)

Simple Values

In SenseTalk there are simple values:

```
5 -- number
sixty-four -- number expressed in words
"hello" -- text string (full international Unicode text allowed)
empty -- constant
0x7F -- hexadecimal number
<3FA64B> -- binary data
```

(see Values)

Tech Note

There are no special "escape codes" for including special characters in text strings in SenseTalk. What you see is what you get.

Text Blocks

Multi-line blocks of text may be enclosed in `{{` and `}}`. This type of text block is particularly useful for dynamically setting the script of an object, or for defining a block of data:

```
set names to {{
Harry Potter
Hermione Granger
Ron Weasley
}}
```

(see Values)

Tech Note

Text blocks can use labels, similar to "here is" blocks in some other languages. This also allows them to be nested.

Operators

Operators combine values into expressions. A full range of common (and some uncommon) operators is available. A `put` command with no destination displays the value of an expression:

```
put 3 + 2 * 7 -- 17
put five is less than two times three -- true
put "a" is in "Magnificent" -- true
put 7 is between 5 and 12 -- true
put "poems/autumn/InTheWoods" split by "/" -- (poems,autumn,InTheWoods)
```

Parentheses can be used to group operations:

```
put ((3 + 2) * 7) is 35 -- true
```

(see Expressions)

Concatenation

Text strings can be joined (concatenated) using `&` or `&&`. The `&` operator joins strings directly; `&&` joins them with an intervening space:

```
put "red" & "orange" -- "redorange"
put "Hello" && "World" -- "Hello World"
```

(see Expressions)

Value assignment

Values can be stored in containers. A variable is a simple container. Either the `put into` or `set to` command may be used to assign a value to a container:

```
put 12 into counter
set counter to 12
```

Variables are created automatically when used; they do not need to be declared first.

(see Containers)

The Put Command

The `put` command can also append values **before** or **after** the contents of a container:

```
put 123 into holder -- "123"
put "X" before holder -- "X123"
put "Y" after holder -- "X123Y"
```

(see Containers)

Typeless Language

SenseTalk is a **typeless** language. Variables can hold any type of value:

```
put 132 into bucket -- bucket is holding a number
put "green cheese" into bucket   -- now bucket holds a text string
```

Values are converted automatically as needed:

```
put ("1" & "2") / 3  -- 4
```

(see Expressions)

Unquoted Strings

Any variable that has not yet had a value stored into it will evaluate to its name. In this way, they can be used as **unquoted strings,** which can be convenient.

```
put Bread && butter -- "Bread butter"
```

(see Containers)

Constants

Some words have predefined values other than their names. These are commonly called “constants” but SenseTalk tries to allow you maximum freedom to use any variable names you choose, so only a very few of these are truly constant; the others can be used as variables and have their values changed.

The actual constants include true, false, up, down, end, empty, and return:

```
put street1 & return & street2 & return & city into address
if line 2 of address is empty then delete line 2 of address
```

The *predefined variables* include numbers and special characters like quote and tab:

```
put 2*pi  -- 6.283185
add one to count
put "Edward" && quote & "Red" & quote && "Jones"  -- Edward "Red" Jones
put comma after word 2 of sentence
```

(see Values)

Comments

Comments can add descriptive information. Comments are ignored by SenseTalk when your script is running. The symbols `--` (two dashes in a row) or `//` (two slashes) mark the remainder of that line as a comment:

```
// this script adds two numbers and returns the sum
params a,b -- this line declares names for the two parameters
return  a+b // return the sum (that's all there is to it!)
```

For longer (or shorter) comments you may enclose the comment in (* and *). This technique is sometimes used to temporarily turn off (or "comment out") part of a script. These "block comments" can be nested:

```
(*
put "the total so far is : " & total -- check the values
put "the average is : " & total / count (* a nested comment *)
*)
```

(see Script Structure)

Chunk Expressions

A *chunk expression* can be used to specify part of a value:

```
put word 2 of "green cheese"  -- "cheese"
put item 3 of "one,two,three,four" -- "three"
put lines 1 to 3 of bigText   -- the first 3 lines
put the first 3 lines of bigText -- also the first 3 lines
put any character of "abcdefg" -- one letter chosen at random
```

Negative numbers count back from the end of a value:

```
put item -3 of "one,two,three,four" -- "two"
put chars 2 to -2 of "abcdefg" -- "bcdef"
```

Chunks of containers are also containers (you can store into them):

```
put "green cheese" into bucket -- "green cheese"
put "blue" into word 1 of bucket -- "blue cheese"
put "ack" into chars 3 to 4 of word 1 of bucket -- "black cheese"
```

(see Chunk Expressions)

Tech Note

SenseTalk's chunk expressions provide capabilities similar to substring functions or slices in other languages, but are much more expressive and powerful.

Lists

You can create a *list* of values by merely listing the values in parentheses separated by commas:

```
(1,2,3)
("John", 67, "555-1234", cityName)
("bread", "milk", "tofu")
```

Lists may include any type of value, including other lists:

```
("Mary", 19, ("555-6545", "555-0684"), "Boston")
```

Lists can be stored in containers:

```
put (2,3,5,7,11,13) into earlyPrimes
```

(see Values)

List Items

Items in a list can be accessed by number. The first item is number 1:

```
put item 1 of ("red", "green", "blue")  -- "red"
put the third item of ("c","d","e","f","g") -- "e"
```

List items in a container are also containers (they can be stored into):

```
put (12,17,32) into numbers    -- (12,17,32)
put 99 into item 5 of numbers   -- (12,17,32,,99)
add 4 to item 2 of numbers     -- (12,21,32,,99)
```

(see Chunk Expressions)

Combining Lists

Lists can be joined using &&&:

```
put ("red", "green", "blue") into colors -- ("red", "green", "blue")
put (12,17,32) into numbers   -- (12,17,32)
put colors &&& numbers into combinedList -- ("red","green","blue",12,17,32)
```

To create a list of nested lists instead of combining into a single list, just use parentheses to create a new list:

```
put (colors,numbers) into nestedList -- (("red","green","blue"), (12,17,32))
```

(see Expressions)

Property Lists

A *simple object* or *property list* is a collection of named values (called *properties*). Each value in an object is identified by its property name:

```
(size:12, color:blue)
(name:"John", age:67, phone:"555-1234", city:"Denver")
```

Objects can be stored in containers:

```
put (size:8, color:pink) into description
```

(see Values, and Lists and Property Lists)

Tech Note

SenseTalk's property lists are similar to collections known as hash tables, associative arrays, dictionaries or records in other languages. However, a SenseTalk property list is an object that can have behaviors as well as data values when certain special properties are set.

Object Properties

Properties in an object can be accessed by name:

```
put property width of (shape:"oval", height:12, width:16) -- 16
```

New properties can be created simply by storing into them:

```
put "red" into property color of currentShape
```

(see Containers)

An object's properties can be accessed in several different ways:

```
put (name:"Michael") into mike -- create an object
put a new object into property cat of mike -- create a nested object
put "Fido" into the name of mike's cat
put mike's cat's name -- Fido
put mike.name -- Michael
```

In addition, an object can access its own properties (or those of an object it is helping) using "me" and "my":

```
put the age of me
if my name begins with "s" then ...
```

(see Objects, Messages, Handlers, and Helpers)

Properties are containers – their contents can be changed:

```
add one to my dog's age -- it's her birthday!
```

(see Containers, and Lists and Property Lists)

Nesting

Lists and objects can be *nested* in complex ways:

```
(size:12, colors:(blue,orange), vendor:(name:"Jackson Industries",
phone:"555-4532"))
```

(see Lists and Property Lists)

Ranges

A range is an expression that indicates a range of values. A range can be stored in a variable:

```
put 13 to 19 into teenRange
put teenRange -- 13 to 19
```

A range can be explicitly converted to a list:

```
put teenRange as list -- (13,14,15,16,17,18,19)
```

Or a range can simply be treated like a list:

```
put item 4 of teenRange -- 16
delete item 1 of teenRange -- teenRange is now a list
```

(see Ranges, Iterators and Each Expressions)

Iterators

A range, a list, or a custom iterator object can be used as an **iterator** to obtain a sequence of values one at a time:

```
put "Z" to "A" into reverseAlphabet
put reverseAlphabet.nextValue -- Z
put reverseAlphabet.nextValue -- Y
put reverseAlphabet.nextValue -- X
```

(see Ranges, Iterators and Each Expressions)

Each Expressions

An **each** expression can generate a list of values of interest:

```
set saying to "Beauty lies beyond the bounds of reason"
put each word of saying where each begins with "b" -- (Beauty,beyond,bounds)
```

Operations can be applied to each value in an each expression

```
put the length of each word of saying -- (6,4,6,3,6,2,6)
put uppercase of each word of saying where the length of each is 6
        -- (BEAUTY,BEYOND,BOUNDS,REASON)
```

(see Ranges, Iterators and Each Expressions)

Repeat Blocks

Repeat blocks are used to repeat a sequence of command statements several times:

```
repeat 3 times
    play "Glass"
    wait one second
end repeat
```

Several types of repeat loops are available, including `repeat while`, `repeat until`, `repeat with` and `repeat with each`:

```
repeat with each item of (1,3,5,7,9)
    put it & tab & it squared & tab & the square root of it
end repeat
```

(see Script Structure)

Tech Note

All loops in SenseTalk use `repeat`. There are different forms of repeat that correspond to most of the popular loop constructs available in other languages.

Conditionals

`if` / **`then`** / **`else`** constructs let scripts choose which commands to execute based on conditions:

```
if hoursWorked > 40 then calculateOvertime
if lastName comes before "Jones" then
    put firstName && lastName & return after firstPageList
else
    put firstName && lastName & return after secondPageList
end if
```

(see Script Structure)

Calling Another Script

A script can *run* another script by simply using the name of the other script as a command:

```
simplify -- run the simplify script
```

(see Script Structure)

Parameters

Parameters can be passed to the other script by listing them after the command name:

```
simplify 1,2,3
```

(see Script Structure)

Run Command

If a script's name contains spaces or special characters, or it is located in another directory, the run command can be used to run it:

```
run "more complex" -- run the "more complex" script
```

Parameters can also be passed using run:

```
run "lib/registerStudent" "Chris Jones","44-2516"
```

(see Working with Messages)

Handlers

A script may include *handlers* that define additional behaviors:

```
to handle earnWages hours, rate
    add hours*rate to my grossWages
end earnWages
```

A script can call one of its handlers as a command:

```
earnWages 40, 12.75
```

A handler in another script can be called using the run command:

```
run ScoreKeeper's resetScores "east", "south"
```

(see Script Structure)

Try/Catch Blocks

A script can catch any exceptions that are thrown when errors occur during execution:

```
try -- begin trapping errors
    riskyOperation -- do something that might raise an error
catch theException
    -- put code here to recover from any errors
end try
```

(see Script Structure)

Exceptions

A script may also *throw an exception*. If an exception thrown by a script is not caught by a try block, script execution will be stopped.

```
throw "error name", "error description"
```

(see Script Structure)

Local and Global Properties

Local and **global properties** control various aspects of script operation. They can be treated as containers, and are

accessed using the word "the" followed by the property name:

```
set the numberFormat to "0.00"
insert "Natural" after the timeInputFormat
```

(see Containers)

File Access

File contents can be accessed directly:

```
put file "/Users/mbg/travel.log" into travelData
```

Files can also be treated as **containers**, allowing data to be written to them or modified in place. If the file does not already exist, it will be created:

```
put updatedTravelData into file "/Users/mbg/travel.log"
add 1 to item 2 of line updateIndex of file "/Users/mbg/updates"
```

(see Working with Files and File Systems)

Sorting

The **sort** command provides expressive and flexible sorting of items in a list or of text chunks in a container:

```
sort callsReceivedList
sort the lines of file "donors" descending
sort the items of inventory in numeric order by the quantity of each
```

(see Working with Text)

Summary

The list of features and brief descriptions provided above give only an introductory summary of the language. SenseTalk offers many other capabilities not mentioned here, plus additional options for these features. The remaining sections of this manual describe all of SenseTalk's features in detail.

The Basics

To use SenseTalk effectively, there are a few basic concepts you will need to understand: values, containers, expressions, and control structures. These are not difficult concepts, but they are important for you to understand in order to take full advantage of the power of SenseTalk and your computer. The next few sections explain these concepts in detail:

Values – describes the different kinds of values that you can work with in SenseTalk, including numbers, text, dates, and so forth.

Containers – describes the different types of containers that can hold values, including local and global variables. It also describes the `put` and `set` commands, which are used to store values in containers, and the ability to store references to containers.

Expressions – introduces expressions and the various operators that can be used to combine and manipulate values in a variety of ways.

Chunk Expressions – describes SenseTalk's powerful text chunk expressions, which make working with the characters, words, lines, and items within text values incredibly easy and natural.

Script Structure – explains the structure of a script and the control structures that allow your scripts to perform complex and repetitive tasks.

Lists and Property Lists – takes an in-depth look at these important multi-value collections and how to work with them.

Ranges, Iterators, and Each Expressions – describes these valuable tools for generating and manipulating many values at once.

Values

Computers deal with a wide variety of data values, including numbers, text, sound, pictures, and so forth. The values you will work with the most in SenseTalk are numbers and text. The ability to combine and organize these values in various ways is also important. SenseTalk's lists and property lists let you do this.

Numbers

Numbers in SenseTalk can be written using numerals:

```
1
634
12.908
-3
.5
18.75
```

or as words:

```
one
six hundred thirty-four
twelve point nine zero eight
negative three
one half
eighteen and three quarters
```

For technical applications, hexadecimal notation (beginning with "0x") and scientific notation (containing "e+" or "e-" followed by the power of 10) are also accepted:

```
0x8ce3
4.58e+6
```

Text

Text in SenseTalk is usually enclosed in straight double quotation marks:

```
"abc"
"The Lord of the Rings"
"Greetings, and welcome to the world of computing!"
```

Full international (Unicode) text is supported. There are a few characters which are hard to represent in quoted text strings, though. In particular, the double quotation mark character itself can't be used between quotation marks, or the computer would get confused. So SenseTalk provides a special word – `quote` – which represents that character. You can use it like this:

```
"John said " & quote & "Hello!" & quote
```

The ampersand (`&`) concatenates text, joining it together into a single longer string of characters, so the expression shown above would result in the following text:

```
John said "Hello!"
```

Another common character which can't be included directly in quoted text is the return character, used to separate one line of text from the next. The word `return` represents the return character in SenseTalk:

```
"This is line 1." & return & "This is line 2."
```

Another way of including these characters in text is to use the special double angle bracket pairs << and >>. By using these pairs of characters instead of the double quotation mark, you can include quotation marks and return characters directly:

```
<<John said "Hello!"
This is line 2.>>
```

You can also use "curly" quotation marks, if you know how to produce them on your keyboard. They must always be paired with the left quotation mark at the beginning of the text and the right quotation mark at the end:

```
“John said "Hello!"”
```

Multi-line Blocks of Text

When you need to incorporate a large block of text spanning many lines into a script, a special mechanism is available using double curly braces `{{ }}`.

This type of block quoting differs from double angle brackets << >>. The quoted content doesn't begin until the line following the opening double brace `{{`, and ends with the line preceding the closing double brace `}}` which must be the first non-whitespace characters on a line.

```
put {{
This is my "quoted" text.
}} into statement-- statement now contains 'This is my "quoted" text.'
```

Neither the return following the opening double brace nor the return preceding the closing double brace are treated as part of the quoted content, so if you need to include a return character at the beginning or end of the quoted content, a blank line must be inserted.

These curly brace quotes are intended to make it easy to include a large quoted block of text within a script. Because the closing braces are only recognized at the beginning of a line, double braces may appear at other places within the quoted content. To allow an even greater range of content to be quoted, the opening braces may be followed by an identifier. The exact same identifier (case-sensitive) must then come immediately before the closing braces to terminate the quoted content. The identifier may include any non-whitespace characters other than curly braces. By using different identifiers, this makes it possible to nest one quoted text block within another.

```
set the script of Talker to {{TalkerScript
on speakUp
put {{InnerQuote
This is my "quoted" text, and I am prepared to use it.
InnerQuote}} into my statement
put statement
end speakUp
TalkerScript}}
```

Logical (Boolean) Values

Logical values (sometimes called Boolean values) express whether something is true or false, yes or no, on or off. In SenseTalk these logical values are represented by the constants `true` and `false`.

There are also a number of operators and expressions which evaluate to true or false (see Expressions). Logical val-

ues are used to make choices about whether or not to perform certain commands contained in an `if...then...else` construct (see "Conditional Statements" in Script Structure).

When testing a condition in an **`if`** statement, SenseTalk will accept any expression which evaluates not only to the constant values `true` or `false` but also to "yes", "no", "on", "off", or "" (empty). The values "true", "yes", and "on" are all treated as true, and "false", "no", "off", and "" are treated as false (none of these are case-sensitive – they are valid in upper, lower, or mixed case). Any other value will cause an exception to be thrown if it is used in a context where a logical value is required.

Constants and Predefined Variables

Words such as `true` and `return` which have pre-defined values that cannot be changed are called "constants". SenseTalk has only seven true constants (`empty`, `return`, `true`, `false`, `up`, `down`, and `end`). In addition, there are a large number of "predefined variables". These are words which have a predefined value, but may also be used as variables (and have their value changed in a script). See Appendix A – Restricted Words for more details.

Some of the SenseTalk words which have pre-defined values are:

WORD:	PRE-DEFINED VALUE:
`empty`	An empty string. This is the same as the text literal `""`
`return`	A "newline" character. This is the same character that is entered in a multi-line field when you press the "Return" key on the keyboard.
`carriageReturn` **`creturn`** **`cr`**	A "carriage return" character. The same as `numToChar(13)`
`linefeed` **`lf`** **`newline`**	A linefeed character. The same as `return`, or `numToChar(10)`
`crlf`	A carriage return followed by a linefeed. The same as `numToChar(13) & numToChar(10)`.
`lineSeparator`	A Unicode line separator character, equal to `numToChar(0x2028)`.
`paragraphSeparator`	A Unicode paragraph separator character, equal to `numToChar(0x2029)`.
`quote`	A straight double-quote character (").
`tab`	A tab character. This is the same character that is entered into a field when you press the "Tab" key. Tab is the same as `numToChar(9)`.
`space`	A single space character (" ").
`comma`	A comma (",").
`slash`	A forward slash ("/").
`backslash`	A backward-leaning slash ("\").
`colon`	A colon (":")
`formfeed`	A formfeed character. The same as `numToChar(12)`
`null`	A null character. The same as `numToChar(0)`
`pi`	The mathematical value pi, used in calculations related to circles. In SenseTalk, `pi` has the value 3.14159265358979323846

WORD:	PRE-DEFINED VALUE:
`true`	The word “true”, which has the logical value TRUE in SenseTalk.
`false`	The word “false”, which has the logical value FALSE in SenseTalk.
`yes`	The word “Yes”, which has the logical value TRUE in SenseTalk.
`no`	The word “No”, which has the logical value FALSE in SenseTalk.
`on`	The word “On”, which has the logical value TRUE in SenseTalk.
`off`	The word “Off”, which has the logical value FALSE in SenseTalk.
`up`	The word “up”.
`down`	The word “down”.
`end`	The value of `the endValue` global property, returned by iterators when no more values are available (default value is " ⓔ ⓝ ⓓ ").
`zero`, **`one`**, **`two`**, ...	The words zero, one, two, three, etc. are predefined as the corresponding numeric values (see Numbers, above).
`today`	The current date, in international date format.
`now`	The current time, in abbreviated international time format.
`date`	The current date, in the same format as the date function.
`time`	The current time, in the same format as the time function.

The words above which are "true constants" include `empty`, `return`, `true`, `false`, `up`, `down`, and `end`. The others are predefined variables, which may have other values stored into them by your script.

Technical Note: return constant

Because text in SenseTalk is based on the standards used by Cocoa and Unix, the `return` constant is actually a linefeed character, not a carriage return character, and is equivalent to `numToChar(10)`. In situations where you need an actual carriage return character (such as when writing text to a device attached to a serial port) you should use `carriageReturn`, `creturn` or `cr`, which is equivalent to `numToChar(13)`.

Custom Predefined Variables

In addition to the predefined variables listed above, SenseTalk automatically loads other variable definitions on start-up, which can be customized to your needs. These definitions are contained in files with a ".predef" extension located in the Resources folder within the SenseTalkEngine framework or other bundle loaded by the host application.

Any files with this extension should be plain text files containing an archived property list. The SenseTalk value() function will be used to read each property list and register its keys and values with the SenseTalk engine as predefined variables.

If two resources provide values for the same predefined variable name, the last value loaded will be the one that is used for that variable. Since the value being loaded is a SenseTalk expression, it is possible to include an expression to check for an already-defined value and use it instead, or incorporate it into the new value.

Predefined Variables Provided

SenseTalk comes with four sets of additional predefined variables installed, which provide convenient names for a number of useful characters and symbols. Not all of these symbols are available in all fonts, but they are all standard Unicode characters which may be useful in certain applications. For example:

```
put copyrightSign && "2008" -- © 2008
```

The symbols provided include:

Common Symbols

ellipsis (…), hyphen (‐), nonBreakingHyphen (‑), figureDash (‒), enDash (–), emDash (—), dagger (†), doubleDagger (‡), bullet (•), triangularBullet (‣), nonBreakingSpace ( ), atSign (@), careOfSign (℅), serviceMarkSign (℠), telephoneSign (℡), tradeMarkSign (™), facsimileSign (℻), numeroSign (№), invertedExclamationMark (¡), invertedQuestionMark (¿), verticalBar (|), brokenBar (¦), sectionSign (§), copyrightSign (©), registeredSign (®), pilcrowSign or paragraphSign (¶), middleDot (·), cedilla (¸), leftDoubleAngleQuotationMark («), rightDoubleAngleQuotationMark (»), checkMark (✓), blackDiamond (◆), lowerLeftPencil (✐), helmSymbol (⎈)

Currency Symbols

centSign (¢), poundSign (£), currencySign (¤), yenSign (¥), euroSign (€), dollarSign ($)

Numeric and Mathematical Symbols

percentSign (%), perMilleSign (‰), perTenThousandSign (‱), degreeSign (°), superscriptOne (^{1}), superscriptTwo (^{2}), superscriptThree (^{3}), microSign (µ), plusSign (+), minusSign (−), multiplicationSign (×), divisionSign (÷), plusOrMinusSign (±), minusOrPlusSign (∓), squareRootSign (√), cubeRootSign (∛), infinitySign (∞), notSign (¬), equalSign (=), almostEqualSign (≈), approximatelyEqualSign (≅), notEqualSign (≠), lessThanOrEqualSign (≤), greaterThanOrEqualSign (≥), fractionOneQuarter (¼), fractionOneHalf (½), fractionThreeQuarters (¾)

Keyboard Symbols

commandKeySymbol (⌘), optionKeySymbol (⌥), controlKeySymbol (^), shiftKeySymbol (⇧), eraseRightKeySymbol (⌦), eraseLeftKeySymbol (⌫), escapeKeySymbol (⎋), returnKeySymbol (⏎), ejectKeySymbol (⏏), appleLogo (), alternativeKeySymbol (⎇), blankKeySymbol (␣), capsLockKeySymbol (⇪), clearKeySymbol (⌧), pageUpKeySymbol (⇞), pageDownKeySymbol (⇟), tabKeySymbol (⇥), tabLeftKeySymbol (⇤), returnLeftKeySymbol (↩), returnRightKeySymbol (↪), leftArrowSymbol (←), rightArrowSymbol (→), upArrowSymbol (↑), downArrowSymbol (↓), contextualMenuKeySymbol (✳)

Lists

SenseTalk understands lists of values, which are enclosed in parentheses and separated by commas:

```
(1,2,3)
("a","b","c")
```

Lists can include any kind of values, including other lists:

```
(12.4, "green", <<John said "Hello!">>, (44,6), (55,2) )
```

Items in a list are accessed by their position number. The first item is number 1:

```
item 1 of ("red", "green", "blue") --  "red"
the third item of ("c","d","e","f","g") -- "e"
```

Lists are very useful for organizing values. SenseTalk provides a number of commands and operators for creating and working with lists (see Lists and Property Lists).

Property Lists

A property list is similar to a list, but instead of containing an ordered list of values, its values are each identified by name:

```
(size:12, color:blue)
(name:"Jason", age:67, phone:"555-1234", city:"Denver")
```

Properties in a property list are accessed by name:

```
property "width" of (shape:"oval", height:12, width:16) -- 16
```

Property lists are actually a simple form of object. Property lists viewed as simple data containers are described in Lists and Property Lists. Objects are described in detail in Objects, Messages, Handlers, and Helpers.

Ranges

A range uses a beginning and ending value to specify a range of numbers, dates, times, or characters:

```
1..200
"May 1" to "June 30"
"A" .. "Z"
```

Ranges can be ascending or descending, and a step value can be given:

```
500 down to 200 by 10
start to finish by step
```

A range value can be used directly as a range, or to generate a list of values. Or it can be accessed like a list itself:

```
put item 12 of "May 1" to "June 30" -- "May 12"
```

Ranges and their uses are described in detail in Ranges, Iterators, and Each Expressions

Special Values

Time Intervals

In some situations it is useful to work with intervals of time. SenseTalk deals with time intervals measured in seconds (including fractions of seconds), but allows you to express them in a natural way, using the terms `weeks`, `days`, `hours`, `minutes`, `seconds`, `ticks` (sixtieths of a second), `milliseconds` (thousandths of a second), and `microseconds` (millionths of a second):

```
wait 2 minutes 12.6 seconds
if the time - startTime < 150 milliseconds then logSuccess
put the date plus one week into nextWeek
put 1 into mtimeout -- number of minutes
put 15 into stimeout -- number of seconds
wait mtimeout minutes and stimeout seconds
```

Byte Sizes

Files and blocks of data are usually measured in bytes, and often get quite large. To simplify working with these data sizes, SenseTalk allows you to express them in a natural way, using the terms `bytes`, `kilobytes` (or `KB`), `megabytes` (or `MB`), `gigabytes` (or `GB`), or `terabytes` (or `TB`) as shown in these examples:

```
if the diskspace is less than 5 megabytes then put "5 MB warning"
if the size of file logFile > 256 KB then trimLogFile
put (the size of file movie) / 1 MB into movieSizeInMegabytes
```

Binary Data

Some applications, such as those that manipulate sounds or images at a low level, may need to deal directly with binary data. For situations where you need to create or compare binary data of any size, SenseTalk provides binary data literals. Data literals are written as any even number of hexadecimal digits enclosed in angle brackets `< >`. Spaces and newlines in the data literal are ignored:

```
put <AF326B47 0058D91C> into value64
```

Containers

The term “container” is used to refer to anything that can store a SenseTalk value and allow that stored value to be changed. SenseTalk containers include local, global, and universal variables, and properties of objects. It is also possible to treat any writable file as a container that can be stored into or modified directly by SenseTalk. In addition, any chunk of a container is also a container (see Chunk Expressions). These capabilities combine with SenseTalk’s `put` and `set` command to provide very powerful data handling.

SenseTalk also supports references to containers. By storing a reference to one container within another container, their values become linked in such a way that changing the value of one container will change the value of the other. This simplifies some operations and allows values to be shared in ways that would not otherwise be possible.

Variables

Variables are named containers. They serve as holders for values, similar to the “memory” function on a calculator. Unlike most simple calculators, which have only a single memory location that holds one number, you can store many values at once in different variables. Also, the value stored in a variable is not limited to a single number, but can be an arbitrarily large amount of data. To keep track of the different variables, each one has a name.

SenseTalk provides three kinds of variables:

- local variables
- global variables
- universal variables

Names of local, global, and universal variables must begin with a letter or an underscore character (_). The name can be any length and may contain letters, digits, and underscores. All variable names are case-insensitive (upper- and lower-case letters are treated as equivalent).

Local Variables

Local variables are the most common. They exist within the context of a single handler in a script. To create a local variable, you simply put something into it in a script. For example, the command `put 5 into foo` will create the local variable **`foo`** if it doesn’t already exist and store the number 5 in it.

Local variables cannot be accessed outside of the handler (or script) where they are used. References to **`foo`** in another handler will be to a separate **`foo`** that is local to that other handler. Local variables are temporary: their values go away when a handler finishes executing.

Parameters declared after the handler name in an `on` or `function` declaration are a special type of local variable. Parameter variables are different from other local variables because when the handler begins executing they already contain values passed by the sender of the message. Other local variables have no initial value when the handler begins executing.

Note

Local variables that have not been assigned a value (and that do not have a special predefined value, as described in Constants) will evaluate to their names. This provides a form of unqouted text string literal, which can be convenient in some instances, but may lead to confusion if you later choose to store something into a variable by that name. Generally, it is best to always use quotes around text literals to avoid any unexpected results.

Undefined Variables and the StrictVariables Global Property

Local variables that have not yet been assigned a value will ordinarily be treated as “unquoted literals” —in other words, their value is the same as their name:

```
put Hello -- displays "Hello"
```

Sometimes this behavior may lead to trouble, such as if you inadvertently misspell the name of a variable later in the script and expect it to have the value previously stored in it. To help with debugging your script in such cases, or if you simply prefer a more rigorous approach, you may set `the strictVariables` global property to true. When this property is set, any attempt to access a variable that has not been explicitly declared or assigned a value will throw an exception rather than returning the variable’s name:

```
put Bonjour into greeting -- stores "Bonjour" in the variable greeting
set the strictVariables to true
put Bonjour into greeting2 -- throws an exception
```

Note

Some local variables have "predefined" values. The `strictVariables` property does not affect the use of these values – you can continue to use them even though you have not explicitly assigned values to those variables (see Constants).

Global Variables

Global variables can be referenced from any handler within a project. Unlike local variables, global variables retain their value between different handlers. While local variables are undefined at the beginning of a handler each time that handler is called, global variables may already have a value that was assigned previously, either by the same handler (on an earlier call to it) or by a different handler.

To create and use global variables, they must be declared as global within each handler where they are used, so that SenseTalk can distinguish them from local variables. This is done using the `global` keyword. The `global` keyword can be used at the beginning of a line, like a command, followed by a list of the names of variables being declared as globals. This type of declaration must appear before any other use of those variables in the handler. Usually, therefore, it is good practice to place global declaration lines at the beginning of a handler, as in this example:

```
to doSomethingUniquely
```

```
    global didItBefore  -- declare the global variable
    if didItBefore is true then -- don't allow it to be done twice
          answer "Can't do it again!"
    else
          dotheThingToBeDone
          put true into didItBefore
    end if
end doSomethingUniquely
```

It is also possible to refer to a global variable without declaring it in advance, by simply preceding its name with the `global` keyword in any expression. Using this approach, our example could be rewritten like this:

```
to doSomethingUniquely
    if global didItBefore is true then
          -- don't allow it to be done twice
          answer "Can't do it again!"
    else
          dotheThingToBeDone
          put true into global didItBefore
    end if
end doSomethingUniquely
```

Note

Unlike local variables, global and universal variables that have not been assigned a value are simply empty – they don't evaluate to their names.

Because global variables are "global" they can be accessed from any handler in any script within a project. This provides a useful means of sharing information between different handlers. Care must be taken, however, not to use the same global variable name for different purposes in different parts of a project.

For example, if you used the doSomethingUniquely handler above to keep an action from being performed twice, and then copied and pasted the same handler into a different script (changing only the dotheThingToBeDone line) so that some other action would also be restricted to running just once, you would have a problem. Both scripts would now be using the same global variable (didItBefore) to keep track of whether their action had already been performed, so the first one that calls its doSomethingUniquely handler will prevent the other one from performing its action at all!

To solve this problem, you would need to use different global variables in each handler, by changing the variable names (didSomethingBefore, didTheOtherThingBefore, etc.) or come up with some other means of distinguishing between the two cases.

GlobalNames function

The `globalNames()` function (or `the globalNames`) may be used to get a list of the names of all global variables that have been assigned values.

Universal Variables

Universal variables are created and used in exactly the same manner as global variables. The difference between global variables and universal variables is defined by the host application in which SenseTalk is running. Typically, universal variables have a broader scope than globals. For example, globals may exist within a single project while

universals may allow information to be shared in common between different projects. Or, as is the case in EggPlant, globals may have their values reset following each script execution while universals retain their values between runs during an entire session.

Universal variables must be declared using the `universal` keyword, either on a separate declaration line sometime before they are referenced in each handler where they are used, or immediately before the variable name within any expression, in a manner identical to that for global variables.

UniversalNames function

The `universalNames()` function (or `the universalNames`) may be used to get a list of the names of all universal variables that have been assigned values. The following example will display the names and values of all universal variables:

```
repeat with each item of the universalNames
   put "Universal " & it & " is " & value("universal " & it)
end repeat
```

Variable Types Summary

In summary, there are three distinct types of variables that you may use in your scripts.

- Global and universal variables must be declared in each handler that uses them, or else preceded by the word `global` or `universal` each time they are used.
- Undeclared variables that are not preceded by the word `global` or `universal` within a handler are local to that handler.
- Local variables have a value within a single execution of a single handler, and are discarded when the handler finishes executing.
- Global variables have a value that persists across handlers. They are discarded at the end of a run.
- Universal variables are similar to global variables, but typically have a larger scope as defined by the host application. For example, in EggPlant universal variables have a value that persists across handlers and across runs. They are discarded when EggPlant quits.

The three types of variables are all distinct (that is, you may have local, global, and universal variables all in existence at the same time with the same name but different values).

Deleting Variables

Variables are created automatically by assigning a value to them. They may also be deleted, using the `delete variable` command, or one of its variants, as shown here:

```
delete variable holder -- makes local variable "holder" undefined again
delete local password
delete global accountNumber
delete universal pendingTasks
```

Metadata in Variables

In some cases variables may contain extra information (also known as metadata) in addition to the ordinary value (the data) that they contain. For example, when a variable holds a date or time value, the actual data (the value) that is stored in the variable is a number that specifies an instant in time. In addition to this time value, the variable also holds a `format` that specifies how the value should be presented to the user or represented when the value is requested as text (see Working with Dates and Times).

As another example, when a variable holds a list, the list of values that it contains is the data (contents) of the container. The metadata in this case is the `currentIndex` value that allows the list to be used as an iterator (see Ranges, Iterators, and Each Expressions).

In each of these cases, in addition to the data contents of the variable, SenseTalk makes the metadata accessible to the script as a property of the variable:

```
put the date into currentDate
put currentDate -- 10/22/10
put currentDate's format -- %m/%d/%y
put "%Y" into the last two characters of currentDate's format
put currentDate -- 10/22/2010
```

Files

Files on your computer's disk can be accessed using a traditional "programming" approach with the file access commands (*open file*, *read from file*, etc.). However, for quick and easy access to the contents of a file, a script may simply access the file directly as a container, without the need for opening or closing it.

```
put file "/tmp/myFile" into fileContents
```

To treat a file as a container, simply refer to the file using the syntax **file** *filename*, at any point in a script where you would use a variable name. Here filename may be either the full pathname of the file, or its local name relative to the current working folder (or a variable or expression that yields its full or relative name). The text contained within the file is its value. When a script puts something into a file, the changed text will appear in the file on the disk when that file is next accessed.

SenseTalk's file access features are described fully in Working with Files and File Systems.

Important Note

A text value stored into a file is permanent — it is saved on the disk automatically. So be careful in how you use this feature — a single `put` or `set` command can wipe out the entire contents of an existing file.

Chunks of Containers

Any chunk of a container is also a container. That is, using chunk expressions, SenseTalk allows you not only to access specific lines, items, words or characters within text, but also to store into them, as long as the source value referenced by the chunk expression is itself a container.

Chunk expressions are described in detail in Chunk Expressions. Here are some examples showing access to a chunk as a container:

```
put "The movie was good" into rating
put "excellent" into the last word of rating
put rating -- The movie was excellent
put "last " before word 2 of rating
put rating -- The last movie was excellent
```

Storing Into Containers

The put, set, and get commands are used to store values into containers. Several other commands will also change the value in a container.

◊ put, set commands

The put ... into command is used to assign a value to a container:

```
put 5 into count
put true into global peace
put customerName into word 2 of line 6 of customerStatement
```

The set command can do the same thing:

```
set count = 5
set global peace to be true
set word 2 of line 6 of customerStatement to customerName
```

Tech Talk

Syntax: put *source* into *container*
set *container* [= | to {be {equal to}} | equal] *source*

Appending Values

The put command (but not set) can also append values at the beginning or end of a container, by specifying before or after instead of into:

```
put "flower" into wordToMakePlural -- flower
put "s" after wordToMakePlural -- flowers
put heading & return before reportBody
put " " & customerLastName after word 2 of line 6 of customerStatement
put "$" before character firstDigit of Amount
```

Tech Talk

Syntax: put *source* [before | after] *container*

Storing Multiple Values At Once

Both the put and set commands can store values into several containers at once. To do this, list the containers separated by commas and enclosed in parentheses or curly braces. If the source value is a single non-list value, that value will be assigned to all of the destination containers:

```
put zero into (a,b,c) -- sets a, b, and c all to zero
set (startTime, endTime) to "noon"
```

If the source value is a list, consecutive items from the source list will be assigned to each of the destination containers in turn. Excess values will be ignored:

```
put (1,2,3) into (a,b,c) -- sets a to 1, b to 2, and c to 3
set (x,y) to (y,x) -- swaps the values of x and y
```

Source values will be skipped if no container is provided to store them into:

```
put (1,2,3) into (a,,c) -- sets a to 1, and c to 3 (2 is ignored)
put "html://foo/bar" split by "/" into (scheme,,root,leaf)
```

Extra values at the end of the source list can be gathered into a single container by using "..." (three dots) after the final container:

```
put (1,2,3,4) into (a,b...) -- sets a to 1, and b to (2,3,4)
```

◊ get command

The get command offers another way to access a value, which can then be referred to as **it**:

```
get the last word of address
if it is "DC" then shipNow
```

Note that **it** is actually a local variable that is also stored into by several other commands besides get, including the ask, answer, convert, and read commands and the "repeat with each" form of the repeat command (see Appendix B – All About "It").

Tech Talk

Syntax: get *source*

Other Commands Modify Container Values

Several other commands will also change the contents of containers, including the `add`, `subtract`, `multiply`, `divide`, `insert`, `replace`, and `delete` commands, and some forms of the `convert`, `read`, `sort`, `split`, and `join` commands:

```
add qty to item productNum of inventoryQty
divide profit by shareholderCount
convert lastUpdate to long date
```

Properties of Objects

Both scripts stored in text files, and objects created dynamically by your scripts as they run can have properties. Most of these properties can be modified by a script. Any modifiable property of an object can be treated as a container.

Script Properties

SenseTalk treats any script stored in a file as an object. All such script objects have a name. The name can be accessed using the `name`, `short name`, `abbreviated name`, and `long name` properties of the object. The short and abbreviated forms return the name of the script file without any file extension; the long form returns the full pathname of the file. The `long id` property of any script or object is a unique identifier that can be used to refer to the object. These properties are read-only.

Script files may also define additional properties in a “properties declaration”. These are treated as modifiable custom properties of the object at runtime, which can be manipulated as containers.

Custom Properties

Any object can also have custom properties assigned to it. All such properties are modifiable, and may therefore be treated as containers, as shown in this example:

```
put new object into sam -- create an object with no properties
put "Sam Adams" into property "name" of sam -- set the name
put 36 into the age of sam -- set the age property to 36
add one to sam's age -- increment sam's age
put "uel" after word 1 of sam's name
put sam.name & ", age " & sam.age -- Samuel Adams, age 37
```

Objects and their properties, including a number of special properties not mentioned here, are described in detail in Lists and Property Lists, and Objects, Messages, Handlers, and Helpers.

Local and Global Properties

In addition to properties of objects, there are also a number of properties which do not belong to any one object, but rather to the SenseTalk runtime environment itself. These include “local” properties such as the numberFormat, the itemDelimiter, and the caseSensitive, and also “global” properties such as the folder, the defaultNumberFormat, the defaultItemDelimiter, the shellCommand, and many others. All of these properties can be treated as containers for changing their values.

Local and Global properties are accessed by using the word "the" followed by the name of the property.

Tech Talk

Syntax: `the` *localOrGlobalPropertyName*

Local Properties

These properties govern behavior locally within a handler. If they have not yet been set to a specific value within the current handler, they will have a default value as indicated for each property. When the value of one of these properties is set in a handler, it will have that value only within the local handler. Handlers which called the local handler, or which are called by it, will not be affected.

`the numberFormat`	the format for displaying numbers; defaults to the value of `the defaultNumberFormat` global property — see "Conversion of Values" in Expressions
`the itemDelimiter`	the delimiter used for text items; defaults to the value of `the defaultItemDelimiter` global property — see Chunk Expressions
`the wordDelimiter`	the characters that separate words in text; defaults to the value of `the defaultWordDelimiter` global property — see Chunk Expressions
`the wordQuotes`	controls how quotes are treated in word chunks; defaults to the value of `the defaultWordQuotes` global property — see Chunk Expressions
`the caseSensitive`	whether text comparisons that don't explicitly specify "considering case" or "ignoring case" are case sensitive or not; defaults to false — see Expressions
`the centuryCutoff`	controls which century two-digit years are assumed to belong to; defaults to the value of `the defaultCenturyCutoff` global property — see Working with Dates and Times
`the listInsertionMode`	whether a list inserted into another is nested in the other list, or is inserted item by item; defaults to the value of `the defaultListInsertionMode` global property — see Lists and Property Lists
`the evaluationContext`	controls how variables will be evaluated in `do`, `send`, `value()` or `merge()` expressions (whether as local, global, or universal variables); defaults to "Local" — see Working with Messages

Global Properties

The following properties are global in scope. They can be changed by any handler at any time, and once they are changed their new value will be in effect from that point forward in all handlers.

Note

Most global properties have a standard initial setting at the start of script execution, as indicated in the discussion of that property. These standard values may be overridden by settings in the user defaults database for the host application, using the property name prefixed by "ST'. For example, the shellCommand property would be set using the name STShellCommand in the user defaults.

`the listFormat`	control formatting of displayed lists — see "Conversion of Values" in Expressions
`the propertyListFormat`	control formatting of displayed property lists — see "Conversion of Values" in Expressions
`the folder` or `the directory`	the current working folder in the filesystem — see Working with Files and FileSystems
`the defaultNumberFormat`	the default format for displaying numbers — see "Conversion of Values" in Expressions
`the defaultItemDelimiter`	the default delimiter for text items — see Chunk Expressions
`the defaultWordDelimiter`	the default delimiter characters for words — see Chunk Expressions
`the defaultWordQuotes`	the default quote characters for word chunks — see Chunk Expressions
`the clockFormat`	set to "12 hour" or "24 hour" to control time formats — see Working with Dates and Times
`the timeFormat`	a property list defining all of the available date/time formats — see Working with Dates and Times
`the timeInputFormat`	a list of formats used in recognizing date and time values — see Working with Dates and Times
`the frontScripts`, `the backScripts`	special lists of objects that receive messages before or after the target object as part of the message passing path — see Objects, Messages, Handlers, and Helpers
`the colorFormat`	the format for displaying colors — see Working with Color
`the namedColors`	the defined colors accessible by name — see Working with Color
`the shellCommand`	the Unix shell to be used by the `shell()` function — see Other Commands and Functions
`the strictVariables`	when set to "true", accessing the value of a local variable that has not been declared or stored into will throw an exception instead of returning the variable's name — see Undefined Variables and the StrictVariables Global Property earlier in this section
`the strictProperties`	when set to "true", accessing an undefined property of an object will throw an exception rather than returning an empty value — see Lists and Property Lists
`the strictFiles`	when set to "true", accessing the contents of a nonexistent file will throw an exception rather than returning an empty value — see Working with Files and FileSystems
`the breakpointsEnabled`	when set to "false", `breakpoint` commands will be ignored — see the `breakpoint` command in Other Commands and Functions
`the throwExceptionResults`	when set to "true", any command or function that would set `the result` to an exception object will throw that exception rather than merely setting `the result` — see the `result` function in Working with Messages
`the resultHistory`	contains a list of the most recent result values — see the `result` function in Working with Messages
`the resultHistoryLimit`	controls the maximum number of items in `the resultHistory` — see the `result` function in Working with Messages
`the exception`	set to empty or a caught exception by a `try` block — see Script Structure

`the watchForScriptChanges`	when set to “true”, SenseTalk will check script files for changes each time a message is sent to an object that was loaded from a script file and reload it if needed (the default value is false) — see Working with Messages
`the folderNamesEndWithSlash`	when set to “true” (the default), folder names are returned with a slash at the end — see Working with Files and FileSystems
`the readTimeout`	the maximum time (in seconds) that a `read` or `open socket` command will take before timing out — see Working with Files and FileSystems
`the defaultStringEncoding`	controls the encoding used when reading and writing text strings in files — see Working with Files and FileSystems
`the duplicatePropertyKeyMode`	controls how duplicate keys in property lists are handled — see Lists and Property Lists
`the umask`	controls the permissions for newly-created files — see Working with Files and FileSystems
`the URLCacheEnabled`	when set to “true” the contents of accessed URLs may be cached for later re-use — see Working with URLs and the Internet
`the URLErrorLevel`	specifies the lowest URL status value that is treated as an error — see Working with URLs and the Internet
`the URLTimeout`	the maximum time (in seconds) that a URL request will take before timing out — see Working with URLs and the Internet
`the endValue`	the value of the `end` constant, returned by iterators when they have no more values to return — see Ranges, Iterators, and Each Expressions

References to Containers

Most of the time, containers in a script are used by storing a *value* directly into a container — or reading a value from one — by simply specifying a variable name (or other container identifier) directly in the script. Sometimes it can be extremely useful, however, to store a *reference* to a container, which can later be used to access the value of that container. The reference holds the *identity* of a container rather than its value, and accessing the reference will read or write the contents of the container that it refers to.

Expert Feature

References are an advanced capability mainly of interest to experienced scripters. Beginning users are invited to skip this topic.

Characteristics of References

References have a few characteristics that you will want to be aware of as you work with them. To begin to understand how references work, consider the following example using a reference to a simple variable:

```
put 32 into age -- age is now a variable containing the value 32
put a reference to age into yearsOnThePlanet -- store a reference
```

```
put yearsOnThePlanet -- 32 (yearsOnThePlanet refers to age's value)
add 1 to age -- (age is now 33)
put yearsOnThePlanet -- 33 (references are dynamic)
```

In the first line above, a simple value (`32`) is assigned to an ordinary variable (`age`). The next line stores a reference to the variable `age` into the variable `yearsOnThePlanet`. Now, any access to the value of the `yearsOnThePlanet` variable will actually be accessing the value of the `age` variable, as illustrated by the next line of the script, The last two lines of the script increment that value of the `age` variable and show that the `yearsOnThePlanet` variable accesses the updated value.

References Bind Tightly

The connection established by a reference is “sticky” (`yearsOnThePlanet` is glued tightly to `age` in the above example) so assignments work in the other direction as well — changing the value of either will change the value of both:

```
put 29 into yearsOnThePlanet
put age -- 29
```

Reference Syntax

References can be made using the words `container`, `reference`, `reference to`, or `refer to`, or using the shorthand symbol ‘`@`’ before the container. All of the following are equivalent:

```
put container thing into watcher
set watcher to be a reference to thing
put @thing into watcher
set watcher to reference thing
set watcher to refer to thing
```

References Can Be Stored in a List or a Property List

References can not only be stored in variables, as shown in the examples so far, but may also be stored as items in a list, or as properties of an object or property list. In this case, accessing the specific item or property that is a reference will access the container it refers to, as shown in this example:

```
put 13 into luckyNumber
put ("horseshoe", container luckyNumber, "penny") into charms
put charms -- (horseshoe, 13, penny)
put 7 into item 2 of charms
put charms -- (horseshoe, 7, penny)
put luckyNumber -- 7
```

References May Refer to Properties, Files, and Chunks

References may refer to any type of container, not just variables. This includes all the different types of container discussed earlier in this section, including properties, files, and even chunks of any container:

```
set highScore to reference item 2 of line 3 of file "/tmp/stats"
put @ item 2 delimited by "." of the numberFormat into decimalFormat
```

References Are Dynamic

When a reference is made to a property of an object or to a chunk of a container (such as "`reference to word 3 of sentence`") that reference is evaluated dynamically each time it is used. This means that if the contents of

`sentence` are changed, the reference will always access the third word of its new contents.

References Are Not "Contagious"

Assignments in SenseTalk are always done by value unless a reference is explicitly indicated. So assigning a variable containing a reference to another variable will not create another reference, but only a copy of the current value of the source container. To create another reference, you must specifically indicate a reference during the asignment (specifying a reference to a reference will result in a second reference to the same source).

```
put 123 into source
put a reference to source into ref -- 123 (ref is a reference)
put ref into number -- 123 (number is NOT a reference)
put @ref into ref2 -- 123 (ref2 is now another reference to source)
put 456 into source
put ref -- 456
put number -- 123
put ref2 -- 456
```

References Can Be Reassigned

Storing a reference into a variable that already contains a reference will simply reassign that variable. It will not change the value that the variable previously referred to:

```
put (123, 456) into (source1, source2)
put a reference to source1 into ref
put ref -- 123
put a reference to source2 into ref -- ref is now reassigned
put 789 into ref -- source1 is unaffected
put (source1, source2) -- (123, 789)
```

To change the container that was referred to into a reference instead of reassigning the original reference variable, you can request that explicitly:

```
put @source1 into the container referred to by ref -- ref still refers to
source2, which now becomes a reference to source1
put "We're Twins" into ref -- assign to ultimate container
put (source1, source2) -- (We're Twins,We're Twins)
```

Deleting By Reference

Deleting a reference to a chunk or property with the `delete` command will delete the chunk or property it refers to (note that this is not the same as deleting the variable containing the reference — see below).

```
put (1,3,5,7,9,11,13) into odds
set num to refer to item 5 of odds
put num -- 9 (num refers to item 5)
delete num
put odds -- (1,3,5,7,11,13)
put num -- 11 (num now refers to the new item 5)
```

References Are Persistent

Once a variable becomes a reference, it stays tied to the container it refers to until it is reassigned as a different reference, or the reference itself is deleted. You might think that you could free up a reference variable for other uses

by simply putting empty into it, but this will merely put empty into the container it refers to! To use the variable for another purpose, you must first delete it, using the `delete variable` command (*not* the `delete` command — see above). If the reference is stored as an item in a list or a property of an object, you can delete that item or property.

```
put "green" into source
set localColor to refer to source
set localColor to "yellow"
put source -- "yellow"
delete variable localColor -- localColor is now undefined (no longer a reference)
set localColor to "blue" -- assigns its own value
put source -- "yellow"
```

There are two exceptional cases where a reference can become a plain variable without explicitly deleting it. Commands that implicitly set the value of the special variable `it` (see the list in Appendix B) will first break any reference stored in `it` before assigning it a new value. Also, if a repeat construct specifies a loop variable that is currently a reference, that reference will be cleared before the variable is used in the repeat.

Using References

References can be used in many ways in your scripts. Here are a few of the most common uses.

Passing Parameters by Reference

One common use of references is to pass a reference as a parameter to another handler. By passing a reference, the called handler is able to change the value in the local handler. Here is an example:

```
put "candy" into tastyFood
pluralize @tastyFood -- pass by reference
put tastyFood
to pluralize aWord
  if aWord ends with "y" then put "ies" into last char of aWord
  else put "s" after aWord
end pluralize
```

When the script above is run, it displays "candies". The variable tastyFood is passed by reference from the initial handler to the pluralize handler. Because it is a reference, when aWord is changed by the pluralize handler, it actually changes the value of the tastyFood variable in the initial handler.

Returning a Reference / Using a Function Result as a Container

Just as passing a reference to another handler gives that handler the ability to modify the contents of the container it refers to, it is sometimes useful for a function to be able to return a reference to a container that the caller can then modify. For instance, a function might return a reference to a global variable or a file selected by the function, or a reference to a chunk of such a container. To do this, there are two things that have to happen. First, the function being called must return a reference. Secondly, the calling handler must indicate that it wants to use the returned value as a container (by default, function return values are always treated by value otherwise).

As an example, suppose you want to insert a value into one of two lists, depending on which list has fewer items (a more realistic example might involve several lists). Here is a script that implements this idea using a function to select the appropriate list:

```
put "a,b,c,d,e,f" into firstBox
put "9,8,7,6" into secondBox
```

```
put ",Here" after container chooseContainer(@firstBox, @secondBox)
put firstBox -- "a,b,c,d,e,f"
put secondBox -- "9,8,7,6,Here"
to chooseContainer @c1, @c2
    if the number of items in c1 < number of items in c2
    then return container c1
    else return container c2
end chooseContainer
```

There are a lot of references in this example. The word "container" (or "@" or "reference") is required before the call to `chooseContainer()` to tell SenseTalk to treat the value returned by the function call as a container. For this example to work, the parameters must also be passed by reference (**`@firstBox`**, **`@secondBox`**) since they originate in the calling script, and of course the selected container must be returned by reference. The result is a function that accepts references to any two containers, and returns a reference to the one containing fewer items. Marking the function's incoming parameters as references (**`@c1, @c2`**) is optional, as it has no functional meaning in the current version of SenseTalk, but it helps to remind users of the chooseContainer function that it should be called with parameters passed by reference.

Determining if a Value is a Reference

In the preceding example, the chooseContainer() function indicates to users that it should be called using references by the use of "@" symbols before its parameters (**`@c1, @c2`**). However, SenseTalk doesn't enforce this in any way. To make the function more robust, it could check whether the parameters were actually passed by reference, and alert the user if it was called incorrectly. The `is a` (or `is not a`) operator can be used for this purpose. Here is an improved version of the function that performs this test:

```
to chooseContainer @c1, @c2
  if c1 is not a reference or c2 is not a reference then
      throw "Not A Reference", "Must pass containers by reference"
  end if
  if the number of items in c1 < the number of items in c2
  then return container c1
  else return container c2
end chooseContainer
```

Repeat With Each ... By Reference

A special "by reference" option of the repeat with each command will set the iterator variable to be a reference to the current element rather than merely its value. This can greatly simplify writing a script that not only iterates over the chunks of a container but makes changes to some of those elements, as in this example:

```
put "come all you good people and make merry" into phrase
repeat with each word of phrase by reference
         if the last char of it is not in "aeiou" then put "e" after it
         if it is "you" then set it to "ye"
end repeat
put phrase -- "come alle ye goode people ande make merrye"
```

Expressions

Expressions combine values using various operators to yield new values. For example, the very simple expression 1+2 uses the '+' operator to produce the value 3. SenseTalk provides a wide range of operators which serve many different needs.

The values used as components in expressions can be of many different types. They may be simple values, such as numbers or text strings (as described in Values), or they may be values that are the contents of containers (Containers). Still another type of value is one provided by a "function". A function is a source of value that retrieves information that may vary. SenseTalk provides many functions for a wide variety of purposes. This section describes how to use functions in SenseTalk expressions.

SenseTalk is a "typeless" language, so containers can store any type of value: numbers, text, dates, lists, etc. Values are automatically converted as needed, so, for example, if you perform a mathematical operation on a text value, SenseTalk will convert that value to a number internally. For converting internal values back into a text format, SenseTalk provides mechanisms that allow you to control the format used.

Operators

Operators are mathematical and logical words and symbols you can use in an expression. The operators are each described in detail later in this section. For reference, they are listed here by category:

Mathematical Operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation (raise to a power)
`squared`	raise to the second power
`cubed`	raise to the third power
`%`	percent
`div`	integer division
`rem`	integer remainder
`modulo`	mathematical modulo
`is a multiple of` `is divisible by`	test for exact multiple of a number
`rounded to`	round to a number of decimal places
`rounded to nearest`	round to the nearest multiple of a value

Comparison Operators

`=`	equal
`is`	equal

`is not`	not equal
`<>`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`between`	tests whether a value is within a range of values

Logical Operators

`and`	logical and
`or`	logical inclusive or
`not`	logical negation
`and if`	short-circuit logical and
`or if`	short-circuit logical inclusive or

Text Operators

`&`	text string concatenation
`&&`	text concatenation with space inserted
`is in`	one string contained in another, or value contained in a list
`contains`	one string contains another, or value contained in a list
`begins with`	one string begins with another, or list begins with a value or sublist
`ends with`	one string ends with another, or list ends with a value or sublist
`is among`	one value is present among the chunks, items, keys or values of another

Property List Operators

`adding`	add properties from one property list to those from another
`replacing`	add properties from one property list to those from another, overriding duplicates
`removing`	remove specified properties from a property list
`retaining`	remove properties from a property list, other than those specified

Date Operators

`ago`	time prior to now
`hence`	time later than now

Miscellaneous Operators

`( )`	grouping
`is a`	tests for types (number, integer, point, rect, date, logical, color)
`there is a`	yields true if the specified object or file exists
`exists`	yields true if the specified object or file exists
`is within`	tests whether a point or rectangle is within a rectangle
`&&&`	list concatenation
`joined by`	converts a list or property list to text
`split by`	converts text to a list or property list
`as`	converts a value to a given internal representation
`(if ... then ... else ...)`	selector expression

Precedence of Operators

Operators in complex expressions are evaluated in a specific order based on their precedence. The precedence of operators, from highest (those that are evaluated first) to lowest (those evaluated last) is as follows:

1st	`( )` *(expressions enclosed in parentheses are evaluated first)*
2nd	*(implicit concatenation – see below)*
3rd	`not - ` *(unary minus, or negation)*
4th	`^ squared cubed % ago hence as`
5th	`* / div rem modulo joined by split by rounded to rounded to nearest`
6th	`+ - adding replacing removing retaining`
7th	`& && is a multiple of is divisible by`
8th	`&&&`
9th	`> < >= <= between contains is in is among is a is within`
10th	`= <> begins with ends with`
11th	`and and if`
12th	`or or if`

When operators at the same level of precedence are used together in an expression, they are evaluated from left to right. In any case, parentheses can be used around a sub-expression to force that group to be evaluated before other parts of the expression (see Uses of Parentheses, below).

Implicit Concatenation

Implicit concatenation occurs when string literals, constants and certain predefined variables appear sequentially in an expression with no intervening operator. For example, this expression:

```
"Line 1" return "This is " "Line 2"
```

will produce the same result as:

```
"Line 1" & return & "This is " & "Line 2"
```

In addition to the constants (return, true, false, empty, up, down) the predefined variables available for implicit concatenation are: space, tab, quote, comma, slash, backslash, newline, linefeed, lf, carriagereturn, creturn, cr, crlf.

Uses of Parentheses

Parentheses are used for several different purposes in SenseTalk expressions.

Grouping Operations

Parentheses can be used to force operations to be performed in a particular order, or to resolve any ambiguity that might otherwise exist. For example, consider this ambiguous expression:

```
the square root of nine plus sixteen
```

If a friend asked you "What is the square root of nine-plus-sixteen?" you would promptly add nine and sixteen to get twenty-five and then take the square root and give the desired answer: "five" (you do that sort of thing all the time, right?). Or, they might ask the question with slightly different emphasis and a brief pause after the word "nine", as "What is the square-root-of-nine, plus sixteen?". In this case you would first take the square root of nine and then add that result to sixteen to get the desired answer: "nineteen".

In the case of a script, there are no vocal clues to tell a reader which of the two possible interpretations to give to this expression. The meaning appears to be ambiguous -- it could be interpreted either way. In fact, SenseTalk's rules of precedence will come into play, and it will evaluate the second way, giving a result of 19. If that wasn't what you intended, you'll need to use parentheses around the part of the expression that you want to be evaluated first:

```
the square root of (nine plus sixteen)
```

Readability is important in a script so that a reader of the script (including yourself at a later date) will be able to understand exactly what the script is doing. So even if the built-in rules give the answer you want, it may be a good idea to include parentheses to make it clear what was intended:

```
(the square root of nine) plus sixteen
```

Forcing Evaluation as an Expression

In certain contexts, a word will be treated as a literal value (the same as though it were in quotes). Enclosing such a word in parentheses forces it to be evaluated as an expression instead (specifically, as a variable name if it is a single word). This is most commonly used in the case of property names.

```
account.balance
```

This expression accesses the "balance" property of the account object. But suppose in your script you sometimes want to access the balance and at other times the availableBalance. Earlier in the script a decision is made about which type of balance to use:

```
set balanceToUse to "balance" -- use full actual balance unless withdrawing
if action is "withdrawal" then set balanceToUse to "availableBalance"
```

To access the balance you might try this:

```
account.balanceToUse
```

Unfortunately, this expression tries to access the property named "balanceToUse" of the account, which is the wrong

name. To use the property name that is stored in the balanceToUse variable, use parentheses around the variable name to force it to be evaluated as an expression:

```
account.(balanceToUse)
```

Required Parentheses

Some expressions require the use of parentheses. For example, a list can be made by listing its items in parentheses, separated by commas. Other examples include property lists, function calls with multiple parameters, and the (if...then...else...) selector expression.

Vector Arithmetic with Lists

Vector arithmetic is supported. You can add or subtract two lists of numbers, provided they both have the same number of items. This will add or subtract the corresponding items of the two lists. Vector operations work with nested lists as well, provided that all corresponding sublists are of equal length.

```
put (1,2,3) + (100,200,300) -- (101,202,303)
```

Multiplication and division of lists also works for lists of equal length.

```
put (100,200,300) / (2,10,100) -- (50,20,3)
```

In addition, you can multiply or divide a list by a single value (sometimes called a ‘scalar’ because it scales all the values in the list).

```
put (1,2,3) * 6 -- (6,12,18)
```

Case Sensitivity

Text comparison operators are usually not case-sensitive, as in this example:

```
put "FLoWeR" is "flower" -- true
```

You can control the case-sensitivity of operations in two ways. Set `the caseSensitive` property to true or false to define whether by default uppercase and lowercase letters are treated differently or not. This property, which is local to each handler, is set to false at the beginning of each handler, so case will ordinarily be ignored.

Case-sensitivity can also be customized for each comparison, overriding the setting of `the caseSensitive` property. To do this, specify `considering case`, `with case`, or `case sensitive` (to force case to be considered), or `ignoring case`, `without case`, or `case insensitive` (to force case to be ignored) for each operator:

```
put "FLoWeR" is "flower" considering case -- false
```

Operator Descriptions

◊ +, plus

What it Does

Adds two numbers or lists of numbers.

When to Use It

Use the + or plus operator to add two values.

Examples

```
put 12 + 97 into someSum
put a squared plus b squared into sumOfSquares
put (12,8) + (4,7) into vectorSum
```

Tech Talk

Syntax: *operand1* + *operand2*
operand1 plus *operand2*

◊ -, minus

What it Does

Subtracts one number or list of numbers from another.

When to Use It

Use the – or minus operator to obtain the arithmetic difference of two values or lists of values. Subtracting one date/time from another will give the difference as a time interval, measured in seconds.

Examples

```
put c^2 - sumOfSquares into difference
put (1,3,5,6) - (1,1,0,2) into diffList
```

Tech Talk

Syntax: *operand1* – *operand2*
operand1 minus *operand2*

◊ *, times

What it Does

Multiplies two numbers or lists, or multiplies a list by a scalar.

When to Use It

Use the * or times operator to obtain the product of multiplying two numbers. When used with two lists of equal

length, the result will be a series of products of the corresponding elements of the two lists. When one operand is a list and the other is a single (scalar) value, the result is a list of values obtained by multiplying each original list element by the scalar value.

Examples

```
put 2 * radius into diameter
put pi times diameter into circumference
put (1,2,3,4) * (2,2,1,3) -- result is (2,4,3,12)
put (1,2,3,4) * 4 -- result is (4,8,12,16)
```

Tech Talk

Syntax: *operand1* `*` *operand2*
operand1 `times` *operand2*

◊ / , divided by

What it Does

Divides one number or list by another, or divides a list by a scalar.

When to Use It

Use the `/` or `divided by` operator to divide one number by another, giving a quotient which may not be a whole number. Compare this to the `div` operator which yields a whole number.

When used with two lists of equal length, the result will be a series of quotients of the corresponding elements of the two lists. When the first operand is a list and the second is a single (scalar) value, the result is a list of values obtained by dividing each list element by the scalar value.

Examples

```
put pi / 2 into halfPi
put (1,2,3,4) / (2,1,1,2) -- result is (0.5,2,3,2)
put (2,4,5,8) / 2 -- result is (1,2,2.5,4)
```

Tech Talk

Syntax: *operand1* `/` *operand2*
operand1 `divided by` *operand2*

If *operand2* is zero this operator will return the value *infinity* which is displayed as “Inf”. Using an infinite value in other calculations will generally give the expected results.

◊ ^ , to the power of , squared , cubed

What it Does

Raises a number to a given power.

When to Use It

Use the `^` or `to the power of` operator to perform exponentiation, or use the `squared` and `cubed` operators to raise a number to the second or third power, respectively.

Examples

```
put a squared + b squared into sumOfSquares
put 6 * x^4 - 2 * y^3 into z
```

Tech Talk

Syntax: *operand1* `^` *operand2*
operand1 `to the power of` *operand2*
operand1 `squared`
operand1 `cubed`

◊ % , percent

What it Does

Treats a number as a percentage, or computes add-on or discount percentages.

When to Use It

Use `%` or `percent` for calculations involving percentages. In its simple form, `%` following a value divides that value by 100 (so `6%` is the same as `.06`). However, if `%` is used following a `+` or `–` operator, the corresponding percent of the value to the left of that operator will be increased or decreased by the specified percent.

Examples

```
put 4% -- .04
put 50 * 4% -- 2
put 50 + 4% -- 52
put 50 - 4% -- 48
put sales plus ten percent into projectedSales
```

Tech Talk

Syntax: *factor* `%`
factor `percent`
value `[ + | - | plus | minus ]` *factor* `[ % | percent ]`

◊ ( )

What it Does

Groups operations within an expression.

When to Use It

Use parentheses to control the order in which operations are performed within an expression. See the precedence list earlier in this section to understand the order in which operations are performed when parentheses are not used. When in doubt, use parentheses to ensure operations are carried out in the desired order. Also see Uses of Parentheses, above.

Examples

```
put 2 * (height + width) into perimeter
```

Tech Talk

Syntax: `(` *expression* `)`

◊ div

What it Does

Divides one number by another, giving the result as an integer.

When to Use It

Use the `div` operator to do integer division returning the quotient as an integer. The companion `rem` operator can be used to find the remainder of such an operation.

Examples

```
put cookieCount div numberOfPeople into cookiesForEach
```

Tech Talk

Syntax: *operand1* `div` *operand2*

Division by zero will yield the result “INF”.

◊ rem

What it Does

Calculates the integer remainder of a division.

When to Use It

Use the `rem` operator to obtain the integer remainder when dividing one integer by another. This is the complement of the `div` operator.

Examples

```
put cookieCount rem numberOfPeople into extrasForMe
```

Tech Talk

Syntax: *operand1* `rem` *operand2*

The result of the `rem` operator will always have the same sign as its first operand.

◊ modulo , mod

What it Does

Performs the mathematical modulo operation.

When to Use It

Use the `modulo` operator (or its abbreviation, `mod`) to obtain the amount by which one number exceeds the next-lower even multiple of another.

Examples

```
put someValue mod modulusValue into extrasForMe
```

Tech Talk

Syntax: *operand1* `modulo` *operand2*

The `modulo` operator is different from the `rem` operator, which gives the remainder of an integer division. When both operands are positive integers, `rem` and `modulo` will yield the same results. Negative numbers and non-integer values are treated much differently by the two operators, however.

◊ is a multiple of , is divisible by

What it Does

Checks whether one number is an exact multiple of another.

When to Use It

Use the `is a multiple of` or `is divisible by` operators to find out if a value is a multiple of another. That is, if the result of dividing one by the other would result in a whole number with no remainder.

Examples

```
put 2895 is a multiple of 5 -- true
put 169 is divisible by 13 -- true
put 98.6 is an exact multiple of 3.14 -- false
if cookieCount is evenly divisible by numberOfPeople then put "Hooray!"
```

Tech Talk

Syntax: *value* `is {not} {a | an} {exact | even} multiple of` *divisor*
value `is {not} {exactly | evenly} divisible by` *divisor*

◊ rounded to, rounded to nearest

What it Does

Rounds a value to a number of decimal places, or to the nearest multiple of another value.

When to Use It

Use the `rounded to` or `rounded to nearest` operators to obtain a rounded value.

Examples

```
put 123.4567 rounded to 2 places -- 123.46
put 123.4567 rounded -1 decimal places -- 120
```

```
put 98.6 rounded to the nearest multiple of 3.14 -- 97.34
put total rounded to nearest .25 into amountDue
```

Tech Talk

Syntax: *value* `rounded` {`to`} *places* {{*decimal*} `places`}
value `rounded to` {`the`} `nearest` {`multiple of`} *nearestMultiple*

These operators provide an alternate syntax for calling the `round()` and `roundToNearest()` functions (see Working With Numbers).

◊ is , are , = , equals , equal to , is equal to , does equal

What it Does

Compares two values for equality, yielding either **true** or **false**.

When to Use It

Use the `is` operator (or any of its synonyms: `are`, `=`, `equals`, `equal to`, `is equal to`, or `does equal`) to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the `considering case` option, or set the `caseSensitive` property to true (see Case Sensitivity earlier in this chapter).

Examples

```
if answer = 7 then ...
if name is "sarah" then ...
if prefix is "Mac" considering case then ...
```

Tech Talk

Syntax: *operand1* `is` *operand2* {`considering case` | `ignoring case`}

As noted above, the default operation when comparing strings with the `is` operator is to ignore case differences. The terms `with` and `without` can be used in place of `considering` and `ignoring`.

When two numeric values are being compared, they will evaluate as equal if the difference between them is less than 0.00000000001 in order to accommodate small inaccuracies which may creep in during calculations.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the `as` operator (or related functions) to explicitly control the type of comparison that is performed (see the `as` operator in this chapter for more details).

◊ is not , are not , <> , does not equal , is not equal to , isn't , aren't , doesn't equal , isn't equal to

What it Does

Compares two values for inequality, yielding either **true** or **false**.

When to Use It

Use the `is not` operator (or any of its synonyms: `are not`, `isn't`, `aren't`, `<>`, `does not equal`, `not equal to`, or `is not equal to`) to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the `considering case` option (see Case Sensitivity earlier in this chapter).

Examples

```
if answer is not 7 then ...
if name isn't "sarah" then ...
if prefix is not "Mac" considering case then ...
```

Tech Talk

Syntax: *operand1* `is not` *operand2* **{considering case | ignoring case}**

As noted above, the default operation when comparing strings with the `is not` operator is to ignore case differences. The terms `with` and `without` can be used in place of `considering` and `ignoring`.

When two numeric values are being compared, they will evaluate as unequal if the difference between them is greater than 0.00000000001 in order to accommodate small inaccuracies which may creep in during calculations.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the `as` operator (or related functions) to explicitly control the type of comparison that is performed (see the `as` operator in this chapter for more details).

◊ is less than , < , comes before , is not greater than or equal to , is earlier than

What it Does

Compares whether a value is less than another, yielding either **true** or **false**.

When to Use It

Use the `less than` operator or one of it synonyms to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the `considering case` option, or set the `caseSen-`

sitive property to true.

Examples

```
if answer < 7 then ...
if name comes before "Beetle" then ...
if prefix is less than "Mac" ignoring case then ...
```

Tech Talk

Syntax: *operand1* {is} less than *operand2* {considering case | ignoring case}

As noted above, the default operation when comparing strings with the less than operator is to ignore case differences. The terms with and without can be used in place of considering and ignoring.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the as operator (or related functions) to explicitly control the type of comparison that is performed (see the as operator in this chapter for more details).

◊ is greater than, > , is more than, comes after, is not less than or equal to, is later than

What it Does

Compares whether a value is greater than another, yielding either **true** or **false**.

When to Use It

Use the greater than operator or one of its synonyms to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the considering case option, or set the caseSensitive property to true.

Examples

```
if answer > 7 then ...
if name comes after "Hannibal" then ...
if prefix is greater than "Mac" ignoring case then ...
```

Tech Talk

Syntax: *operand1* is greater than *operand2* {considering case | ignoring case}

Tech Talk

As noted above, the default operation when comparing strings with the `greater than` operator is to ignore case differences. The terms `with` and `without` can be used in place of `considering` and `ignoring`.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the `as` operator (or related functions) to explicitly control the type of comparison that is performed (see the `as` operator in this chapter for more details).

◊ is less than or equal to , <= , does not come after , is not greater than , is not later than, is at most, is no more than

What it Does

Compares whether a value is less than or equal to another, yielding either **true** or **false**.

When to Use It

Use the `less than or equal to` operator or one of its synonyms to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the `considering case` option, or set the `caseSensitive` property to true.

Examples

```
if answer <= 8 then ...
if name does not come after "Frank" then ...
if the number of items in guestList is no more than 12 then ...
if prefix is not greater than "Mac" considering case then ...
```

Tech Talk

Syntax: *operand1* `<=` *operand2* `{considering case | ignoring case}`

As noted above, the default operation when comparing strings with the `less than or equal to` operator is to ignore case differences. The terms `with` and `without` can be used in place of `considering` and `ignoring`.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the `as` operator (or related functions) to explicitly control the type of comparison that is performed (see the `as` operator in this chapter for more details).

◊ is greater than or equal to , >= , does not come before , is not less than , is not earlier than, is at least, is no less than

What it Does

Compares whether a value is greater than or equal to another, yielding either **true** or **false**.

When to Use It

Use the `greater than or equal to` operator or one of its synonyms to compare two values. If both values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the `considering case` option, or set the `caseSensitive` property to true.

Examples

```
if answer >= 7 then ...
if name does not come before "Zoo" then ...
if customer's age is at least 17 then admitToRMovie
if prefix is not less than "Mac" considering case then ...
```

Tech Talk

Syntax: *operand1* `>=` *operand2* {`considering case` | `ignoring case`}

As noted above, the default operation when comparing strings with the `greater than or equal to` operator is to ignore case differences. The terms `with` and `without` can be used in place of `considering` and `ignoring`.

When the two operands are both of the same internal type, such as date/time values, lists, property lists, or trees, the values will be compared directly according to the rules for that type of value. Use the `as` operator (or related functions) to explicitly control the type of comparison that is performed (see the `as` operator in this chapter for more details).

◊ between , is between , is not between , comes between , does not come between

What it Does

Tests whether a given value falls or does not fall within a pair of other values, yielding either **true** or **false**.

When to Use It

Use the `between` operator or one of its synonyms (or antonyms) to test whether a "featured" value falls within a specified pair of values. The `between` operator is equivalent to "*value* >= *lowEndValue and value* <= *highEndValue.*" The tested value is compared to both end values. If it falls between them, or is equal to either end value, the

result is **true**. The end values of the range may be specified in acending or descending order.

Examples

```
if answer is between 7 and 11 then ...
if wd does not come between "Zoo" and "Zygote" ignoring case then ...
if height is between minAllowedHeight and maxAllowedHeight then ...
```

Tech Talk

Syntax: *value* {is} {not} between *endValue1* and *endValue2* {considering case | ignoring case}
value [comes between | does {not} come between] *endValue1* and *endValue2* {considering case | ignoring case}

The value being tested is compared to both of the "end values". If it is equal to either end value, or is both greater than one and less than the other, then the between expression evaluates to true. Otherwise, it evaluates to false. EndValue1 may be greater or less than endValue2. Internally, the tested value is compared to each end value independantly, so it is possible that different types of comparisons may be used for each (numeric for one and textual for the other, for example).

If both end values are valid as numbers, a numeric comparison is performed. Otherwise a textual comparison is made. Ordinarily, textual comparisons are not case-sensitive. To force a case-sensitive comparison, use the considering case option, or set the caseSensitive property to true.

◊ and , and if

What it Does

Evaluates two conditions, yielding **true** if both conditions are true, and **false** otherwise.

When to Use It

Use the and or and if operators to test for two or more conditions being true at once.

Examples

```
if x > 7 and x < 12 then ...
if operation is "test" and if fullValidate(system) then ...
```

Tech Talk

Syntax: *operand1* and *operand2*
operand1 and if *operand2*

Tech Talk

Both the `and` and the `and if` operators will yield a logical value of **true** if and only if both of their operands are true. The `and` operator always fully evaluates both of its operands, however, while the `and if` operator "short-circuits" if *operand1* is **false** and only evaluates *operand2* if *operand1* is **true**. In the second example above, the fullValidate() function will only be called if operation is equal to "test".

◊ or , or if

What it Does

Combines two conditions, yielding **true** if either one is true.

When to Use It

Use the `or` operator or the `or if` operator to test for any of several conditions.

Examples

```
if x < 7 or x > 12 then ...
if status = 99 or if file "N37" contains "ruby" then ...
```

Tech Talk

Syntax: *operand1* `or` *operand2*
operand1 `or if` *operand2*

Both the `or` and the `or if` operators will yield a logical value of **true** if either or both of their operands are true. The `or` operator always fully evaluates both of its operands, however, while the `or if` operator "short-circuits" if *operand1* is **true** and only evaluates *operand2* if *operand1* is **false**. In the example above, the search of file "N37" for "ruby" will only be performed if status is not equal to 99.

◊ not

What it Does

Negates a condition.

When to Use It

Use the `not` operator to obtain the opposite of a true or false condition.

Examples

```
if not showGreeting then ...
```

Tech Talk

Syntax: `not` *operand*

◊ &

What it Does

Joins (concatenates) two values.

When to Use It

Use the `&` operator to create a text string by combining values one after another.

Examples

```
put "The answer is:" & answer
```

Tech Talk

Syntax: *operand1* `&` *operand2*

Both operands are converted to text representations if they are not already text before concatenation.

◊ &&

What it Does

Joins (concatenates) two values with a space between them.

When to Use It

Use the `&&` operator to combine strings separated with a space.

Examples

```
put "Dear" && correspondent & "," into openingLine
```

Tech Talk

Syntax: *operand1* `&&` *operand2*

Both operands are converted to text representations if they are not already text before concatenation.

◊ &&&

What it Does

Joins two lists or values into a single list of values.

When to Use It

Use the `&&&` operator to combine two lists, to combine values into a list, or to append or prepend a value to an existing list.

Examples

```
put (1,2,3) &&& (4,5) into oneList -- (1,2,3,4,5)
put oneList &&& 6 -- (1,2,3,4,5,6)
put 0 &&& oneList -- (0,1,2,3,4,5)
put 12 &&& 42 into luckyList -- (12,42)
```

Tech Talk

Syntax: *operand1* `&&&` *operand2*

The result of this operation is always a list, even if one of the operands is empty.

◊ is in , is not in , isn't in

What it Does

Tests for the presence or absence of one value within another, giving **true** or **false**.

When to Use It

Use the `is in` operator to check whether a text string is present in another string, or whether a single value or sequence of values is present in a list, range, or property list. Ordinarily, this operator is not case-sensitive. To force case-sensitivity, use the `considering case` option.

Examples

```
if "--help" is in commandLine then ...
if "Johnson" is not in indexList then ...
put 12 is in ((11,12),(13,14)) -- true
put (2,3) is in (1,2,3,4,5) -- true
put (2,3) is in 1..5 -- true
```

Tech Talk

Syntax: *targetValue* `is` {`not`} `in` *sourceValue* {`considering case` | `ignoring case`}

Tech Talk

When *sourceValue* is a list, this operator tests whether any of its values is equal to *targetValue*. If *targetValue* is also a list, it tests whether a consecutive sequence of the values in *sourceValue* are equal to the values in *target-Value*.

When *sourceValue* is a range, it is treated the same as a list (as would be generated by converting the range to a list) with the values in that list checked for the presence of *targetValue*. This is different from the `is within` operator which checks whether a value lies anywhere between the start and end values of the range.

When *sourceValue* is a property list (object), the behavior can be determined by the object itself, or the built-in `containsItem` function (see "Checking Object Contents" in Objects, Messages, Handlers, and Helpers for details).

Otherwise, *sourceValue* is evaluated as text, and tested to see if it contains *targetValue* as a substring. To force a substring text search when *sourceValue* is a list or property list, use the `asText` function or `as text` operator to convert it to text.

◊ contains , does not contain , doesn't contain

What it Does

Tests for the presence or absence of one value within another, giving **true** or **false**.

When to Use It

Use the `contains` operator to check whether a text string is present in another string, or whether a single value or sequence of values is present in a list, range, or property list. Ordinarily, this operator is not case-sensitive. To force it to be case-sensitive, use the `considering case` option. This performs exactly the same operation as the `is in` operator, but allows the relationship to be expressed the other way around. Use whichever one feels more natural in a given context.

Examples

```
if commandLine contains " -x " considering case then ...
if word n of partNums doesn't contain "q" then ...
put nameList contains "Mayfield" into shouldInvite
put 5..17 by 2 contains 8 -- false, since only odd numbers are generated
put ("abcd","defg") contains "abc" -- false
put ("abcd","defg").asText contains "abc" -- true
```

Tech Talk

Syntax: *sourceValue* `contains` *targetValue* `{considering case | ignoring case}`
sourceValue `does {not} contain` *targetValue* `{considering case | ignoring case}`

Tech Talk

When *sourceValue* is a list, this operator tests whether any of its values is equal to *targetValue*. If *targetValue* is also a list, it tests whether a consecutive sequence of the values in *sourceValue* are equal to the values in *target-Value*.

When *sourceValue* is a range, it is treated the same as a list (as would be generated by converting the range to a list) with the values in that list checked for the presence of *targetValue*. This is different from the `is within` operator which checks whether a value lies anywhere between the start and end values of the range.

When *sourceValue* is a property list (object), the behavior can be determined by the object itself, or the built-in `containsItem` function (see "Checking Object Contents" in Objects, Messages, Handlers, and Helpers for details).

Otherwise, *sourceValue* is evaluated as text, and tested to see if it contains *targetValue* as a substring. To force a substring text search when *sourceValue* is a list or property list, use the `asText` function or `as text` operator to convert it to text, as shown in the last example above.

◊ is among , is not among , isn't among

What it Does

Tests for the presence or absence of one value as a whole chunk, list item, key or value within another.

When to Use It

Use the `is among` operator to check whether a particular value is equal to one of the chunks of a text value, one of the list items of a list, or one of the keys or values of a property list. Ordinarily, this operator is not case-sensitive. To force it to be case-sensitive, use the `considering case` option.

Examples

```
put "cat" is among the items of "dog,cat,mouse" -- true
put "be" is not among the words of "bell jibe amber" -- true
if "cost" isn't among the keys of part then set part's cost to 10
```

Tech Talk

Syntax: *targetValue* `is` {`not`} `among` {`the`} *chunkTypes* `of` *sourceValue* {`considering case` | `ignoring case`}

ChunkTypes can be any of characters, chars, words, lines, items, text items, list items, keys, values, or bytes.

◊ begins with , does not begin with , doesn't begin with

What it Does

Tests for the presence or absence of a value at the beginning of another, giving **true** or **false**.

When to Use It

Use the `begins with` operator to check whether or not a text string begins with a particular sequence of characters, or whether a list begins with a particular value or sequence of values. Ordinarily, this operator is not case-sensitive. To force it to be case-sensitive, use the `considering case` option.

Examples

```
if sentence begins with "Joshua " considering case then ...
if word n of productNames does not begin with "q" then ...
if (9,12,14,23) begins with (9,12) then put "yes" -- yes
```

Tech Talk

Syntax: *operand1* `begins with` *operand2* {`considering case` | `ignoring case`}
operand1 `does` {`not`} `begin with` *operand2* {`considering case` | `ignoring case`}

◊ ends with , does not end with , doesn't end with

What it Does

Tests for the presence or absence of a value at the end of another, giving **true** or **false**.

When to Use It

Use the `ends with` operator to check whether or not a text string ends with a particular sequence of characters, or whether a list ends with a particular value or sequence of values. Ordinarily, this operator is not case-sensitive. To force it to be case-sensitive, use the `considering case` option.

Examples

```
if sentence ends with "?" then ...
if word n of plurals does not end with "es" then ...
if scoresList ends with (98,99,100) then resetScores
```

Tech Talk

Syntax: *operand1* `ends with` *operand2* {`considering case` | `ignoring case`}
operand1 `does` {`not`} `end with` *operand2* {`considering case` | `ignoring case`}

◊ is a , is not a , isn't a , is all , is not all , isn't all

What it Does

Checks whether a value is valid as a particular type, or analyzes the contents of a value.

When to Use It

Use the `is a` operator (or one of its synonyms) to test the contents of a container. Specifically, you can test whether a value is or is not a **number**, **integer**, **even number**, **odd number, positive number, negative number**, positive integer, negative integer, **point**, **rectangle**, **date**, **time**, or **boolean**. A variable can be tested to see whether it is a **list**, a **range**, an **iterator**, a **file**, a **folder**, a **tree**, or an **object**. You can also test whether a character or all characters in a value are **digits**, **letters**, **alphanumeric**, **uppercase**, **lowercase**, **punctuation**, **blank** (or **whitespace**), **blankOrReturn** (or **whitespaceOrReturn**), or **controlChars**. You can tell if a value is actually a reference to another container, by testing whether or not it is a **reference**. In addition, the `is a` operator can also be used to test for custom object types if the object defines an **objectType** property (see Objects, Messages, Handlers, and Helpers).

Examples

The following expressions all yield "true":

```
put pi is a number
put pi is not an integer
put -12 is an even number
put 5683 is an odd number
put 98.6 is a positive number
put 0 isn't a positive number
put -13.2 is a negative number
put 144 is a positive integer
put -1 is a negative integer
put "123, 12.5" is a point
put "123, 12.5, 245, 25" is a rectangle
put (snow is greater than rain) is a boolean
put (a,b,c) is a list
put 14..94 is a range
put (a,b,c) is an iterator
put "July 4, 1776" is a date
put "/System/Library/Fonts/Courier.dfont" is a file
put "/System" is a folder
put (partnum:"4X56N32", qty:14) is an object
put 6 is a digit
put character 2 of "4X56N32" is a letter
put "J946Ux" is an alphanumeric
put "a" is a lowercase
put "ABCdef" isn't all uppercase
put "(),.;:!?[]{}%\'/" & quote is all punctuation
put space is a blank
put space & tab & return is all blankOrReturn
put tab is a controlChar
put @foo is a reference
put (radius:23, objectType:("Shape", "Circle")) is a "Circle"
```

Tech Talk

Syntax: *valueToTest* `is` {`not`} `a` *typeIdentifier*
valueToTest `is` {`not`} `all` *typeIdentifier*

An error will be raised if *typeIdentifier* is not one of the following valid identifiers, or an expression that evaluates to one of the built-in identifiers listed below, unless *valueToTest* is an object or property list.

If the *valueToTest* is an object, the `is a` operator evaluates to the value returned by sending an "isObjectType" function message to the object, with *typeIdentifier* as a parameter. The default implementation of this function checks the "objectType" property of the object contains the *typeIdentifier*. If a property list has an "objectType" property, it may be a single value or a list of values. If *typeIdentifier* is equal to any item in the objectType list, the `is a` operator will evaluate to true, otherwise it will be false.

IDENTIFIER:	TRUE WHEN THE VALUE TO TEST IS:
number	a number
integer or int	a "whole" number without any fractional part
even number	a whole number that is evenly divisible by 2
odd number	a whole number that is not evenly divisible by 2
positive number	a number that is greater than zero
negative number	a number that is less than zero
positive integer	a whole number that is greater than zero
negative integer	a whole number that is less than zero
point	a list of two numbers, or two numbers separated by commas
rectangle rect	a list of four numbers, a list of two points, or four numbers separated by commas
date time	a value other than a single number that can be converted to a date or time value
list	a list
range	a range
iterator	an iterable value such as a list or range
object propertyList	an object or property list
tree	a tree
reference	a reference to another container
file	a file object or file name of an existing file, not a folder
folder directory	a file object or file name of an existing folder, not a plain file
boolean logical	"true" or "false", "yes" or "no", "on" or "off"

The following identifiers can be used to test the type of characters in a text value:

IDENTIFIER:	TRUE WHEN EVERY CHARACTER OF THE VALUE TO TEST IS:
digit digits	a digit: 0,1,2,3,4,5,6,7,8, or 9
letter letters	an upper- or lower-case letter
alphanumeric	a letter or a digit
uppercase	an upper-case letter
lowercase	a lower-case letter
blank whitespace	a space or a tab
blankOrReturn whitespaceOrReturn	a space or a tab or a return
punctuation	a punctuation character such as , . ! ? ; :
controlChar controlChars	a hidden control character such as return, tab, formfeed, etc.

◊ ago , hence

What it Does

The `ago` and `hence` operators produce a date/time value that is the specified length of time in the past or future, respectively.

When to Use It

Use these operators for time comparisons (using the `earlier than` or `later than` operators), or any time you need a date/time value that is a specific amount of time before or after the present moment.

Examples

```
put three minutes hence into expirationTime
if modificationDate of file updateLog is earlier than 5 days ago then ...
```

Tech Talk

Syntax: *timeInterval* `ago`
timeInterval `hence`

TimeInterval is typically given as a time interval value (see Time Intervals in Values). However, it can actually be any value or an expression in parentheses that yields a number, which will be treated as the number of seconds. The `ago` operator results in a time value that is the indicated length of time earlier than the present moment (in the past). The `hence` operator yields a time value that is the indicated length of time later than the present moment (in the future). The resulting value is set to use the Abbreviated International Time format when it is converted to text.

◊ there is a , there is not a , there isn't a , there is no , exists , does not exist , doesn't exist

What it Does

Tests for the existence of a file, folder, variable, object or object property.

When to Use It

Use one of the `there is a` or `exists` operators to find out whether a specified object, object property, variable, file or folder exists, and take appropriate action. In the case of variables, this operator returns true if the the variable has been assigned a value.

Examples

```
if there is a folder "BankReport" then ...
if there is no file "secretpasswords" then ...
if there is not an object "printHelper" then ...
if there is a property cost of material then ...
if there is a variable controller then ...
if file "answers" doesn't exist then create file "answers"
if property sequence of part exists then add 1 to part's sequence
```

Tech Talk

Syntax: `there is a` *thingThatMayExist*
`there is not a` *thingThatMayExist*
`there isn't a` *thingThatMayExist*
`there is no` *thingThatMayExist*
thingThatMayExist `exists`
thingThatMayExist `does not exist`
thingThatMayExist `doesn't exist`

where *thingThatMayExist* is one of:
`file` *fileName*
`folder` *folderName*
`object` *objectIdentifier*
`property` *propertyName* **`of`** *someObject*
`variable` *localOrDeclaredVariableName*
`global` *globalVariableName*
`universal` *universalVariableName*

◊ is within , is not within , isn't within

What it Does

Tests whether a point lies within a rectangle, whether one rectangle is completely contained by another, or whether a value falls within a given range.

When to Use It

The `is within` operator can be used for checking whether a specified x,y coordinate point is within a particular rectangular area, or whether one rectangular area is enclosed by another. It can also be used to determine whether a value is between the start and end values of a range.

Examples

```
if mousePoint is within windowBorder then ...
if lastLoc + (12,8) is within (10,10,90,50) then ...
if windowRect is not within screenRect then ...
if day is within 1..lastValidDay then ...
```

Tech Talk

Syntax: *point* `is {not} within` *rectangle*
rectangle1 `is {not} within` *rectangle2*
value `is {not} within` *range*

All forms of the `is within` operator test for containment **inclusive** of the edges or endpoints of the containing rectangle or range. So, for example `9 is within 5..9` will evaluate as true.

Points are always specified as a pair of numbers representing the x and y coordinates of the point. Usually, these two values are given as a two-item list such as (12,42) but a text string containing two numbers separated by commas can also be used, such as “12,42”.

Rectangles are specified as 4 numbers, representing the locations of the left, top, right, and bottom of the rectangle, such as (5,18,105,118) — this rectangle would actually be a square, with both width and height of 100. You can also think of the four numbers as describing two points, which are opposite corners of the rectangle. A rectangle can also be specified as a list of 2 points.

◊ (if ... then ... else ...) selector expression

What it Does

Evaluates to one of two values depending on some condition.

When to Use It

The `(if ... then ... else ...)` operator, also known as a “selector expression” can be used to select one of two values or an optional value within an expression, or to select one of two containers.

Examples

```
put "Delivered " & count & " widget" & (if count>1 then "s")
put (if a>b then a else b) into bigger
```

The second example above is equivalent to:

```
if a>b then put a into bigger else put b into bigger
```

The selector expression can also be used anywhere a container is expected, provided that both the `then` and `else` expressions specify containers:

```
add 100 to (if a<b then a else b) -- add to whichever is smaller
```

This is equivalent to:

```
if a<b then add 100 to a else add 100 to b -- add to whichever is smaller
```

Tech Talk

Syntax: (if *condition* **then** *expression1* **{else** *expression2***})**

Please note that the selector expression, although it looks very similar, is not the same as the `if ... then ... else ...` control structure which controls whether statements are executed or not. Note that parentheses are always required around a selector expression, and it must always include a `then` expression. If the `else` expression is omitted, empty is assumed (that is, it's the same as saying "else empty"). The `else` expression is required when specifying a container.

◊ adding, replacing, removing, retaining properties

What it Does

Each of these four operators works with a property list and produces a new property list containing a modified set of properties. The new property list is derived by adding or replacing properties coming from a second property list, by removing specified properties, or by removing all but a selected set of properties.

When to Use It

Use the `adding properties` operator to combine the properties of two different property lists or objects to create a new property list. The `replacing properties` operator combines the properties of two different property lists or objects to create a new property list, overriding properties in the first property list with any corresponding properties from the second property list. The `removing properties` operator removes some properties from one property list or object to create a new property list. The `retaining properties` operator creates a new property list from another, containing only the indicated properties from the original property list.

Examples

```
put (A:1,C:3) into firstProps -- start with a simple property list
put firstProps adding properties (B:2, C:99, D:4) into newProps
put firstProps -- unchanged: (A:1, C:3)
put newProps -- (A:1, B:2, C:3, D:4)
put newProps removing properties ("B","C") -- (A:1, D:4)
put newProps replacing (B:22, D:44) -- (A:1, B:22, C:3, D:44)
put newProps retaining ("B","D","E") -- (B:2, D:4)
```

Tech Talk

Syntax: *sourcePropList* `adding` `{property | {the} properties {of}}` *additionalPropList*
sourcePropList `removing` `{property | {the} properties {of}}` *propsToRemove*
sourcePropList `replacing` `{property | {the} properties {of}}` *propsToReplace*
sourcePropList `retaining` `{property | {the} properties {of}}` *propsToRetain*

If a property appears in both property lists when `adding`, the properties in *sourcePropList* always take precedence over those in *additionalPropList*. The reverse is true when `replacing`, with properties from *propsToReplace* taking precedence over those in *sourcePropList*.

When `removing` properties, the *propsToRemove* may be the name of a single property, a list of property names, or a property list (whose values will be ignored but whose keys will be used as the properties to remove). Trying to remove properties that don't exist has no effect. The `retaining` operator is similar, but instead of specifying the properties to be removed, *propsToRetain* identifies the properties from the original property list that should be retained, with all others being discarded. The word "property" or "properties" is optional in all of these operators.

See Also: the `add properties`, `replace properties`, `remove properties`, and `retain properties` commands in Lists and Property Lists.

◊ joined by, split by

What it Does

The `joined by` operator combines the elements of a list or property list into a text string. The `split by` operator does the reverse, taking a text string and producing a list or property list from it.

When to Use It

The `joined by` and `split by` operators can be used to convert lists or property lists to or from a text format. They may be useful to convert structured data to or from a text form used for archiving in a disk file, or for working with text such as a file path, which could be split into individual components in a list for manipulation and then joined back together as text again.

The word `combined` may be used in place of `joined`, and either `with` or `using` may be used instead of `by` in either operator if you prefer.

Examples

```
put path split by "/" into components
set newPath to components combined using "/"
put (a:1, b:2) joined with ";" and "=" -- "a=1;b=2"
```

Tech Talk

Syntax: *sourceStructure* `[joined | combined] [by | with | using]` *separator1* `{and` *separator2*`}`
sourceText `split [by | with | using]` *separator1* `{and` *separator2*`}`

Tech Talk

When working with property lists, two separators should be used. The first indicates the text separator between elements, and the second specifies the text separator between each key and its corresponding value. To split or join a list, only a single separator is needed.

◊ As operator

What it Does

The `as` operator converts a value to a specified internal representation.

When to Use It

The `as` operator can be used to tell SenseTalk to treat a value as a particular type, or force conversion to a different representation at a particular point in a script. For example, the `as text` operator can be used to force textual comparisons of values that might otherwise be compared as numbers.

Examples

```
if today is "April 15" as date then payTaxes
performTextOperation (hours * rate) as text
put "007" is equal to "7.0" -- true (numeric comparison)
put "007" is equal to "7.0" as text -- false
```

Tech Talk

Syntax: *sourceValue* `as` `[text | number | date | time | color | data | {a} {property} list | {an} object | {a} tree]`

Internally, the `as` operator calls the `asText`, `asNumber`, `asDate`, `asTime`, `asColor`, `asData`, `asList`, `asObject`, and `asTree` functions.

The existence of the `as` operator does not imply that SenseTalk is a "typed" language. In fact, it is an "untyped" language, with values converted automatically to whatever internal representation is needed at any time. In practice, there are only a few relatively rare situations in which you will need to use this operator to explicitly force the internal representation of a value.

For example, SenseTalk won't automatically perform a date/time comparison just because two values could be treated as date/time values. To compare two values as date/times, both values must be in that representation, otherwise they will be compared as text, giving very different results (April comes before January alphabetically, for instance). One way to ensure this is by specifying `as date` or `as time` after both values as needed, as shown in the first example above.

◊ As operator

Similarly, `as data` can be used to perform direct comparisons of raw binary data values rather than their text representations. It is most often used when reading or writing binary data files to keep the data in binary format. The `byte` chunk type can then be used to access individual bytes or byte ranges within the data.

The `as list` and `as property list` (or `as object`) operators are somewhat different. They don't merely indicate how a value should be treated, but for any value that isn't already in the requested representation they will evaluate that value's text as an expression (in the same manner as the `value()` function) to produce the requested structure. The `as tree` operator will similarly evaluate text as XML (equivalent to calling the `treeFromXML()` function).

When the value to be converted is an object, it may control its representation in each format if it implements an `asText`, `asNumber`, etc. handler, or has special properties that apply for the requested type. See the relevant `as...` function documentation for full details.

See the section on "Conversion of Values" later in this section for more information on automatic conversion of values, and when you might use the `as` operator.

Functions

A "function" is a fancy name for a source of value that may vary. SenseTalk provides many functions for a wide variety of purposes. Some functions may produce a different value each time they are used, such as `the time` which provides the current time of day. Other functions may produce a different value depending on what "parameters" are passed to them – using the same function with different parameter values will produce different results.

The functions that are built-in as part of the SenseTalk language are described throughout this manual, especially in the section titled "Commands and Functions". The host application environment in which SenseTalk is running may provide other predefined functions. In addition, you can write your own functions that can be used by your scripts.

Calling Functions

To use a function value, a script "calls" that function, either on its own or as part of an expression. SenseTalk provides several different ways to call a function – you can use whichever one seems most natural in a given situation. The simplest type of function call is one without parameters that is not being sent to a particular object. Such a function call can be made in two ways: either by naming the function followed by an empty pair of parentheses (indicating no parameters); or by using the word "the" followed by the function name:

```
put date() -- displays the date
put the date -- displays the date
```

To call a function with a single parameter, either include that parameter in the parentheses after the function name, or follow the function name with "of" and the parameter:

```
put length("abcd") -- 4
put the length of "abcd" -- 4
```

In the second case above, because "of" is used, SenseTalk is able to recognize it as a function call even without the word "the":

```
put length of "abcd" -- 4
```

Because many functions (such as the "length" function) calculate some attribute of a value, SenseTalk also allows the property access notation to be used to call a function:

```
put "abcd" 's length -- 4
put "abcd".length -- 4
```

Calling a function with more than one parameter is similar to the one-parameter case. All of the following will pass 4 parameters to the average function:

```
put average(2,4,6,8) -- 5
put the average of 2 with (4,6,8) -- 5
put 2's average(4,6,8) -- 5
put 2 .average(4,6,8) -- 5
```

Although some of these seem a bit odd in this case, there may be other situations where those forms will seem more natural. (Note that in the last example above, a space is needed after the number 2 to prevent the period from being interpreted as a decimal point.)

The name of a function to call can be generated dynamically by using an expression in parentheses in place of a function name:

```
set funcToCall to "length" -- name of the function
put (funcToCall) of "abcd" -- 4
```

A list of functions can be called on the same value at once, producing a corresponding list of results:

```
put (length, uppercase, lowercase) of "Test" -- (4,"TEST","test")
```

Example

Here is an example function handler that takes one parameter (`aWord`) and returns a modified copy of its value:

```
function plural aWord
   if aWord ends with "s" then put "es" after aWord
   else if aWord ends with "y" then put "ies" into the last char of aWord
   else put "s" after aWord
   return aWord
end plural
```

Here is an example showing how this function could be called from a script:

```
ask "What type of object are you carrying?"
put it into itemType
ask "How many do you have?"
put it into howMany
if howMany is not 1 then put the plural of itemType into itemType
answer "You have " & howMany && itemType
```

Conversion of Values

Typed or Typeless?

SenseTalk is a “typeless” language, in the sense that all values can be treated as text, and you never need to declare that a given variable will hold a particular type of value, such as numbers or dates or text, or lists of values. Internally, though, SenseTalk may hold these values in different forms, converting from one representation to another as needed. Understanding how and when these conversions occur, and the global properties that control the formatting, will allow you to take control of this process when necessary.

Automatic Conversion

SenseTalk converts values automatically to an appropriate internal representation as needed. When performing an arithmetic operation such as addition, for instance, the two values being added will be evaluated as numbers. The resulting value will be kept internally in numeric form until it is needed as text.

In most situations, SenseTalk's automatic value conversion will do what you want and provide the desired result. Occasionally, though, there can be surprises, so it's helpful to understand when these conversions take place and how you can control them. Consider the following example:

```
put ((1 + 2) & 4) + 5
```

When this statement is executed, SenseTalk first adds the numbers 1 and 2, getting the value 3, which is temporarily stored as a number. The next operation that is performed is to concatenate this value with 4. This is a text operation, so both values are converted to their text representation before being joined into a single text string. Finally, this result is converted back to a number so that it can be added to the number 5.

The final result displayed will be 39. Or will it? It turns out that the actual outcome may be 39, or 309, or 3009 or some other number, or it may result in an error, depending on the setting of `the numberFormat` property when this statement is executed. Let's see how this can happen.

`The numberFormat` (described in detail below) controls how a numeric value will be formatted when it is converted to a text representation, including such things as the number of decimal places to show, and whether leading zeros should be displayed. In our example, the numbers 3 and 4 were converted to text form in order to concatenate them. The default setting of the numberFormat would convert them to the text strings "3" and "4", resulting in the concatenated text "34". Using a numberFormat that includes leading zeros (setting it to "00", for example) would cause the numbers 3 and 4 to be represented in text form as "03" and "04", which concatenate as "0304". In order to add 5 to this string, it is converted to a number (304) giving the final result of 309.

Similar conversions happen when values stored internally as date or time values, or entire lists or property lists of values, are needed in a text format.

Binary Data Conversion

Data can be read from a file in a binary format containing the "raw" bytes of data by specifying `as data` when accessing the file:

```
put file "/tmp/aFile" as data into dataBytes
```

Some operations, such as extracting a range of bytes from the data, will use the data in its raw format:

```
put bytes 1 to 4 of dataBytes into firstBytes
```

Other operations, such as displaying its value, will automatically convert the data to text. When converting between binary data and text, `the defaultStringEncoding` is used to interpret the data as characters, or (in the other direction) to encode characters of text as data.

Explicit Conversions

Properties such as `the numberFormat` let you control the form a value will take when it is converted to text, but don't let you control *when* that will occur. Look at this script fragment:

```
set the numberFormat to "0.00"
put amount1 + amount2 into total
displayOutput total
```

If the intent is to pass `total` to the displayOutput command formatted with two decimal places, the script above will fail. The problem is that `total` is represented internally as a number and will be passed to displayOutput in that form, where the displayOutput handler's numberFormat will determine how it ultimately gets formatted. To force these numbers to be converted to text using the local numberFormat setting, the `as text` operator can be used:

```
set the numberFormat to "0.00"
put amount1 + amount2 into total
displayOutput total as Text -- convert total to text while passing it
```

Similarly, the `as number`, `as date`, `as time`, `as data`, or `as color` operators, or the related `asText()`, `asNumber()`, `asDate()`, `asTime()`, `asData()`, and `asColor()` functions, can be used to force a value to be evaluated explicitly in those formats.

◊ the numberFormat local property

What it Does

`The numberFormat` property specifies the number of decimal places to use, and number of leading and/or trailing zeros to show when a numeric value (such as the result of a mathematical operation) is converted to a textual representation.

Examples

```
set the numberFormat to "#.###" -- up to 3 decimal places
set the numberFormat to "0.00" -- 2 decimal places always
set the numberFormat to "00.##"
put 3.41 -- displays 03.41
set the numberFormat to "00.###"
put 3.41 -- displays 03.41
set numberFormat to "0.000"
put 3.41 -- displays 3.410
set numberFormat to "0"
put 3.41 -- displays 3
```

Tech Talk

Syntax: `set the numberFormat to` *formatExpression*
`get the numberFormat`

The *formatExpression* is an expression which yields a string made up of a combination of 0's and/or #'s, and optionally a decimal point. The 0's to the left of the decimal point indicate how many places will appear in the number, being filled with 0 if the place has no value in the number being formatted. Use #'s to the right of the decimal point to have decimal places appear if they have value. Use 0's to have decimal places appear whether they have value or not.

The numberFormat property is local to each handler. Setting its value in one handler will not affect its value in other handlers called from that handler, or vice versa. When each handler begins running, the numberFormat in that handler is initially set to the value of the defaultNumberFormat global property. The defaultNumberFormat is originally set to "0.######", but may be changed if desired.

◊ the listFormat global property

What it Does

The listFormat global property is a property list usually holding three values (prefix, separator, and suffix) that define the format used to convert a list into a text format. By default they are set to "(", ",", and ")" respectively. When displaying a list as text, it is surrounded by the prefix and suffix values, with the separator value used between each item of the list.

An optional quotes value defines the manner in which values in the list are quoted. By default, the quotes value is not set for lists, so the setting of the defaultQuoteFormat global property (see below) controls quoting of list values.

An optional indent value may be set to a text value that will be used for indenting items in the list. When set to anything other than empty, this will cause individual list items to be displayed on separate lines with values indented by multiples of the indent value according to the nesting of the structure.

Examples

```
set the listFormat.prefix to "[["
set the listFormat's suffix to "]]"
put (1,2,3,4) -- [[1,2,3,4]]
set the listFormat's separator to " & "
set the listFormat's quotes to ("<",">")
put (5,6,7,8) -- [[<5> & <6> & <7> & <8>]]
set the listFormat's indent to "  "
put (1,(10,20,30),2) --
    [[
      <1> &
      [[
        <10> &
        <20> &
        <30>
```

```
    ]] &
    <2>
]]
```

For compatibility with earlier versions, the listPrefix , the listSeparator, and the listSuffix properties provide direct access to the prefix, separator, and suffix properties of the listFormat. The split by and joined by operators (described in Expressions), provide ways to explicitly convert text to lists and vice versa. For more information on working with lists, see Lists and Property Lists.

◊ the propertyListFormat global property

What it Does

The propertyListFormat global property is a property list that may contain these property values: prefix, entrySeparator, keySeparator, suffix, quotes, emptyRepresentation, indent and asTextEnabled. The settings of these values define the format used to convert a property list (object) into a text format. If asTextEnabled is true (the default) when the text representation of an object is needed, the object is sent an asText function message to get its text representation. If the object does not respond to this message its asText or asTextFormat property will be used to obtain a text representation.

If asTextEnabled is false, or if none of the "asText" mechanisms yield a value, then the other values are used. In that case, the object's properties are listed (in alphabetical order of its keys) with each key (property name) preceding its value separated by the keySeparator (with a default value of ":") and the entries separated by the entrySeparator (which defaults to ", "). Property values are quoted according to the quotes value (which defaults to "Standard" for property lists — see the defaultQuoteFormat global property, below). The entire text is surrounded by the prefix and suffix (default values "(" and ")"). If the object has no properties, the value of the emptyRepresentation property (default "(:)") will be used as its text representation.

If the keySeparator is set to empty, the object's keys will not be listed, only its values.

The indent value may be set to a text value that will be used for indenting entries in the object. When set to anything other than empty, this will cause each entry to be displayed on a separate line with each line indented by multiples of the indent value according to the nesting of the structure.

Examples

```
replace properties (prefix:"[[", suffix:"]]", entrySeparator:"; ", \
   keySeparator:" is ") in the propertyListFormat
put (A:1,C:3,B:2) -- displays [[A is "1"; B is "2"; C is "3"]]
delete the propertyListFormat's keySeparator
set the propertyListFormat.quotes to "None"
put (A:1,C:3,B:2) -- displays [[1; 2; 3]]
set the propertyListFormat.emptyRepresentation to "[[no properties]]"
put (:) -- displays [[no properties]]
```

For compatibility with earlier versions, the plistPrefix, the plistEntrySeparator, the plistKeySeparator, the plistSuffix, and the emptyPlistAsText properties provide direct access to the prefix, entrySeparator, keySeparator, suffix and emptyRepresentation properties of the propertyListFormat. For more information on converting property lists to a text representation, see Lists and Property Lists.

◊ the defaultQuoteFormat global property

What it Does

The `listFormat` and `propertyListFormat` global properties both include an optional `quotes` value that specifies how values in those containers are quoted when converted to text. `The defaultQuoteFormat` global property provides the quoting behavior when either of these `quotes` values is not given (which is the default case for lists, but not for property lists).

The value of any of these quoting properties can be empty or "None" for no quoting, "Standard" for standard quoting, or may be any other string or a list of two strings. If a list of two strings is given, they are used as the opening and closing quote (before and after the value) respectively. If a single string is given it is used as both the opening and closing quote. Standard quoting will automatically choose either a straight double quote character (") or double angle brackets ("<<",">>") depending on the value's content. Whenever values are quoted, any value containing the final quote string or a return character will automatically be turned into a valid SenseTalk expression that will yield the proper value.

Initially, until changed by your script, `the defaultQuoteFormat` is set to "None".

Examples

```
set the defaultQuoteFormat to "#"
put (1,2,3,4) -- (#1#,#2#,#3#,#4#)
set the defaultQuoteFormat to "Standard"
put (<<"Hello">>,42,"GO"&return) -- (<<"Hello">>,"42","GO" & return)
set the defaultQuoteFormat to "None"
put (<<"Hello">>,42,"GO"&return) --   ("Hello",42,GO          )
```

◊ numberWords, ordinalWords, timeInterval, timeIntervalWords, byteSize, and byteSizeWords() functions

What it Does

These functions convert a number, an interval of time (in seconds), or a file size (in bytes) to a friendlier format using words. The text that results from any of these functions (except ordinalWords) can be converted back to a number using the `value()` function (below).

Examples

```
put numberWords(90) -- "ninety"
put numberWords(427.8) -- "four hundred twenty-seven point eight"
put ordinalWords(90) -- "ninetieth"
put timeInterval(90) -- "1 minute 30 seconds"
put timeIntervalWords(90) -- "one minute thirty seconds"
put timeIntervalWords(11520) -- "three hours twelve minutes"
put byteSize(5242880) -- "5 megabytes"
put byteSizeWords(90) -- "ninety bytes"
```

Other value conversions

In addition to numbers, lists, and property lists, there are some other values that may have non-textual internal representations. These internal values will be converted to text automatically as needed. In addition to the value types discussed earlier in this section, these include values representing dates and times, and also color values. There are

also some formatting functions not mentioned earlier that can be used to convert values explicitly from one format to another.

Color values may be represented internally in a binary format. When displayed or converted to text, the current setting of `the colorFormat` global property is used to control the format. Colors are described in detail in Working with Color.

Date and time values in SenseTalk are often represented using an internal format that includes both the actual date/time value and a text format that is used to convert the value to text when needed. To convert a date/time value from one format to another, the `formattedTime` function or the `convert` command can be used, along with the settings in `the timeFormat` global property. Dates and times are described in detail in Working with Dates and Times.

The `merge()` and `format()` functions, described in Working with Text, are very versatile functions that can be used for a variety of general formatting needs. The `standardFormat()` function described in that section is useful for converting any value to a text format suitable for archiving.

Evaluating Expressions at Runtime

◊ value function

What it Does

Returns the value of its parameter evaluated as a SenseTalk expression. One use of this would be to accept text entered by a user that might contain an expression (such as "12.95 + 6%") and calculate its value. Another would be to read a file containing a list or property list and quickly convert it from the text format stored in the file into a useful collection of values.

Examples

```
put the value of "51+93" into sum -- sets sum to 144
put value("total is greater than quarter" & bestQtrNum) into best
put value("(" & commaSeparatedText & ")") into myList
put the value of file "storedProperties" into pList
```

Tech Talk

Syntax: `value(`*expr*`)`
`{the} value of` *expr*

The expression is evaluated in the context of the current handler. It may include variables (such as `total` in the second example above), operators, function calls, and so forth. If the `evaluationContext` property is set to "Global" or "Universal" then variables will be treated as global or universal rather than local. If an error occurs while obtaining the value of the expression, the `result()` function will return an exception object describing the problem.

See Also: the `merge` function, in Working with Text, and the `do` command in Working with Messages.

Chunk Expressions

For easily accessing elements within text, SenseTalk provides a very powerful type of expression called a chunk expression. Chunk expressions give you great flexibility in referring to any portion of text, from a single character to a range of lines, in a very natural way. Here are some brief examples:

```
... character 3 of partNum ...
... the second word of companyName ...
... items 2 to 4 of "a,b,c,d,e,f,g,h" ...
... the first 3 lines of file "log" ...
```

You can use chunk expressions to grab lines, text items, words or characters from any container or text value. You can also describe part of a list or part of a binary data value using a chunk expression.

Chunk expressions can also be combined into more complex expressions, such as,

```
... chars 1 to 3 of the last word of line 2 of myText ...
```

Chunk Types

Chunk expressions let you work with all of these chunk types:

characters	individual characters within text
words	words separated by any amount of white space (spaces, tabs, returns) within text
lines	paragraphs separated by any of several standard line endings (CR, LF, CRLF, etc.)
text items	portions of text separated by commas
list items	the individual items in a list
bytes	the bytes within binary data

In addition, you can specify custom delimiters to be used in identifying text items, lines, and words, giving even greater functionality. These three text chunk types each have distinctive types of delimiters: text items are delimited by a single text string, lines are delimited by any of a list of text strings, and words are delimited by any number and combination of characters from a set of characters.

Characters

The simplest type of chunk is the character chunk. A character is simply one character of text, including both visible and invisible characters (invisible characters include control characters such as tab, carriage return, and linefeed characters). The word `character` may be abbreviated as `char`.

```
put "The quick brown fox" into animal
put character 1 of animal -- "T"
put the last char of animal -- "x"
put chars 3 to 7 of animal -- "e qui"
```

Words

A single word is defined as a sequence of characters not containing any whitespace characters, or a sequence of characters contained in quotation marks. A range of words includes all characters from the first word specified through the last word specified, including all intervening words and whitespace. Whitespace characters are spaces,

tabs, and returns (newlines).

```
put " Sometimes you feel like a nut; sometimes you don't." into slogan
put the second word of slogan -- "you"
put word 6 of slogan -- "nut;"
put words 1 to 3 of slogan -- "Sometimes you feel"
```

Note that quoted phrases are ordinarily treated as a single word, including the quotation marks:

```
put <<Mary said "Good day" to John.>> into sentence
put the third word of sentence -- <<"Good day">>
```

The set of characters that are used to identify words can be changed to something other than space, tab, and return by setting `the wordDelimiter` property. The quote characters used to identify a quoted word (or whether word quoting should be disabled completely) can be specified with `the wordQuotes` property.

◊ the wordDelimiter local property

What it Does

`The wordDelimiter` property specifies the set of characters recognized as separators between words in a container.

> **Note**
>
> You can also specify a delimiter directly in a word chunk expression, by including a "delimited by" clause, as described under "Custom Chunks" below.

Examples

```
set the wordDelimiter to ".,!?;:" & space & tab
put "A man, a plan, a canal. Panama!" into palindrome
set the wordDelimiter to " ,."
put word 4 of palindrome -- "plan"
```

> **Tech Talk**
>
> **Syntax:** `set the wordDelimiter to` *expression*
> `get the wordDelimiter`
>
> Characters in the `wordDelimiter` are used to identify words when evaluating a chunk expression. Any sequence of these characters, in any order or combination, may appear between words.
>
> The `wordDelimiter` property is local to each handler. Setting its value in one handler will not affect its value in other handlers called from that handler, or vice versa. When each handler begins running, the `wordDelimiter` in that handler is initially set to the value of the `defaultWordDelimiter` global property. The `defaultWordDelimiter` is originally set to the space, tab, and return characters, but may be changed if desired.

◊ the wordQuotes local property

What it Does

The wordQuotes property specifies the quote character or characters used to identify quoted "words" in a word chunk. By default any word beginning with a double quote character will include all characters up to and including the next double quote character (including any enclosed wordDelimiter characters).

Examples

```
set the wordQuotes to empty -- disable quoting

put "Hi, I'm [[my long name]]" into format
set the wordQuotes to ("[[","]]")
put word 3 of format -- [[my long name]]

set sentence to <<Jack said "Let's go see a movie.">>
put word 3 of sentence -- "Let's go see a movie."
set the wordQuotes to empty -- disable quoting
put word 3 of sentence -- "Let's
```

Tech Talk

Syntax: set the wordQuotes to *expression*
get the wordQuotes

The wordQuotes can be set to a list of two values for the beginning and ending quote delimiters, or to a single value which will be used as both the beginning and ending delimiters. It can also be set to empty or "None" to disable word quoting, or to "Standard" to restore the default quoting (with double quotation marks used for quotes).

The wordQuotes property is local to each handler. Setting its value in one handler will not affect its value in other handlers called from that handler, or vice versa. When each handler begins running, the wordQuotes in that handler is initially set to the value of the defaultWordQuotes global property. The defaultWordQuotes is originally set to use straight double quote characters, but may be changed if desired.

Lines

A line chunk expression allows you to specify one or more lines or paragraphs of text within the subject text, where lines are initially defined as the characters between return characters.

```
put "line 1" & return & "line 2" & return & "line 3" into text
put the second line of text -- "line 2"
put line 6 of text -- ""
put lines 2 to 3 of text -- "line 2" & return & "line 3"
```

The set of line endings (delimiter strings) that defines what a line is can be changed to something other than the default by setting the lineDelimiter property. Setting the lineDelimiter to empty will cause it to be set to a list of standard line endings (CRLF, Return, CarriageReturn, LineSeparator, ParagraphSeparator).

◊ the lineDelimiter local property

What it Does

The `lineDelimiter` property specifies the list of delimiter strings recognized as separators between lines of text.

When to Use It

You may want to change the line delimiter to include more than one type of line ending. The default setting only includes the Return character (technically, this is the LineFeed character). This will work fine for most text that a script works with. Text files produced on different platforms (Mac, Windows, Linux) may use other line endings. To make it easy to work with a variety of files you can set the lineDelimiter to empty. This is a shortcut that will automatically set it to a list of standard line endings: (CRLF, Return, CarriageReturn, LineSeparator, ParagraphSeparator).

Another reason you might want to change the lineDelimiter would be to take advantage of the fact that it provides a list of delimiters, in order to access chunks of text that may be separated by several different characters.

Note

You can also specify a list of delimiters directly in a line chunk expression, by including a “delimited by” clause, as described under “Custom Chunks” below.

Examples

```
set the lineDelimiter to empty -- use all standard line endings

put "C:\songs/Solas/BlackAnnis" into songPath
set the lineDelimiter to ("/","\")
put line 2 of songPath -- "songs"
```

Tech Talk

Syntax: `set the lineDelimiter to` *delimiterList*
`get the lineDelimiter`

The `lineDelimiter` is a list of strings which may separate line items when evaluating a line chunk expression. The order of the items in the list is important, as they are matched in the order given. For example to match the common line endings CR, LF, and CRLF, be sure to list CRLF before CR. Otherwise if CRLF is encountered in text it will match CR first and be treated as two line endings in a row.

Setting `the lineDelimiter` to `empty` is treated as a special case that will set it to a list of standard line endings: (CRLF, Return, CarriageReturn, LineSeparator, ParagraphSeparator) where CRLF is ASCII 13 followed by ASCII 10, Return is ASCII 10, CarriageReturn is ASCII 13, LineSeparator is Unicode 0x2028 and ParagraphSeparator is Unicode 0x2029.

The `lineDelimiter` property is local to each handler. Setting its value in one handler will not affect its value in other handlers called from that handler, or vice versa. When each handler begins running, the `lineDelimiter` in that handler is initially set to the value of the `defaultLineDelimiter` global property. The `defaultLineDelimiter` is originally set to Return (ASCII 10), but may be changed if desired.

Text Items

An item within text is usually defined as the portion of text between commas:

```
put "A man, a plan, a canal. Panama!" into palindrome
put item 2 of palindrome -- " a plan"
```

The separation (delimiter) character can be specified as something other than a comma, by setting `the itemDelimiter` property.

◊ the itemDelimiter local property

What it Does

`The itemDelimiter` property specifies the character recognized as the separator between text items in a container.

> **Note**
>
> You can also specify a delimiter directly in an item chunk expression, by including a “delimited by” clause, as described under “Custom Chunks” below.

Examples

```
set the itemDelimiter to "/"
put "A man, a plan, a canal. Panama!" into palindrome
set the itemDelimiter to "."
put item 2 of palindrome -- " Panama!"
```

Script

The following function takes a full file path as input and returns just the directory of the file. It does this by parsing the path into items using “/” as the item delimiter.

```
function pathOfFile filename
   set the itemDelimiter to "/"
   delete the last item of filename
   return filename
end pathOfFile
```

> **Tech Talk**
>
> **Syntax:** `set the itemDelimiter to` *expression*
> `get the itemDelimiter`
>
> The `itemDelimiter` is used to separate text items when evaluating a chunk expression. Most often, a single character is used as a delimiter, but it may also be set to a longer sequence of characters if desired.

Tech Talk

The `itemDelimiter` property is local to each handler. Setting its value in one handler will not affect its value in other handlers called from that handler, or vice versa. When each handler begins running, the `itemDelimiter` in that handler is initially set to the value of the `defaultItemDelimiter` global property. The `defaultItemDelimiter` is originally set to "," (a comma), but may be changed if desired.

List Items

The word `items` can also refer to the elements in a list.

```
put ("red", "green", "blue") into colors
put item 2 of colors -- "green"
```

SenseTalk decides whether `item` refers to text items or list items depending on whether the value is a list or not. When referring to items within a value which is a list, SenseTalk will automatically assume the reference is to list items, not text items. However, if the `itemDelimiter` is set to "" (empty), `items` will refer to list items rather than text items. You may explicitly refer to `list items` or `text items` instead of the more generic `items` if you need to control the way items are treated. This is especially important if you are trying to create a list by putting values into individual items, like this:

```
put 1 into myText -- 1
put 2 into item 2 of myText -- "1,2"
```

The code above will generate a text string, with the middle character being the itemDelimiter (unless the itemDelimiter has been set to `empty`). To generate a list instead of text, specify `list item`:

```
put 1 into myList -- 1
put 2 into list item 2 of myList -- (1,2)
```

See Lists and Property Lists, for more information on working with lists.

Bytes

A `byte` chunk can be used to refer to a portion of binary data.

```
put <3f924618> into binaryData
put byte 2 of binaryData into b2 -- <92>
```

See Working with Binary Data, for more information on byte chunks.

Custom Chunks

The standard `word`, `line`, and `text item` chunks are useful for many things just as they are. Sometimes you may have text in specific formats that you would like to divide in other ways, however. For example, many programs can produce data files containing several values separated by tab characters on each line of the file.

One way to work with such data would be to set `the itemDelimiter` to `tab` and then access the `items` of each line. But suppose that each tab-separated item contains several values separated by commas. To access these values individually would require switching the `itemDelimiter` back and forth between tab and comma.

SenseTalk offers an easier alternative for such cases, by specifying the delimiter to be used as part of each chunk, using the phrase `delimited by`:

```
add 1 to item 3 delimited by "," of item 5 delimited by tab \
   of line 18 of file complexDataFile
```

The same syntax may be used with `line` chunks if you like:

```
get line 6 delimited by creturn of oddLineBreakText
```

The delimiters used to separate text items and lines are not restricted to a single character:

```
put item 2 delimited by "<>" of "12<>A19<>X" -- "A19"
```

Custom delimiters are also allowed with `word` chunks, but the behavior is different than with items and lines. Words are normally separated by spaces, tabs, and line breaks. Any number of these "whitespace" characters may appear in sequence between two words. If you specify a custom delimiter for a word chunk, the "words" will be delimited by any number and combination of the characters contained in the delimiter string you supply:

```
put word 2 delimited by "<>" of "12><<>>A19><>X" -- "A19"
```

The following example may help to illustrate the difference between the use of custom delimiters for line chunks (which treat each delimiter string found as a separate chunk) and for word chunks (which treat each sequence of delimiter characters as a single word break:

```
put each line delimited by ("<",">") of "12><<>>A19><>X" -- (12,,,,,A19,,,X)
put each word delimited by "<>" of "12><<>>A19><>X" -- (12,A19,X)
```

Chunk Syntax

Single Chunks

Chunk expressions for all types of chunks can be expressed in several different ways. You can describe a single chunk element:

```
put item 17 of scores into quiz3
put word 4 of "Mary had a little lamb" -- "little"
```

Negative numbers can be used to count backwards from the end of the source, with -1 indicating the last chunk element, -2 for the next-to-last, and so forth:

```
get item -2 of "apple,banana,pear,orange" -- "pear"
```

Tech Talk

Syntax: `chunk` *number* **`of`** *expression*

Note on Syntax

In this and the following syntax descriptions, **chunk** is used to represent any of the chunk types: character (or its abbreviation, char), word, line, item, text item, or list item. Similarly, **chunks** represents the plural version of these terms. *number* is a factor which evaluates to a positive or negative number, and *expression* is the source or destination value which the chunk refers to. Wherever the word "of" is shown, you may use either “of” or “in”, whichever seems natural to you at the time.

Ordinal Chunks

Chunk elements can also be referred to by their ordinal number (first, second, ... , millionth):

```
get the third item of (9,12,13,42) -- 13
```

Last, Penultimate, Middle, Any

In addition to numeric ordinals, there are four “special ordinals” that can be used -- `last`, `penultimate`, `middle` (or `mid`), and `any`. `Last` will identify the last chunk element (this is the same as specifying -1). `Penultimate` identifies the next-to-last chunk element (equivalent to specifying -2). `Middle` will select the element closest to the middle, based on the total number of chunk elements of that type. `Any` selects an element at random from among all those of the indicated type:

```
put any item of "cow,ant,moose,goat,donkey,elephant"
get the middle character of "serendipity" -- "d"
put the penultimate word of "Peace is its own reward." -- "own"
put the last line of gettysburgAddress into finalParagraph
```

Tech Talk

Syntax: {the} *ordinal* chunk **of** *expression*

Chunk Ranges

A chunk expression can refer to a range of chunk elements. The chunk range will include the beginning and ending elements specified, along with everything in between:

```
put characters 3 to 5 of "dragon" -- "ago"
put words 1 to 2 of "Alas, poor Yorick!" -- "Alas, poor"
put items middle to last of (1,2,3,4,5,6,7) -- (4,5,6,7)
put the second to penultimate items of (1,2,3,4,5,6) -- (2,3,4,5)
put items -3 to -1 of (1,2,3,4,5,6) -- (4,5,6)
```

A range value can also be used to access a range of chunks:

```
put chars 4..6 of "incredible" -- "red"
set fruitRange to 7..10
put chars fruitRange of "now appearing" -- "pear"
```

For ranges at the beginning or end of a value, you can also specify the “first n” or “last n” chunk elements:

```
put the last three items of (1,2,3,4,5,6) -- (4,5,6)
get the first 4 characters of "abracadabra" -- "abra"
```

Tech Talk

Syntax: chunks *firstNumberOrOrdinal* **to** *lastNumberOrOrdinal* **of** *expression*
chunks *range* **of** *expression*
{the} first *number* chunks **of** *expression*
{the} last *number* chunks **of** *expression*

Multiple Chunks (Lists of Chunks)

You can access several distinct chunks of a source value as a list, by specifying a list of the ordinal chunk numbers:

```
put items (1,3,5,7) of scores into oddResults
put words (4,1,2,-1) of "Mary had a little lamb"-- (little,Mary,had,lamb)
```

The result of accessing multiple chunks is always a list. Negative numbers and the special ordinals (middle, any, etc.) can also be used:

```
get items (-2,"middle") of "apple,banana,pear,orange,peach"-- (orange,pear)
```

Tech Talk

Syntax: chunks *indexlist* **of** *expression*

Working with Chunks

Storing Into Chunks

In addition to accessing a portion of a value, chunk expressions can also be used to store into a portion of a value, provided the thing being accessed is a container.

```
put "Jack Peterson" into name
put "d" into char 3 of last word of name
put "e" into char -2 of name
put "Olaf" into first word of name
put name -- "Olaf Pedersen"
```

You can also store something before or after a chunk:

```
put "The plant is growing" into phrase
put "egg" before word 2 of phrase
put " purple" after word 1 of phrase
```

```
put phrase -- "The purple eggplant is growing"
```

Storing Into Chunk Ranges

When storing into chunk ranges, the entire range will be replaced:

```
put "The great grey green gooey goblin" into monster
put "ugly" into words 2 to 5 of monster
put monster -- "The ugly goblin"
```

Storing Into Non-existent Chunks

If you store something into a chunk which is beyond the end of the container you are storing into, SenseTalk does its best to accommodate you. The results are different for different types of chunks. For text items beyond the number of items in the container:

```
put "mercury,venus,mars" into gods
put "saturn" into item 5 of gods
put gods -- "mercury,venus,mars,,saturn"
```

Here, the word "saturn" was put into the 5th text item of a value that previously had only 3 text items. To accommodate the request, two addtional commas were automatically inserted before the word "saturn" so that it would become the new 5th item. The actual character inserted will match the current setting of `the itemDelimiter` property. When storing into list items beyond the end of a list, the results are similar:

```
put (dog, cat, mouse) into pets
put rabbit into item 7 of pets
put pets -- (dog,cat,mouse,,,,rabbit)
```

For lines, the behavior is very similar to that for text items. But since `the lineDelimiter` can be a list of several possible delimiters, any one of which could indicate a new line, it can't be used to provide the inserted delimiter. Instead, a separate global property called `the lineFiller` provides the delimiter string (by default, Return) that will be inserted as many times as needed to fill the text out to the requested line number.

For word chunks beyond the end of the text, a simple delimiter is not enough. Since a word delimiter can be any amount of whitespace, simply inserting more spaces won't add more words. So `the wordFiller` global property provides a placeholder "word" (by default, "?") to insert along with spaces to fill out the text to the desired number of words (see the full description below for more options with this property):

```
put "one two three" into someWords
put "seven" into word 7 of someWords
put someWords -- "one two three ? ? ? seven"
```

For character chunks, `the characterFiller` global property (by default, ".") provides text to be repeated as needed to fill the text out to the desired character position:

```
put "abcdefg" into alpha
put "z" into character 26 of alpha
put alpha -- "abcdefg.................z"
```

When a *negative* chunk number larger than the number of chunks is used, the result is similar to the above descriptions for all chunk types, but with fillers or delimiters added at the *beginning* of the value to achieve the expected result:

```
put "abc" into backfill
```

```
put "X" into character -7 of backfill
put backfill -- "X...abc"
```

◊ the characterFiller, the lineFiller, the wordFiller global properties

What it Does

The `characterFiller`, `the lineFiller`, and `the wordFiller` global properties specify the behavior when a container is extended by storing into a character, line, or word chunk (respectively) that lies beyond the end of the container's contents.

How to Use It

`The characterFiller` is text which is repeated as needed to fill the container to the desired length. Set it to a single character or to a longer string if desired. The default value of `the characterFiller` is "." (a single period).

`The lineFiller` property provides the line delimiter to use when filling a container to the requested number of lines. Typically it should be set to one of the values listed in `the lineDelimiter` property. The default value of `the lineFiller` is the Return constant.

`The wordFiller` can be set to a single value or to a list of two values. When set to a single value it is a filler word that will be repeated as needed to reach the word number being stored into. In this case the inserted words will be separated by the first character of `the wordDelimiter` property (by default this is a space character). If `the wordFiller` is set to a list of two values, the first value will be the filler word, and the second value will be used as the delimiter between inserted words. The default value of `the wordFiller` is "?".

Examples

```
put "zig" into test
set the characterFiller to "/\"
put "zag" into character 9 of test
put test -- "zig/\/\/zag"
set the lineFiller to "+"
put "more" into line 5 of test
put test -- "zig/\/\/zag++++more"

set the wordFiller to "umm..."
put "Hello and" into greeting
set word 5 of greeting to "welcome!"
put greeting -- "Hello and umm... umm... welcome!"
```

Tech Talk

```
Syntax: set the characterFiller to fillCharacter
        get the characterFiller
        set the lineFiller to lineDelimString
        get the lineFiller
        set the wordFiller to placeholderWordOrList
        get the wordFiller
```

Storing Into Multiple Chunks

You can store into multiple chunks at once by supplying a list of chunk numbers:

```
put "The great grey green gooey goblin" into monster
put "G" into chars (5,11,16,22,28) of monster
put monster -- "The Great Grey Green Gooey Goblin"
```

You can store multiple values at once by supplying a list of values as well as of chunk numbers:

```
put ("Old","Ugly") into words (5,2) of monster
put monster -- "The Ugly Grey Green Old Goblin"
```

Deleting Chunks

Chunks of containers, besides being stored into, can also be deleted. This is done with the `delete` command (described in detail in Working With Text):

```
put (dog, cat, gorilla, mouse) into pets
delete item 3 of pets -- (dog, cat, mouse)
put "My large, lumpy lout of a lap dog is lost." into ad
delete words 2 to 7 of ad -- "My dog is lost."
```

Counting Chunks

To find out how many of a given chunk type are present in some value, use the `number` function:

```
get the number of characters in "extraneously" -- 12
put number of words in "I knew an old woman" -- 5
if the number of items in list is less than 12 then ...
```

◊ number function

What it Does

The `number` function counts the number of characters, words, lines, text items, list items, keys, values, or bytes in a value.

How to Use It

Use this function whenever you need to determine how many of a particular chunk type are present in a value. If the value is empty, the result will always be zero.

Examples

```
put "I wept because I had no answers, until I met a man who had no
questions." into quote
put the number of characters in quote -- 72
put the number of words in quote -- 16
put the number of items in quote -- 2
put the number of lines in quote -- 1
```

Tech Talk

Syntax: {the} number of chunks [in | of] *expression*

In addition to the usual text chunks and bytes, when *expression* is an object or property list `chunks` can be "keys" or "values" to count the number of keys or values that are defined in the object.

Testing for Presence of a Chunk Value – Is Among

You can find out whether a particular value is present as one of the chunks of another value using the `is among` or `is not among` operator.

◊ is among operator

What it Does

The `is among` operator tests whether a particular value is present among the characters, words, lines, text items, list items, keys, values, or bytes in a value.

How to Use It

Use this operator to determine whether a target value is present among the chunks of a particular type in a value. This will only return true if the target value is equal to one of the specified chunks. Contrast this with the `is in` or `contains` operators which will only test whether one text string is a substring of another (see the second example below).

Examples

```
put "be" is among the words of "To be or not to be" -- true
put "be" is among the words of "I believe I am a bee" -- false
put 7 is among the items of (5,5+1,5+2,5+3) -- true
put "M" is not among the characters of "Avogadro" -- true
```

Tech Talk

Syntax: *targetValue* is {not} among {the} chunks of *sourceValue* {considering case | ignoring case}

In addition to the usual text chunks, when *expression* is an object or property list `chunks` can be "keys" or "values" to test whether *targetValue* is one of the keys or values of the object.

Determining Chunk Position of a Value

You can find the ordinal position of characters, words, lines, text items, and list items within a value (searches are case-insensitive unless “considering case” or “with case” is specified). The number 0 will be returned if the target expression is not found:

```
put "The rain, in Spain, is mainly in the plain" into text
put the character number of "t" within text -- 1
put character number of "t" within text considering case -- 34
put the text item number of " in Spain" within text -- 2
put the word number of "mainly" within text -- 6
put the line number of "another line" within text -- 0
```

To find the word, line, or item number that contains a value (rather than one that is equal to the value), use the word `containing` instead of `of`:

```
put the word number of "main" within text -- 0
put the word number containing "main" within text -- 6
put the text item number containing "Spain" within text -- 2
```

Tech Talk

Syntax: {the} chunk number of *targetValue* within *sourceValue* {considering case | ignoring case}
{the} chunk number containing *targetValue* within *sourceValue* {considering case | ignoring case}

Counting Occurrences of a Chunk Value

To count how many times a particular chunk value occurs within a source value, use the `number of occurrences` or `number of instances` function:

```
put the number of occurrences of "a" among the chars of "banana" -- 3
put the number of instances of "be" among \
   the words of "to be or not to be" -- 2
put the number of occurrences of 15 among the \
   items delimited by "-" of "315-15-4152" -- 1
```

If a specific chunk type is not named, `characters` are assumed unless the source value is a list or an object, in which case `list items` or property `values` are assumed, respectively:

```
put number of occurrences of "a" in "banana" -- 0
put the number of instances of 3 in (1,3,5,6,3,2) -- 2
put number of occurrences of "Do" in "Do,re,mi,do" -- 2
```

For case-sensitive comparisons, use “considering case” (or set the caseSensitive property to true):

```
put number of instances of "Do" in "Do,re,mi,do" considering case -- 1
```

As a special case, “among the characters of” can be used not only to count occurrences of a single character, but of a sequence of characters:

```
put number of instances of "na" among the chars of "banana"   -- 2
```

Tech Talk

Syntax: {the} number of {occurrences of} *targetValue* among {the} chunks of *sourceValue* {considering case | ignoring case}

Iterating Over All Chunks in a Value

To do something with each of the chunks within a value, use the `repeat with each` form of the repeat command (which is also described in Script Structure). Here is a short example:

```
repeat with each line in file "/tmp/output"
   if the first word of it is "Error:" then put it
end repeat
```

Extracting a List of Chunks Using "each" Expressions

Note: For more information about "each" expressions, see Each Expressions in Ranges, Iterators, and Each Expressions.

Any expression of the form `each` *chunkType* `of` *sourceValue* will yield a list containing all of the chunks of that type (if *chunkType* is omitted, `item` will be assumed):

```
put each character of "Sweet!" -- ("S","w","e","e","t","!")
put each word of "Wisdom begins in wonder"-- ("Wisdom","begins","in","wonder")
```

More interestingly, an `each` expression can be part of a larger expression. Within the larger expression, operators apply to each item of the list rather than to the list as a whole:

```
put "Z" & each character of "Cat" -- ("ZC","Za","Zt")
put 2 + each item of "1,2,5,6" -- (3,4,7,8)
put the length of each word in "Wisdom begins in wonder"
         -- (6,6,2,6)
put each word of "Wisdom begins in wonder" begins with "w"
         -- (true,false,false,true)
```

Parentheses limit the scope of the larger `each` expression, limiting the behavior to applying to the list as a whole rather than to each individual item:

```
put sum of the length of each word in "Wisdom begins in wonder" -- (6,6,2,6)
put sum of (the length of each word in "Wisdom begins in wonder") -- 20
```

An `each` expression can also include a `where` clause to select a subset of the items in the list. The word `each` can be used within the where clause to refer to each source item:

```
put each word of "Wisdom begins in wonder" \
   where each begins with "w" -- ("Wisdom","wonder")
put each item of (1,2,3,4,5,6,7,8,9) where the \
   square root of each is an integer -- (1,4,9)
```

Tech Talk

Syntax: `each chunk of` *sourceExpr* **{**`where` *conditional***}**

Script Structure

When a script is executed, the statements in the script are normally executed one at a time, in the order they appear in the script. This section presents information on control structures and commands which may conditionally alter the basic flow of execution.

Control structures form the foundation of a script. They define the basic framework which lets you create handlers, declare global and universal variables, and control the execution of your script through conditional statements and repeat loops, among other things. They also provide control over the flow of messages through the system.

In addition to the control structures presented here, another important part of structuring scripts is the use of handlers to deal with different messages. Handlers are presented in Objects, Messages, Handlers, and Helpers.

Statements and Comments

Statements

A SenseTalk script is a sequence of statements or commands which contain your instructions. Each statement is typed on a separate line, in the form of an imperative command that begins with a verb, such as:

```
put "Hello, World!" into greeting
wait 3 seconds
put "EggPlant says: " & greeting
add 5 to total
```

A simple SenseTalk script is a series of statements, typically one on each line of a script. Blank lines are ignored. When a script is run, or executed, SenseTalk begins performing the actions described by each statement in turn, from the first to the last, unless directed to do otherwise (such as by the flow control constructs described later in this section).

Continuing Long Statements on the Following Line

A longer than usual SenseTalk statement may be continued to the next line by entering a "\" (backslash) character at the end of a line. Here is a single statement written on 3 lines:

```
put "Hello! This is a simple contrived example " \
    & "to show how a long statement " \
    & "may be continued across several lines."
```

Comments

Comments may be introduced into a script at any point, to provide notes to the reader or developer of the script. Comments begin with `--` (two dashes) or `//` (two slashes) or `#` (a pound sign) and continue to the end of the line. Block comments, enclosed in `(*` and `*)` can be used to insert comments into the middle of a line, or to span multiple lines:

```
(* This script doesn't do much, but it illustrates several types of
comments (* including nested comments *) that may be used. *)
wait 2 seconds -- make the user think the computer is thinking hard!
put (* "Hello!" *) "Bonjour!" // let's change the greeting to French
```

Conditional Statements

Conditionals allow your script to make choices, carrying out some actions only under certain conditions, and other actions under other conditions.

◊ if ... then ... else ...

All forms of the `if` statement evaluate a condition expression, which must evaluate to a logical value (either **true** or **false** or one of the equivalent values “yes” or “no”, or “on” or “off”). An empty value is also treated as false. If the condition is **true** (or “yes” or “on”) then the statement or statementList following the word `then` is executed. If the condition is **false** (or “no” or “off” or empty) then the statement or statementList following the word `else` (if it is present) is executed.

The `if` statement may take any of the following forms (note that *statement* is a single statement, while *statementList* may be multiple statements, each on its own line): The `else` portion is always optional.

form 1 (single statement):

if *condition* **then** *statement* {**else** *statement* }

form 2 (multi-line single statement):

```
if condition
then statement
{else statement }
```

form 3 (several statements):

```
if condition then
     statementList
{else
     statementList }
end if
```

form 4 (chained conditionals):

```
if condition1 then
     statementList
else if condition2 then
     statementList
{else
     statementList }
end if
```

The final form shown above (form 4) allows testing for a series of mutually exclusive conditions. Any number of conditions can be tested, by chaining as many `else if` blocks as needed, optionally followed by an `else` block before the closing `end if` to catch any cases that didn’t match any of the tested conditions.

In forms 3 and 4, where the word `then` appears at the end of the line, it may be omitted for simplicity, if desired.

Examples:

```
if true then put "Yes!" -- always puts "Yes!"
if balance < 1000 then put "The balance is getting low"
if date of transaction is between date("January 1") and date("June 30")
then put "First half" into period
else put "Second half" into period
```

Repeat Loops

One of the great strengths of computers is their ability to perform repetitive tasks with ease. SenseTalk provides several different types of `repeat` loops for this purpose.

A `repeat` loop is used any time you want to execute one or more statements repeatedly some number of times. The statements to be repeated are preceded by one of the `repeat` statements described below, and must always be followed by an `end repeat` statement to mark the end of the loop. Repeat loops may be nested.

◊ repeat forever

This form of repeat loop will repeat indefinitely until terminated. Usually, you don't really want your script to keep looping forever (a condition known by programmers as an "infinite loop"), so `repeat forever` loops typically include at least one `exit`, `return`, or `pass` statement that will break out of the loop when some condition has been met: The word `forever` is optional.

Example:

```
repeat forever
   get nextValue(partList)
   if it is empty then exit repeat -- all done
   doSomethingWith it
end repeat
```

Tech Talk

Syntax: repeat {forever}
statementList
end repeat

◊ repeat number times

This form repeats the number of times specified by the expression *number*.

Example:

```
repeat 6 times
   put "SenseTalk is fun!"
```

```
end repeat
```

Tech Talk

```
Syntax: repeat {for} number {times}
            statementList
        end repeat
```

◊ repeat until condition

This form of repeat loop will execute until the condition expression evaluates to a value of **true**. The condition is evaluated before the first and each subsequent execution of the loop.

Example:

```
repeat until list is empty
    put the last item of list
    delete the last item of list
end repeat
```

Tech Talk

```
Syntax: repeat until condition
            statementList
        end repeat
```

◊ repeat while condition

This form of repeat loop will execute as long as the condition expression evaluates to a value of **true**. The condition is evaluated before the first and each subsequent execution of the loop.

Example:

```
repeat while x < 100
    put x
    add x to x
end repeat
```

Tech Talk

Syntax: repeat while *condition*
statementList
end repeat

◊ repeat with variable = start to finish

This form of repeat sets a variable to each of a sequence of values. The term `up to` or `down to` may be used instead of the word `to` to indicate whether the loop variable should be incremented (increased) or decremented (decreased) each time through the loop. If neither direction is specified, "up to" is assumed. A `step` option lets you specify an amount other than 1 by which the loop variable will be changed:

Examples:

```
put "Countdown:"
repeat with n=10 down to 1
   put n
   wait one second
end repeat
put "BOOM!"
```

```
repeat with n=1 to the number of lines in file "/tmp/legalDoc"
   put n & ": " & line n of file "/tmp/legalDoc"
end repeat
```

```
repeat with rate = 4.75 to 8.5 step 0.25
   put "Rate: " & rate & tab & "Payment: " & calcPayment(rate)
end repeat
```

```
repeat with discount = 50 down to 0 step 5
   insert (amount - discount%) after discountList
end repeat
```

Tech Talk

Syntax: repeat [with | for] *variable* **[= | as | from]** *start* **{up | down}** to *finish*
{step *stepAmount***}**
statementList
end repeat

The value of *variable* is set to the value of *start* before the first execution of the loop. *Variable* is then incremented (or decremented for "down to") by the value of *stepAmount* (or by 1, if no *stepAmount* is given) before each subsequent repetition. The value of *variable* is compared to the value of *finish* before each execution. When *variable* is greater than *finish* (or less than *finish* for "down to") the loop is terminated. If *variable* is a reference, it will be reset to an ordinary variable before use.

Note

When a repeat loop using a loop variable is nested inside another, be sure to use a different variable for each loop, in order to avoid conflicts.

◊ repeat with each

The `repeat with each` form is perhaps the most powerful and useful of all of SenseTalk's repeat loops. This form of repeat makes it very easy to step through each of the values in a list, the words in a sentence, the lines of text in a file, or many other subdivisions of a value.

Examples:

```
repeat with each item of myAccountList
    set property balance of it to zero --Set the balance of each account in a
list of account objects to zero:
end repeat

put empty into condensedList -- start with nothing; Gather all non-blank lines of
a file into a container:
put file "/tmp/somefile" into sourceText -- read the file
repeat with each line of sourceText
    if it is not empty then put it & return \
    after condensedList
end repeat--Count the number of times each word occurs in a text file:

answer file "Select a file to count its words" -- get the file to be counted:
if the result is "Cancel" then exit handler
put it into sourceFile
put (:) into wordCounts -- start with an empty property list
repeat with each word of file sourceFile
    add 1 to property (it) of wordCounts
end repeat
put "Word counts in file " & sourceFile & ":"
repeat with each item of the keys of wordCounts
    put it && "appears" && wordCounts.(it) && "times"
end repeat

repeat with each line of file "/tmp/example" by reference
    if it begins with "#" then delete it--Delete every line that begins
with "#" in a text file:
end repeat
```

Tech Talk

Syntax: **repeat with each** *subitemType* **[of | in]** *containingItem* **{by reference}**
statements
end repeat

repeat with *variable* **= each** *subitemType* **[of | in]** *containingItem* **{by reference}**
statements
end repeat

repeat with each {*subitemType*} *variable* **[of | in]** *containingItem* **{by reference}**
statements
end repeat

These repeat formats will execute the statements within the repeat loop (until the matching `end repeat` statement) exactly once for each object or part (as specified by *subitemType*) that is contained within the specified *containingItem*. Virtually any of the combinations of *subitemType* and *containingItem* that are allowed with the `number` function can be used (that is, anything that can be counted within some containing object or container).

In all forms of the `repeat with each` statement, *subitemType* indicates the type of item being iterated over (if omitted in the third format shown above, "item" is assumed), and *containingItem* is the specific object or container whose parts are being iterated over.

In the second and third formats, *variable* is the name of a variable which will be assigned the identifier or contents of a particular subitem each time through the repeat loop. In the first format, where no specific variable is specified, the variable `it` is used. If `by reference` is specified, *variable* is set to be a reference to each subitem. Otherwise, if *variable* (or `it`) is already a reference, it will be reset to be an ordinary variable before being used as the loop variable.

In cases where *subitemType* is a type of object, the long id of the specific subObject will be assigned to `it` or *variable* each time through the loop. In cases where *subitemType* is a type of text chunk, the actual chunk of text will be stored in `it` or in *variable*.

Note

When a repeat loop using a loop variable is nested inside another, be sure to use a different variable for each loop. In particular, watch out for nested repeats which implicitly use `it` as a loop variable.

◊ repeatIndex function

What it Does

The `repeatIndex()` function returns the current iteration count of the innermost executing repeat loop.

Examples

```
if repeatIndex() is a multiple of 100 then put repeatIndex()
```

Tech Talk

```
Syntax: repeatIndex()
        the repeatIndex
```

The `repeatIndex()` can be used in any type of repeat loop. Its value is set to one the first time through the loop and increases by one on each iteration. If used inside nested loops it will report the index of the innermost loop that it is in.

Flow Control

There are a number of statements that can affect the flow of statement execution within a loop. In addition to the statements listed here, `return` and `pass` statements will terminate execution of a repeat loop (see Working with Messages).

◊ next repeat

Causes any statements following the `next repeat` statement, down to the `end repeat`, to be skipped and execution to jump directly back to the beginning of the current (innermost) repeat loop. Execution then continues with the next iteration.

◊ exit repeat

Terminates execution of the current (innermost) repeat loop. Script execution proceeds with the next statement following `end repeat`.

◊ exit handler

Terminates execution of the current handler, returning immediately to the handler that called it. This may also take the form `exit` handlerName where handlerName is the name of the current handler, or `exit on`, `exit function`, `exit getProp`, or `exit setProp` to exit from the corresponding type of handler.

◊ exit all

Terminates execution of the current handler, the handler that called it, and all other handlers further up the call stack.

Pausing Script Execution

Script execution can be paused, either to wait for some external condition, or simply to delay for a specified length of time, using one of the `wait` statements.

◊ wait while, wait until

The `wait until` command will pause script execution until a given condition occurs. The `wait while` command waits until a condition is no longer true. In both cases, the condition expression is evaluated repeatedly, and should be an expression whose value will eventually change to the awaited outcome.

```
wait until the time is "12:50 PM"
wait until temperature(hell) < 32
wait while the sound is not done
```

Tech Talk

Syntax: wait while *condition*
wait until *condition*

◊ wait

The `wait` command will pause script execution for a specified length of time. The script simply goes to sleep for the time interval indicated, then wakes up and continues with executing the following statement.

The timeInterval is any expression that evaluates to a number of seconds. Since SenseTalk supports time interval expressions which evaluate to seconds, you can express the time in a natural fashion, using the terms `weeks`, `days`, `hours`, `minutes`, `seconds`, `ticks`, `milliseconds`, and `microseconds` (see "Time Intervals" in Values). If a time unit is not specified, seconds are assumed.

Examples

```
wait 20 ticks -- 1/3 of a second
wait 2 days 7 hours 14 minutes and 28.6 seconds
wait 3 milliseconds
wait 1.5   -- assumes seconds, since no unit was specified
```

Tech Talk

Syntax: wait *timeInterval*

Error Handling

During script execution, errors may occur. These are generally called "runtime errors" because they occur while the script is running, as opposed to syntax errors which are detected before a script is run. When a runtime error occurs, it "throws an exception". When an exception is not caught, it causes script execution to terminate and an error message to be displayed. The `try...catch...end try` control structure allows you to catch these exceptions so your script to handle the error condition in the manner you choose. The `throw` command can be used to throw your

own exceptions, or to re-throw an exception that you caught.

◊ try ... catch ... end try

What it Does

The `try` statement protects a block of statements from being terminated by any exceptions that may be thrown while they are running. If such an exception is thrown, control immediately transfers to the first statement following the `catch` statement. If no exception is thrown, the catch block is skipped, and execution continues with the next statement following `end try`.

Example

```
to testExceptions
   try
        doSomething
        throw "Bad Problem", "Something went wrong"
        doSomethingElse -- this will not get executed
   catch anException
        -- do processing here to handle the error
        put anException -- shows "Bad Problem"
        put anException.reason -- shows "Something went wrong"
   end try
   -- now test the single-statement try:
   try to set product to 5*amt -- throws an exception
   if the exception is not empty then put the exception
end testExceptions
```

Tech Talk

Syntax: **try**
statementsToTry
{
catch *{exceptionVariable}*
statementsToHandleErrors
}
end [try | catch]

try *{to} singleStatementToTry*

Basically, any exception which is thrown in the “try” portion will cause execution to transfer directly to the “catch” part of the script. If no exception is raised, the catch portion is skipped. Exceptions may be thrown by SenseTalk itself — for example, if there is no “doSomething” handler an Unknown Command exception will be thrown — or directly by your script using the `throw` command as shown. If an exception is not caught in the handler where it occurs, it may be caught by another handler which called the first one. For example, if an exception is thrown in the doSomething handler and not caught there, it will be caught here in the testExceptions handler.

◊ try ... catch ... end try

The exceptionVariable name on the `catch` line (anException in the example) is optional. If supplied, it receives an exception object whose text value is the name of the exception. The exception object's `name` property also contains the name of the exception that was thrown ("Bad Problem" in the example), and its `reason` property typically describes the reason for the exception in more detail ("Something went wrong" in the example). The caught exception is also available in `the exception` global property. See the description of the exception global property below for more details on exception objects.

If the `catch` portion is omitted, exceptions thrown in the try block will interrupt the flow of the script, preventing the remainder of statements within that try block from being executed, but will otherwise be ignored, and execution will continue with the next statement after the try block.

The single statement version of the `try` statement provides an easy way to execute a single command without terminating the script if it throws an exception. If an exception is thrown, execution proceeds normally with the next statement. The exception is available in `the exception` global property (described later in this section), which otherwise is set to empty.

◊ throw

What it Does

Throws an exception, which will cause script execution to terminate with an error message unless the exception is caught. A reason may be given with the `throw` command by supplying it as a second parameter, or a property list with name and reason properties may be used instead:

Examples

```
throw "BadProblem","Something is seriously messed up!"
throw (name:"Error Code 97", reason:"Invalid Phone Number")
```

Tech Talk

Syntax: throw {exception} *exceptionName* **{,** *exceptionReason* **{,** *additionalReason* **...}}**
throw {exception} *exceptionObject*

If an *exceptionObject* is given, it should typically have at least a name property, and usually a reason property, as shown in the example above. In any case, the `throw` command will ensure that a complete exception object is created and thrown, using the information that you supply, plus some additional details.

◊ the exception global property

What it Does

This property contains information about an exception thrown in the context of the latest try block.

Examples

```
try
    throw "Invalid Access", "Name or password is wrong"
catch
    put the exception -- " Runtime Error at line 2: Invalid Access - Name or
password is wrong"
    put the exception's name -- "Invalid Access"
    put the exception's reason -- "Name or password is wrong"
end try
```

Tech Talk

Syntax: the exception

At the beginning of each try statement, `the exception` is set to empty. If an exception is thrown in the try block, the exception object is put into `the exception`. It is then available within the catch portion of the try block, or any time thereafter until the next try statement.

An exception is an object (a property list). The exception object has a text value that will display a brief description if the exception is simply rendered as text. However, it also contains a number of other pieces of useful information, including one or more of the following properties:

name	the exception name or identifier
`reason`	the reason given for the exception
`location`	a textual description of the location in the script where the exception occurred
`callStack`	a list of stack frame objects providing information about the call sequence to the point where the exception occurred, as returned by the callStack() function
`scriptError`	a scriptError object containing specific pieces of information about the error

Because `the exception` property is global property (which is a container), a script is free to modify its contents at any time. This may be useful when an exception is caught, allowing you to modify the caught exception or add additional properties to it, for example, and then throw it again.

Note: The standardFormat()

The `standardFormat()` *function can be used to display all properties of an exception object.*

Deprecated functions

The `exceptionName()`, `exceptionReason()` and `exceptionLocation()` functions return the name, reason, and location of the caught exception within a catch block. They are now obsolete, though, and their use is discouraged -- please use `the exception` instead.

◊ tryDepth function

What it Does

This function returns the level of nesting of try blocks in effect at the current point in script execution. This can be used to quickly determine whether a thrown exception will be caught at some higher level of the call stack. A return value of zero indicates that there are currently no try blocks, so any exception that is thrown will cause script execution to terminate.

Examples

```
try
    put the tryDepth -- 1
catch
    put the tryDepth -- 0 (outside of the 'try' portion)
end try
```

Tech Talk

Syntax: `tryDepth()`
`the tryDepth`

Declaring global and universal variables

◊ global

Use a `global` statement to declare global variables, which can be accessed from any script within a document. Global variables must be declared within each handler where they are used. See Containers for more information on global and universal variables.

Tech Talk

Syntax: `global` *{variable {, variable...} }*

◊ universal

Use a `universal` statement to declare universal variables, which can be accessed from any script during the current run of the host program. Universal variables must be declared within each handler where they are used. See Containers for more information on global and universal variables.

Tech Talk

`Syntax: universal` *{variable {, variable...} }*

Lists and Property Lists

Lists and property lists provide two very convenient and powerful ways to organize and manipulate data within your scripts. This section presents the key things you need to know in order to fully exploit their power.

For more information on working with lists, also see Chunk Expressions, and the discussion of the `repeat with each...` control structure in Script Structure.

Property lists are actually the simplest form of objects. For complete information on objects, see Objects, Messages, Handlers, and Helpers.

In addition to lists and property lists, SenseTalk also provides a hierarchical tree structure, which is described in Working with Trees and XML.

Lists

Creating Lists

A list can be created in a script by simply listing two or more expressions separated by commas and enclosing the list in parentheses. Parentheses or braces are required around lists. This makes the list syntax clear and consistent, and allows for easily constructing nested lists:

```
put (1,3,5,7,9) into oddList
put (("dog", "Fido"), ("cat", "Cleo")) into nestedList
```

For longer lists, you can enclose the list in curly braces { } which allows the list to span multiple lines:

```
set searchPaths to {
    "/Library/WebServer/Documents",
    "~/Documents",
    "/Users/hemingway/Documents/temp/ImportantStuff" }
```

Tech Talk

Syntax: `(` *{expr { , expr}... }* `)`
`{` *{expr { , expr}... }* `}`
{an} `empty list`

Here, *expr* can be any expression, including nested lists, or may be omitted. If any *expr* is omitted, leaving sequential commas with nothing in between, an empty item is created. A final comma at the end of a list is ignored.

List Contents

Lists can include any type of value, including other lists and property lists. Expressions can be used in constructing lists:

```
put ("geranium", pi * (2 * radius), (age:42)) into mixedList
```

Combining Lists

To join two lists as one (without nesting), the `&&&` operator may be used:

```
put (3,4,5) into tr1 -- (3,4,5)
put (5,12,13) into tr2 -- (5,12,13)
put tr1 &&& tr2 into longList -- (3,4,5,5,12,13)
put (tr1,tr2) into twoTriples -- ((3,4,5),(5,12,13))
put (1,(2,(3,4)),5) into nestedList -- (1,(2,(3,4)),5)
```

Accessing List Items

Accessing items within lists, including ranges of items and lists of items, is fully supported using the item and items chunk expressions (or their more explicit forms, list item and list items). Accessing a single item yields the item at that index. Accessing a range of items always yields a list, even when the range specifies a single item:

```
put (1,2,3,4,5) into list -- (1,2,3,4,5)
put list items 3 to last of list -- (3,4,5)
put item 2 of list -- 2
put items 2 to 2 of list -- (2)
put items (4,2,5) of list -- (4,2,5)
```

Note that the term `item` will, by default, refer to text items rather than list items if the variable in question is not known to be a list. Using the explicit term `list item` will always treat the variable as a list even if it contains a non-list value (in which case it will be treated like a one-item list containing that value).

Converting Lists to Text

When a list needs to be accessed as text (such as when it is displayed by a `put` command), SenseTalk converts it automatically to a text representation. By default, this will be done by enclosing the entire list in parentheses, with the items in the list separated by commas. You can change this formatting by setting the `prefix`, `separator`, and `suffix` values of `the listFormat` global property. The `prefix` is the text that will be used before the list (usually a left parenthesis), `separator` is the text inserted between items, and `suffix` is the text appended after the list. These may be set to any text you like, including `empty`.

```
set the listFormat to (prefix:">>", separator:":", suffix:"<<")
put (1,2,3,4,5) -- displays >>1:2:3:4:5<<
```

When a list is converted to text, each value within the list will also be converted to a text representation. You can control the quoting of these values by setting the `quotes` value of `the listFormat` global property. For details on how this is done, see the section “Conversion of Values” in Expressions.

The `split` and `join` commands and functions and the `asText` function, described in detail in Working with Text, and the corresponding `split by` and `joined by` operators (described in Expressions), provide ways to explicitly convert text to lists and vice versa. The `value` function can also produce a list from text (see Expressions). The `standardFormat` function can be used to convert a list to text in a format that will produce the original list again when that text is evaluated by the `value` function.

Single-Item Lists

When a single value is enclosed in parentheses in a script, the parentheses are treated as a grouping operator, so that value will not be treated as a list. To make SenseTalk recognize a single item in parentheses as a list, include a comma after the value:

```
put (12,) into shortList -- this makes a list with one item
```

For most purposes, a list containing a single item is treated the same as a single (non-list) value. For example, if myList is (4) — that is, a list containing a single item which is the number 4 — then it will work just fine to say `put myList + 2 into theSum` — theSum will then contain the number 6. Normally, when accessing such a list as text the value will be shown in parentheses. To avoid this behavior, you can set the `prefix` and `suffix` properties of `the listFormat` to empty.

Empty Lists

An empty list can be created using an empty pair of parentheses, or the phrase `empty list`:

```
put () into newlist -- newlist is now a list with no items
put 16 into item 1 of newlist -- (16)
put empty list into item 2 of newlist -- (16,())
```

Inserting Items into a List

The `insert` command is used for inserting items before, into, or after a list or any item of a list. It is somewhat similar to the `put` command, but always treats the destination container as a list:

```
put "abc" into myList -- abc (not actually a list)
put "de" after myList -- abcde (still not a list)
insert "xyz" after myList -- (abcde,xyz)
insert 24 before item 2 of myList -- (abc,24,xyz)
```

When inserting a list of items into another, there are two possible ways the insertion can occur. The default mode – known as “item by item” – is to insert each item from the source list into the destination list:

```
put (1,2,3,4) into list -- (1,2,3,4)
insert (A,B) after item 2 of list -- (1,2,A,B,3,4)
```

It is also possible to insert a list into another so as to create a nested list, by using the `nested` keyword:

```
put (1,2,3,4) into list -- (1,2,3,4)
put (A,B) into otherList -- (A,B)
insert otherList nested after item 2 of list -- (1,2,(A,B),3,4)
```

The standard insertion behavior can be controlled by setting `the listInsertionMode` local property or `the defaultListInsertionMode` global property to either “nested” or “item by item”:

```
put (1,2,3) into aList
set the listInsertionMode to "nested"
insert (4,5) into aList -- (1,2,3,(4,5))
```

The `insert` command is described in detail later in this section.

Replacing Items in a List

While the `insert` command is specifically tailored for working with lists, it only adds new values into a list. To replace existing items in a list with new values, use the `set` or `put into` commands.

Note: `Put before` and `put after`

Unlike `put into`, the `put before` and `put after` commands are rarely used with lists. Their behavior when the destination is a list is to concatenate a text value before or after the first or last item of the list. More often what is wanted is to insert new items before or after a list, using the `insert before` and `insert after` commands.

Using the `put into` or `set` commands to put one or more new values into a list will replace the entire contents of the destination with the new values. The final result can take several forms. If the destination container is simply a variable, that variable's previous contents are discarded, and replaced by a copy of the new value(s). If the source is a single value and the destination is a single item or a range of items within a list, those items are replaced by the new value:

```
put (1,2,3,4) into aList
put 99 into items 2 to 3 of aList -- (1,99,4)
```

If the new value is a list rather than a single value, the result will be a nested list if the destination is a single item:

```
put (1,2,3,4) into aList
put (a,b) into item 2 of aList -- (1,(a,b),3,4)
```

If the destination is a range of items (even a 1-item range), those items are replaced by the new values:

```
put (1,2,3,4) into aList
put (a,b,c) into items 2 to 3 of aList -- (1,a,b,c,4)
put (x,y) into items 1 to 1 of aList -- (x,y,a,b,c,4)
```

These results can be controlled explicitly by specifying "item by item" or "nested":

```
put (1,2,3,4) into aList
put (a,b,c) nested into items 2 to 3 of aList -- (1,(a,b,c),4)
set item 1 of aList to (x,y) item by item -- (x,y,(a,b,c),4)
```

The `listInsertionMode` and `defaultListInsertionMode` properties, described earlier, also apply when the destination is a range of items and "nested" or "item by item" is not stated explicitly (some of the examples shown above assumed the default "item by item" setting for these properties).

You can store values into several items at once by specifying a list of indexes to store into:

```
set notes to "c,d,e,f,g"
put "#" after items (1,4) of notes
put notes -- "c#,d,e,f#,g"
set items (3,1) of notes to ("c","a")
put notes -- "a,d,c,f#,g"
```

You can store a single value into multiple items at once:

```
put (1,2,3,4) into aList
put "Egg" into items (6,4,2) of aList -- (1,Egg,3,Egg,,Egg)
```

Deleting Items from a List

The `delete` command can be used to delete one item or a range of items from a list, using the standard chunk expression syntax:

```
delete item 4 of searchPoints
delete the last 2 items of toDoList
```

Counting the Items in a List

The `number` function can be used to find out how many items are in a list, by asking for the number of items (if the source is known to be a list) or explicitly for the number of list items (if it may not be a list):

```
put the number of items in (1,3,4,5,9) -- 5
put the number of list items in phoneNum into phoneCount
```

Determining the Location of an Item in a List

The `item number of ... within ...` expression will give the item number of a value within a list, or zero if that item is not found in the list:

```
put ("a","b","c","d","e") into myList
put the item number of "c" within myList -- 3
```

Performing Arithmetic on Lists

Vector arithmetic is supported by lists. You can add or subtract two lists of numbers, provided they are both the same length. This will add or subtract the corresponding items of the two lists. Vector operations work with nested lists as well.

```
put (12,4,6,22) into numList
add (1,2,3,4) to numList -- results in (13,6,9,26)
```

Multiplication and division of lists also works for lists of equal length.

```
put (6,7,14,33) / (3,2,1,3) -- (2,3.5,14,11)
put (4,5,6) * (9,8,7) -- (36,40,42)
```

You can also multiply or divide a list by a single value (sometimes called a ‘scalar’ because it scales all the values in the list).

```
put (6,7,14,33) / 2 -- (3,3.5,7,16.5)
```

List Comparisons

The comparison operators treat lists differently than single values. The =, <>, <, >, <=, and >= operators go through lists item-by-item, comparing individual values to determine equality or inequality. For equality, there must be the same number of items in both lists, and each matching pair must be equal. For inequality, the first unequal pair of items determines the order. If the lists are of unequal length but are equal for every item in the shorter list, the longer list is considered greater. A plain (non-list) value is treated like a list with a single item when it is compared to a list.

For example:

```
put (3-2, "02.00", "A") = (1,2,"a") -- true unless caseSensitive is set
```

```
put (1,2,2) < (1,2,3) -- true
put (1,2) < (1,2,0) -- true
put (1,2,3) < (1,3) -- true
put (1,2,3) < 5 -- true, because 1 is less than 5
```

The `begins with`, `ends with`, `is in`, and `contains` operators all check for the presence of whole values or sequences of values in the list, whenever the value being searched is a list. These operators all look deeply into nested lists. When the value being looked for is also a list, its items must appear in the same order within the list or one of its sublists in order to match.

For example:

```
put ("a","b","c") begins with ("a","b") -- this is true
put ("a","b","c") begins with "a" -- this is also true
put (1,2,3) ends with (2,3) -- true
put (1,2,3) ends with 3 -- true
put (11,12,13) ends with 3 -- this is false
put (11,12,13) ends with 13 -- true
put ("some","choice","words") contains ("choice","words") -- true
put ("some","choice","words") contains "words" -- true
put ("some","choice","words") contains "word" -- false
put (3,4) is in ((1,2),(3,4,5)) -- true
put (2,3) is in ((1,2),(3,4,5)) -- false (not in the same sub-list)
```

Iterating Over Items in a List

To perform actions with each of the values in a list, you can step through the list's values using a `repeat with each item` loop. This will set the variable `it` to each of the values of the list in turn:

```
repeat with each item of (1,2,3,4,5,10,20,30,40,50)
        put it & tab & it squared & tab & it cubed
end repeat
```

To use a variable other than `it`, include a variable name after the word `item`:

```
repeat with each item x of (1,2,3,4,5,10,20,30,40,50)
        put x & tab & x squared & tab & x cubed
end repeat
```

To modify items in a list, including changing their values or even deleting them, specify `by reference`. This will set the loop variable to be a reference to each of the items instead of a copy of their values:

```
set numList to (10,11,12,13,14,15,16)
repeat with each item of numList by reference
        if it is an odd number then delete it
        else divide it by 2
end repeat
put numList -- (5,6,7,8)
```

Selecting List Items Using "each" Expressions

The use of `each` expressions to obtain a list of selected chunks of a value was described in the section " Extracting a List of Chunks Using "each" Expressions" in Chunk Expressions. Similarly, to extract a sublist of selected items

from a list, use an `each` expression with a `where` clause:

```
set numList to (9,10,11,12,13,14,15,16)
put each item of numList where each is an odd number -- (9,11,13,15)
put each item of numList where sqrt(each) is an integer -- (9,16)
```

Note: For more information about "each" expressions, see Each Expressions in Ranges, Iterators, and Each Expressions.

Applying Operations to Lists Using "each" Expressions

To perform an operation on each item in a list, giving a list of the resulting values, use an `each` expression (with or without a `where` clause) as an element of a larger expression:

```
set numList to (9,10,12,16)
put each item of numList + 3 -- (12,13,15,19)
put sqrt of each item of numList -- (3,3.162278,3.464102,4)
```

The expression can be quite complex, with multiple operations being applied to each item from the `each` expression. Enclosing parentheses will limit the scope of the each expression: operators within the parentheses will apply to each item of the list; operators outside the parentheses will apply to the list as a whole.

◊ insert command

What it Does

Inserts a value or list of values at the end of a list, or before or after a specific item of the list.

Examples

```
insert 5 into myList
insert newName before item index of nameList
insert (3,5) nested after pointList -- creates a nested list
insert "myBuddy" before my helpers
```

Tech Talk

Syntax: `insert` *expr* `{nested | item by item} [before | into | after]` *container*

Expr is an expression whose value will be inserted into container. If `before` is specified, the new value is inserted at the beginning of the list. If `after` is specified, the new value is inserted at the end of the list. If `into` is specified, the new value is inserted into the list at an appropriate position (currently always at the end of the list).

If *container* is specified as a particular item of a list (e.g. `insert x after item 2 of myList`), the `before` and `after` options will insert the new value into the list before or after that item, respectively. The `into` option, however, will treat that existing item as though it were the list being operated on. If it is a single value, it will become a list (nested within the main list), with the new value as its second item. If the targeted item was already a nested list, the new value will be inserted into that list.

If *container* is specified as a range of items in a list, the `before` option will insert the new value before the first item in the range, and both the `after` and `into` options will insert the new value after the last item in the given range.

Tech Talk

If *expr* yields a list of items rather than a single item, that list is inserted as a unit making a nested list if the `nested` option was specified. If `item by item` was specified each item from that list is inserted individually into the destination list. If neither option is specified, the type of insertion is controlled by the setting of the local property `the listInsertionMode` if it has been set to either “nested” or “item by item”, or by `the defaultListInsertionMode` global property, which is initially set to “item by item”.

The `insert` command never removes or replaces anything in the destination container. The container always becomes a list when something is inserted into it, even if it was empty or undefined to begin with.

◊ asList function

What it Does

The `asList` function (called by the `as a list` operator) returns the value of its parameter converted to a list.

Examples

```
put file "scores" as a list into testScores
```

Tech Talk

Syntax: `{the} asList of` *factor*
`asList(`*expr*`)`

When the `asList` function is called with an object (property list) as a parameter, it first checks whether the object has an "asList" property. If so, its value is returned. If not, and the object has an "asListExpression" property, the value of that property is evaluated as an expression (equivalent to calling the `value()` function) to obtain the list value. If the object has neither of these properties, an "asList" function message is sent exclusively to the object and its helpers, and its return value is used.

If the target is not an object (or doesn't have an "asList" or "asListExpression" property or an "asList" function handler) and it is not already a list, the target's string value is evaluated as an expression (equivalent to calling the `value()` function) to obtain the list value.

See Also: the discussion of “Conversion of Values” and the `as` operator in Expressions.

Property Lists

Property lists are similar to lists – both are collections of values. The fundamental difference is that a list is a simple sequence of values, while each value in a property list is identified by a name or label, called its "key".

```
put (5,12) into myList
put (x:5, y:12) into myPropList
```

Another difference, which is less obvious but can be even more significant, is that a property list is actually a simple form of a SenseTalk "object", which means it can have behaviors as well as properties. Objects are described in detail in a later section. The remainder of this section deals with property lists primarily as data containers.

Creating Property Lists

You can create a property list by simply listing the keys and values for each of its properties, like this:

```
put (name:"Elizabeth", age:14) into daughter
```

Each property's key precedes its associated value, separated by a colon, with key/value pairs separated by commas, and the entire list enclosed in parentheses.

An empty property list can be specified using a colon in parentheses `(:)` or the phrase `empty object` or `empty property list`:

```
put (:) into newPList
```

A property list (because it is an object) can be helped by other objects (such as objects defined in text files — helpers are described in detail in the next section):

```
put (name:"Hank", age:47) helped by parent,actor into candidate
```

A property list can also use curly braces { } in place of parentheses, which allows the list to span multiple lines without the need to use line continuation ("\") characters:

```
set detective to { name: "Sherlock Holmes",
   address: "221 B Baker Street, London",
   remarks: "Enjoys playing violin, solving mysteries"
}
```

When a property list occurs as the last item of a regular list, the parentheses can be omitted around the property list. This yields a syntax that has the appearance of a list/property list hybrid, but the named properties must come at the end. The following two lines are equivalent:

```
put (5, 12, (color:"Green", size:42))
put (5, 12, color:"Green", size:42)
```

Duplicate Keys and the DuplicatePropertyKeyMode Global Property

If the same key is repeated within a property list, `the duplicatePropertyKeyMode` global property controls how it is handled. If this property is set to "error" (the default), an exception is thrown; if it is "first" or "last", only the first (or last) value given for that key will be used, and any duplicate keys will simply be ignored; or if it is "list", then all of the values supplied for that key will be accepted and combined into a list.

Property List Contents

Just like lists, property lists can include any type of value, including lists and other property lists. An expression can be used in assigning a value:

```
put ( label:"12592-A", area:pi*r^2, dimensions:(9,8,42), color:(top:"Red",
sides:"Black") ) into currentPart
```

Accessing the Properties in a Property List

The properties in a property list can be accessed in several different ways, as shown in this example:

```
put (firstName:"Joseph", age:50) into joe
put property age of joe -- 50
put the age of joe -- 50
put joe's firstName -- Joseph
put joe.firstName -- Joseph
```

The last three examples above (using dot (`.`) and apostrophe-S (`'s`) as well as the simple "`of`" syntax) all invoke a special SenseTalk mechanism that can be used either to access the requested property or to call a function on an object. The "property ... of" syntax will always access a property (never call a function). To use the value of a variable (or an expression) as the property name with any of these forms, the variable name must be enclosed in parentheses, otherwise it will be treated as the literal name of the property (or function) being accessed.

When accessing a property of a simple property list, the effect of all four syntaxes is the same – they will access a specific property. Property names are never case-sensitive.

Undefined Properties

Accessing a property that hasn't been set in a property list will simply return empty, unless `the strictProperties` global property has been set to true. When `the strictProperties` is true, any attempt to access a property that has not been previously set will throw an exception.

```
put (name:"Whiskers") into myCat
put the color of myCat -- shows "" (if the strictProperties is false)
```

Accessing Multiple Properties as a List

You can access multiple properties at once, as a list of values, by specifying a list of keys:

```
set pitch to (a:440, b:493.88, c:523.25, d:587.33, e:659.26)
put pitch's (e,d,c,d,a) -- (659.26,587.33,523.25,587.33,440)
```

Setting or Changing Property Values

Every property in a property list is a container, so its value can be changed using any command that alters the contents of a container (such as `put`, `add`, `sort`, etc.):

```
put (firstName:"Joseph", age:50) into joe
put "ine" after joe's firstName -- changes "Joseph" to "Josephine"
subtract 7 from the age of joe
```

You can also set multiple properties at once, by specifying a list of keys:

```
set joe's (height, weight) to ("5ft. 9in.", 143)
```

Adding New Properties

New properties can be added to a property list by simply storing something into them. Nested property lists will be created as needed:

```
put "Olivier" into joe.lastName
put "Tabby" into the name of joe's cat
```

You can add all of the properties from one property list into another using the `adding properties` operator or `add properties` command. Existing properties are left unchanged by these operations:

```
put plainCar adding (color:"Red", top:"Blue") into colorfulCar
add properties of smartPerson to newHireRequirements
```

To override any existing properties with new values, use the `replacing properties` operator or `replace properties` command instead:

```
put colorfulCar replacing (color:"Aqua") into customCar
replace property (phone:newPhoneNumber) in clientRecord
```

Removing Properties

An existing property can be removed from a property list by deleting it using the `delete` command:

```
delete joe.hobbies
```

Multiple properties can be removed at once using the `removing properties` operator or `remove properties` command:

```
put coloredCar removing properties ("color", "top") into plainCar
remove properties of executive from newHireRequirements
```

Removing a property which doesn't exist in the source property list has no effect.

Counting the Properties in a Property List

The `number` function can be used to find out how many properties are in a property list, by asking for the number of keys, values, or properties:

```
put the number of keys in (x:5,y:12) -- 2
put the number of values in phoneBook into phoneCount
```

The `number of occurrences` function counts how many times a particular value or key is present:

```
put the number of occurrences of 5 among the values in (x:5,y:12) -- 1
```

Listing Property Names – the Keys Function

The `keys` function provides an alphabetical list of the names of the properties in a property list:

```
repeat with each item of keys(mike)
   put it & ": " & property (it) of mike
end repeat
```

Listing Property Values – the Values Function

A list of the values of all of the properties in a property list can be obtained by using the `values` function. The values are listed in alphabetical order of their keys:

```
repeat with each item of mike's values
    if it contains "Gold" then put "Found " & it
end repeat
```

Passing one or more property names or lists of property names to the values function will list only the requested property values in the order requested:

```
put mike's values("favoriteColor", "hobby", "favoriteMovie")
```

Iterating Over the Properties in a Property List

To perform actions with each of the values in a property list, you can step through the keys or values using a `repeat with each` loop. This example will set the variable `key` to each of the keys in a property list in turn, and sets `value` to each of the values:

```
set phoneBook to (Mark:"555-1234", John:"555-2345", Eli:"555-3456")
repeat with each key of the keys of phoneBook
    set value to phoneBook.(key)
    put key & ":" && value -- work with
end repeat
```

Checking for a Key or Value in a Property List

The `is among` operator can be used to find out whether a property list contains a particular key or value:

```
put "x" is among the keys of (x:5,y:12) -- true
put 12 is among the values of (x:5,y:12) -- true
```

The `contains` or `is in` operators can also be used in some situations. The meaning of these operators when working with property lists can be defined by the object itself, or configured by setting `the objectContainsItemDefinition` global property (see "Checking ObjectContents" in Objects, Messages, Handlers and Helpers). The default behavior of the `contains` or `is in` operators with a property list, however, is to check for a substring of the property list's text value (as described below).

Converting Property Lists to Text

When a property list is accessed as text (such as when it is displayed by a `put` command), it is converted automatically to a text representation. If the property list includes an `asText` property, the value of that property will be used as the text representation of the object. If not, but it includes an `asTextFormat` property, that value is used as a merge format to produce the text representation (see the `merge` function in Working with Text for a description of the merge format).

If neither `asText` nor `asTextFormat` properties are present, a text representation will be created using, by default, the values of `the propertyListFormat` global property. The text will be enclosed by the values of the propertyListFormat's `prefix` and `suffix` properties (open and close parentheses, by default). The properties will be listed with keys separated from values by the `keySeparator` (a colon by default), and the key/value pairs separated from each other by the `entrySeparator` (a comma). Values are enclosed in quotes. The default text representation of an empty property list is given by the `emptyRepresentation` property (which defaults to "(:)"). You can change the default formatting by setting these values of `the propertyListFormat` global property.

They may be set to any text you like, including `empty`.

To obtain a text representation of all of the keys and values of a property list, the `standardFormat` function can be used to produce a standardized representation of the property list suitable for storing in a file. This text is in a format that will produce the original property list again when that text is evaluated by the `value` function (see the section "Conversion of Values" in Expressions).

Note: Overriding text representations

An object, including a property list with a script or helpers, may override this mechanism entirely if it handles an `asText` message to provide its own text representation. See Objects, Messages, Handlers, and Helpers.

The following example illustrates the use of `the propertyListFormat` global property:

```
set the propertyListFormat to (prefix:"The value of ", \
   keySeparator:" is ", entrySeparator:" and the value of ", \
   suffix:".", emptyRepresentation:"The value is empty.")
put (x:12, y:13) -- The value of x is "12" and the value of y is "13".
```

When a property list is converted to text, each value within the list will also be converted to a text representation. For details on how this is done, see the section "Conversion of Values" in Expressions.

The `split` and `join` commands and functions and `asText` function, described in detail in Working with Text, and the corresponding `split by` and `joined by` operators (described in Expressions) provide ways to explicitly convert text to property lists and vice versa. The `value` function can also produce a property list from a standard text representation (see Expressions).

◊ add properties, replace properties, remove properties, and retain properties commands

What they do

The `add properties` command adds the properties of one property list or object into an existing property list or object if those properties are not already present. The `replace properties` command is similar, but will override existing properties with new values. The `remove properties` command removes the indicated properties from an existing property list or object. The `retain properties` command removes any properties that are not explicitly requested to be retained.

When to use them

The `add properties`, `replace properties`, `remove properties` and `retain properties` commands can be used to change the properties of an existing property list or object. To combine or remove properties without affecting an existing object, use the `adding properties`, `replacing properties`, `removing properties` or `retaining properties` operators instead (see Expressions).

◊ add properties, replace properties, remove properties, and retain properties commands

Examples

```
put (A:1, C:3) into myObj
add properties (B:2, C:99, D:4) to myObj
put myObj -- (A:1, B:2, C:3, D:4)
replace properties (B:"bunny", D:"dog") of myObj
put myObj -- (A:1, B:bunny, C:3, D:dog)
remove property (B:765) from myObj -- (A:1, C:3, D:dog)
remove properties ("B","C") from myObj -- (A:1, D:dog)
retain properties ("C","D","E") of myObj -- (D:dog)
```

Tech Talk

Syntax: **add {the} properties {of}** *additionalPropList* to *sourcePropList*
replace {the} properties {of} *replacementPropList* **[of | in]** *sourcePropList*
remove {the} properties {of} *propertiesToRemove* **from** *sourcePropList*
retain {the} properties {of} *propertiesToRetain* **[of | in]** *sourcePropList*

If a property in *additionalPropList* is already present in *sourcePropList* that property is ignored. If *additionalPropList* is empty, the command does nothing.

If a property in *replacementPropList* is already present in *sourcePropList* that property is replaced with the new value. Other values from *replacementPropList* are added to those in *sourcePropList*. If *replacementPropList* is empty, the command does nothing.

The *propertiesToRemove* may be the name of a single property, a list of property names, or a property list. If a property list is given its values will be ignored but its keys will be used as the list of properties to remove. Trying to remove properties that are not present in *sourcePropList* has no effect.

The *propertiesToRetain* may be the name of a single property, a list of property names, or a property list. If a property list is given its values will be ignored but its keys will be used as the list of properties to retain. Trying to retain properties that are not present in *sourcePropList* has no effect.

◊ asObject function

What it Does

The `asObject` function (often called by the `as a property list` or `as an object` operator) returns the value of its parameter converted to an object (property list).

Examples

```
put file "cust2497" as a property list into customer
```

Tech Talk

Syntax: **{the} asObject of** *factor*
asObject(*expr***)**

When the `asObject` function is called with a tree as a parameter, it converts the tree to a property list representation, according to the setting of `the treeFormat`'s `useStandardFormat` property, otherwise the parameter's string value is evaluated as an expression (equivalent to calling the `value()` function) to obtain the property list value.

See Also: the discussion of "Conversion of Values", the `as` operator in Expressions, and Working with Trees and XML for everything about trees.

Objects and Messages

The material covered up to this point is enough to allow you to write scripts and accomplish many tasks. To fully understand SenseTalk and leverage its power, there are a few more concepts to master: messages and handlers, and objects and their helpers.

Objects, Messages, Handlers and Helpers – introduces the powerful concept of Objects, and describes how to create them, access their properties, and send messages to them. Object Helpers, which allow objects to "help" others, are also described.

Working with Messages – describes commands and constructs that deal with sending and handling messages.

Ranges, Iterators, and Each Expressions

Ranges

Defining a Range

A range in SenseTalk can be used to indicate a range of values. A range is specified by giving a start value and an end value, using either a double-dot operator (two periods in a row) or the word to between the two values:

```
put 1 to 100 into firstHundred
set validRange to 100..200
```

A range may optionally include a step value, using the word by or step or step by (the use of step values is described in "Using a Range to Generate a List", below):

```
put 1 to 99 step by 2 into oddNumbers
set evenNumbers to 0..100 by 2
```

Tech Talk

Syntax: {from} startValue [to | ..] endValue { [{step{ping}} {down} by | step] stepValue }

Simple Uses of Ranges

In its simplest uses, a range merely specifies two values. You can access the start and end values directly, as properties of the range:

```
set myRange to 10 .. 20 --  (10 to 20)
put myRange.start -- 10
put myRange's end -- 20
```

The is within operator can be used to test whether another value falls within the range:

```
put 13 is within myRange -- true
put 18.975 is within myRange -- true
put 9.2 is within myRange -- false
```

A range is inclusive -- that is, it includes both the start and end values. So any value between 10 and 20, including the values 10 and 20, would be within our example range:

```
put 10 is within myRange -- true
put 20 is within myRange -- true
```

The `is a range` operator can be used to test whether a value is a range:

```
put myRange is a range -- true
```

Using a Range to Generate a List

A range can also be used as a sequence generator to automatically generate a sequence or list of values beginning with the start value of the range and continuing until its end value. The simplest way to cause a range to generate its values is merely to use the range in a context where a list is expected:

```
put item 5 of 10 to 20 -- 14
put the last 3 items of 100..200 by 2 -- (196,198,200)
```

If a step by increment is specified for the range, its absolute (positive) value is used when generating a sequence as the increment between sequential items in the list. If a step by increment isn't specified for the range, a value of 1 is assumed. The increment value of a range can be accessed or changed using the step property of the range:

```
set the step of myRange to 3
```

To generate all of the values in the range at once, use the as list operator to request it as a list:

```
put 10 .. 20 as a list -- (10,11,12,13,14,15,16,17,18,19,20)
put 10 to 20 by 2 as list -- (10,12,14,16,18,20)
```

A range's endpoints can be specified in either order, for example as a range from 10 to 20 or from 20 to 10. For purposes of the is within operator the two are equivalent. The difference is the order in which values will be generated if the range is treated as a list or used as an iterator.

```
put 20 .. 10 by 2 as a list -- (20,18,16,14,12,10)
```

Using the contains and is in Operators with Ranges

The contains and is in operators, when used with a range, will treat the range as a list. In this case, a range only contains a value if that value is one of the distinct values generated by the range. Note that the is in and is within operators behave quite differently in this way:

```
set range to 10..20
put range contains 13 -- true
put range contains 13.5 -- false
put 13.5 is within range -- true (13.5 is between 10 and 20)
put 13.5 is in range -- false (it's not one of the values
                          10,11,12,13,14,15,16,17,18,19, or 20)
```

Date/Time Ranges

A date or time range works similarly to a numeric range. Both the start and end values must be valid as date or time values for it to be recognized as a time range:

```
set Q1 to "Jan 1" to "Mar 31"
put "Feb 3" is in Q1 -- true
```

When a time range is created without explicitly giving a step increment, the step value will be assumed to be 1 day, 1 minute, or 1 second, depending on the start and end values. If the start and end values are a day apart or more, a step value of 1 day will be used. If they are within 24 hours of each other but more than a minute apart a step value of 1 minute will be used, and if the range is 60 seconds or less a step value of 1 second will be used.

For more natural readability of date/time ranges, the step value may be specified as a plural time unit like "days" instead of "1 day" or "minutes" instead of "1 minute":

```
set MondaysIn2009 to "2009-01-05" to "2009-12-31" by weeks
```

When a date/time range is accessed as a list, each value produced will be a date/time value using the same format

as the start value.

```
put item 33 of "Jan 1" to "March 31" -- "Feb 2"
put item 33 of "January 1" to "Mar 31" -- "February 2"
```

Character Ranges

A range can also specify a range of Unicode characters. One common use of character ranges is to specify the alphabet:

```
set alphabet to "a".."z"
set upperAndLowerLetters to "A".."Z" &&& "a".."z"
put "z" to "t" as list -- ("z","y","x","w","v","u","t")
```

Using a Range as a Chunk Index

A range can be used to select certain characters, words, items, etc. of a value:

```
put chars 1..5 of "abcdefgh" -- "abcde"
put chars 1..5 by 2 of "abcdefgh" -- "ace"
put items 2..6 by 2 of "a,b,c,d,e,f,g,h" -- "b,d,f"
put items 2..6 by 2 of (a,b,c,d,e,f,g,h) -- (b,d,f)
put chars 8..1 of "abcdefgh" -- "hgfedcba"
put chars 8..1 as list of "abcdefgh" -- (h,g,f,e,d,c,b,a)
```

Iterators

Iteration is the process of stepping through a sequence of values. An iterator is an entity that provides such a sequence of values, one at a time. In SenseTalk, lists and ranges can both be used as iterators. In addition, you can create your own custom iterator objects. There are several ways that you can step through each of the values supplied by an iterator and work with each one in turn.

Iterating Using Repeat With Each

When you need to work with each value supplied by an iterator and perform some actions with that value, use a repeat with each loop. For example, here's a loop that uses each value from a list:

```
repeat with each color in ("red", "orange", "yellow", "green")
   put "I think " & color & " is a very pretty color."
end repeat
```

To iterate over all of the values in a range, use repeat with each just as you would with a list:

```
repeat with each letter in "A" to "G"
   repeat with each digit in 1 to 3
        put letter & digit -- A1, A2, A3, B1, B2, ...
   end repeat
end repeat
```

Iterating Using Each Expressions

When you need to process each value (or a selected set of values) from a list or other iterator to produce a list of result values, an each expression can be very effective. The result of an each expression is always a list. See Each Expressions below.

Iterating Using NextValue

The nextValue() function will return the next sequential value from an iterator. When the sequence is exhausted and no more values are available from that iterator, the special constant value end is returned.

```
put 100 to 200 by 50 into range
put range's nextValue -- 100
put range's nextValue -- 150
put range's nextValue -- 200
put range's nextValue --  ⓔ ⓝ ⓓ
```

Modifying Iteration Using CurrentIndex

Both lists and ranges have a currentIndex property. When a list or range is first created, the value of the currentIndex is zero. It is also reset to zero at the start of a repeat with each loop or an each expression for that iterator.

During iteration, the currentIndex property is incremented each time a value is retrieved from the iterator. The iterator's currentIndex can be accessed to determine the item number of the current value in the source list or range.

```
set foo to (a,b,c,d,e)
put foo's currentIndex & each item of foo -- (1a,2b,3c,4d,5e)
```

What's more, the currentIndex can be changed to alter the course of iteration. For example, this repeat loop will only display the odd numbers in the range:

```
set myRange to 1..100
repeat with each item of myRange
    put it
    add 1 to myRange.currentIndex -- skips the next item
end repeat
```

Changing a List During Iteration

While iterating over the items in a list using repeat with each, an each expression, or the nextValue function, SenseTalk uses the currentIndex property of the list to keep track of the iteration. Whenever items are inserted, deleted, or replaced in a list, the currentIndex property is adjusted appropriately to reflect the change. So it is possible to add, remove, or replace items in a list during iteration without causing problems.

Custom Iterators

Lists and ranges are the most common sources used as iterators. You can also create a custom iterator by making a script or object with a nextValue handler that returns the next value in a sequence. The values returned can be anything, and the implementation may be as simple as returning the nextValue from a list that is a property of the object, or may be the next result calculated from some internal state maintained by the object. Iterators of this type are sometimes called "generators" because they generate values one at a time as needed.

To be used as an iterator, there are few rules an object must follow. First, it must have an objectType of "iterator". This tells SenseTalk to call the object's nextValue handler to obtain each value for a repeat with each loop or an each expression. Second, of course, it must have a nextValue handler to supply each value in the sequence. The nextValue handler should return the value end when the end of the sequence has been reached and no more values are available. If it doesn't do this, a repeat loop should include an exit repeat statement to break out of the loop at some point or it will continue forever. An each expression should not be used with an iterator that doesn't eventually return end.

In addition to a nextValue handler, a custom iterator may optionally also have a startIteration handler. If present, this

handler will be called before nextValue is called by either a repeat with each loop or an each expression to allow the iterator to set up any initial conditions, or to reset its state following any earlier iteration.

Here is an example of a custom iterator that generates values from the Fibonacci sequence, in which each value is the sum of the two preceding values (note that although the Fibonacci sequence is infinite, this iterator only returns the values up to ten thousand, allowing it to be used with an each expression):

```
set FibonacciGen to {a:0, b:1, objectType:"iterator", script:{{
to handle nextValue
   if my b is more than ten thousand then return end
   set (my a, my b) to (my b, my a + my b)
   return my a
end nextValue
}} }
set fib to a new FibonacciGen
repeat 10
   insert fib's nextValue into firstTen
end repeat

put firstTen -- (1,1,2,3,5,8,13,21,34,55)
put fib. nextValue -- 89
put each item of fib -- (144,233,377,610,987,1597,2584,4181,6765)
```

Passing an Iterator As a Parameter

An iterator can be passed as a parameter to another script or handler. It is passed with its current state intact, including the value of the currentIndex property of a list or range, and any properties of a custom iterator. So if the local script has been using values one at a time from the iterator (by using nextValue), the called script can continue using later values in the sequence.

Keep in mind that SenseTalk ordinarily makes copies of any parameter values, so the called script will receive its own copy of the iterator. To share the iterator so that both the local and the called scripts will be using a single sequence of values from the iterator, pass the iterator by reference. For example:

```
set takeANumber to 1..20
put takeANumber's nextValue -- 1
put takeANumber's nextValue -- 2
scriptThatTakesTwoNumbers @takeANumber -- takes 3 and 4
put takeANumber's nextValue -- 5
```

Without the "@" to pass takeANumber by reference, the final call to takeANumber's nextValue would return 3 instead of 5. By passing a reference here, both scripts share the same sequence of values rather than splitting off a copy to the called script.

The `is an iterator` operator can be used to test whether a value (such as a parameter that was received) can be iterated:

```
if param(1) is not an iterator then throw "An Iterator Is Required"
```

Restarting Iteration

To restart iteration over a range or list beginning with the first value again, set its currentIndex to zero. The next time you access its nextValue, the first value in the range or list will be returned.

```
set the currentIndex of sequence to zero -- reset iteration
```

Some custom iterators may provide a mechanism for starting iteration again, and some may not. A custom iterator that is designed to be reusable by repeat with each loops and each expressions should implement a startIteration handler that can be called to reset it. Other custom iterators may implement a currentIndex property that can be reset just as you would for a list or range. But neither of those is required in a basic iterator.

Assigned List Values

Lists in SenseTalk grow dynamically as items are inserted or values are assigned to items beyond the previous extent of the list. This allows values to be assigned to items that may be widely dispersed in a list, as in this example:

```
set list to ("a","b")
put "g" into item 7 of list
put "z" into item 26 of list
put list -- (a,b,,,,,g,,,,,,,,,,,,,,,,,,z)
```

The list in this example has 26 items, although only 4 of them have assigned values. The other items are empty. Iterating over the items of this list will access all 26 items. In some cases it may be useful to be able to access only those list items that have explicitly been assigned a value. The nextAssignedValue function does this. It works like the nextValue function, but instead of simply incrementing the currentIndex property by 1 and returning the value at that index, it advances the currentIndex to the next item in the list that has an assigned value and returns that value. So, continuing the example above:

```
put list's nextAssignedValue -- a
put list's currentIndex -- 1
put list's nextAssignedValue -- b
put list's currentIndex -- 2
put list's nextAssignedValue -- g
put list's currentIndex -- 7
put list's nextAssignedValue -- z
put list's currentIndex -- 26
put list's nextAssignedValue --  ⓔ ⓝ ⓓ
```

Since both the nextValue and nextAssignedValue functions advance the currentIndex property of the list, you can mix calls to the two functions to obtain either the next sequential value of the list (whether assigned or not) or the next assigned value of the list after the current index as needed.

Note: Empty values in a list

An empty value is not the same as an unassigned value. If an empty value is stored into an item of a list, that empty value will be returned by the `nextAssignedValue` function.

Each Expressions

An each expression is a powerful mechanism for collecting or generating information for each value or a selected subset of the values provided by a list, range, or custom iterator. A single each expression can do a lot of work, in a very simple and readable way.

Here are some examples:

```
put each item of 1..100 where each is a multiple of 7
    -- (7,14,21,28,35,42,49,56,63,70,77,84,91,98)
put each item of 1 to 100 where the square root of each is an integer
    -- (1,4,9,16,25,36,49,64,81,100)
```

If numbers don't inspire you, perhaps some examples with words will be more interesting:

```
put "Mary Mary quite contrary how does your garden grow" into rhyme
put each word of rhyme
    -- (Mary,Mary,quite,contrary,how,does,your,garden,grow)
put each word of rhyme where the length of each is 4
    -- (Mary,Mary,does,your,grow)
put each word of rhyme where each ends with "ary"
    -- (Mary,Mary,contrary)
put the length of each word of rhyme where each ends with "ary"
    -- (4,4,8)
```

Facts About Each

The result of an each expression is always a list. At its simplest, an each expression merely accesses each character, word, line, text item, or list item from a source value and creates a list containing those values.

A "where" clause lets you select items that meet some condition. Within a where clause, the variable "each" refers to each value from the source in turn. Only values for which the where clause evaluates to true will be included in the resulting list.

An each expression can be used with any chunk type (list items, text items, words, lines, characters).

Tech Talk

Syntax: `each chunk of` *sourceValue* `{where` *conditional*`}`
`each chunk of` *sourceValue* `{(where` *conditional*`)}`

The *conditional* expression is usually an expression involving the special variable `each`, which is set to each chunk of *sourceValue* in order to select which values to include in the resulting list. The where clause may be enclosed in parentheses for readability or separation from other parts of a statement.

Each Expression Within a Larger Expression

When an each expression is embedded within a larger expression, other operators outside of the each expression itself will be applied to each of the values in the list generated by the each expression rather than to the list as a whole. For example:

```
put the length of each word of "four score and twenty"
    -- (4,5,3,6)
```

Here, the each expression itself – each word of "four score and twenty" – generates the list ("four", "score", "and", "twenty"). Rather than calling the length function on that list, though, the fact that it was generated by an each

expression causes the length function to be called on each item in that list, resulting in a list of the individual word lengths.

Limiting the Scope of an Each Expression

The fact that an each expression spreads its influence to the enclosing expression, causing the entire expression to apply to each item in the resulting list adds tremendously to their power. However, there are times when it is important to be able to limit this effect in order to get the desired result. For example, to count the number of 3-letter words in some text, you might try this:

```
set text to "Paris in the the Spring"
put the number of items in each word of text where the length of each is 3
```

However, rather than returning the number 2, this will result in the list (1,1). This is because the each expression returns the list ("the","the") and then the "number of items in" operator is applied to each item in this list, resulting in two 1's (since the word "the" is a single item).

To limit the scope of the each expression's influence, enclose it in parentheses:

```
put the number of items in (each word of text where the length of each is 3)
```

This will cause "the number of items in" to be applied to the result of the each expression as a whole list, giving the desired result – the number 2 – which is the number of 3-letter words in the text.

Expanding Scope with For Each Expressions

The fact that parentheses limit the scope of an `each` expression restricts the type of expressions that can be used. For example you can't call a function using parentheses and have it apply to each value, because the parentheses limit the scope of the each. So the function would be called for the resulting list as a whole rather than for each value. A `for each` expression can be used to overcome this problem.

```
put round(each,2) for each item of 2.2 to 9.1 by 1.376 --
(2.2,3.58,4.95,6.33,7.7,9.08)
```

Here, an expression is given that uses the special **`each`** variable, followed by an each expression beginning with the words "for each". The expression that comes before "for each" has no special restrictions, and can use the each variable more than once. This gives complete flexibility in the kinds of operations that can be performed.

```
set text to "an ancient anteater sat with my antiquarian aunt"
put "Longest words in text:"
get each & " has " & length(each) & " letters" for each word of text \
        where length(each) > 4
put it joined by return
```

which displays:

```
Longest words in text:
ancient has 7 letters
anteater has 8 letters
antiquarian has 11 letters
```

Tech Talk

Syntax: *resultExpression* `for each` **`chunk`** `of` *sourceValue* `{where` *conditional*`}`
resultExpression `(for each` **`chunk`** `of` *sourceValue* `{where` *conditional*`})`

The *conditional* expression is usually an expression involving the special variable `each`, which is set to each chunk of *sourceValue* in order to select which values to include in the resulting list.

ResultExpression is usually also an expression using the special variable `each`. It is evaluated for each of the values produced by the each expression to produce the final list of values for the overall expression. The for each clause may be enclosed in parentheses to enhance readability or provide separation from other parts of a statement.

Nested Each Expressions

An each expression may be enclosed in another each expression. While this can sometimes become confusing, it may be useful when working with nested lists or other nested structures. For example, to obtain the lengths of each word – not for a single phrase, but for each of a series of phrases – you could do it like this:

```
set phrases to {{
universal truth
magic is in the eye of the beholder
all is fair in love and war
}}
put the length of each word of phrases
put the length of each word of each line of phrases
```

The first put statement above gets all of the word lengths in a single list: (9,5,5,2,2,3,3,2,3,8,3,2,4,2,4,3,3). The second put statement, by using nested each expressions results in a nested list: ((9,5),(5,2,2,3,3,2,3,8),(3,2,4,2,4,3,3)).

Where clauses can be used in nested each expressions, but care must be taken to match each where to the nearest each. For example, if we only wanted to see the word lengths for words that are longer than 3 characters, we could do this:

```
put the length of each word of each line of phrases \
        where true where length of each > 3
```

Here, the first where clause, "where true" is needed even though it simply selects every entry. This is because without it, the other where clause would be applied to each line, but we care about the lengths of the individual words, not the lengths of the lines. The statement shown above produces the correct result: ((9,5),(5,8),(4,4)).

Combined Each Expressions

Two each expressions may be combined as a part of a larger expression to multiply the effect, producing a nested list of results:

```
put each item of "A".."C" & each item of 1..4
        -- ((A1,A2,A3,A4),(B1,B2,B3,B4),(C1,C2,C3,C4))
```

```
put each item of 1..10 times each item of 1..10 into timesTable
```

RepeatIndex() in each expressions

When the repeatIndex() function is used within the context of an each expression, it evaluates to the number of the current item from the source value. This can sometimes be useful. For instance, the following example will associate a different number with each character of a string in producing a list:

```
put each char of "abcdefg" & repeatIndex() -- (a1,b2,c3,d4,e5,f6,g7)
```

This also means that if you need the repeatIndex() value from an enclosing loop inside an each expression, you will need to first assign the repeatIndex() value to a variable.

Objects and Messages

The material covered up to this point is enough to allow you to write scripts and accomplish many tasks. To fully understand SenseTalk and leverage its power, there are a few more concepts to master: messages and handlers, and objects and their helpers.

Objects, Messages, Handlers and Helpers **–** introduces the powerful concept of Objects, and describes how to create them, access their properties, and send messages to them. Object Helpers, which allow objects to "help" others, are also described.

Working with Messages **–** describes the nuts and bolts of the commands and constructs that deal with sending and handling messages.

Objects, Messages, Handlers and Helpers

This section introduces one of the most powerful concepts in SenseTalk: objects. The SenseTalk object model is described, along with the related topics of how messages are used by objects to communicate with other objects, how messages are handled by an object, and how helper objects can be used to help other objects, thereby leveraging their functionality across many objects.

Objects

SenseTalk was designed to enable people to create their own software, without years of study to master the intricate and arcane details of programming. The earlier sections of this manual focused on SenseTalk scripts primarily as simple lists of statements that tell the computer what you want it to do. For many people that basic approach to scripting will be sufficient for their needs.

For those who care to learn just a little bit more, though, SenseTalk's object model offers a much richer environment, without adding greatly to its complexity.

This section gives an overview of the modular structure of SenseTalk, and introduces the key concepts and terminology that you'll want to be familiar with as you proceed with learning how to write your own "object-oriented" SenseTalk software.

Setting the Stage

SenseTalk is a "scripting" language. You create SenseTalk software by writing scripts that describe what different elements of your system will do. The usual way to do this in SenseTalk is to create a number of different "objects" which will be the actors in your system. Each object has its own script that tells what it does in the system. By taking this modular approach, each script can be relatively short and self-contained, which makes even complex systems relatively easy to work with.

The objects in your system will interact by sending and receiving messages. An object responds to a message if it has a handler for that message. A script consists of a series of handlers, one for each message that the object will respond to. Any other messages will be ignored by the object.

One SenseTalk object can help another. When you have a number of objects that need to have similar behaviors or abilities, you can create a helper object which embodies those shared behaviors and abilities. Each of the other objects can then be helped by the helper object, eliminating the need for them to each have those behaviors defined in their own scripts. This saves you a lot of time and effort. It also makes it easy to update a behavior by simply changing the script of the helper object. All of the other objects will then "inherit" the changed behavior.

It should be noted, for those who may be familiar with other object oriented languages, that SenseTalk is somewhat different. Most other languages define "classes" of objects and generally only implement behavior at the class level, for all objects in that class. In SenseTalk, each individual object has its own script, so it can have its own unique behavior.

SenseTalk has no classes, but its helpers provide a similarly rich set of capabilities by allowing objects to use (or inherit) functionality provided by any number of other objects. This all-object (classless) approach is both simpler and more versatile than class-based systems.

Objects Defined

SenseTalk is an Object-Oriented Language. SenseTalk scripts describe the behavior of objects. Depending on the host environment, an object may be something visible which can be directly manipulated and interacted with, such as

a button or an image on the screen, or it may be more abstract, such as an object representing a bank account or an exercise schedule.

Whether an object represents something visual or something that is purely conceptual, all objects in the SenseTalk world share certain characteristics. Any object can have attributes, called properties, which store information that is important to that object. An object may also have behaviors, which are defined by the object's script and define what the object can do.

In the case of a picture shown on a computer screen, its properties might include such things as its height and width as well as the colors of all the dots that form the image itself. An object representing a person's contact information would include properties such as their name, telephone number, and mailing address.

An object's behaviors as defined by its script might include such things as turning an image upside down, or sending an email message to a person on a mailing list. Other "behaviors" of an object may be more passive, such as providing information about the object which is not directly represented in its properties. For example, an object representing a person might include a property that stores their birth date. The object's script could provide the person's age by accessing the current date and calculating the number of years that have passed since their birth date.

Property Lists

Property lists were described in an earlier section as a collection of values identified by their keys. It was mentioned there that a property list is actually a simple object.

A property list is an object that's mainly properties. Behaviors can be added to a property list, though, either by assigning helpers, or by setting the script property to a valid script. Helpers are described later in this section. To set the script of an object, simply store the text of one or more handlers into the "script" property of that object:

```
put (width:7, length:12) into myRect
set the script of myRect to {{
function area
         return my width * my length
end area
}}
put myRect's area -- 84
```

Script Files

In many SenseTalk scripting environments, each script is stored in a text file on the disk. In these environments, each such file is a SenseTalk object, and the contents of the file is the object's script.

A script consists of a sequence of SenseTalk commands (also called *statements*) which define the behaviors of that object. Here is the simplest and shortest complete SenseTalk script you are likely to encounter:

```
put "Hello, World!"
```

A script file is an object that's mainly behaviors. The script above constitutes a very simple object, with a single behavior that will be invoked when the script is run: displaying the words "Hello, World!".

In addition to behaviors, a script file may also include property declarations if desired. When an object is created by loading a script file from disk, any property declarations in that script will define initial values for the object's properties. A `properties` declaration takes the form shown in this example:

```
Properties
   name: "Charlie Brown",
   birthDate:"May 14, 1942",
   hairColor: brown,
   numberOfSiblings: 1,
```

```
    helpers: ("Linus", "Lucy")
end properties
```

This property declaration defines five properties of the object, and assigns them initial values. Values that contain spaces and special characters must be enclosed in quotation marks, as shown here for the name and birthDate properties. Simple values do not require quotes, but may be quoted if you prefer. Some properties may be assigned a list of values, as shown here for the helpers property.

Script Folders

A folder can also serve as a SenseTalk object. When treating a folder as an object, each script file within the folder is treated as a handler of that object. If a script with the special name "_initialHandler_" exists in that folder, it will be loaded and used as the folder object's initial handler, and any properties defined in that script will be used as the initial property values of the folder object.

Using Objects

Creating Property Lists

A simple property list (an object with properties but no handlers) can be created by simply listing its properties and values in the format shown here:

```
put (x:44, y:108, z:-19) into point2
```

Creating Simple Objects

An object can be helped by other objects (helpers are described in detail later in this section):

```
put (name:"Hank", age:47) helped by parent,actor into person
```

Creating Initialized Objects

Fully-initialized objects can be created with a `new object` expression:

```
put new object with (width:14, length:9) into dimensions
```

When an object is created using `new object` as shown in the example here, the result is essentially the same as the simple objects shown earlier. The only difference is that an “initialize” message is sent to the newly-created object. Of course, it won’t have a handler to respond to that message, unless it has a helper with such a handler. You can create the object with one or more helpers, like this:

```
put new object with (partNum:1234) helped by Part into aPart
```

Creating with Prototype Objects

A more common way to use a `new object` expression is to specify a “prototype object”, as shown in the following example:

```
set child to be a new Person with (name:"Penny", age:6)
```

In this example, Person is a prototype object. Prototype objects (if they want) can control exactly how the new object is constructed (the details of how this works are given later in this section). In the usual case, however, the new object will have the properties given in the script plus copies of any other properties from the prototype, and will be helped by the prototype object. So, unless the Person object overrides the usual behavior, the statement above would be equivalent to this:

set child to a new object with ((name:"Penny", age:6) adding properties of (object Person)) helped by Person

Accessing Object Properties

An object's properties can be accessed in several different ways, using a dot (`.`) or apostrophe-S (`'s`) and the property name after the object, or the property name followed by "`of`" before the object:

```
put (make:"Yamaha", model:"U3", finish:"Walnut") into piano
put "My piano is a " & piano.make && piano's model
put "It has a pretty " & finish of piano & " finish"
```

Object properties are containers. Properties can be added, deleted, or their values changed, as described in detail for property lists in Lists and Property Lists.

Using "Me" and "My" to Access an Object's Own Properties

It is very common for an object to need to access its own properties from within its script. Rather than referring to itself by name, it can do this using the terms "`me`" and "`my`":

```
put the age of me
if my name begins with "S" then return "Smilin' " & my name
```

Undefined properties and the StrictProperties global property

Ordinarily, if you access a non-existent property of an object, SenseTalk will simply return `empty` as the value of that property. Occasionally, this may lead to trouble, such as if you inadvertently misspell the name of a property. To help with debugging your script in such cases, or if you simply prefer a more rigorous approach, you may set the `strictProperties` global property to true. When this property is set, any attempt to access a property of an object that has not been previously set will throw an exception.

```
put a new object into emptyObj
put (property abc of emptyObj) is empty -- displays 'true'
set the strictProperties to true
put (property abc of emptyObj) is empty -- throws an exception
```

Using "Object" to Ensure Object Access

In most contexts, SenseTalk can recognize when an object is needed and will treat a string value as the name of a script object. However, there are times when the meaning may be ambiguous. In these situations, the word "object" can be used to indicate that a value should be treated as an object name:

```
put "Person"'s greeting -- since "Person" is text, not an object, the greeting
function will be called
put (object "Person")'s greeting -- treat "Person" as an object name, and
access it's greeting property
```

Messages

Objects “do things” only when they receive a “message”. An object can send messages to itself, or they may be sent to it by another object, or by the environment in which the script resides. For example, when a script is invoked from the command line, it is sent a message based on the name of the script, which causes the script to run.

In the next section, Handlers, we will look at how messages are received and handled by an object. To begin with, though, all you need to know is that when a message is sent to an object, it can either receive and handle that message, or it can ignore it. So now, let’s take a look at messages themselves, and how they are sent.

Sending Messages

When a script is running, it sends messages constantly as it executes its commands. A message is always a single word, and the simplest message-sending statement is a single-word command, which is the name of the message to be sent. For example, the second line of this script sends the message “greetTheUser”:

```
put "Hello, World!"
greetTheUser
put farewellMessage()
```

In this script, the `put` commands on the first and third lines also send messages. In fact, with the exception of a few fundamental SenseTalk control structures, *almost every command in a script sends a message*. Function calls, such as the call to the “farewellMessage()” function above, also send messages.

Command Messages and Function Messages

SenseTalk actually sends two different types of messages. Command messages are sent by commands, such as “greetTheUser” in the example above. Function messages are sent by function calls, such as “farewellMessage” in the example. The two message types are identical except for the type of handler that will receive them (see the section on Handlers below).

Where are Messages Sent?

We said earlier that messages are used to allow objects to communicate with one another, so you might be wondering where the “greetTheUser” and “put” messages above are being sent. What object will receive these messages? The answer is very simple, though perhaps somewhat surprising: the object containing these commands will send these messages to itself!

While at first it may not appear to be terribly useful for an object to send messages to itself, in practice any message which is not handled by the object is passed along for possible handling by other objects or ultimately by one of SenseTalk’s built-in commands or functions.

Messages can also be sent directly to some other particular object, if desired. This is done using the `send` or `run` commands or the square-bracket messaging syntax (described in the next section), or by using one of the integrated property and function access calls (described more fully later in this section):

```
get investor's balance("Checking") -- sends "balance" to investor
add paycheck.netIncome() to it -- sends "netIncome" message to paycheck
```

Parameters and Results

While each message is identified by a single word – the message name – it can also include additional information in the form of “parameters”. The command `put “Hello, World!”` sends the string “Hello, World!” as a single

parameter along with the “put “message. To send more than one parameter, just list them with commas in between:

```
WaitFor 15,"BigBlueButton"
```

To include parameters with a function call message, list them inside the parentheses after the function name:

```
put roundToNearest(salary/12, 0.01) into monthlyPayTothePenny
```

As a convenience, when passing a single property list as a parameter, it is not necessary to enclose the property list in parentheses. This gives the effect of passing parameters by name:

```
addToInventory item:"candles", quantity:6, color:"Blue"
set article to fetchClipping(author:"Lewis", title:"Lost")
```

The parameters are actually passed as a single property list when one of these forms is used.

Named and unnamed parameters can be combined in a single call, provided that the named parameters come last. The named parameters are sent as a property list which is passed as the last parameter:

```
addAtPosition 23, item:"paper", quantity:500, color:"White"
```

A list of values can be passed as individual parameters, by specifying `as parameters` after the list:

```
calculateVariance bigListOfNumbers as parameters
```

This can be especially useful for passing along the list of parameters received by the current handler:

```
return otherFunction(parameterList() as parameters)
```

Information Returned by a Message

Function messages, and sometimes command messages, return a value. The value returned by a function message becomes the value of the function call expression. So, in the last example, the value returned by the “roundToNearest” function message is the value that is put into the `monthlyPayTothePenny` variable. The value returned can be a single value, or may in some cases be a list or property list containing multiple values.

The Result

When a value is returned from a command message, it is available on the following line as `the result`. Because the value of `the result` is reset to `empty` for each statement, a given result can only be accessed on the very next statement. If the value returned is an exception object (a property list with an objectType of “exception”) the caller will throw the exception if `the throwExceptionResults` global property is set to true.

The Message Path

When a message is sent to an object, the message may actually be passed along to several different objects. Each object along the path which the message follows will be given a chance to handle the message. When an object with a handler for that message is found, the instructions in that handler are carried out. Usually, that is the end of that message. In some cases, however, a handler may choose to pass the message on along the path, allowing other objects the opportunity to handle that same message (see the `pass` commands in Working with Messages).

The Message Path

When an object receives a message, its script is checked to see if it contains a handler for that message. If so, that handler is executed. If not, the message is passed to each of the object's helpers in turn, to see if any of them have a handler for the message. When checking whether one of the helpers has an appropriate handler, its list of helpers is also consulted, as well as their helpers (if any) and so forth.

Looking at the larger picture, the message path may also include other objects, both before and after the target object and its helpers. In the standalone SenseTalk environment, these include the objects in `the frontScripts` and `the backScripts`. In other environments, additional objects may also be included.

TechTalk

Syntax: **`insert`** *object* **`into back`**
`insert` *object* **`into the backScripts`**
`start using` *object*

`remove` *object* `from back`
`stop using` *object*

`insert` *object* `into front`
`insert` *object* `into the frontScripts`

`remove` *object* `from front`

Objects in the backScripts list only receive messages not handled by any other objects in the message path. They are commonly used to provide basic functionality which may be needed by many different objects. Objects in the frontScripts list get the chance to intercept and handle any message before it even reaches the target object. This capability is rarely used, but may be quite powerful in certain situations.

After all the objects in the message passing path have had a chance to handle it, a message will be delivered to the host application and SenseTalk itself. If the message corresponds to a built-in command or function, that command or function will be processed. Some messages are recognized by SenseTalk but ignored. If the message is neither recognized and handled by SenseTalk nor recognized and ignored, the host application will look for a script file with the same name as the message. Typically, it will look at least in the same folder where the calling script is located, and possibly in other folders as well. If a script file cannot be found to handle the message, an error is raised.

The Message Path

In total, then, when a message is sent to an object (the target object), the full path of that message is to the following objects, in this order:

- objects in the frontScripts in the order in which they were inserted (with the object most recently inserted in front receiving the message last) and their helpers
- the target object and its helpers
- objects in the backScripts (with the object most recently inserted in back receiving the message last) and their helpers
- the host application and SenseTalk itself
- other script files in folders identified by the host application

At each step along the way, if a handler for the message is found, that handler is run. If such a handler does not pass the message on, it is considered handled and goes no further. However, a handler may also choose to pass the message along using the `pass` command. If it does, the message continues along the message passing path as though it had not been handled, and later objects in the path will have a chance to receive that message.

See The Target, below.

The Target

Because a message may be handled by any of several different objects along the message path, those objects may need to be able to find out what object the message was actually sent to. This is the purpose of `the target`, which identifies the target object. For example, an object inserted into the backScripts might have an "incrementAccessCount" handler that increments a counter each time it is run, like this:

```
on incrementAccessCount
   add 1 to the accessCount of the target
end incrementAccessCount
```

Here, rather than maintaining a single counter in its own accessCount property, it is updating the accessCount property of whatever object received the "incrementAccessCount" message.

Handlers

Handlers are "message listeners". An object must have a handler for a particular message in order to receive that message and act on it.

Handling Messages

An object's behaviors are defined by the way it handles different messages that may be sent to it. When an object

receives a message, it will respond and perform a scripted action, but only if it has a “handler” for that particular message. All scripts consist of one or more handlers for the various messages that the object is interested in handling (including the “initial handler” of a script, which handles a message by the same name as the script). If an object receives a message for which it doesn’t have a matching handler, it ignores that message.

The following is a very simple handler for a “greetTheUser” message:

```
to greetTheUser
    put "Welcome to SenseTalk. Happy scripting!"
end greetTheUser
```

Command, Function, and Generic Handlers

There are three primary types of message handlers: command, function, and generic handlers. Command handlers begin with the word `on` and handle messages sent by commands. A command handler typically performs a series of actions. Function handlers, which begin with the word `function`, handle function call messages, and return a value. Generic handlers begin with `to handle` (or simply `to`) and can handle both command and function messages. It should be noted that a command handler may also return a value, and a function handler may perform actions in addition to returning a value. The only real difference between the handler types is the kind of messages they will handle: a command handler will never be run as a result of a function message, and a function handler will never be called by a command message.

Initial Handlers

In addition to explicitly-declared handlers beginning with the `to`, `on`, and `function` keywords, a script may have an “initial handler” consisting of all of the statements from the beginning of the script (after skipping any properties declarations at the very beginning of the script) to the first explicit handler or properties declaration. The initial handler is treated as a generic “to handle” type of handler, which will handle any messages with the same name as the script. In the case of an unnamed object (one that isn't loaded from a script file) its initial handler can be called using the `run` command or function.

Receiving Passed Parameters

When a handler expects to receive parameters passed to it by the calling script, it can list names for those parameters following the message name (or in a `params` declaration on the first line within the handler – particularly useful in an initial handler). For example, here is a function that will calculate the quotient of two numbers, after first checking for a zero divisor:

```
function quotient dividend, divisor
        if divisor is zero then return zero
        else return dividend / divisor
end quotient
```

The words `dividend` and `divisor` in this example are treated like local variables within the quotient handler, with initial values already assigned to them from the first two parameters passed by the calling function. In SenseTalk, commands and functions can be called with any number of parameters, regardless of what a handler may be expecting. If fewer values are passed in by the calling script than the number of named parameters in a handler, the named parameters for which there are no initial values will be set to empty.

If more values are passed to a handler than the number of named parameters that it declares, the additional values can be accessed from within the handler by using the `param()` or `parameterList()` functions. The number of parameters that were passed in can be obtained by the `paramCount()` function. Here is an example that uses these functions to find the median (middle) value from among all of the parameter values that are passed to it, ignoring any values that are not numbers;

```
to handle medianNumber
    set numList to be an empty list
    repeat with n=1 to the paramCount -- step through all parameters
          if param(n) is a number then insert param(n) into numList
    end repeat
    sort the items of numList in numeric order
    return the middle item of numList
end medianNumber
```

Another way to handle a variable number of parameters passed to a handler is to list one or more variable names after the handler name (or in a `params` declaration), and follow the last variable name with three dots (`...`). By doing this, that variable will be set to a list containing all of the additional parameters:

```
to quoteAndJoin joiner, stuffToJoin...
    get stuffToJoin joined by (quote & joiner & quote)
    return quote & it & quote
end quoteAndJoin
```

Parameters Passed as Containers (by Reference)

Sometimes it is useful for a handler to be able to change one or more of the values in the calling script. In Sensetalk, this can only be done if the calling script allows it, by passing one or more parameters "by reference" or as "containers" (see the description of References in Containers for more details). To illustrate, here is a very simple command handler that swaps two values:

```
on swapValues a,b
    put a into temp
    put b into a
    put temp into b
end swapValues
```

If called in the usual way, this command would have no effect in the calling script:

```
swapValues x,y -- this command leaves x and y unchanged
```

If the calling script explicitly passes its parameters as containers, though, their values can be changed:

```
swapValues container x, container y -- this swaps the values in x and y
```

Returning Results

As noted earlier, a function handler will return a value to the calling script, which is used as the value of the function call expression. If a function handler does not explicitly return a value, its return value will simply be empty. Usually, though, the reason to have a function is to provide a value, so function handlers most often end with a `return` statement. A return statement must include one parameter — the value to be returned (which may be an expression). If needed, multiple values may be returned as a list, as shown here:

```
return (min(source), max(source), average(source))
```

Passing Messages

A handler may choose not to handle a message it has received, or may perform some actions and then pass the message on to other scripts later in the message passing path, using a `pass` command:

```
pass message
```

There are several variations of the pass command, described in Working with Messages.

Handling Any (<any>) Messages

Each handler will handle messages with only one name. So, a “to drive” handler will only be called when a “drive” message is sent. On rare occasions, it may be helpful to create a handler that can receive messages with any name. This can be done by specifying “<any>” instead of a message name. This capability may be particularly useful in a getProp handler (described later in this section) or in a script in the frontScripts.

There are a few special rules associated with these <any> handlers. If a script has a handler for a specific message, that handler will be called rather than the <any> handler — the <any> handler will only be called for messages that would otherwise not be handled by that script. Due to the unique ability of an <any> handler to handle any message that comes along, it is specially blocked by SenseTalk from calling itself. Otherwise it would quickly become a recursive nightmare as each command in an ‘on <any>’ handler would cause it to call itself.

The use of <any> handlers should be approached with caution in order to avoid blocking messages unintentionally. The `param(0)` function can be used to obtain the name of the actual message sent, in order to select the appropriate action or to pass messages that the handler is not intended to deal with:

```
on <any>
    if param(0) does not begin with "x_" then pass message
    -- put code here to handle commands beginning with x_
end on <any>
```

Handling Undelivered Messages: Advanced

When a message is sent for which no handler is found, rather than immediately raising an error, SenseTalk sends an `undeliveredMessage` message to the target of the original unhandled message. An object can implement an undeliveredMessage handler to try passing the original message to some other object which may be able to handle it (see the `pass original message to ...` command, in Working with Messages), or dealing with the problem in some other way. If the undeliveredMessage handler executes a “pass undeliveredMessage” command, the usual error will be raised.

A **pass original message to** *object* command may be used to try passing an undelivered message to some other object. If that object handles the message, that will be the end of it, and execution of the current handler will end. If the object does not handle the original message, execution will continue. In this way, an undeliveredMessage handler can attempt to deliver the original message to one or more other objects which may be able to handle it.

Handling Undelivered Messages: Advanced

```
to handle undeliveredMessage
   repeat with friend = each item in my friends
        pass original message to friend
   end repeat
   pass undeliveredMessage -- give up and let it fail
end undeliveredMessage
```

Helpers

SenseTalk objects do not need to stand on their own. They can be "helped" by other objects which supply some or all of their behavior. For example, if you create a person object which has a handler to calculate the person's age based on their birth date, that person object could be used as a helper for other objects, enabling them to also perform the age calculation without needing their own handler for that purpose.

Here is a sample script for an object which is helped by two other objects, named Rex and Sue:

```
sayHello -- let's start out by being friendly!
properties
   helpers: (Rex, Sue),
   birthDate: "May 14, 1942",
end properties
on sayHello
   put "Greetings!  My age is " & calculateAge()
end sayHello
```

This object's `sayHello` handler calls a function named `calculateAge`. When the sayHello handler is called, it will call calculateAge(). This object doesn't have a calculateAge function handler, but if one of its helpers, say Rex, does have such a function handler, that handler will be run on behalf of this object, to calculate the age.

Who is Me? What is This Object?

An object and its helpers work closely together, acting very much like a single object (think of the Three Musketeers' slogan, "All for one, one for all"). When a handler in a helper is being run, it is treated as though it were a handler of the original object. In this context, `me` and `my` refer to the object being helped rather than the helper.

This means that any statements executed in the helper's script will target their messages to the object being helped (and only indirectly, then, to the helper). It also means that if the helper uses `me` or `my` to access properties, it will be accessing the properties of the helped object. For cases where a script may need to target itself or access its own properties, the term `this object` may be used instead of `me`.

Helpers' Place in the Message Path

An object's helpers are also closely tied to it in the message path. When an object receives a message that it doesn't have a handler for (or which it handles and then passes along), that message will go next to the object's helpers. Only if none of the helpers handle the message will it be passed along to other objects in the message path, such as to those in the backScripts.

Objects Designed to be Helpers

Any object can be a helper to one or more other objects. Suppose, for example, you have an object called Rex that has a calculateAge() function handler. Then suppose you have another object, Luna, that needs the same calculateAge() functionality. By adding Rex to Luna's list of helpers, Luna gains that capability. But Rex may have other handlers as well, which may not all be desirable to include as part of Luna's behaviors. How can you provide just the functionality that Luna needs?

One good way to deal with this situation is to separate out those behaviors (handlers) that may be useful to several different objects, and create a new object designed specifically to serve as a helper. For example, you might move the calculateAge() handler out of Rex into a Person object, and then add Person to both Rex and Luna's helper lists. Each object can have any number of helpers, so separating related groups of behaviors out into helper objects can be a very powerful tool, allowing you to re-use that functionality in various combinations in different objects.

Designing an Object to be a Helper

Many helper objects are quite simple, having nothing more than a script with a few handlers. A Person object, for example, might start off with just a single handler, like the calculateAge() function that we've been talking about:

```
to calculateAge -- returns my current age in years
    return (the date - my birthDate) / 365.25 days
end calculateAge
```

More on Me (oh My!)

One very important thing to understand about helpers is that when the terms `me` and `my` are used in the script of a helper, they refer to the object being helped and its properties, not to the helper or its properties (use `this object` if necessary to refer to the helper). This usually means that a handler written as part of an object will also perform correctly when helping another object, without any changes.

Making Objects Based on Other Objects

Simply using one object as a helper to another in order to borrow the first object's behavior is very easy, often requiring no special effort on your part. Once you've decided to separate some of the functionality of your scripts into an object that will serve mainly as a helper to other objects, there are some additional capabilities you may want to take advantage of.

Earlier in this section, we saw how a simple object could be created using a `new object` expression. It was mentioned that the more common way to use a `new object` expression is to indicate a particular type of object that you want to create by providing the object it should be based on, as shown in this example:

```
put new Person with (name:"Elizabeth", age:14) into daughter
```

Here, Person is the name of an object (known as a "prototype" object) being used to define the newly-created object. In this example SenseTalk will look for an object named "Person" and send that object a `makeNewObject` function message with the initial properties supplied as a parameter. This gives the prototype object the chance to control exactly what the new object will be like. Typically, the new object will be based on the protytype object itself, by having the prototype object as its helper, thus inheriting all of its behaviors, and by receiving a copy of the prototype's regular properties (other than its script and helpers). Let's look more closely at exactly how this works.

The Role of a Prototype Object

If a prototype object has a "makeNewObject" function, that function should create and return a new object. If it doesn't handle the makeNewObject message, SenseTalk's built-in makeNewObject function will be called, which will create a new object helped by the prototype object (Person, in our example), set any properties of that object that were supplied in the `new` expression ("name" and "age" in this case), add any properties from the prototype that aren't already present, and then send the object an "initialize" message. So, the effect of the built in behavior is essentially the same as this makeNewObject function:

```
to makeNewObject initialProperties
    get new object with initialProperties helped by me
    add properties of me to it
    send "initialize" to it
    return it
end makeNewObject
```

If a different behavior is needed, simply write a custom `makeNewObject` function in your prototype object. You might, for instance, want to assign some default initial property values, include additional helpers, or perhaps even call on a different prototype object to make the new object.

Notice that the default behavior is to create a new object that is helped by the prototype object. So any object can be used as a prototype object, and will become the primary helper of new objects created from it.

Initializing the Properties of a New Object

When a new object is created from a prototype, by default it inherits all of the behaviors of the prototype object, and also copies of its properties. It also receives any properties supplied with the `new` expression (which will override any corresponding property values from the prototype). A prototype object may also provide additional properties for the new object. One way to do this, as already mentioned, would be to write a custom makeNewObject function, although this is rarely needed. A much simpler (and better) choice is to take advantage of the "initialize" message that is sent to the new object by the built-in makeNewObject function (the "initialize" message is sent only to the object and its helpers, not through the full message path). Here is an example initialize handler that provides some default property values:

```
to initialize
   add properties (time:"12:00", priority:"Normal") to me
end initialize
```

If this handler is in a prototype Appointment object that has a month property with a value of "May", then a new appointment can be created like this:

```
put a new Appointment with (time:"8:30") into meeting
```

This will create a new object helped by Appointment, with its time property set to "8:30" and its priority set to "Normal". Because the `add properties` command doesn't replace any properties that are already present, any values supplied with the `new` expression will take priority over those provided by this initialize handler. The new object will also have a month property with the value "May" that was copied from the prototype object.

Creating Empty Objects

Sometimes you may want to create a new object that doesn't automatically receive copies of all of the properties from the prototype. To do this, use the word `empty` before the prototype:

```
put a new empty Appointment with (time:"8:30") into meeting
```

Here, the new object will be helped by Appointment, with its time property set to “8:30”, and its priority property set to "Normal" (from the initialize handler). It will not receive the month property (or any other properties) copied directly from the prototype object.

Displaying Objects as Text

When an object is displayed as text, by default its keys and values are displayed in a format determined by the plistPrefix, plistSuffix, plistKeySeparator, plistEntrySeparator properties, as described in Lists and Property Lists. However, objects can represent themselves as text strings in any way they like, by implementing a handler for an `asText` function as part of their script, like this:

```
to handle asText
   return "Part number " & my partNum
end asText
```

If an object does not handle the `asText` message, but does have an “asText” property, the value of that property will be used as the string representation of the object. This is very convenient for some simple objects that may not have a script at all.

For more flexibility, if an object has neither an “asText” handler nor an “asText” property, but it has an “asTextFormat” property, the value of that property is evaluated by the merge() function to provide the text representation of the object. This provides a more dynamic solution that can combine the values of other properties of the object:

```
set account to {type:"Savings", balance:1234.25,
         asTextFormat:"[[my type]] Account: [[my balance]]"}
put account -- "Savings Account: 1234.25"
add 5050.50 to account's balance
put account -- "Savings Account: 6284.75"
```

Checking Object Contents

When a script checks whether or not an object contains a particular value using either the `contains` or `is in` operators, the object (or one of its helpers) may implement a function to perform the test in any way it wants. When either of these operators is used to check for the presence of a value in an object or property list, a `containsItem` function message is sent to that object. If it implements a handler for this message, that handler will be passed two parameters – the value to search for, and a boolean indicating whether the test should be case-sensitive or not – and it should return `true` or `false` to indicate whether the indicated value is present in the object or not.

If an object does not handle the `containsItem` message, SenseTalk's implementation of that function will be called. The behavior of the built-in function can be configured by setting `the objectContainsItemDefinition` global property. If this property is set to "AsTextContains" (the default behavior) the containsItem function returns true if the object's text value (as described above) contains the search value. When set to "NestedValueContains", the operator is applied to each of the object's values and the expression will yield true if any them contains the search value. Finally, a setting of "KeyOrValueEquals" will return true if the search value is equal to one of the object's keys or to one of its values.

Early Helpers: Advanced

An object's helpers provide additional behavior by potentially handling any messages sent to the object that it doesn't already handle on its own. The object can override any handlers provided by its helpers by supplying its own handlers. Occasionally, it may be useful to be able to work the other way around, providing an object with a helper that can override behaviors defined by the object itself. Early Helpers do this. Any messages received by an object are sent first to the object's early helpers, then to the object itself, and finally to its helpers.

Assigning Helpers to an Object

We have already seen a number of ways that helpers can be assigned to an object when it is first created: by including a list of helpers in a properties declaration of a script file; by listing them in a 'helped by' clause following a property list or a `new object` expression; or by creating an object from a prototype object that becomes its first helper.

An object's list of helpers – and its list of early helpers – can also be modified dynamically at any time by a script. Simply access the `helpers` or `early helpers` property and insert or delete objects as needed:

```
insert CleverTricks into the helpers of Fritz
```

Properties

Most of the properties of a SenseTalk object are ordinary containers that can store any type of value, and have no special inherent meaning outside of the way they are used in your scripts. There are a few special properties, however, that do affect the way an object is treated by SenseTalk. These include the `asText`, `asTextFormat`, `objectType`, and `script` properties which show up as ordinary properties of your object but have special meaning. There are also a number of "hidden" object properties that are not listed by the `keys` function but can be accessed and in some cases changed by your script. These include the `long id`, `name`, and `helpers` properties.

Referring to an Object's Properties

An object's properties can be accessed in several ways, as described for property lists in Lists and Property Lists. These include the dot (`.`), apostrophe-S (`'s`) and `of` syntaxes. Ordinary properties are treated as containers – you can store any type of value into them, or modify their values just like those in a variable or other container.

Property and Function Integration: Advanced

Accessing a property of an object and calling a function on an object may seem to be completely different tasks, but in many ways they are quite similar. SenseTalk allows objects to define "property" values using a function that can dynamically calculate the value of the property. To allow the designer of an object complete flexibility in how it is implemented, calling a function on an object and accessing a property look the same in the script that is doing the accessing. To see how this works, consider the following three examples (which are all equivalent):

Property and Function Integration: Advanced

```
put gumdrop.color
put gumdrop's color
put the color of gumdrop
```

Any of these examples will perform the following sequence of actions to determine the value of the expression. First, if `gumdrop` is an object, a 'color' function message is sent directly to the gumdrop object (note: this message is not passed through the usual message passing path, but only to the object and its helpers). If the gumdrop object has a `function color` handler, that handler will be called and the value that it returns will be the value of the expression. If it does not handle the color message, then gumdrop's `color` property will be accessed instead (including calling getProp or setProp as described below). Finally, if there is no `color` property (or if gumdrop was not an object), a 'color' function message is sent with the gumdrop object as a parameter. This message is sent to `me`, that is, to the object whose script is executing.

Because of this sequence of actions, the `gumdrop` object's designer may choose to implement the object's color using a `function color` handler, a `getProp color` handler, or an actual `color` property.

If the word `property` is used, the property of the object will be accessed without calling the function (but getProp or setProp will still be invoked as appropriate):

```
put property color of gumdrop -- avoid calling the color function
```

Parameters may also be passed, but the syntax for doing so is different for function calls or property access:

```
put gumdrop.color("sprinkles") -- calls the color function
put gumdrop's color("sprinkles") -- calls the color function
put the color of gumdrop with "sprinkles" -- unified access
put property color of gumdrop with "sprinkles" -- property
```

Here, the parameter (the word "sprinkles") will be passed as a parameter to the color function in the first two examples. The use of parentheses without the word "with" always signifies a function call. The third example (using the "of" syntax) invokes the unified process described above, passing the parameter to the color function, and also to the getProp handler if it is called. If the final step of sending a 'color' function message to `me` is invoked as part of that process, both the gumdrop object and the word "sprinkles" will be passed as parameters.

Multiple parameters may also be passed (always in parentheses):

```
put gumdrop.color("red", "pink") -- function call
put gumdrop's color("red", "pink") -- function call
put the color of gumdrop with ("red", "pink") -- unified
put property color of gumdrop with ("red", "pink") -- property access
```

Special Properties

There are several object properties with special meaning to SenseTalk. Some of these properties are "hidden" in the sense that they are not listed by the keys or values functions and will not appear in the default text representation of an object, but they can be directly accessed as properties of an object.

ObjectType

Setting an object's "objectType" property identifies it as a particular type of object for the `is a` operator. For example, the object caught by the `catch` keyword when an exception is thrown has its objectType property set to "exception" so if this object is in a variable named "error" then the expression `error is an "exception"` would evaluate to true. The objectType property may be set to a list of types if an object may be considered to be of more than one type, as shown here:

```
set shape to (width:7, height:7, objectType:("rectangle",
"square"))
put shape is a "rectangle" -- true
put shape is a "square" -- true
```

In fact, when the value being tested by the `is a` operator is an object, SenseTalk sends an "isObjectType" function message to the object, with the type as a parameter. If the object has a `function isObjectType` handler, it can determine dynamically whether it is an object of the given type and return true or false accordingly. The default SenseTalk implementation of this function checks the "objectType" property of the object as described above.

AsText

The `asText` property, if set for an object, will be used as the text representation of the object if it doesn't handle an "asText" function message:

```
set shape to (objectType:"Square", size:7, asText:"Seven Square")
put shape -- displays "Seven Square"
```

AsTextFormat

If an object has neither an "asText" function handler nor an `asText` property, SenseTalk will check for an `asTextFormat` property. If this property is set for an object, its value will be used as a format string to generate the text representation of the object using the `merge()` function (see Working with Text). This provides a simple way to achieve a dynamic text representation of an object:

```
set account to {balance:1234.25, type:"Savings",
   asTextFormat:"[[my type]] Account: [[my balance]]"}
put account -- "Savings Account: 1234.25"
add 5050.50 to account's balance
put account -- "Savings Account: 6284.75"
```

If an object has neither an asText function handler nor an asText nor asTextFormat property, the default representation using `plistPrefix` etc. will be used (see Lists and Property Lists):

Script

An object's `script` is a property of the object, and may be changed. Setting the script will change the object's behaviors to those of the handlers in the new script. Handlers that are already executing will not be changed until the next time they are called.

```
set Fido to (script:<<play "Basso">>)
set the script of Fido to guardDogScript
```

Helpers, Early Helpers

The `helpers` and `earlyHelpers` properties are hidden properties holding lists of objects. When one of these

properties is changed, such as by inserting a new helper object, each item in the list is checked to be sure it is an object, and an exception is thrown if it is not. The items in this list are always reported as the objects' long ids.

Name, Long Name, Short Name, Abbreviated Name, Long ID

The `name` property, along with its variants when preceded by the adjectives `long`, `short`, or `abbreviated`, and the `long id` property, are all hidden properties that identify the object. All of them except the `name` property are read-only. For a script file object, the long name and long id are both the full path name of the file on disk, the name and abbreviated name are the file's name including extension, the short name is the file's name without the extension if the extension is a recognized script extension in the application.

Folder, Directory

Script file objects also have a `folder` (or `directory`) hidden read-only property that provides the full path of the folder where the script file is located.

Working with Messages

SenseTalk is an object-oriented language which works by passing *messages* from one object to another, as described in the last section. When a message is sent to an object, it can respond if it has a *handler* for that message. If the object does not have a handler for a particular message, that message is passed along to some other object.

This section provides the details about all of the specific mechanisms that may be used to deal with sending, handling, or otherwise managing messages in your scripts.

Handlers

A SenseTalk script is made up of “handlers” A handler is a part of a script that defines how the script will handle a particular message that is sent to it. There are three primary types of handlers: command handlers (sometimes called “on” handlers), function handlers, and generic handlers (also known as “to” handlers). There are also two special types of handlers – “getProp” and “setProp” handlers – which were described in the section on “Properties” in the previous section.

If a script contains both a generic handler and a specific handler (“on” or “function”) for the same message name, the specific handler will always receive priority and receive the message. The “to” handler will still receive messages of the other type. If a script contains all three types of handlers for a particular message, the “on” handler will receive command messages by that name, the “function” handler will receive function messages, and the “to” handler will never be called.

◊ to, to handle

What it Does

The `to` or `to handle` keyword declares a generic handler that can receive both command and function messages:

```
to {handle} messageName {{with} paramName1 {, paramName2...} }
    statements
end messageName
```

When the script receives either a command message or a function message matching *messageName* the *statements* of this handler will be executed. The incoming parameter values passed to the handler will be assigned to the corresponding parameter names (*paramName1* , etc.) declared following the messageName.

Examples

```
to handle increaseSize amount
    if amount is empty then add 1 to my size else add amount to my size
end increaseSize
```

If the handler needs to take different actions depending on how it was called, the `messageType()` function can be used to find out whether the handler was called as a command or as a function.

◊ on

What it Does

The on keyword is used to declare a command handler, and an end keyword ends it, like this:

```
on handlerName {paramName1 {, paramName2...} }
  statements
end handlerName
```

When an object receives a command message matching handlerName the statements of that handler will be executed. The incoming parameter values passed to the handler will be assigned to the corresponding parameter names (*paramName1* , etc.) declared following the handlerName .

Examples

```
on addToTotal newAmount
    add newAmount to global total -- store total in a global variable
end addToTotal
```

The example above will handle an “addToTotal” command message. If a parameter is passed with the message, its value will be available in the “newAmount” variable, otherwise that variable will be empty.

◊ function

What it Does

The function keyword declares a function handler, like this:

```
function functionName {paramName1 {, paramName2...} }
    statements
    return returnValue
end functionName
```

When the script receives a function message matching *functionName* the *statements* of this handler will be executed. The incoming parameter values passed to the handler will be assigned to the corresponding parameter names (*paramName1* , etc.) declared following the *functionName*. The *returnValue* will be passed back to the calling script as the value of the function call.

Examples

```
function getTotal
    return global total
end getTotal
```

Initial Handlers

In addition to handlers which are specifically declared using to, on or function, any lines at the beginning of a script are referred to as the object’s *initial handler*. Many scripts contain only an initial handler, and none that are named explicitly using to, on or function. The initial handler is always treated as a generic “to handle” handler, that can respond to both command and function messages.

All statements up to the first named handler in the script (if any) are treated as if they were part of an unnamed initial handler. When the script is loaded directly from a file, the initial handler is assigned the same name as the name of

the file, less any extension and illegal characters. For objects other than script files, the initial handler is assigned the name “<initialHandler>”.

Note: Identical names

If the script actually contains a named handler by the same name as the script (such as “to handle scriptFile-Name”), that handler will take precedence, and the initial lines of the script prior to the first named handler will be ignored.

Script-Object Caching and the WatchForScriptChanges Global Property: Advanced

When a message is sent to a script object that resides on disk, SenseTalk reads that script file and caches the script in memory. If the object then receives another message, SenseTalk can check very quickly whether it has a handler for that message. In some (fairly rare) situations it may be desirable to have SenseTalk check for updates to the script during a run. In these situations, you can set `the watchForScriptChanges` global property to true (the default setting is false). This will cause SenseTalk to check the file for updates each time the object receives a message. If the file has been updated, it will be read again, and its new handlers will be used. The executing version of a handler is not changed while it runs.

◊ getProp, setProp

In addition to the standard command and function handlers, an object’s script may include two special types of handlers for working with the object’s properties: getProp handlers for providing the value of a property, and set-Prop handlers for receiving a new value for a property.

Whenever a property of an object is being read, a getProp message for that property is sent to the object. If it handles the message, the value it returns is used as the value of that property. When the value of a property is changed, a setProp message is sent to the object, with the new value as a parameter. If the object has a setProp handler for that message, it is called, otherwise the property is set directly.

For example, here is a handler that will supply the “area” property of an object, based on its length and width:

```
getProp area
   return my length * my width
end area
```

An object may also want to control setting a property. Here, setting the “area” of an object will actually change its length:

```
setProp area newArea
   set the length of me to newArea / my width
end area
```

◊ getProp, setProp

If an object accesses one of its own properties from within a getProp or setProp handler for that property SenseTalk will access the property directly rather than calling the getProp/setProp handler recursively.

◊ my direct property

For efficiency, it is sometimes helpful for an object to be able to access its own properties without sending any function or getProp/setProp messages. This can be done using the special syntax `my direct propName` (or `my direct property propName`) as in this getProp handler:

```
getProp age -- my age in years
   return (the date - my direct birthDate) div 365.25 days
end age
```

Note that `direct` can only be used with `my` and is therefore limited to an object accessing its own properties. Access to another object's properties will always involve calling getProp or setProp.

Parameters and Results

Parameters are values that are passed to commands or functions. When you invoke a command, you may pass parameters to that command by listing them following the command name, separated by commas:

```
doSomethingImportant 31,"green",style
```

The example above passes 3 parameters to the doSomethingImportant command: the number 31, the text "green", and the variable `style`. Parameters are passed to functions in a similar way, by listing them inside parentheses following the name of the function being called. Two commas in a row imply an empty parameter, and a final comma at the end of a list is ignored, so the following passes 3 parameters ("silverBar", "", and 16):

```
get verifyQuantity("silverBar",,16,)
```

When passing a variable as a parameter, only its value is passed, so the local variable's contents cannot be changed, unless it is passed by reference, as described in References to Containers.

Instead of passing individual parameters, you may pass all of the values contained in a list by specifying `as parameters`. This will pass the values individually, rather than passing the list as a single entity:

```
updateAllItems thingsToBeUpdated as parameters
```

A handler can declare and use the parameters it receives, and supply results, as described below.

◊ params declaration

What it Does

Declares some named incoming parameters for the initial handler of a script. The `params` declaration must be the first statement in the script. It assigns names to the incoming parameters. The parameters may still also be accessed using the `param`, `paramcount`, and `params` functions as described below.

Examples

```
params width,height
```

Tech Talk

Syntax: params *paramName1 {, paramName2 ...}*

◊ param function

What it Does

Returns the value of one of the parameters passed to the current handler, specified by its ordinal position in the parameter list. A handler may receive any number of incoming parameters, regardless of the number of named parameters that it declared. The `param` function can be used to retrieve their values.

Examples

```
get the param of 1 -- gets the first parameter
put param(0) -- shows the handler name
```

Tech Talk

Syntax: the param of *numExpr*
param(*numExpr***)**

◊ paramCount function

What it Does

Returns the number of parameters passed to the current handler.

Examples

```
repeat with n=1 to the paramCount
   put "Parameter " & n & " is " & param(n)
end repeat
```

Tech Talk

Syntax: the paramCount
paramCount()

◊ params function

What it Does

Returns a text string consisting of the handler name followed by the values of all of the parameters passed to the current handler. Parameter values other than lists and property lists are each enclosed in quotes.

Examples

```
put the params -- might show: test "John Doe","27",("JD","Bud")
```

Tech Talk

```
Syntax: the params
        params()
```

◊ parameterList function

What it Does

Returns a list containing all of the parameters that were passed to the current handler. Sometimes this may be a more convenient way of working with the parameters that were passed in than using the `paramCount` and `param` functions.

Examples

```
repeat with each item of the parameterList
   add it to total
end repeat
return func(the parameterList as parameters) -- pass our params to func
```

Tech Talk

```
Syntax: the parameterList
        parameterList()
```

◊ messageType function

What it Does

Returns a value indicating the type of message being handled, such as “Command” or “Function”. This can be used to enable a generic (“to handle”) handler to take different actions depending on how it was called. Other possible values that may be returned include “GetProp” and “SetProp”.

Examples

```
if the messageType is "Function" then return theAnswer
```

Tech Talk

Syntax: the messageType
messageType()

◊ return command

What it Does

Returns the result of a function when used in a function handler, or sets `the result` if used in a message handler.

Examples

```
return pi*diameter
```

Tech Talk

Symtax: return {*expression***}**

The `return` statement terminates execution of the current handler. If an *expression* is not given, `empty` is returned.

◊ result function

What it Does

Returns the result (if any) set by the previous command. Most commands which operate on files, for example, will set `the result` to empty if they succeed, or to a message indicating what went wrong if they fail.

Examples

```
if the result is not empty then throw "Error", the result
```

Tech Talk

Syntax: the result
result()

At the beginning of each statement `the result` is set to the result produced by the previous statement. To determine the result of an operation, therefore, you must call this function as part of the very next statement executed.

When a `return` statement is executed as part of a handler that was called as a command (rather than as a function), the return value sets `the result` in the script which called it.

The following commands will set `the result` to something other than empty under some conditions: `answer`, `ask`, `convert`, `copy`, `create`, `delete`, `move`, `open`, `post`, `read`, `rename`, `replace`, `seek`, `shell`. It can also be set when accessing a file or URL. In testing the result to determine whether the previous command succeeded, it is usually best to check whether the result is empty, rather than looking for specific words in the result, since the exact wording of an error message may change in a future release.

If `the throwExceptionResults` global property is set to true, any time a command sets `the result` to an exception object (which happens on most error conditions), that exception will be thrown rather than becoming available through this function as described above.

Whenever a non-empty result is set, it is inserted into `the resultHistory` global property. This makes it possible for a script to retrieve a result value from an earlier statement. Care must be used, however, to match a result to a particular statement, as empty values are not included. A limited number of results are retained, as specified by `the resultHistoryLimit` global property. Once this limit is reached, older results are discarded.

◊ handlerNames function

What it Does

Returns a list of the names of each of the handlers in an object's script. The order in which the handlers are listed is undefined.

Examples

```
put handlerNames of Account
```

Tech Talk

Syntax: the handlerNames of *anObject*
handlerNames(*anObject***)**

Passing Messages

◊ pass command

What it Does

Passes the current message along to the next object in the message passing path (as described in the previous section). This terminates execution of the current handler.

Examples

```
pass message
```

Tech Talk

Syntax: pass *handlerOrFunctionName*
pass {message | on | function | getProp | setProp}

If *handlerOrFunctionName* is given (rather than the generic term "message"), the name supplied must be the same as the name of the current handler. If `on`, `function`, `getProp`, or `setProp` is used, they must match the current handler type.

Pass ... and continue

What it does

Use `pass ... and continue` to pass the current message along to the next object in the message passing path, just as the `pass` command does, but allow the current handler to continue running after the message has been handled elsewhere.

Examples

```
pass message and continue
```

Tech Talk

Syntax: pass *handlerOrFunctionName* **and continue**
pass message and continue

The current handler resumes executing after the message is handled by some other object later in the message passing path. Any value returned by the other object is available in `the result` immediately after the **pass ... and continue** command.

Pass ... and continue

After executing a **pass ... and continue** command, any later attempt to pass that message (using any of the pass commands) will be unable to actually pass the message along, as no message is ever delivered to the same object twice.

Pass ... without helping

What it Does

Use `pass ... without helping` to pass the message along to the next object following the helpee in the message passing path. This differs from the `pass` command when used in a helper. In essence, the helper is saying "I'm not able to help with this message". The helpers' helpers are not given a chance to handle the message, but the message is passed to later helpers of the object being helped, and to other objects later in the message passing path.

Examples

```
pass message without helping
```

Tech Talk

Syntax: pass *handlerOrFunctionName* **without helping**
pass message without helping

The current handler stops running when this command is executed.

Pass original message to ...

What it Does

Use `pass original message to ...` to pass the current message directly to some other object. This differs from other forms of the `pass` command, which pass the current message along to the next object in the message passing path.

Examples

```
pass original message to alterEgo
```

Pass original message to ...

Tech Talk

Syntax: pass original message to *object* **{and continue}**

This form of the pass command is intended for use within undeliveredMessage handlers to try passing the original (undelivered) message to some other object for handling. If that object handles the message, that will be the end of it, and execution of the current handler will end. If the other object does not handle the message, execution will continue in the current handler. In this way an undeliveredMessage handler can attempt to deliver the original message to one or more other objects which may be able to handle it. See “Undelivered Messages” below.

If `and continue` is specified, the current handler resumes executing after the message is passed, whether or not it was handled by the other object.

Exiting a Handler

◊ exit handler

What it Does

Terminates execution of the current handler. Execution continues in the calling handler.

Examples

```
exit handler
exit myFancyFunction
```

Tech Talk

Syntax: exit *handlerOrFunctionName*
exit [handler | script | on | function | getProp | setProp]

If *handlerOrFunctionName* or a handler type is given, it must match the name or type of the current handler.

◊ exit all, exit to top

What it Does

Stops execution of all handlers (the current handler, plus the handler which called it, and so forth).

Examples

```
exit all
```

Tech Talk

```
Syntax: exit all
        exit to top
```

Running Other Scripts

The simplest way to run another script is merely to use the name of that script as a command, along with any needed parameters. For example, the following command will run the `tellMeMore` script:

```
tellMeMore 1,2,3
```

A specific handler within a script can be run by using the script name, a dot or apostrophe-s, and the handler name, followed by any parameters:

```
tellMeMore's elucidation "terse",5
utilities.insertCommas @myNumber, 3
```

For more complex situations, the following commands and functions may be useful.

◊ run command

What it Does

The `run` command can be used to run a specified script when a simple command as shown above won't work, such as when a full pathname is needed or the script's name contains spaces or other special characters, or when the name of the script to be run is in a variable. The following command will run the script's initial handler:

```
run "/tmp/testScript.st"
```

Parameters may be passed:

```
run "common/startup" 1200,true,"salvador"
```

A specific handler other than the script's initial handler may be called, using 's:

```
run "common/finance" 's amortize 143000,6.75,360
```

The `run` command may be used with any object, not just a script file:

```
run account's accrueInterest 30, 4.125
```

The preceding example is equivalent to:

```
send accrueInterest 30, 4.125 to account
```

Although the `run` command provides similar functionality to that of the `send` command, it offers an alternative syn-

tax for running a script, and also gives you the ability to run the initial handler of a script, which the `send` command does not provide. In addition, the `run` command runs the script's initial handler directly, without calling front scripts, back scripts, or helpers.

Tech Talk

```
Syntax: run scriptName {,} {parameters}
        run scriptName 's handlerName {parameters}
```

Calls the *scriptName* script, or the *handlerName* handler of that script, as a command. *HandlerName* is usually just the simple name of a handler without quotes, but may also be given as a quoted literal, or as an expression in parentheses to construct the handler name at run time.

◊ run function

What it Does

Calls the initial handler of a script or other object explicitly as a function. This also allows calling a script whose name is generated dynamically at runtime. The `run` function needs to be called in a way that sends the run message to the target object, such as using dot notation.

Examples

```
set doubler to (script:"return 2*param(1)")
put doubler.run(12) -- 24
get ("test" & testNumber).run(value1, value2) -- call script dynamically
```

Tech Talk

```
Syntax: object.run(parameters)
        {the} run of object {with parameters}
```

◊ do command

What it Does

Evaluates an expression as one or more SenseTalk statements and executes those statements within the current handler. This is an advanced command that is seldom needed, as there are usually simpler ways to achieve the same results. In some situations, however, `do` can perform actions that might otherwise be difficult or impossible to achieve. The `do` command is typically used to execute commands generated by the script at runtime.

◊ do command

Examples

```
do "put total into quarter" & currentQtrNumber
do customCleanupCode
```

Tech Talk

Syntax: do *expression*

The do command evaluates *expression* as one or more SenseTalk statements and executes those statements as though they were part of the current handler. This means that the statements contained in *expression* can refer to, use, and modify any local variables of the handler (unless the evaluationContext global property is set to "Global" or "Universal" in which case undeclared variables will be treated as global or universal rather than local variables).

Using the do command to execute some code is slower than including the same statements directly in your script, because the SenseTalk compiler is called to compile the code each time before it is executed. The impact on performance is especially significant if the do command will be executed many times, such as inside a repeat loop.

◊ do AppleScript command, doAppleScript function

What it Does

Executes one or more statements written in the AppleScript scripting language that are contained in a variable or generated by your script at runtime. Any final value or error will be available in the result.

Examples

```
do acquireDataScript as AppleScript
put "EggPlant" into app -- name of the app we want to leave showing
do AppleScript merge of {{
   tell application "Finder"
        activate
        set visible of every process whose name is not "[[app]]" to false
   end tell
}}
put doAppleScript(merge("get [[A]] + [[B]]")) into sumDoneTheHardWay
```

Tech Talk

Syntax: **do AppleScript** *expression*
do *expression* **as AppleScript**
get doAppleScript(*expression***)**

The `do AppleScript` command evaluates *expression* as AppleScript statements and executes those statements by calling the AppleScript interpreter . Values can be passed in at any point within the AppleScript code by enclosing local SenseTalk variable names and expressions in double square brackets `[[ ]]` within the code and using the `merge` function to insert their values before running the AppleScript.

When run using the `doAppleScript` function, the final value of the AppleScript expression will be the value returned by the function. Any errors will be returned as an exception object in `the result`. When `do AppleScript` is run as a command, `the result` will contain either an exception object or the final value of the expression.

The AppleScript code is run on the local machine. To send messages to another machine using this mechanism, the other machine will need to have remote events turned on in the Sharing panel of the System Preferences, and your AppleScript code will need to send commands to that machine using a tell statement looking something like this:

```
tell application "AppName" of machine "eppc://userName:password@xxx.xxx.xxx.xxx"
```

Consult an AppleScript reference for more details.

◊ send command

What it Does

Use the **send** command to send a message to another object.

Examples

```
send "Hello" to newObject
send "calculateLoan 10,9,5000" to "Loan"
send accrueInterest 30, 4.625 to account
```

Tech Talk

Syntax: **send** *message* **{to** *object***}**
send *message parameters* **{to** *object***}**

Message is an expression which is evaluated as text. The message sent may include parameters, either within the message string (as in the “calculateLoan” example above) or separately (as in the “accrueInterest” example above). If an *object* is not specified, the message is sent to the object **me** (that is, the object containing the script).

◊ target function

What it Does

Returns the long id of the object to which the current message was sent. Useful in a handler farther down the message passing path when you need to know what the original target of the message was.

Examples

```
if the target is not me then ...
```

Tech Talk

```
Syntax: the target
        target()
```

As of SenseTalk version 1.1, using the syntax **the target** accesses the target object directly. The **target**() syntax calls a function which (if not overriden by another script in the message path) will return the long id of the target object. This will function equivalently for most purposes, although it may be slightly less efficient in some cases.

◊ [ ] (square bracket messaging)

What it Does

Use square brackets to send a function-type message to a specific object. The square bracket syntax can be used by itself as a statement, or like a function call as part of an expression. In both cases, however, the message it sends is a function message, so it must be handled by a **to** or **function** handler (not an on handler).

Examples

```
[currentLoan computeInterest:10,9,5000]
answer "The result is: " & [calculator getResult]
```

Tech Talk

```
Syntax: [ object functionMessage ]
        [ object functionMessage: parameters]
```

Commands and Functions

Working with Text – details the text functions and commands which are available for manipulating strings of text.

Working with Numbers – documents the mathematical functions and commands which are available for performing various numeric calculations in SenseTalk.

Working with Dates and Times – describes how SenseTalk scripts can use and manipulate values representing dates and times.

Working with Files and File Systems – explains the extensive facilities available in SenseTalk for reading and writing data in files, and for working with files and folders in the file system.

Working with URLs and the Internet – describes how SenseTalk can be used to access resources on the Internet and to manipulate URL strings.

Working with Trees and XML – describes the SenseTalk tree structure and how it can be used to read, write, and manipulate XML data.

Working with Color – describes facilities provided by the STColor XModule to enable your scripts to work with values representing colors.

Working with Binary Data – explains mechanisms for working with binary (non-textual) data in your scripts.

Other Commands and Functions – describes commands and functions for interacting with the user and with the system.

Working with Text

SenseTalk has very strong text handling capabilities. Its chunk expressions, described in Chunk Expressions, provide a powerful and intuitive means of accessing and manipulating specific portions of a text string. In addition, there are a number of commands and functions for obtaining information about text, converting between text and other data formats, and manipulating text at a high level. This section describes these commands and functions in detail.

◊ capitalized function

What it Does

The `capitalized` function returns text with the first letter of each word capitalized.

Examples

```
put capitalized of "now and then"  -- shows "Now And Then"
```

Tech Talk

Syntax: {the} capitalized of *stringFactor*
capitalized(*stringExpr***)**

See Also: the `uppercase` and `lowercase` functions, later in this section.

◊ charToNum function

What it Does

Returns the numeric code (in Unicode) representing the first character of its parameter.

Examples

```
put charToNum("a")  -- 97
```

Tech Talk

Syntax: {the} charToNum of *textFactor*
charToNum(*textExpr***)**

See Also: the `numToChar` function, later in this section.

◊ delete command

What it Does

The `delete` command deletes a chunk of text or one or more occurrences of a target text string within a container. In its simplest form, it will delete every occurrence of the target text, regardless of case. More advanced options allow you to specify a chunk by its location, or to tell how many occurrences of a target string – or even to indicate a particular occurrence – to delete, and to specify exact case matching.

Examples

```
delete the first 2 characters of line 15 of output
delete the last "s" in word 3 of sentence
delete every occurrence of " ugly" in manual
delete the third "i" within phrase considering case
```

Tech Talk

Syntax: **delete** *chunk* **[of | in]** *container*
delete {*Options***}**

Options:
[in | within | from] *container*
{ all {occurrences of} | every {occurrence of} } *targetText*
{the} [first | last] *howMany* **{occurrences of}** *targetText*
{the} *ordinalTerm* **{occurrence of}** *targetText*
occurrence *ordinalNumber* **of** *targetText*
[with | considering] case
[without | ignoring] case

For the first form of the `delete` command, chunk can be any chunk expression describing the part of container that should be deleted. See Chunk Expressions for a full description.

For the second form of the `delete` command, exactly one of the options defining the *targetText* must be supplied, as well as the *container* where the deletions will occur. The **case** options are optional. The options used may be specified in any order, although only one option of each type may be given.

You must include the `in` *container* or `within` *container* option in such a delete command. The *container* can be any container, including a variable, a portion of a variable (using a chunk expression), or a text file. If *container* is a variable, its contents may be of any sort, including a list or property list. The `delete` command will delete the indicated text, searching through all values nested to any depth within such containers.

You must include one of the *targetText* options in a delete command, to specify what will be deleted. Simply providing a *targetText* expression will cause the command to delete every occurrence of the value of that expression within *container*, so use this option cautiously. You may optionally precede *targetText* by `all {occurrences of}` or `every {occurrence of}` in this case if you like, which makes the impact clearer.

If the `first` or `last` options are used, *howMany* will specify the number of occurrences of *targetText* that should be deleted, starting at either the beginning or end of *container* respectively.

Tech Talk

If *ordinalTerm* or *ordinalNumber* is given, only a single occurrence of *targetText* will be deleted. The *ordinalTerm* should be an ordinal number, such as `first`, `second`, `third`, and so forth, or one of the terms `middle` (or `mid`), `penultimate`, `last`, or `any`. The *ordinalNumber* should be an expression which evaluates to a number. If it is negative, the `replace` command will count backward from the end of *container* to determine which occurrence to delete.

If `considering case` is specified, only occurrences of *targetText* within container that match exactly will be considered for deletion. The default is to delete any occurrence of *targetText* regardless of case.

The `delete` command sets a result (as returned by `the result` function) that indicates the number of occurrences that were deleted.

See Also: the `delete variable` command in Containers, and the `delete file` command in Working with Files and File Systems.

◊ format function

What it Does

Returns a formatted text representation of any number of values, as defined by a template string. The template consists of text intermixed with special formatting codes to specify such things as numbers formatted with a defined number of decimal places, text values formatted with extra spaces to fill a defined minimum length, and more.

Examples

```
format("The interest rate is %3.2f", interestRate)
format(reportTemplate, day(date), month(date), description, amount)
format("%x", maskValue) -- converts maskValue to hexadecimal
```

Tech Talk

Syntax: format(*template, value1, value2, ...***)**

The *template* string can include any format codes supported by the standard Unix printf command, as summarized below. In addition, certain “escape sequences” beginning with a backslash character are translated as follows: `\e` — escape character; `\a` — bell character; `\b` — backspace character; `\f` — formfeed character; `\n` — newline character; `\r` — carriage return character; `\t` — tab character; `\v` — vertical tab character; `\'` — single quote character; `\\` — backslash character; `\`*num* — character whose ASCII value is the 1-, 2-, or 3-digit octal number *num*.

A format code begins with a percent sign (`%`) followed by optional modifiers to indicate the length and number of decimal places for that value, and ends with a letter (`d`, `i`, `u`, `o`, `x`, `X`, `f`, `e`, `E`, `g`, `G`, `b`, `c`, `s`, `a`, `A`, or `@`) that specifies the type of formatting to be done (see the table below). Two percent signs in a row (`%%`) can be used to produce a percent sign in the output string.

Following the percent sign, and before the letter code, a format may include a number indicating the output length

for this value. The length may be followed by a decimal point (‘.’) and another number indicating the “precision” — this is the number of decimal places to display for a number, or the maximum number of characters to display from a string. Either the length or precision may be replaced by an asterisk (‘*’) to indicate that that value should be read from an additional parameter.

Before the length, the format code may also include any of the following modifier codes as needed:

- ‘–’ — a minus sign indicates the value should be left-aligned within the given length
- ‘+’ — a plus sign indicates that signed number formats should display a plus sign for positive numbers
- ‘ ‘ — a space indicates that signed number formats should include an extra space for positive numbers
- ‘0’ — a zero indicates that leading zeros (rather than spaces) should be used to fill the specified length
- ‘#’ — a pound sign affects specific numeric formats in different ways as described below

The following table lists the format codes that are recognized, their meaning, and examples:

d or i	signed (positive or negative) decimal integer (‘#’ has no effect):
	`format("%4d", 27)` —> `" 27"`
	`format("%+-4d", 27)` —> `"+27 "`
	`format("%04i", 27)` —> `"0027"`

u	unsigned (must be positive) decimal integer (‘#’ has no effect): `format("%u", 27)` —> `"27"`
o	unsigned octal integer (‘#’ increases precision to force a leading zero): `format("%#o", 27)` —> `"033"`

x or X	unsigned hexadecimal integer (‘#’ prepends ‘0x’ or ‘0X’ before a non-zero value):
	`format("%x", 27)` —> `"1b"`
	`format("%#X", 27)` —> `"0X1B"`

f	signed fixed-precision number (‘#’ forces decimal point to appear, even when there are no digits to the right of the decimal point):
	`format("%f", 27)` —> `"27.000000"` (default precision is 6 decimal places)
	`format("%7.3f", 27.6349)` —> `" 27.635"`
	`format("%+*.*f", 7, 2, 27.6349)` —> `" +27.63"`
	`format("%#-5.0f", 27)` —> `"27. "`

e or E	signed number in exponential notation with 'e' or 'E' before the exponent (‘#’ forces decimal point to appear, even when there are no digits to the right of the decimal point)
	`format("%e", 27)` —> `"2.700000e+01"`
	`format("%9.2E", 0.04567)` —> `" 4.57E-02"`

g or G	signed number in fixed (the same as 'f') or exponential (the same as 'e' or 'E') notation, whichever gives full precision in less space ('#' forces decimal point to appear, even when there are no digits to the right of the decimal point; trailing zeros are not removed)
	`format("%g", 27)` —> `"27"`
	`format("%+g", 0.04567)` —> `"+0.04567"`

c	single character `format("%-2c", "hello")` —> `"h "`

s	text string
	`format("%6s", "hello")` —> `" hello"`
	`format("%-3.2s", "hello")` —> `"he "`

b	text string with backslash escape sequences expanded `format("%b", "\tHello\\")` —> `"  Hello\"`
a or A	signed number printed in scientific notation with a leading 0x (or 0X) and one hexadecimal digit before the decimal point using a lowercase p (or uppercase P) to introduce the exponent *(not available on Windows)*
@	any value, displayed in its usual text format

See Also: the `merge` function, later in this section.

◊ join command and function

What it Does

The `join` command (or its synonym `combine`) joins the items of a list, or the properties of a property list, into a single text value, using the specified delimiters. If the source value is a container, that container will receive the text string resulting from this command. If the source value is *not* a container, the variable `it` will receive the text that results from combining the elements of the source value. The `with quotes` option may be used to enclose each value in quotes.

When called as a function, `join` (or `combine`) returns the joined text value, for convenient use as part of an expression.

Examples

```
join (1,2,3) using ":" -- note: the source value is not a container
put it -- 1:2:3
combine (1,2,3) using ":" with quotes
put it -- "1":"2":"3"
put (1,2,3) into numbers -- start with a list in a container
join numbers using ":" with quotes ("<<",">>")
put numbers -- <<1>>:<<2>>:<<3>>
combine (a:12,b:5) using ";" and "/"
put it -- a/12;b/5
join (a:12,b:5) with "##" using quotes -- only 1 delimiter
put it -- "12"##"5"
```

Tech Talk

Command Syntax:
[join | combine] *source* **{[using | with | by]** *listDelimiter* **{and** *keyDelimiter***}**
{[using | with] quotes {*quotes***}} }**

Function Syntax:
[join | combine] (*source* **{,** *listDelimiter* **{,** *keyDelimiter* **{,** *quotes***} } })**

If *source* is a container (a variable), that variable becomes a text value with the results of the join operation. Otherwise, the variable `it` receives the resulting text.

If *source* is a list, its items are combined using *listDelimiter* between the items. If source is a property list and only a *listDelimiter* is given, the property values are combined using *listDelimiter* between the values. If both *keyDelimiter* and *listDelimiter* are given, the result will contain the keys (property names) of each property before its value, using *listDelimiter* to separate each entry, and *keyDelimiter* to separate each key from its value within an entry.

If neither *listDelimiter* nor *keyDelimiter* is specified, the items or properties of source will be combined into a text string, using the current values of the `listSeparator`, `plistEntrySeparator`, and `plistKeySeparator` properties as appropriate. This applies to nested lists and property lists as well.

If `using quotes` or `with quotes` is specified, each value will be enclosed in quotation marks. If a quotes value is supplied, it is used instead of standard double *quote* marks. If *quotes* is a list, the first item will be used before each value, and the second item after each value.

See Also: the `split` command and function, later in this section, and the `joined by` operator in Expressions.

◊ length function

What it Does

Returns the number of characters in some text.

Examples

```
put length("SenseTalk")  -- 9
if the length of name is less than 3 \
        or the length of name is greater than 12 then
   put "Invalid name!"
end if
```

Tech Talk

Syntax: {the} length of *textFactor*
length(*textExpr***)**

◊ lowercase (or toLower) function

What it Does

The `lowercase` function (and its synonym `toLower`) returns an expression converted to all lowercase (non-capital) letters.

Examples

```
put lowercase of "Hi There!"  -- shows "hi there!"
```

Tech Talk

Syntax: {the} lowercase of *stringFactor*
lowercase(*stringExpr***)**

See Also: the `uppercase` and `capitalized` functions, elsewhere in this section.

◊ merge function

What it Does

Scans a source text and performs evaluation and substitution of expressions enclosed in merge delimiters, returning the merged text. The standard merge delimiters are double square brackets ("[[" and "]]"). Any expressions within the source text enclosed in [[and]] are replaced by their value. Expressions are evaluated in the current context, so any local or global variables within a merge expression will have the same value they have in the handler which called the merge function.

In addition to simple expressions, the text between merge delimiters may include any SenseTalk statements. This can be very powerful, for such things as including repeat loops around portions of your text.

Examples

```
put "Jorge" into name
put merge("Hello [[name]], how are you?") -- displays "Hello Jorge, how are
you?"
put merge of "[[repeat with n=0 to 99]][[n]],[[end repeat]]" \
    into numbersList
put merge(template) into file "/Library/WebServer/Documents/report.html"
```

Tech Talk

Syntax: merge(*source, delimiter1, delimiter2***)**

The last two parameters are optional: only a *source* string is required. If *delimiter1* and *delimiter2* are given, they are used in place of "[[" and "]]" as the merge delimiters. If only *delimiter1* is given, its value is used for both the opening and closing delimiters. If the `evaluationContext` property is set to "Global" or "Universal" then variables will be treated as global or universal rather than local during the merge process.

◊ numToChar function

What it Does

Returns the character represented by a given numeric code (in Unicode).

Examples

```
put numToChar(97)  --shows “a”
```

Tech Talk

Syntax: **{the} numToChar of** *numFactor*
numToChar(*numExpr***)**

See Also: the `charToNum` function, earlier in this section.

◊ offset, range functions

What it Does

The `offset()` function returns the offset location of a target text within a source text, or the offset of a target value or list of values within a source list. The `range()` function works identically but returns the range (start and end locations) where the target is found in the source. If the target value being searched for is not found in the source, the number 0 or empty range 0 to 0 is returned. Three optional parameters to `offset()` or `range()` allow them to find occurrences of a substring beyond a certain point, to control whether the search is case-sensitive or not, and to search backwards from the end of the source string.

A natural syntax is also available for writing an offset or range expression as an English-like phrase.

Examples

```
put offset("the", "Hi there!")  -- 4
put range("the", "Hi there!")  -- 4 to 6
get the offset of "eggs" within recipe
put offset("th", "Hi there, from Thoughtful Software.", 5, FALSE)
    -- 16 ("th" matches "Th" in Thoughtful)
put offset("th", "Hi there, from Thoughtful Software.", 5, TRUE)
    -- 0 (case-sensitive: not found, "th" does not match "Th"-- and search starts
after the "th" in "there")
put offset of "th" within "Hi there, from Thoughtful Software" \
         after 5 considering case -- 0 (same as above, using natural syntax)
put offset of "th" within "Hi there, from Thoughtful Software" \
         before end -- 16 (backwards search)

put the range of "cat" in "concatenation" into catRange
put catRange -- 4 to 6
put chars catRange of "concatenation" -- "cat"

put the range of (c,d,e) in (a,b,c,d,e,f,g) -- 3 to 5
put the range of (c,d,e) in (a,b,c,d,X,e,f,g) -- 0 to 0
```

Tech Talk

Syntax: **offset**(*target, source*)
offset(*target, source, beyondPosition, caseSensitive, reverse*)
range(*target, source*)
range(*target, source, beyondPosition, caseSensitive, reverse*)

Natural Syntax:
{the} offset of *targetFactor* **[in | within]** *sourceFactor* **{ [before | after]** *[beyondPosition* **| {the} end] } {considering case | ignoring case}**
{the} range of *targetFactor* **[in | within]** *sourceFactor* **{ [before | after]** *[beyondPosition* **| {the} end] } {considering case | ignoring case}**

The *beyondPosition, caseSensitive*, and *reverse* parameters are optional: only a *target* string or value (to search for) and a *source* string or list (to search in) are required.

If *beyondPosition* is specified, the search will begin at the next character of the source string (or next item of the source list) beyond that position. To search from the beginning of the source, a value of zero should be used (the default). The value returned by the function is always the position (or range) where the target string was found within the source string, or zero if it was not found. If *beyondPosition* is given as a negative number, it indicates a position relative to the end of the source. In this case the return value will also be given as a negative number indicating the offset from the end of the source.

The *caseSensitive* parameter (if specified) should evaluate to true or false. If it is true, the search will only find exact case matches. The default (if not specified) is false. If the natural syntax is used, the search will be case sensitive if `considering case` is specified and case insensitive if `ignoring case` is specified. The default is to ignore case differences.

The *reverse* parameter (if specified) should evaluate to true or false. Using `before` in the natural syntax is equivalent to setting *reverse* to true. If it is true, a reverse search will be performed, starting at the end of the source (or at the character before *beyondPosition*) and searching toward the beginning. The default (if not specified) is false.

◊ replace command

What it Does

The `replace` command replaces one or more occurrences of an old (target) text string within a container with a new (replacement) text string. In its simplest form, it will replace every occurrence of the old text with the new text, regardless of case. More advanced options allow you to specify how many occurrences — or even to indicate a particular occurrence — of the old text to replace, and to specify exact case matching.

Examples

```
replace "Johnson" by "Johansson" in bigReport
replace the last "s" in word 3 of sentence with empty
replace every occurrence of " he " in manual with " she "
replace the first 2 "i" in phrase considering case by "I"
```

Tech Talk

Syntax: replace *Options*

Options:

```
[in | within] container
{ all {occurrences of} | every {occurrence of} } oldText
{the} [first | last] howMany {occurrences of} oldText
{the} ordinalTerm {occurrence of} oldText
occurrence ordinalNumber of oldText
[with | by] newText
[with | considering] case
[without | ignoring] case
```

A number of options must be specified as part of the `replace` command. Both *oldText* and *newText* must be supplied, as well as the *container* where the substitutions will occur. Other options are optional. The options may be specified in any order, although only one option of each type may be given.

You must include the `in container` or `within container` option in any replace command. The *container* can be any container, including a variable, a portion of a variable (using a chunk expression), or a text file. If *container* is a variable, its contents may be of any sort, including a list or property list. The `replace` command will replace text in all values nested to any depth within such containers.

You must include the `with newText` or `by newText` option to supply the new text that will be substituted for the selected occurrences of the old text within the container.

You must include one of the *oldText* options in a replace command, to specify what will be replaced. Simply providing an *oldText* expression will cause the command to locate every occurrence of the value of that expression within container, and replace each one with the value of *newText*. You may optionally precede *oldText* by `all {occurrences of}` or `every {occurrence of}` if you like.

If the `first` or `last` options are used, *howMany* will specify the number of occurrences of *oldText* that should be replaced, starting at either the beginning or end of *container* respectively.

If *ordinalTerm* or *ordinalNumber* is given, only a single occurrence of *oldText* will be replaced by *newText*. The *ordinalTerm* should be an ordinal number, such as `first`, `second`, `third`, and so forth, or one of the terms `middle` (or `mid`), `penultimate`, `last`, or `any`. The *ordinalNumber* should be an expression which evaluates to a number. If it is negative, the `replace` command will count backward from the end of *container* to determine which occurrence to replace.

If `considering case` is specified, only occurrences of *oldText* within *container* that match exactly will be considered for replacement. The default is to match regardless of case.

The `replace` command sets a result (as returned by `the result` function) that indicates the number of occurrences that were replaced.

See Also: Chunk Expressions, for other ways to replace portions of text.

◊ rtfToText function

What it Does

Converts rich text in the RTF format to plain text.

Examples

```
put the rtfToText of file "Picasso.rtf" into picassoText
```

Tech Talk

Syntax: **{the} rtfToText of** *richText*
rtfToText(*richText***)**

The value of *richText* should be text encoded in the RTF format. The `rtfToText` function removes all of the formatting information and returns just the plain text contents from *richText*.

◊ sort command

What it Does

The `sort` command sorts the contents of a container.

Examples

```
sort nameList
sort ascending items delimited by tab of phoneList
sort the lines of file "scores" numerically \
   in reverse order by the last item of each
sort the first 7 items of deck by each.rank
sort jobs by (city of each, salary of each)
```

Tech Talk

Syntax: sort {*Options***} {by** *sortByExpr***}**

Tech Talk

Options:

{ {the} {[first | last] *count*} *chunkTypes* of } *containerToSort*
{in} [ascending | descending | reverse | reversed] {order}
[numerically | {in} [numeric | numerical] {order}]
[chronologically | {in} [dateTime | chronologic | chronological] {order}]
[alphabetically | {in} [text | alphabetic | alphabetical] {order}]
[with | considering] case
[without | ignoring] case

A number of options may be specified as part of the `sort` command. Only the *containerToSort* is required. The additional options are optional, and may be specified in any order, although only one sort order and one comparison type may be given.

If *chunkTypes* is specified, the sort will be performed on the chunks of that type (characters, words, text items, lines, or list items) within the *containerToSort*. If not specified, the items of the container will be sorted (either list items if it is a list, or text items). A delimiter may be specified for items, lines, or words.

If `ascending` or `descending` (or `reversed`) is specified, the result of the sort will be arranged by increasing or decreasing values, respectively. If the order is not specified, the sort will be performed in ascending order.

The comparison type used in the sort – whether numeric, alphabetic, or chronologic – can be indicated in several different ways. If the comparison type is not specified, the container will be sorted by comparing textual values, ignoring distinctions between uppercase and lowercase letters.

If `numerically` or one of its variant forms is specified, values are sorted based on their numeric values. If `chronologically` or one of its variants is specified, values are sorted by evaluating them as a particular time on a given date. Dates without specific times are treated as noon on that date. Times without dates are assumed to refer to the current date (today).

If `alphabetically` or one of its variants is specified (or if no comparison type is given) you may indicate whether or not case should be considered in the comparisons by specifying `considering case` or `ignoring case`.

Finally, if `by sortByExpr` is specified, it must come at the end of the command, following any other options. The *sortByExpr* is usually an expression involving the special variable `each` which refers to each of the elements being sorted.

For example, if the variable nameList in your script contains a list of people's names in which each name consists of a first name and a last name separated by a space, then `sort the items of nameList` would sort the list alphabetically by the entire name of each person, resulting in a list in first-name order. The command `sort the items of nameList by the last word of each`, however, would sort the list in order by last names.

Tech Talk

When two or more elements being sorted have identical values, the `sort` command leaves their order unchanged. This enables several sort commands to be used in sequence to do sub-sorting. Note that the sort operations must be done in reverse order, beginning with any sub-sorts, since the final sort will determine the primary order of the result. For example, given a list of objects representing people, where each object has both firstName and lastName properties, the following code will sort them into order by last name, and among those with the same last name, they will be sorted by first name:

```
sort people by the firstName of each
sort people by the lastName of each
```

Another way to do sub-sorting, with the only limitation being that every level of the sort is done in the same order (ascending or descending), is to sort by a list of elements, beginning with the primary element:

```
sort people by (the lastName of each, the firstName of each)
```

◊ split command and function

What it Does

The `split` command splits text into a list of values, or into a property list of keys and values, by separating the text at specified delimiters. If the source value is a container, that container will become a list (or property list) as a result of this command. If the source value is not a container, the variable `it` will receive the list (or property list) resulting from splitting the source value.

When called as a function, `split` returns the resulting list or property list, for convenient use as part of an expression.

Examples

```
split "apple;banana;orange;cherry" by ";"
put it -- (apple,banana,orange,cherry)
put "x=7;y=2" into coordinate
split coordinate using ";" and "="
put coordinate -- (x:7, y:2)
split "a/2, b/15, a/9" by ", " and "/"
put it -- (a:(2,9), b:15)
put split(filePath, "/") into pathComponents
```

Tech Talk

Command Syntax:
split *sourceText* **{[by | using | with]** *listDelimiter* **{and** *keyDelimiter***}}**

Function Syntax:
split (*sourceText* **{,** *listDelimiter* **{,** *keyDelimiter***}})**

Tech Talk

If *sourceText* is a container (a variable), that variable becomes a list or property list containing the results of the split operation. Otherwise, the variable `it` receives the resulting list or property list.

If *listDelimiter* is given, but *keyDelimiter* is not, the result will be a list, splitting the *sourceText* at each occurrence of *listDelimiter* to obtain the list items. If both *keyDelimiter* and *listDelimiter* are given, the result will be a property list, using *listDelimiter* to separate each entry, and *keyDelimiter* to separate each key from its value within an entry. If neither *listDelimiter* nor *keyDelimiter* is specified, the sourceText will be split into a list, using the current value of the `itemDelimiter` property as the delimiter.

When creating a property list, if the same key occurs more than once in *sourceText*, the value associated with that key will become a list containing all of the values found for that key (see the final Example above).

See Also: the `join` command and function, earlier in this section, and the `split by` operator in Expressions.

◊ standardFormat function

What it Does

The `standardFormat` function returns a text representation of any value, in a format suitable for archiving. For any type of value, this function should return a text value that can be supplied as the parameter to the `value()` function to retrieve the original value.

Examples

```
put standardFormat(97)  -- "97"
put the standardFormat of (1,"cow",2+3) -- {"1", "cow", "5"}
put standardFormat of (name:Jack, age:17) -- {age:"17", name:"Jack"}
put standardFormat(Jack & return & Jill) -- "Jack" & return & "Jill"
put standardFormat(Jack & quote & Jill) -- <<Jack"Jill>>
```

Tech Talk

Syntax: **{the} standardFormat of** *factor*
standardFormat(*expr***)**

In general, `standardFormat()` will perform “Standard” quoting of values, and force the use of standard formats for lists, property lists, and trees. However, you shouldn’t rely on the value returned by this function to always be in a specific format — the exact format may change from one version of SenseTalk to another, and may also depend on other factors, including the current internal representation of the value. It should always be true, though, that `value(standardFormat(someValue))` is equal to `someValue`.

◊ standardFormat function

See Also: `the defaultQuoteFormat`, `the listFormat`, and `the propertyListFormat` global properties.

◊ text, asText function

What it Does

The `text` function (or `asText`) returns the value of its parameter in text format.

Examples

```
set the numberFormat to "00000"
put (4 * 13)'s text into formattedNumber    --"00052"
```

Tech Talk

Syntax: **{the} text of** *factor*
text(*expr***)**

See Also: the discussion of "Conversion of Values" and the `as` operator in Expressions.

◊ textDifference function

What it Does

Calculates the "Levenshtein Distance" between two words or phrases. This is one measure of how different two text strings are from each other. If the two values are equal, their textDifference will be zero. If one character must be changed to a different character, or one character inserted or deleted, in order to turn one string into the other, then the difference will be 1. A greater number indicates a greater difference between the two strings. The largest number that will be returned is the length of the longer string, which would indicate that the two strings are completely different. This function is case insensitive, unless the caseSensitive global property is set to true.

Examples

```
put textDifference("Hobbit", "habit")  -- 2
put textDifference("Hobbit", "hobo")  -- 3
put textDifference("cheeseburger", "squash")  -- 10
```

◊ textDifference function

Tech Talk

Syntax: textDifference(*string 1, string 2***)**

◊ uppercase, toUpper function

What it Does

The `uppercase` function (and its synonym `toUpper`) returns an expression converted to all uppercase (capital) letters.

Examples

```
put uppercase("Hi There!")  -- "HI THERE!"
```

Tech Talk

Syntax: {the} uppercase of *stringFactor*
uppercase(*stringExpr***)**

See Also: the `lowercase` and `capitalized` functions, earlier in this section.

Working with Numbers

SenseTalk supports mathematical operations through its mathematical operators, commands, and functions. The mathematical operators (+, -, *, etc.) are described in Expressions. This section documents the commands and functions which operate on numbers. It also describes the representation of geometric points and rectangles in SenseTalk, and some functions for working with them.

Arithmetic Commands and Functions

There are four arithmetic commands:

```
add
subtract
multiply
divide
```

Use them to modify values stored in containers. These commands perform the same arithmetic functions as the +, –, *, and / operators. The difference is that these commands take one of their operands from a container and store the result of the calculation back into that container.

◊ add

What it Does

The add command lets you add one number or list to another that is stored in a container.

When to Use It

Use the add command when you want to add a number to the value in a container, replacing the value in the container by the sum. Lists of values can be added, provided that both the source and destination lists contain the same number of items. Each item from the source list is added to the corresponding item of the destination container.

Examples

```
add amount to dollarsVariable
add 37 to item 2 of line 3 of scores
add speed * time to item 1 of distances
add (10,5) to centerPoint
```

Tech Talk

Syntax: add *numExpr* to {*chunk* **of**} *container*

NumExpr is a source expression. It can be a number, any formula, or another container. Chunk is a chunk expression describing part of a container (lines, words, items, or characters). Container is any container.

See Also: the subtract command, below.

◊ subtract

What it Does

The `subtract` command lets you subtract a value from a number that is stored in a container, or a list of values from a corresponding list of numbers.

When to Use It

Use the `subtract` command when you want to subtract a number from the value in a container, replacing the value in the container by the result.

Lists of values can be subtracted, provided that both the source and destination lists contain the same number of items. Each item from the source list is subtracted from the corresponding item of the destination container.

Examples

```
subtract checkAmt from accountBalance
subtract 1 from property CountDown of gameController
subtract rate * pmt from line 4 of amortization
subtract (1,1,2) from boxDimensions
```

Tech Talk

Syntax: subtract *numExpr* **from** {*chunk* **of**} *container*

NumExpr is a source expression. It can be a number, any formula, or another container. *Chunk* is a chunk expression describing part of a container (lines, words, items, or characters). *Container* is any container.

See Also: the `add` command, earlier in this section.

◊ multiply

What it Does

The `multiply` command lets you multiply a number that is stored in a container by another number. A list of values may be multiplied by another list or by a single (scalar) value.

When to Use It

Use the `multiply` command when you want to multiply a value in a container by another number, replacing the value in the container by the product.

Lists of values can be multiplied, provided that both the source and destination lists contain the same number of items, or that the source is a single value. Each item in the destination container is multiplied by the corresponding item of the source list, or by the source value.

Examples

```
multiply score by weightingFactor
multiply accountBalance by 1 + interestRate
```

```
multiply item 3 of line x of table by 2
```

Tech Talk

Syntax: multiply {*chunk* **of**} *container* **by** *numExpr*

NumExpr is a source expression. It can be a number, any formula, or another container. *Chunk* is a chunk expression describing part of a container (lines, words, items, or characters). *Container* is any container.

See Also: the `divide` command, below.

◊ divide

What it Does

The `divide` command lets you divide a number that is stored in a container by another number. A list of values may be divided by another list or by a single (scalar) value.

When to Use It

Use the `divide` command when you want to divide a value in a container by another number, replacing the value in the container by the quotient.

Lists of values can be divided, provided that both the source and destination lists contain the same number of items, or that the source is a single value. Each item in the destination container is divided by the corresponding item of the source list, or by the source value.

Examples

```
divide score by totalCount
divide item 1 of balances by 12
```

Tech Talk

Syntax: divide {*chunk* **of**} *container* **by** *numExpr*

NumExpr is a source expression. It can be a number, any formula, or another container. Chunk is a chunk expression describing part of a container (lines, words, items, or characters). Container is any container.

See Also: the `multiply` command, earlier in this section.

Arithmetic Functions

Use these functions to manipulate numbers in a variety of ways.

◊ abs function

What it Does

Returns the absolute value of its numeric parameter. The absolute value is the magnitude of a number regardless of its sign — it is always positive or zero.

Examples

```
put abs(-11) -- shows 11
if height is negative then put abs(height) into height
```

Tech Talk

Syntax: the abs of *numFactor*
abs(*numExpr***)**

◊ annuity function

What it Does

Calculates the present value of an ordinary annuity with payments of one unit, based on the specified interest rate per period and the number of periods.

Examples

```
put annuity(10%, 32) -- shows 9.526376
```

Tech Talk

Syntax: annuity(*interest, periods***)**

See Also: the compound function, later in this section.

◊ atan function

What it Does

Returns the trigonometric arctangent of its parameter as an angle expressed in radians.

Examples

```
put atan(19) -- shows 1.518213
```

Tech Talk

Syntax: the atan of *numFactor*
atan(*numExpr***)**

◊ average function

What it Does

Returns the average of its parameters.

Examples

```
put average(8, 10, 12) -- shows 10
if the average of (x, y, z) is greater than z then
   put "Z is below average!"
end if
```

Tech Talk

Syntax: {the} average of *numList*
average(*numList***)**

numList may either be a list of numbers, an expression which evaluates to a list of numbers separated by commas, or a combination of these, nested to any depth.

See Also: the median function, later in this section.

◊ compound function

What it Does

Computes the principal plus accrued interest on an investment of 1 unit, based on the specified interest rate and the number of periods.

Examples

```
put compound(7.25%, 6) -- shows 1.521892
put initialInvestment * compound(6.7%, 12) into currentValue
```

Tech Talk

Syntax: compound(*interest, periods***)**

See Also: the `annuity` function, earlier in this section.

◊ cos function

What it Does

Returns the trigonometric cosine of its parameter, which is an angle expressed in radians.

Examples

```
put cos(18) -- shows 0.660317
```

Tech Talk

Syntax: **{the} cos of** *numFactor*
cos(*numExpr***)**

◊ exp function

What it Does

Returns the natural exponential of its parameter (that is, the mathematical constant **e** raised to the power of the parameter).

Examples

```
put exp(2) -- 7.389056
```

Tech Talk

Syntax: **{the} exp of** *numFactor*
exp(*numExpr***)**

◊ exp1 function

What it Does

Returns one less than the natural exponential of its parameter (that is, the mathematical constant e raised to the power of the parameter, minus 1).

Examples

```
put exp1(2) -- 6.389056
```

Tech Talk

Syntax: **{the} exp1 of** *numFactor*
exp1(*numExpr***)**

◊ exp2 function

What it Does

Returns 2 raised to the power of its parameter.

Examples

```
put exp2(8) -- 256
```

Tech Talk

Syntax: **{the} exp2 of** *numFactor*
exp2(*numExpr***)**

◊ frac function

What it Does

Returns the fractional part of a number. Use the `trunc()` function to get the whole number part of a value.

Examples

```
put frac(81.236) -- .236
```

Tech Talk

Syntax: **{the} frac of** *numFactor*
frac(*numExpr***)**

Note: `Trunc()` and `frac()`

The `trunc()` *and* `frac()` *functions are defined such that trunc(x) + frac(x) is always equal to x.*

◊ ln function

What it Does

Returns the natural logarithm of its parameter.

Examples

```
put ln(2) -- 0.693147
```

Tech Talk

Syntax: **{the} ln of** *numFactor*
ln(*numExpr***)**

◊ ln1 function

What it Does

Returns the natural logarithm of 1 more than its parameter.

Examples

```
put ln1(2) -- 1.098612
```

Tech Talk

Syntax: **{the} ln1 of** *numFactor*
ln1(*numExpr***)**

◊ log2 function

What it Does

Returns the base 2 logarithm of its parameter.

Examples

```
put log2(256) -- 8
```

Tech Talk

Syntax: **{the} log2 of** *numFactor*
log2(*numExpr***)**

◊ maximum, max function

What it Does

Returns the highest number from a list. The `maximum` function may be abbreviated as `max`.

Examples

```
put max(4, 6, 5, 7, 3) -- 7
if the maximum of (x, y, z) is z then
    put "Z is the greatest!"
end if
```

Tech Talk

Syntax: **{the} max{imum} of** *numList*
max{imum}(*numList***)**

numList may be a list of numbers, an expression which evaluates to a list of numbers separated by commas, or a combination of these, nested to any depth.

See Also: the `minimum` function, below.

◊ median function

What it Does

Returns the median (middle value) of its parameters, or the average of the two middle values.

Examples

```
put median(1, 1, 8, 9, 12) -- 8
put the median of "2,7,8,10" -- 7.5
```

Tech Talk

Syntax: **{the} median of** *numList*
median(*numList***)**

numList may either be a list of numbers, an expression which evaluates to a list of numbers separated by commas, or a combination of these, nested to any depth. If numList contains an odd number of numbers, the median is the middle value of the sorted list of numbers, otherwise it is the average of the two middle values.

See Also: the `average` function, earlier in this section.

◊ minimum, min function

What it Does

Returns the lowest number from a list. The `minimum` function may be abbreviated as `min`.

Examples

```
put min(4, 6, 5, 7, 3) -- 3
if the min of (x, y, z) is z then put "Z is the smallest!"
```

Tech Talk

Syntax: **{the} min{imum} of** *numList*
min{imum}(*numList***)**

numList may be a list of numbers, an expression which evaluates to a list of numbers separated by commas, or a combination of these, nested to any depth.

See Also: the `maximum` function, earlier in this section.

◊ random function

What it Does

Returns a randomly generated integer between 1 and the value of its parameter, or between two values.

Examples

```
put random(12) -- returns any number from 1 to 12
put random(20,30) -- gets a number from 20 to 30, inclusive
put (random(100) / 100) into randomPercentage
```

Tech Talk

Syntax: **{the} random of** *numFactor*
random(*numExpr* **{,** *secondExpr***}** **)**

Use the `reset random` command (below) to change the sequence of random numbers.

◊ reset random command

What it Does

Resets the random number generator sequence used by the `random` function and whenever SenseTalk selects things at random.

Examples

```
reset random with seed 27
reset random
```

Tech Talk

Syntax: reset random {{with | from} {seed} *seedExpr***}**

By setting a specific *seedExpr* value for the random number generator, you can obtain a repeatable sequence of "random" events. This can be very useful for testing purposes. Use the `reset random` command without a seed value to get an unpredictable sequence.

◊ round function

What it Does

Returns the value of its parameter rounded to the nearest whole number. An optional second parameter may be supplied to specify the number of decimal places to round to. A negative number of places will round to the left of the decimal point.

Examples

```
put round(6.5)  -- 7
put round(6.49) -- 6
put round(6.49 , 1) -- 6.5
put round(2389 , -2) -- 2400
```

Tech Talk

Syntax: {the} round of *numFactor*
round(*numExpr, decimalPlaces***)**

See Also: the `roundToNearest` function, below, and the `rounded to` operator in Expressions.

◊ roundToNearest function

What it Does

Returns the value of its first parameter rounded to the nearest whole multiple of its second parameter.

Examples

```
put roundToNearest(643,100) -- 600
put roundToNearest(643,25) -- 650
```

Tech Talk

Syntax: roundToNearest(*numExpr, nearestMultiple***)**

See Also: the `round` function, above, and the `rounded to nearest` operator in Expressions.

◊ sin function

What it Does

Returns the trigonometric sine of its parameter, which is an angle expressed in radians.

Examples

```
put sin(18) --  -0.750987
```

Tech Talk

Syntax: {the} sin of *numFactor*
sin(*numExpr***)**

◊ square root , sqrt function

What it Does

Returns the square root of its parameter.

Examples

```
put sqrt(16) -- 4
put the square root of nine -- 3
```

Tech Talk

Syntax: {the} square root of *numFactor*
{the} sqrt of *numFactor*
sqrt(*numExpr***)**

◊ sum function

What it Does

Returns the sum of its parameters.

Examples

```
put sum("8,1", (10,11), 12) -- shows 42
if the sum of (x, y, z) is more than 100 then
    put "The sum exceeds 100"
end if
```

Tech Talk

Syntax: **{the} sum of** *numList*
sum(*numList***)**

numList may be a list of numbers, an expression which evaluates to a list of numbers separated by commas, or a combination of these, nested to any depth.

◊ tan function

What it Does

Returns the trigonometric tangent of its parameter, which is an angle expressed in radians.

Examples

```
put tan(18) --  -1.137314
```

Tech Talk

Syntax: **{the} tan of** *numFactor*
tan(*numExpr***)**

◊ trunc function

What it Does

Truncates a number, returning the integer part of its parameter, and discarding any fractional part. Use the `frac` function to get the fractional part of a value.

Examples

```
put trunc(6.8) -- 6
```

```
put trunc(6.49) -- 6
```

Tech Talk

Syntax: **{the} trunc of** *numFactor*
trunc(*numExpr***)**

Note: `Trunc()` **and** `frac()`

The `trunc()` *and* `frac()` *functions are defined such that trunc(x) + frac(x) is always equal to x.*

Points and Rectangles

SenseTalk understands the concepts of geometric points and rectangles. Any list of two numbers, or a text string consisting of two numbers separated by a comma, can be treated as a point. The first number represents the X coordinate of the point, and the second number the Y coordinate:

```
put (150,47) into pointA
put "250,98" into pointB
```

A rectangle can be represented by any list of two points, by a list of four numbers or by a text string consisting of four numbers separated by commas (representing two points). The two points indicate two opposite corners of the rectangle (either the top-left and bottom-right corners, or the bottom-left and top-right corners, listed in either order).

```
put (pointA,pointB) into myRect
put "15,28,19,72" into bounds
```

The `is within` operator can be used to test whether a point lies within a rectangle or one rectangle is completely within another (see the full description in Expressions):

```
if (18,35) is within bounds then scoreHit
```

◊ x, y, width, height, origin, and size functions

What it Does

These functions can be used to extract the various component values of a point or rectangle. The functions `x()` and `y()` can be used with points and rectangles to obtain the x and y coordinates of the point, or of the origin point of the rectangle. The `origin()` and `size()` functions can be used with rectangles to obtain the origin point (the minimum x and y values) and the size (a list of two numbers representing the width and height, respectively) of the rectangle. The `width()` and `height()` functions can be used with rectangles or sizes to obtain the width or height.

Examples

These functions are most commonly used with the dot (.), apostrophe-S (‘s) or “of” syntax, similar to accessing a property of an object, but keep in mind that these functions return read-only values only:

```
put pointA.x into horizontalLocation
```

```
put the y of pointA into verticalLocation
put myRect's origin into topLeftCorner
put the size of bounds into outerRect
put height of imageRect into verticalSpaceNeeded
```

Tech Talk

Syntax:

{the} x of *pointOrRectangle*	**x(***pointOrRectangle***)**
{the} y of *pointOrRectangle*	**y(***pointOrRectangle***)**
{the} width of *rectangle*	**width(***rectangle***)**
{the} height of *rectangle*	**height(***rectangle***)**
{the} origin of *rectangle*	**origin(***rectangle***)**
{the} size of *rectangle*	**size(***rectangle***)**

◊ top, bottom, left, right, topLeft, topRight, bottomLeft, bottomRight, center, topCenter, bottomCenter, leftCenter, and rightCenter functions

What it Does

These functions can be used to find the coordinates of various parts of a rectangle.

The `top()`, `bottom()`, `left()` and `right()` functions return a single number which is the y coordinate of the top or bottom edge, or the x coordinate of the left or right edge, respectively, of a rectangle.

The `topLeft()`, `topRight()`, `bottomLeft()` and `bottomRight()` functions return the coordinates of the point at the indicated corner of a rectangle.

The `center()` function returns the coordinates of the center of a rectangle, and the `topCenter()`, `bottomCenter()`, `leftCenter()` and `rightCenter()` functions return the coordinates of the point at the center of the indicated edge of a rectangle.

Examples

These functions are most commonly used with the dot (.), apostrophe-S ('s) or "of" syntax, similar to accessing a property of an object, but keep in mind that these are not properties, but functions which return read-only values:

```
put boundingBox.top into highestEdge
put the bottomRight of doorFrame into anchorPoint
put myRect's center into centerPoint
put the leftCenter of bounds into alignmentPoint
```

Tech Talk

Syntax:

{the} top of *rectangle*	**top(***rectangle***)**
{the} bottom of *rectangle*	**bottom(***rectangle***)**
{the} left of *rectangle*	**left(***rectangle***)**
{the} right of *rectangle*	**right(***rectangle***)**
{the} topLeft of *rectangle*	**topLeft(***rectangle***)**
{the} topRight of *rectangle*	**topRight(***rectangle***)**
{the} bottomLeft of *rectangle*	**bottomLeft(***rectangle***)**
{the} bottomRight of *rectangle*	**bottomRight(***rectangle***)**
{the} center of *rectangle*	**center(***rectangle***)**
{the} topCenter of *rectangle*	**topCenter(***rectangle***)**
{the} bottomCenter of *rectangle*	**bottomCenter(***rectangle***)**
{the} leftCenter of *rectangle*	**leftCenter(***rectangle***)**
{the} rightCenter of *rectangle*	**rightCenter(***rectangle***)**

Working with Dates and Times

This section describes the commands and functions that SenseTalk provides for working with dates and times. One date or time can be subtracted from another using the minus (`-`) operator, to get their difference in seconds. A time interval may also be added to a date or time to obtain a new date/time value (see Time Intervals in Values).

Dates, Times, and Time Intervals

SenseTalk makes no fundamental distinction between a “date” and a “time” — both are treated as precise instants in the flow of time, or points along a time-line whose origin (zero value) was at the stroke of midnight at the beginning of January 1, 2001, Coordinated Universal Time. Any date or time before that instant is treated internally as a negative value, and later times as a positive value indicating the number of seconds since the origin.

SenseTalk can recognize dates and times expressed in a wide variety of formats such as "4/22/67" or "1967-04-22 18:00" (see `the timeInputFormat` later in this section for details). A "Natural" format allows even more variations, such as "May 15, 2004 10:57 PM", or even "yesterday", "today", or "next Tuesday in the afternoon". The words `today` and `now` (without enclosing quotes) can be used to indicate the current date or the current date and time (unless they are used as variables and assigned some other value).

Whenever a date value is supplied without a time of day, it is taken to mean noon of that day. When a time is given without a date, it is assumed to mean the indicated time on the current date (today). All dates and times are assumed to be in the local time zone, as currently set on the machine where the script is running.

A “time interval” is a length of time. SenseTalk always measures time intervals in seconds, but provides time interval expressions using the words `weeks`, `days`, `hours`, etc. to make it easy to express times more naturally (see Time Intervals in Values). Time intervals can be used with the `ago` and `hence` operators to produce a time value that is a specific length of time in the past or future.

Most of the SenseTalk date/time functions return a value that is not merely a representation of a point in time, but one that also encapsulates a time format. Such a “formatted date/time value” has the advantage that it will retain the same format when date/time arithmetic is performed on it.

> **Note: Date/time values**
>
> Because there is no real difference between dates and times, this manual sometimes refers to either a date or a time as “a date/time value”.

Date/Time Arithmetic

Adding and Subtracting Time Intervals

Starting from any date (or time), you can obtain a different date/time by simply adding or subtracting a time interval:

```
put "today" + 2 weeks into dueDate
subtract 3 hours 14 minutes 22 seconds from timer
```

Calculating Date or Time Differences

By subtracting one date or time value from another, you can easily calculate the number of days between dates or the elapsed time for some process. The difference is always a time interval expressed in seconds, but you can convert it to a different unit (such as days) by dividing it by the number of seconds in that unit (which can be expressed using a time interval expression such as `1 day`, for example):

```
put (expirationDate - "today") / 1 day into daysRemaining
put the time into startTime -- start timing here
run "somethingTimeConsuming" -- whatever you want to time
put the time - startTime into secondsElapsed
```

Date or Time Comparisons

The SenseTalk comparison operators ("is", "=", "comes before", "<", and the like) ordinarily treat the two values being compared as text, unless they are both numbers or it "knows" they are both date or time values. Because comparisons usually treat values as text, the following will not produce the desired result:

```
if the date is between "Sep 21" and "Dec 21" then put "Happy Autumn"
```

To persuade SenseTalk to perform date or time comparisons, use the date() or time() functions to convert the text to an internal date/time representation. This will work (note that when the year isn't specified, the current year is assumed, so this example will work in any year):

```
if the date is between date("Sep 21") and date("Dec 21") then
   put "Happy Autumn!"
end if
```

◊ date, asDate functions

What it Does

Returns the current date, or the date value for a given expression. The `long date` function returns a verbose version of the date, including the current day of the week and the full name of the month. `Abbreviated date` and `short date` variants provide the date in other formats.

Examples

```
put the date -- "10/07/95"
put the short date -- "10/7/95"
put the abbrev date -- "Sat, Oct 7, 1995"
put the long date -- "Saturday, October 7, 1995"
put date("May 14, 1942") -- "05/14/42"
put asDate("May 14, 1942") -- "May 14, 1942"
```

Tech Talk

Syntax: **the { [long | short | abbr{ev{iated}}] } date {of** *dateExpr*}
date(*dateExpr***)**
asDate(*dateExpr***)**

Tech Talk

The value returned by the `date` function, when converted to text, will automatically display a formatted date, as shown in the Examples. Its value may also be treated as a number, representing the exact date and time when the function was called, for use in date/time calculations. When *dateExpr* is given, returns a value representing noon on the given day (if *dateExpr* includes a time of day, it is ignored).

The `asDate` function also converts the value given in *dateExpr* to a date, but instead of assigning it the standard date format, the `asDate` function will derive the format from the way *dateExpr* itself is formatted. The asDate function can also be called using the `as {a} date` operator.

See Also: the `time()` and `seconds()` functions, later in this section.

◊ dateItems function

What it Does

Returns the current date, or the date value for a given expression, using one of the dateItems formats. These formats present a date and time as a comma-separated text list. The `short dateitems` returns six items: the year, month, day, hour, minute, and second. The `dateitems` (without an adjective) returns seven items, with the seventh being the day of the week (0-6, where Sunday is 0). The `abbreviated dateitems` adds the timezone offset in HHMM format, and the `long dateitems` returns that same information, but with the timezone name rather than offset.

Examples

```
put the dateitems -- "1995,10,07,17,50,22,6"
put the short dateitems -- "1995,10,07,17,50,22"
put the abbrev dateitems -- "1995,10,07,17,50,22,6,-0600"
put the long dateitems -- "1995,10,07,17,50,22,6,America/Denver"
put dateitems("May 14, 1942") -- "1942,05,14,12,00,00,4"
```

Tech Talk

Syntax: the { [long | short | abbr{ev{iated}}] } dateitems {of *dateExpr***}**
dateitems(*dateExpr***)**

The value returned by the `dateitems` function, when converted to text, will automatically display a formatted date, as shown in the Examples. Its value may also be treated as a number, representing the exact date and time when the function was called, for use in date/time calculations.

See Also: the `date` , `time`, `internet date`, `international time`, and `local time` functions, elsewhere in this section.

◊ UTCOffset (or secondsFromGMT) function

What it Does

Returns the difference in seconds between the current local time and Coordinated Universal Time or UTC (the synonymous `secondsFromGMT` refers to the historical term Greenwich Mean Time or GMT). If called with one parameter which is a date, it returns the local difference from UTC on the given date (which may vary depending on whether or not daylight savings is in effect on that date).

Examples

```
put the UTCOffset -- returns "-25200" (in MST)
put UTCOffset("June 4, 2001") / 1 hour -- returns -6
```

Tech Talk

Syntax: the UTCOffset {of *aDate***}**
UTCOffset(*aDate***)**

◊ seconds function

What it Does

Returns the current number of seconds since the beginning of January 1, 2001. The `long seconds` function returns a more precise version of the seconds, including the current fraction of a second. `Abbreviated seconds` and `short seconds` variants are also available, which provide values rounded to the microsecond (6 decimal places) and millisecond (3 decimal places), respectively.

Examples

```
put the seconds -- 62899676
put the long seconds -- 62899676.90865231
```

Tech Talk

Syntax: the { [long | short | abbr{ev{iated}}] } seconds
the seconds of *dateTimeValue*
seconds({*dateTimeValue***})**

If a *dateTimeValue* is given, the number returned will be the number of seconds since the beginning of January 1, 2001 until the given time (a negative number if the given time is earlier than 2001).

See Also: the `date` , `time`, and `ticks` functions, elsewhere in this section.

◊ ticks function

What it Does

Returns the number of ticks (1/60 second) since the **SenseTalk** engine was started.

Examples

```
if the ticks is greater than 36000 then
   put "SenseTalk was started more than 10 minutes ago."
end if
```

Tech Talk

Syntax: the ticks
ticks()

See Also: the `seconds` function, above.

◊ time, asTime functions

What it Does

Returns the current time of day, or the time value of a given expression. The `long time` function returns a longer version of the time, including the seconds. `Abbreviated time` and `short time` variants provide the time in other formats.

Examples

```
put the time  -- shows "02:38 PM"
put the short time  -- shows "02:38"
put the abbrev time -- shows "02:38:32"
put the long time   -- shows "02:38:32 PM"
put time("7:35:42") -- shows "07:35 AM"
put asTime("7:35:42")     -- shows "07:35:42"
```

Tech Talk

Syntax: the { [long | short | abbr{ev{iated}}] } time {of *timeExpr***}**
time(*timeExpr***)**
asTime(*timeExpr***)**

The value returned by the `time` function, when converted to text, will automatically display a formatted time, as shown in the Examples. Its value may also be treated as a number, representing the exact date and time when the function was called (or the exact date and time represented by its parameter), for use in date/time calculations or comparisons.

Tech Talk

If `the clockFormat` global property is set to “24 hour”, the format used by all of the time functions will not include “AM” or “PM” but instead will indicate hours between 00 and 23.

When *timeExpr* is given, that value is evaluated and returned as an internal time representation. It can also be called with the name `asTime` in this way, for consistency with other conversion functions.

The `asTime` function also converts the value given in *timeExpr* to a time, but instead of assigning it the standard time format, the `asTime` function will derive the format from the way *timeExpr* itself is formatted. The asTime function can also be called using the `as {a} time` operator.

See Also: the `date`, `seconds`, and `local time` functions, elsewhere in this section.

◊ year function

What it Does

Returns the year number of a given date, or the current year.

Examples

```
put the year into currentYear
put year("4 July 1776") -- "1776"
```

Tech Talk

Syntax: the year {of *dateTimeValue***}**
year(*dateTimeValue***)**

See Also: the `date`, `month`, `day`, and `dayOfYear` functions.

◊ month function

What it Does

Returns the month number (from 1 to 12) of a given date, or the current month.

Examples

```
put the month into monthNum
put month("4 July 1776") -- "7"
```

Tech Talk

Syntax: **the month {of** *dateTimeValue***}**
month(*dateTimeValue***)**

See Also: the `date`, `year`, and `day` functions.

◊ day function

What it Does

Returns the day number (from 1 to 31) of a given date, or the current date.

Examples

```
put day() into dayNum
put the day of "4 July 1776" -- "4"
```

Tech Talk

Syntax: **the day {of** *dateTimeValue***}**
day(*dateTimeValue***)**

See Also: the `date`, `year`, `month`, `dayOfWeek`, `dayOfYear`, and `dayOfCommonEra` functions.

◊ dayOfWeek function

What it Does

Returns the weekday number (from 0 to 6) of a given date, or of the current date. The number 0 represents Sunday.

Examples

```
put dayOfWeek() into weekdayNum
put the dayOfWeek of "4 July 1776" -- "4" (Thursday)
```

Tech Talk

Syntax: **the dayOfWeek {of** *dateTimeValue***}**
dayOfWeek(*dateTimeValue***)**

See Also: the `date`, `year`, `month`, `day`, `dayOfYear`, and `dayOfCommonEra` functions.

◊ dayOfYear function

What it Does

Returns the day number within a year (from 1 to 366) of a given date, or the current date.

Examples

```
put dayOfYear() into dayNum
put the dayOfYear of "4 July 1776" -- "186" (the 186th day of the year)
```

Tech Talk

Syntax: the dayOfYear {of *dateTimeValue***}**
dayOfYear(*dateTimeValue***)**

See Also: the `date`, `year`, `month`, `day`, `dayOfWeek`, and `dayOfCommonEra` functions.

◊ dayOfCommonEra function

What it Does

Returns the day number of a given date, or of the current date, since the beginning of the Common Era (January 1 of the year 1). This function can be useful for calculating the number of elapsed days between any two dates.

Examples

```
put the dayOfCommonEra into todayNum
put dayOfCommonEra("4 July 1776") -- shows "648491"
put "The U.S. has been independent for " & \
    dayOfCommonEra() - dayOfCommonEra("4 July 1776") & " days"
```

Tech Talk

Syntax: the dayOfCommonEra {of *dateTimeValue***}**
dayOfCommonEra(*dateTimeValue***)**

See Also: the `date`, `year`, `month`, `day`, `dayOfWeek`, and `dayOfYear` functions.

◊ hour function

What it Does

Returns the hour number (from 0 to 23) of a given time value, or the current hour.

Examples

```
put the hour into hourNum
put hour("5:37:22 PM") -- "17"
```

Tech Talk

Syntax: the hour {of *dateTimeValue***}**
hour(*dateTimeValue***)**

See Also: the `time`, `minute`, `second`, and `seconds` functions.

◊ minute function

What it Does

Returns the minute number (from 0 to 59) of a given time value, or the minute within the current hour.

Examples

```
put minute() into minutesPastTheHour
put the minute of "5:37:22 PM" -- "37"
```

Tech Talk

Syntax: the minute {of *dateTimeValue***}**
minute(*dateTimeValue***)**

See Also: the `time`, `hour`, `second`, and `seconds` functions.

◊ second function

What it Does

Returns the number representing the second (from 0 to 59) of a given time value, or the current second.

Examples

```
put the second into currentSecondNumber
put second("5:37:22 PM") -- "22"
```

Tech Talk

Syntax: the second {of *dateTimeValue***}**
second(*dateTimeValue***)**

Note that the value returned by the second() function will always be a number from 0 to 59. This is quite different from the seconds() function, which returns the number of seconds since the beginning of January 1, 2001.

See Also: the `time`, `hour`, `minute`, `ticks`, `millisecond`, `microsecond` and `seconds` functions.

◊ millisecond, microsecond functions

What it Does

The `millisecond()` function returns a number from 0 to 999 indicating the millisecond (thousandth of a second) of the current time, or of a given time value. The `microsecond()` function returns a number from 0 to 999999 indicating the microsecond (millionth of a second) of the current time, or of a given time value.

Examples

```
put millisecond() into currentMillisecond
put the millisecond of startTime
set tempName to the hour & the minute & the second & the microsecond
put microsecond(previousTime)
```

Tech Talk

Syntax: the millisecond {of *dateTimeValue***}**
millisecond(*dateTimeValue***)**
the microsecond {of *dateTimeValue***}**
microsecond(*dateTimeValue***)**

See Also: the `time`, `hour`, `second`, and `seconds` functions.

◊ convert command

What it Does

The `convert` command converts a date/time value to different date and/or time formats. If the source value being converted is a container, the contents of the container are replaced with the converted value. Otherwise, the result is placed in the variable `it`.

In either case (except when converting to one of the “seconds” formats) the resulting value internally is a date/time value with the requested format. This allows you to add or subtract time intervals from the result while retaining the same format. The actual formats used are defined in `the timeFormat` global property, and can be changed if

desired. The “seconds” formats have no corresponding value in the timeFormat, so converting to seconds, long seconds, etc. will result in a fixed string rather than a formatted time value.

Examples

```
convert "8/14/02" to long date -- sets value of 'it'
convert the time to short date and short time -- sets value of 'it'
convert expirationDate to date -- changes the value of expirationDate
convert line 2 of file "/tmp/resultLog" to abbreviated local time
```

Tech Talk

Syntax: convert *source* **to** *format* **{and** *format***}**

The following values for *format* may be used:

Format	Example Value
date	11/09/04
short date	11/9/04
long date	Tuesday, November 9, 2004
abbreviated date	Tue, Nov 9, 2004
time	03:17 PM
short time	03:17
long time	03:17:25 PM
abbreviated time	03:17:25
local time	03:17 PM America/Denver
short local time	03:17 PM -0700
long local time	03:17:25 PM America/Denver
abbreviated local time	03:17:25 PM -0700
dateitems	2004,11,09,15,17,25,2
short dateitems	2004,11,09,15,17,25
long dateitems	2004,11,09,15,17,25,2,America/Denver
abbreviated dateitems	2004,11,09,15,17,25,2,-0700
simple date	11/09
short simple date	11/9
long simple date	November 9
abbreviated simple date	Nov 9
basic date	Nov 9, 2004
short basic date	Nov 9 04

Format	Example Value
long basic date	November 9, 2004
abbreviated basic date	November 9 04
basic time	Nov 9, 2004 03:20 PM
short basic time	Nov 9 04 3:20 PM
long basic time	November 9, 2004 03:20:26 PM
abbreviated basic time	November 9 04 03:20:26 PM
common date	9 Nov 2004
short common date	9 Nov 04
long common date	09 November 2004
abbreviated common date	09 Nov 2004
common time	9 Nov 2004 08:29 AM
short common time	9 Nov 04 08:29 AM
long common time	09 November 2004 08:29 AM
abbreviated common time	09 Nov 2004 08:29 AM
C time	Tue Nov 9 15:32:28 2004
short C time	Nov 9 15:32 2004
long C time	Tuesday November 9 15:32:28 2004
abbreviated C time	Nov 9 15:32:28 2004
international date	2004-11-09
short international date	2004-11
long international date	2004-11-09
abbreviated international date	2004-11
international time	2004-11-09T08:29:25-0700
short international time	2004-11-09T08:29-0700
long international time	2004-11-09T08:29:25.894-0700
abbreviated international time	2004-11-09 08:29:25 -0700
internet date	Tue, 9 Nov 2004 08:29:25 -0700
short internet date	9 Nov 2004 08:29 -0700
long internet date	Tuesday, 9 November 2004 08:29:25 -0700
abbreviated internet date	9 Nov 2004 08:29:25 -0700
seconds	121732349
long seconds	121732348.572803789
abbreviated seconds	121732348.572804

Format	Example Value
short seconds	121732348.573

If `the clockFormat` global property is set to “24 hour”, all time formats will use a 24-hour clock format (e.g. “15:17”) instead of using “AM” or “PM” (e.g. “03:17 PM”).

The dateitems formats can be especially useful for working with calendar dates. A date/time represented in dateitems format consists of 7 numbers delimited by commas, representing the year, month, day, hour, minute, second, and day of the week (with 0 representing Sunday, and 6 representing Saturday). The long and abbreviated dateitems formats also include time zone information. The short dateitems omits the day of the week.

SenseTalk doesn’t provide time interval expressions for months or years because they vary in length. To calculate a calendar date that is some number of years or months away from another, you can convert a date to dateitems, add or subtract from the desired item, and then convert back. For example, this script sets a date 18 months in the future, on the same day of the month as the current date:

```
put the date into futureDate -- start with today's date
convert futureDate to dateItems
add 18 to item 2 of futureDate -- item 2 is the month
convert futureDate to long date -- back to a more friendly format
```

Here is a more complex example — a calculateAge function similar to one presented in the discussion of Helpers in an earlier chapter. This version does a better job of dealing with leap years:

```
function calculateAge birthDate -- calculate age in years for a given birthDate
    convert birthDate to dateItems -- change it to yr,mon,day,hr,min,sec
    split birthDate by comma -- convert to a list
    convert the date to dateItems -- today's date in 'it' as dateItems
    split it by comma -- convert to a list
    subtract birthDate from it -- subtract one list of values from the other
    -- if today's day of month is less than birthDate's, then subtract a month:
    if item 3 of it < 0 then subtract 1 from item 2 of it
    -- then if today's month is less than birthDate's, subtract a year:
    if item 2 of it < 0 then subtract 1 from item 1 of it
    return item 1 of it -- the difference in years
end calculateAge
```

The exact format of all of the date and time formats can be customized within a script by setting the appropriate properties within `the timeFormat` global property, described later in this section.

◊ basic date, basic time functions
common date, common time functions
international date, international time functions
internet date functions
local time functions
simple date functions
C time functions

What it Does

Each of these functions returns a formatted value representing either the current moment in time or a given moment in time, using a format that is unique to the particular function used. There are four variations of each function, using the function name alone, or preceded by one of the adjectives `long`, `short`, or `abbreviated` (which can be shortened to `abbrev` or `abbr`). All of the different formats are shown in the table for the `convert` command, above.

The `basic date` formats begin with the month name (or abbreviated name) followed by the day, then the year. The `basic time` formats add the time of day.

The `common date` and `common time` formats are similar to the `basic` formats, but show the day before the month name rather than after it.

The `international date` and `international time` formats present the full date, the year and month, or the full date and time, in a manner that complies with the international ISO 8601 standard (see http://www.ietf.org/rfc/rfc3339.txt) except for the `abbreviated international time`, which presents a format used widely on the internet that varies slightly from the standard.

The `internet date` formats present a date and time in a manner that complies with the date and time specification of the internet message format as defined in RFC 2822 (http://www.ietf.org/rfc/rfc2822.txt), except that the `long internet date` shows full weekday and month names.

The `local time` formats present the current time of day, including time zone information.

The `simple date` formats present the date in a very simple format that includes only the month and day but omits the year.

The `C time` (or `CTime`) formats present the date and time in a format used by some C-based systems (including Python).

Examples

```
put the basic date -- Jan 4, 2008
put the basic date of "10/2/1869" -- Oct 2, 1869
put the long basic date -- January 4, 2008
put the basic time -- Jan 4, 2008 03:20 PM
put the common date -- 4 Jan 2008
put the common time -- 4 Jan 2008 03:20 PM
put the simple date -- 01/04
put the long simple date -- January 4
```

```
put the international date -- 2005-10-19
put the short international date -- 2005-10
put the international time -- 2005-10-19T18:32:31-0600
put the abbrev international time -- 2005-10-19 18:32:31 -0600
put the internet date -- Wed, 19 Oct 2005 18:21:47 -0600
put the short internet date -- 19 Oct 2005 18:21 -0600
put the long internet date -- Wednesday, 19 October 2005 18:21:47 -0600
put the local time -- 05:42 PM -0700
put the short local time -- 05:42 PM US/Mountain
put the long local time -- 05:42:12 PM US/Mountain
put the C time -- Fri Sep 17 13:38:17 2010
put the long CTime -- Friday September 17 13:39:48 2010
```

Tech Talk

Syntax: **the { long | short | abbr{ev{iated}} } basic [date | time] {of** *dateExpr*}
the { long | short | abbr{ev{iated}} } common [date | time] {of *dateExpr*}
the { long | short | abbr{ev{iated}} } international [date | time] {of *dateExpr*}
the { long | short | abbr{ev{iated}} } internet date {of *dateExpr*}
the { long | short | abbr{ev{iated}} } local time {of *dateExpr*}
the { long | short | abbr{ev{iated}} } simple date {of *dateExpr*}
the { long | short | abbr{ev{iated}} } [c time | ctime] {of *dateExpr*}

The value returned by each of these functions contains both a number and a format. When used as text, it will automatically be presented in the format shown in the table for the `convert` command, above. The returned value may also be treated as a number, for use in date/time calculations. When called without a parameter, the numeric value represents noon on the current date (for the `date` functions), or the exact date and time when the function was called (for the `time` functions).

When a *dateExpr* is given, that expression is evaluated as a date and/or time, and the value returned will represent that date and time (or noon on that date, for the `date` functions).

See Also: the `date` , `time`, and `dateitems` functions, earlier in this section, and the `formattedTime` function, below.

◊ formattedTime function

What it Does

The `formattedTime` function returns a date/time value using a custom format that you supply. An optional second parameter is the date/time that you would like to format. If a date/time value is not given, the result will be the current time, using the supplied format.

Examples

```
put formattedTime("Calendar for %Y") into calendarTitle
```

```
put the formattedTime of "It's now day %j of %Y!"
put formattedTime(the timeFormat's longDate, sentDt) into dateSent
put formattedTime("%Y%m%d_%H%M%S", logTime) into logFileName
```

Tech Talk

Syntax: the formattedTime of *customFormat*
formattedTime(*customFormat {,dateTimeValue}* **)**

See Also: the `convert` command, and the specific date and time format functions, above.

◊ monthNames and weekDayNames functions

What it Does

The `monthNames()` function returns a list of the names of the months used when formatting dates. The `weekDayNames()` function returns a list of the names of the days of the week. Both functions can be called using the adjectives `long`, `short`, and `abbreviated` to return variations on these. In each case, the long form is the same as not specifying any adjective, the abbreviated form returns a list of three-letter abbreviations rather than the full name, and the short form returns numeric representations.

Examples

```
put monthNames() into monthList
put the abbreviated monthNames
put item dayNum of weekDayNames() into dayName
put the short weekDayNames -- displays the numbers 0 to 6
```

Tech Talk

Syntax: the {long | short | abbreviated} monthNames
monthNames()
the {long | short | abbreviated} weekDayNames
weekDayNames()

See Also: the `formattedTime` function and the `convert` command.

◊ the timeFormat global property

What it Does

The `timeFormat` global property is a property list holding all of the standard date and time formats supported by the `convert` command and the various formatted date and time functions. You can change any of these formats if you choose.

Examples

```
put the timeFormat.ShortDateItems -- "%Y,%m,%d,%H,%M,%S"
set the timeFormat's time12 to "the time is now %I:%M in the %p"
put the long date -- "Wednesday, October 19, 2005"
delete the first word of the timeFormat's longDate
put the long date -- October 19, 2005
```

Tech Talk

Syntax: the timeFormat

To see a list of all of the formats included in `the timeFormat` issue this command:

```
put the keys of the timeFormat
```

Any of the formats can be examined or changed (as shown in the examples above). Use caution when making changes to avoid deleting any formats that you need, or accidentally replacing the entire set of formats with one value.

A date/time format may contain any characters. The following are special placeholders that indicate the parts of the date/time value being formatted that will appear in the formatted text representation:

Format Placeholder	Will Be Replaced By
%%	a percent sign (%)
%a	abbreviated weekday name
%A	full name of the day of the week
%b	abbreviated month name
%B	full month name
%c	complete localized date and time
%d	day of the month as two-digit number (01-31)
%e or %1d	day of the month as one- or two-digit number (1-31)
%F	fraction of a second to three places (000-999)
%H or %1H	hour based on a 24-hour clock, as two-digit number (00-23) or with leading zero suppressed (`%1H`)
%I or %1I	hour based on a 12-hour clock, as two-digit number (01-12) or with leading zero suppressed (`%1I`)
%j or %1j	day number within the year, as three-digit number (001-366) or with leading zero suppressed (`%1j`)
%m or %1m	month as a two-digit number (01-12) or with leading zero suppressed (`%1m`)
%M or %1M	minute of the hour as a two-digit number (00-59) or with leading zero suppressed (`%1M`)
%p	AM or PM
%S or %1S	second of the minute as a two-digit number (00-59) or with leading zero suppressed (`%1S`)

%w	weekday number (0-6), where Sunday is 0
%x	date using the date representation for the locale
%X	time using the time representation for the locale
%y or %1y	year without century as a two-digit number (00-99) or with leading zero suppressed (`%1y`)
%Y	full year number (with century — currently a 4-digit number
%z	time zone offset in hours and minutes from GMT (HHMM)
%Z	time zone name

◊ the timeInputFormat global property

What it Does

The `timeInputFormat` global property is a list that specifies all of the date and time formats that are used in recognizing whether a text string is valid as a date/time value. By default, this property is dynamically linked to the values of `the timeFormat` global property, so that date or time strings can be recognized in any format contained there.

Examples

```
insert "%d/%m/%y" before the timeInputFormat -- prefer European interpretation
of dates
set the timeInputFormat to "Natural" -- most formats accepted
set the timeInputFormat to "" -- reset default behavior
```

Tech Talk

Syntax: the timeInputFormat

Whenever SenseTalk tries to interpret a value as a date/time value that isn't already in that format internally, it follows a three step process. First, if the value is a number, it is treated as the number of seconds since the beginning of January 1, 2001. Next, if the value is in one of the dateItems formats (a list or comma-delimited text list of 5, 6, 7, or 8 numbers), it is interpreted accordingly. Finally, the value is treated as text and SenseTalk goes through each item in `the timeInputFormat` in the order they are listed to see if the value matches that format. The first matching format is used to translate the value.

This process of interpreting dates/times is used by many operations, including such things as the `is a date` operator, the `date()` function and related functions, and operations like addition that may try to implicitly interpret a string as a date/time value.

When setting the value of `the timeInputFormat`, you may set it to a single format value (or "Natural" – see below), to a list of format values, or to a property list containing format values. Setting it to empty will dynamically link it to `the timeFormat` values again (its initial default state). If you set the value to a list of formats, take care to order them with longer formats first, since interpretation of time values will always use the first matching format even it if only matches part of the text.

When set to a property list (or linked to `the timeFormat`), only the values from the property list will be used. SenseTalk will order the formats in the sorted key order of the property list, except when one format is a prefix of another it will always be placed later in the list in order to allow the longer format to match first. Also, if any format allows two-digit years (using %y), the equivalent format for four-digit years (%Y) will automatically be added to the list ahead of the two-digit version of that format.

See the description of `the timeFormat` global property, above, for definitions of all of the symbols that have special meaning in a date/time format string.

In addition to specific format strings, the value "Natural" may be used to accept input in many formats, including some natural language phrases such as "yesterday" or "at lunch on Tuesday". Be aware that this setting, while using sophisticated techniques to arrive at a "best guess" of the value, may in some cases may be overly aggressive about treating values as dates/times. When used in a list, "Natural" should be the last item in the list to allow other specific formats to match in a controlled priority order first.

◊ format property

What it Does

Each date/time value has a display format associated with it. The format can be accessed directly by using the `format` property of the value. If the value is stored in a variable, the format property can also be set.

Examples

```
put date().format -- %m/%d/%y
put (long date)'s format -- %A, %B %e, %Y
set the format of dueDate to the timeFormat.InternationalDate
```

Tech Talk

Syntax: the format of *dateTimeValue*
dateTimeValue **. format**
dateTimeValue **'s format**

◊ the centuryCutoff local property

What it Does

`The centuryCutoff` local property controls the interpretation of two-digit years when converting strings to dates.

Examples

```
set the centuryCutoff to 26
put international date of "3/16/26" -- 2026-03-16
put international date of "3/16/27" -- 1927-03-16
```

Tech Talk

Syntax: the centuryCutoff

When a two-digit year is encountered, it is assumed to be in the present year or a future year up through the year indicated by the centuryCutoff. Otherwise, a two-digit year that is greater than the centuryCutoff is taken to be in the past (either earlier in the present century or in the previous century). By default the centuryCutoff is set to the defaultCenturyCutoff global property, which is initialized to 10 years in the future. To disable this functionality and allow two-digit years to represent years in the first century of the common era, set the centuryCutoff to a negative number or to empty.

Working with Files and File Systems

SenseTalk provides a number of facilities for working with files and file systems. You can read and write the contents of files, create and delete files and folders in the file system, and obtain various information about files and file systems. This section describes all of these facilities in detail.

SenseTalk also provides for reading and writing data through sockets, and through standard input and output streams. These facilities are also described in this section (see the `open socket`, `close socket`, and `read` and `write` commands in the section “File, Socket, and Stream Input and Output”).

Referring to Files in a Script

To refer to a file in a script, just use the word `file` followed by an expression that evaluates to the name of the file. You can refer to folders in a similar way, using the word `folder` instead of `file`.

```
open file "/etc/passwd"
move file "runlog24" into folder "archivedLogs"
```

Tech Talk

Syntax: **file** *filePath*
folder *filePath*
directory *filePath*

The *filePath* may be the text of a path in either Mac/Linux or Windows format, or it may be a list in which each item is one component of the path.

The name given may be either the full (absolute) path name of the file or folder, or the path relative to the current working folder (see `the folder` global property below). SenseTalk determines the full “absolute” path name of the file based on the name given, according to the following rules:

- if the name begins with a slash (/) or the first item in the path list is "/" it is already an absolute file name
- if the name begins with a drive letter and colon followed by a slash (like "C:/") or the first item in the path list is a drive letter and colon and the second item is a slash it is already an absolute file name
- if the name begins with a tilde and a slash (~/) it represents a path relative to the user’s home folder
- if the name begins with a tilde (~) followed by a user name it represents a path relative to that specific user’s home folder
- if the name begins with a period and a slash (./) or two periods and a slash (../) it represents a relative path from either the current working folder or the current working folder's parent folder, respectively
- otherwise, it represents a file or path within the current working folder

◊ the folder global property

You can access or change the current working folder using the global property `the folder` (or `the directory`):

```
set the folder to "/tmp/myWorkArea" -- set the working folder
put the folder & "myFileName" into filePath
```

The value returned by `the folder` will end with a slash, unless `the folderNamesEndWithSlash` global property is set to false. The slash at the end makes it easy to create a full path name by simply appending a file name, as shown in the example above.

Tech Talk

Syntax: the folder
the directory

The value returned by accessing the folder is actually a fileDescription object, but can simply be treated as a string for most purposes. Its string value is the full path to the current working folder.

Note

The words **folder** and **directory** are used interchangeably throughout this manual, and within SenseTalk scripts — wherever the word **folder** is used in a script, you may use the word **directory** instead.

◊ folder, directory function

What it Does

Returns the parent folder of a given file path.

When to Use It

Use the `folder` function (or its synonym, `directory`) to obtain the path to the folder containing a given file or folder.

The value returned will end with a slash, unless `the folderNamesEndWithSlash` global property is set to false. The slash at the end makes it easy to create a full path name by simply appending a file name.

Examples

```
put the folder of myFile into myFolder
put folder(someFile) & "siblingFileName" into newFile
```

Tech Talk

Syntax: the folder of *filePath*
folder(*filePath***)**

If the *filePath* given is not an absolute path it will be taken relative to the current working folder, as given by `the folder` global property. The `folder` function may also be called with a fileDescription object (such as returned by the `files()` or `fileDescription()` functions) or with a script file object, to obtain the parent folder of the indicated file.

The value returned by the folder function is actually a fileDescription object, but can simply be treated as a string for most purposes. Its string value is the full path to the folder.

◊ lastPathComponent function

What it Does

Returns the local name of a file system object, removing the path to the parent folder.

When to Use It

Use the `lastPathComponent` function when you have the full path name of a file or folder and want to obtain just the local name, without the full path.

Examples

```
put the lastPathComponent of fullPath into fileName
put lastPathComponent("/Users/jc/Documents/Jan24.data")-- "Jan24.data"
```

Tech Talk

Syntax: the lastPathComponent of *filePath*
lastPathComponent(*filePath***)**

◊ fileExtension function

What it Does

Returns the file extension from a file name.

When to Use It

Use the `fileExtension` function when you have the full name of a file or folder and want to obtain just the file extension. The extension is the part of the name following the last period. The period is not included in the extension that is returned.

Examples

```
put the fileExtension of fileName into extension
put fileExtension("/Users/jc/Documents/Jan24.data") -- "data"
```

Tech Talk

Syntax: the fileExtension of *fileName*
fileExtension(*fileName***)**

◊ pathList function

What it Does

Returns a file path as a list of individual path components in a standard format.

When to Use It

Use the `pathList` function when you have the full or partial path name of a file or folder and want to obtain a path list containing the individual components of that path in a standard format.

Examples

```
put the pathList of filePath into filePathList
put pathList("/Users/sj/Documents/MLK.txt") -- (/,Users,sj,Documents,MLK.txt)
```

Tech Talk

Syntax: the pathList of *filePath*
pathList(*filePath***)**

◊ filePath, windowsFilePath functions

What it Does

The `filePath` function returns a file path as a string in a standard (Mac/Linux/web) format, with slashes as the separator between path components. The `windowsFilePath` function is similar, but returns a file path string in Windows format, using backslashes as the separator between components.

When to Use It

Use the `filePath` or `windowsFilePath` function when you have a full or partial path name of a file or folder in any format and want to obtain either a standard Mac/Linux/UNIX text representation of that path that uses slashes between path components, or a Windows text representation that uses backslashes between path components.

Examples

```
put the filePath of fullPath into stdPath
put filePath("\wiki\en\Home") -- "/wiki/en/Home"
put windowsFilePath("/Admin/theo/x32.jpg") -- "\Admin\theo\x32.jpg"
put filePath of ("a","b","c") -- "a/b/c"
put windowsFilePath of ("a","b","c") -- "a\b\c"
```

Tech Talk

Syntax: **{the} filePath of** *filePath*
filePath(*filePath***)**
{the} windowsFilePath of *filePath*
windowsFilePath(*filePath***)**

◊ resolvedFilePath function

What it Does

Returns a file path as a string in a "resolved" standard format, which is the full absolute path to the file, taking into account the current folder if necessary, and resolving components such as ".." and "~".

When to Use It

Use the `resolvedFilePath` function when you want to determine the full actual path fo a file or folder. Because of path mechanisms such as "~" which refers to the user's home folder and ".." which refers to the parent folder of any folder it is possible to have multiple different paths which all refer to the same location. The resolvedFilePath can be used to obtain a standard representation of the path which can be used to compare paths to see if they represent the same file, for example.

Examples

```
put the resolvedFilePath of fileName into resolvedName
if resolvedFilePath(it) is resolvedFilePath(safeFile) then ...
```

Tech Talk

Syntax: **the resolvedFilePath of** *filePath*
resolvedFilePath(*filePath***)**

◊ fileDescription function

What it Does

Returns a fileDescription object containing information about a given file.

◊ fileDescription function

When to Use It

Use the `fileDescription()` function to obtain a packet of information about a file or folder. The value returned is a fileDescription object. A fileDescription is a SenseTalk object (property list) that appears as the short name of the file if displayed as text, but knows its full path and also contains many pieces of additional information about the file.

Examples

```
put fileDescription("/tmp/data") into fileInfo
put fileInfo is a fileDescription -- true
put fileInfo -- "data"
put the long name of fileInfo -- "/tmp/data"
put fileInfo's NSFileSize into dataSize
```

Tech Talk

Syntax: the fileDescription of *filePath*
fileDescription(*filePath***)**

The *filePath* may be the full path to a file, or the file name relative to the current working folder. The value returned is a fileDescription object (a property list with objectType set to "fileDescription"). A fileDescription object contains an asText property that is set to the local name of the file, so displaying the object will simply show the file's name.

Each fileDescription object also holds a number of additional items of information. In particular, the "long name" property contains the full path name of the file. Other properties include the parent folder where the file is located, and such information as the file size, owner, and permissions. Use the keys() function (or delete the object's asText property before displaying it) to find out exactly what information is available.

SenseTalk commands and functions that work with files, such as the `copy file` and `rename` commands and the `diskSpace()` function, recognize fileDescription objects that are used in place of file names, and will use the long name to identify the actual file. In this way, fileDescription objects can serve as file identifiers that can be stored in variables, passed as parameters, and so forth.

FileDescription objects can also be obtained from the `files()` and `folders()` functions, which each return a list of fileDescriptions.

Accessing a File as a Container

The simplest way to work with the contents of a file is to access the file directly as a SenseTalk container. Using this approach you can read an entire file with a single command:

```
put file "/etc/passwd" into passwordInfo
```

Or you can write a file just as easily:

```
put "0,0,0,0" into file "/tmp/testing/counters"
```

The command above will create a file named “counters” in the directory “/tmp/testing” and write the value “0,0,0,0” into it. If the “/tmp/testing” directory does not exist, it will also be created. If there was already a file “/tmp/testing/counters”, its previous contents will be completely replaced by the new value, so be careful when using this approach.

You can also access any part of the text in a file, using chunk expressions:

```
add 1 to item 2 of line 1 of file "/tmp/testing/counters"
```

This command will read the current contents of the file, add 1 to the second item on the first line, and store the modified value back into the file.

If a command attempts to write to a file and fails for some reason (such as insufficient privileges for writing to the file), `the result` will be set to an error message. The value of `the result` will also be set to an error message when reading a file as a container, if the file does not exist or cannot be accessed. The value of the file expression will be treated as empty in this case.

Treating a file as a container is very easy and works very well for many situations. Occasionally, it may not be the most efficient approach to use if your script needs to do a significant amount of reading or writing in a file. In these cases you may prefer to use the `open file`, `read from file`, `seek in file`, `write to file`, and `close file` commands, described under **File Input & Output Commands** later in this section.

Configuring File Behavior

When accessing a file as a container, text is interpreted during both reading and writing according to the setting of `the defaultStringEncoding` global property, which is descibed in detail near the end of this section. To read or write a file as binary data instead of as text, specify `as data`:

```
put file "/tmp/datafile" as data into myData
put contents as data into file "/tmp/binaryFile"
```

◊ the strictFiles global property

When reading from a nonexistent file, the default behavior is simply to act as though that file were empty. Sometimes this behavior may lead to unexpected results. For example, if a file name is entered incorrectly, a script will simply treat it as empty rather than giving an error indicating that the file could not be found.

To provide stricter control over the use of files at runtime, `the strictFiles` global property may be used. When this property is set to `true`, reading a nonexistent file as a container will throw an exception rather than simply returning empty. This property is initially set to `false`.

Checking the Existence of a File or Folder

You can check whether a file exists using the `there is a file fileName` or `file fileName exists` operators, or their negative counterparts `there is not a file fileName`, `there is no file fileName` or `file fileName does not exist` to determine that a file or folder doesn’t exist:

```
if file "tempWorkFile" exists then delete file "tempWorkFile"
if there is a file "scores" then
```

```
    put file "scores" into highScores
else
    put "100,100,100,100,100" into highScores
end if-- create an empty data file if it does not already exist:
if there is no file "data" then put "" into file "data"
```

You can check for the existence of a folder in the same way.

```
if folder "/tmp/work" does not exist then initWorkFolder
```

File System Commands and Functions

Several commands and functions provide access to the file system on the machine where the script is running (or a locally mounted file system), enabling your script to create, move, copy, rename, and delete files and folders, and to obtain information about the files and folders in the system.

◊ create file, create folder, create link

What it Does

Creates a new file or folder in the file system, or a symbolic link to an existing file or folder.

When to Use It

Use the `create folder` command to create a new folder on the disk. Use the `create file` command to create an empty file. Use `create link` to create a link (sometimes called an alias or a symbolic link) which looks like an independent file, but is actually a reference to a different file on the disk.

> **Note: `Open file` command**
>
> To create a file, you may also use the `open file` command to open it and the write command to write to it; or simply put something into the file.

Examples

```
create a new folder "/tmp/myWorkArea"
create folder "/tmp/myWorkArea/subdir" with (groupName:"admin",
permissions:"rwxrwxr-x")
create file "/tmp/myWorkArea/testData"
create link "tasty" to file "juicy"
```

Tech Talk

Syntax: create {a} {new} [file | folder | directory] *fileOrFolderName* **{with** *properties***}**
create {a} {new} link linkName to [file | folder | directory] *fileOrFolderName*

The *fileOrFolderName* expression must yield either an absolute path name or a path name relative to the current folder. The file, folder, or link being created must not already exist. If its parent folder does not exist, it will also be created.

If the **with** *properties* option is used, properties should be a property list specifying initial values for any of the following properties: **ownerName**, **groupName**, **permissions**, **creationDate**, **modificationDate**, and for files: **typeCode**, **creatorCode**, **fileExtensionHidden**, **appendOnly**, or **locked**. See the section “Accessing File Properties” later in this section for more information about setting these properties.

If the command fails, `the result()` function will be set to return a non-empty value indicating the error.

◊ delete file, delete folder

What it Does

Permanently removes a file or folder from the disk.

When to Use It

Use the `delete` command to destroy a file, or to destroy a folder including all of its contents. This command is permanent and irreversible — use with caution.

Examples

```
delete file "testData27"
delete folder "/tmp/myWorkArea"
```

Tech Talk

Syntax: delete [file | folder | directory] *fileOrFolderName*

The *fileOrFolderName* expression must yield the name of an existing file or folder. Deleting a folder will delete all of the files and folders within it as well.

If the command fails, `the result()` function will be set to return a non-empty value indicating the error.

◊ rename file, rename folder

What it Does

Changes the name of a file or folder.

When to Use It

Use the `rename` command to change the name of a file or folder.

Examples

```
rename folder "/tmp/myWorkArea" as "oldWorkArea"
rename file sourceFile as sourceFile && "backup"
```

Tech Talk

Syntax: rename [file | folder | directory] *originalName* **as** *newName*

The *originalName* expression must yield the name of an existing file or folder. If newName is not a full path name, it is taken to be relative to the folder where the source file or folder is located.

If the command fails, `the result` function will be set to return a non-empty value indicating the error.

◊ copy file, copy folder

What it Does

Makes a duplicate copy of an existing file or folder.

When to Use It

Use the `copy` command any time you want to make a complete copy of a single file or of a folder and all of its contents. There are three forms of the `copy` command: `copy ... into ...`, `copy ... as ...`, and `copy ... to ...`. The first form, using the preposition `into`, makes a copy of the source file or folder with the same name as the original in a different destination folder. If the destination folder does not exist, it will be created.

The second form of `copy`, using the preposition `as`, allows you to assign a different name to the copy. The copy may be created in the same folder as the source, or in a different folder. The final form of `copy`, using the preposition `to`, behaves just like `copy ... into ...` if the destination is an existing folder, otherwise it behaves like `copy ... as ...`.

Examples

```
copy file results into folder resultsArchiveFolder
copy file "/tmp/testFile" as "/tmp/testFileCopy"
copy folder planFldr to "~/Documents"
```

Tech Talk

Syntax: **copy [file | folder | directory]** *sourceName* **[into | to] {folder | directory}** *destinationFolder*
copy [file | folder | directory] *sourceName* **[as | to]** *destinationName*

The sourceName expression must yield the name of an existing file or folder. If sourceName is not an absolute path, it is assumed to be relative to the current working folder. If the destinationFolder or destinationName is not an absolute path, it is assumed to be relative to the parent directory of the source file or folder.

If the command fails, `the result` function will be set to return a non-empty value indicating the error.

◊ move file, move folder

What it Does

Moves a file or folder to a new location in the file system.

When to Use It

Use the `move ... into ...` command to move a file or folder into a different parent folder without changing its name. The `move ... to ...` command – similar to the `copy ... to ...` command – will assign a new name to the file or folder being moved unless the destination is an existing folder, in which case the source file or folder will be moved into the destination folder without changing its name.

Examples

```
move file "/tmp/testFile" into folder "archives"
```

Tech Talk

Syntax: **move [file | folder | directory]** *sourceName* **[into | to] {folder | directory}** *destinationFolder*

If *sourceName* is not an absolute path, it is assumed to be relative to the current working folder. If the *destinationFolder* is not an absolute path, it is taken to be relative to the parent folder of the source file or folder. If the *destinationFolder* does not exist, it will be created.

If the `move` command fails, `the result` function will be set to return a non-empty value indicating the error.

◊ files function

What it Does

Returns a list of all of the files in the current working folder, or in some other specified folder.

When to Use It

Use the `files` function to find out what files exist in a given folder in the file system. You can easily iterate over all of the files in the list that is returned to work with each file in turn.

If the `files` function is called without any parameter (or with an empty parameter), it returns a list of files in the current working folder. If a parameter is given, it should be the name of an existing folder, and `files` will return the list of files in that folder.

Examples

```
put files("/tmp") into tmpFileList-- backup all ".st" files into backupFolder
repeat with each item of the files -- look at all files in working folder
   if it ends with ".st" then copy file it into backupFolder
   else put "Not backed up: " & it & " of size " & it.NSFileSize
end repeat
```

Tech Talk

Syntax: the files {of *folder***}**
files(folder)

The `files` function returns a list containing one item for each of the non-folder entries in *folder* (or in the current working folder, if *folder* is not specified). Each item in the returned list is a fileDescription object (a property list with objectType set to “fileDescription”). The asText property of each fileDescription is set to the local name of the file, so displaying it will simply show the file’s name.

Each fileDescription object also holds many additional items of information. In particular, the “long name” property contains the full path name of the file. Other properties include the parent folder where the file is located, and such information as the file size, owner, and permissions. Use the keys() function (or delete the object’s asText property before displaying it) to find out exactly what information is available.

SenseTalk commands and functions that work with files, such as the `copy file` and `rename` commands and the `diskSpace()` function, recognize fileDescription objects that are used in place of file names, and will use the long name to identify the actual file. In this way, fileDescription objects can serve as file identifiers that can be stored in variables, passed as parameters, and so forth.

◊ folders function

What it Does

Returns a list of all of the sub-folders in the current working folder, or in a specified folder.

When to Use It

Use the `folders` function to find out what folders exist within a given folder in the file system. You can easily iterate over all of the folders in the list that is returned to work with each folder in turn.

If the `folders` function is called without any parameter, it returns a list of folders in the current working folder. If a parameter is given, it should be the name of an existing folder, and `folders` will return the list of sub-folders within that folder.

Examples

```
put folders() into subfolderList-- show the files in each subfolder of the
current folder
repeat with each item of the folders
    put "Folder " & it & return & the files of it
end repeat
```

Tech Talk

Syntax: the folders {of *parentFolder***}**
folders(parentFolder)

The `folders` function returns a list containing one item for each of the sub-folder entries in *parentFolder* (or in the current working folder, if *parentFolder* is not specified). The *parentFolder* expression must yield the name of an existing folder.

Each item in the returned list is a fileDescription object (property list) which shows the local name of the folder when displayed, but also contains many additional items of information about the folder, such as its modification date and permissions settings. See the `files` function, above, or the `fileDescription` function, near the beginning of this section, for more information about fileDescription objects.

◊ filesAndFolders function

What it Does

Returns a list of all of the files and sub-folders in the current working folder, or in a specified folder.

When to Use It

Use the `filesAndFolders` function to find out what files and folders exist within a given folder in the file system. You can easily iterate over all of the items in the list that is returned to work with each file or folder in turn.

If the `filesAndFolders` function is called without any parameter, it returns a list of files and folders in the current working folder. If a parameter is given, it should be the name of an existing folder, and `filesAndFolders` will return the list of files and sub-folders within that folder.

Examples

```
put filesAndFolders(appPath) into appContents -- show the files in the current
folder and its subfolders
```

```
repeat with each item of the filesAndFolders
   if it is a folder then
         put "Folder: " & it & " -- " & the files of it
   else
         put "File: " & it
   end if
end repeat
```

Tech Talk

Syntax: the filesAndFolders {of *parentFolder*}
filesAndFolders(parentFolder)

The `filesAndFolders` function returns a list containing one item for each of the file and sub-folder entries in *parentFolder* (or in the current working folder, if *parentFolder* is not specified). The *parentFolder* expression must yield the name of an existing folder.

Each item in the returned list is a fileDescription object (property list) which shows the local name of the file or folder when displayed, but also contains many additional items of information about that item, such as its modification date and permissions settings. See the `files()` function, above, or the `fileDescription()` function, near the beginning of this section, for more information about fileDescription objects.

◊ diskSpace function

What it Does

Returns the number of bytes of free space available on a disk.

When to Use It

Use the `diskSpace()` function to find out how much free space (in bytes) is available on a file system.

If the `diskSpace()` function is called without any parameter, it returns the amount of free space in bytes in the file system containing the current working folder. If a parameter is given, it should be the name of an existing file or folder, and `diskSpace()` will return the amount of free space on the file system containing that file or folder.

Examples

```
put diskSpace() into spaceRemaining
if the diskspace is less than a Megabyte then
   put "You have less than a megabyte of space remaining!"
end if
if diskSpace("/Volumes/sparky") is less than 1000 then
   put "Less than a thousand bytes left on /Volumes/sparky!!"
end if
```

Tech Talk

Syntax: the diskSpace {of *fileOrFolder***}**
diskSpace(*fileOrFolder***)**

If *fileOrFolder* is not specified, the `diskSpace()` function returns the amount of space available on the volume containing the current working folder, as indicated by `the folder` global property.

Accessing File Properties

You can access a number of different properties of a file or folder. The following properties are accessible:

File Property Name	Description
name	the name of the file, such as “myFile.txt”
short name	the name without the extension — “myFile”
long name	the full path name of the file, such as “/tmp/myFile.txt”
display name	the name that should be displayed to the user
folder directory	the full path name of the folder containing the file
size	the size of the file in bytes
creation date	the date/time when the file was created
modification date	the date/time when the file was last changed
permissions	a string denoting the file permissions (see below)
owner permissions	”read”, “write”, and/or “execute”
group permissions	”read”, “write”, and/or “execute”
other permissions	”read”, “write”, and/or “execute”
locked immutable	the locked state of the file
entry type	“File”, “Folder”, “SymbolicLink”, “Socket”, “CharacterSpecial”, “BlockSpecial”, or “Unknown”
type code	HFS type code, as a 4-character string
creator code	HFS creator code, as a 4-character string
owner name	the login name of the owner of the file
group name	the name of the group the file is owned by
owner id	the user id number of the file’s owner
group id	the group id number of the file’s owner

File Property Name	Description
link destination	for symbolic links, this gives the relative path name of the linked-to file

The **permissions** property is a string, similar to that reported by the 'ls -l' command in Unix. It consists of 9 characters, "rwxrwxrwx" indicating read, write, and execute permissions for the owner, group, and others. Any permissions which are not present are replaced by dashes, so a permissions value of "rwxr-xr-x" for example would indicate a file that is readable and executable by everyone but writable only by the owner. The special permissions flags are indicated by additional items appended with separating commas when they apply. These include "setuid", "setgid", and "sticky". So, the permissions value for a setuid file might be "rwxr-xr-x,setuid".

Examples

```
put the short name of file "/tmp/Friday.rtf" -- "Friday"
add the size of file datafile1 to filesizeTotal
```

Tech Talk

Syntax: the *fileProperty* **of [file | folder]** *fileName*

The properties **creation date**, **modification date**, **owner permissions**, **group permissions**, **other permissions**, **permissions**, **type code**, **creator code**, **owner id**, and **group id** may be set (using the `set` command) as well as retrieved. All other file properties are read-only.

To access the properties of a link directly (rather than the file or folder it links to), use the word **link** instead of **file** or **folder**.

The value of the folder property will end with a slash, unless `the folderNamesEndWithSlash` global property is set to false. The slash at the end makes it easy to create a full path name by simply appending a file name.

Asking the User to Choose a File

There are several commands that interact with the user to allow them to select or specify particular files or folders that your script will work with:

```
answer file
answer folder
ask file
ask folder
```

The `answer file` and `answer folder` commands display a standard Open panel for the user to select an existing file or folder, respectively. The `ask file` and `ask folder` commands displays a standard Save panel for the user to select a location in the file system and enter a new file name.

◊ answer file

What it Does

Displays an Open Panel so the user may select an existing file, and returns the full path name of the selected file in the variable `it`.

When to Use It

Use the `answer file` command any time you want the user to supply the name of an existing file. You can then open the file (with the `open` command) or use it in other ways.

The full path name of the file selected by the user will be returned in the variable `it`. If multiple files were selected, a list is returned, with each file name selected by the user in a separate item in `it`. If no file is selected (that is, the user clicks the "Cancel" button in the panel), the variable `it` will be empty following the `answer file` command and `the result` function will return "Cancel".

Examples

```
answer file
answer file "Select Account File" with "Dec03"
answer multiple files "Choose Data Sets" in folder "/Users/mmalone/data"
answer file with button label "Select" of type "png", "tif", or "tiff" \
        in folder "~/Pictures" title "Choose an Image"
```

Tech Talk

Syntax: **answer {multiple | single} file {***Options***}**

Options: **{prompt}** *promptExpr*
[title | titled] *titleExpr*
of type factor {or *factor***}...**
with *defaultFile*
in [folder | directory] *defaultFolder*
allow multiple
{with} {button} label *buttonLabel*

The simplest form of the answer file command is `answer file`. The word **multiple** or **single** may be used before the word **file** to specify whether or not the user should be allowed to select multiple files at once. If not specified, only a single file may be selected.

A number of additional options may be specified as part of this command. None of these options is required, and they may be specified in any order, although no option may be specified more than once.

If the **prompt** or **title** options are specified, the expression given will be used as the title to be displayed at the top of the Open panel. If not given, the word "Open" will be used. If both are specified, only the *promptExpr* will be used.

Tech Talk

One or more file types may be specified using the **of type** option. Several different types may be listed by using commas, the word **or**, or both between types. If this option is included, only files whose extension matches one of the specified types will be accepted (a file extension is the part of the file name following the last period in the name). File extensions may be specified with or without the period (e.g., “eps” and “.eps” are both acceptable). Specify `empty` or “” to include files without any extension.

If **with** *defaultFile* is specified, that file will be selected if it exists. If the value includes a folder as well as a file name, that will be the initial folder shown in the Open panel.

The **in folder** option specifies the initial folder whose contents will be shown in the Open panel.

If the **label** option is specified, *buttonLabel* is used as the label of the default button in the Open panel. If not specified, the button’s label will be “Open”.

If **allow multiple** is specified, then the user will be able to select several files at once. This option is an alternative to the answer multiple files syntax.

◊ answer folder, answer directory

What it Does

Displays an Open Panel so the user may select an existing folder, and returns the full path name of that folder in the variable `it`.

When to Use It

Use the `answer folder` command any time you want the user to supply the name of a folder.

The full path name of the folder selected by the user will be returned in the variable `it`. If multiple folders were selected, a list is returned with each folder name in a separate item of `it`. If no folder is selected (that is, the user clicks the “Cancel” button), `it` will be empty following the `answer folder` command, and the `result` function will return “Cancel”.

Examples

```
answer folder "Select the working directory:"
answer multiple folders "Choose source paths" \
   in folder "~/Documents"
```

Tech Talk

Syntax: **answer {multiple | single} [folder | directory] {***Options***}**

Options: **{prompt}** *promptExpr*
[title | titled] *titleExpr*
with *defaultPath*
in [folder | directory] *defaultFolder*
allow multiple
{with} {button} label *buttonLabel*

The `answer folder` command is almost the same as the `answer file` command, except that the user will be presented with an Open panel which allows folders to be selected, rather than files.

A number of additional options may be specified as part of this command. None of these options is required, and they may be specified in any order, although no option may be specified more than once. See the option descriptions under the `answer file` command above. File type extensions may not be specified with this command.

The value returned will end with a slash, unless `the folderNamesEndWithSlash` global property is set to false. The slash at the end makes it easy to create a full path name by simply appending a file name.

◊ ask file, ask folder, ask directory

What it Does

Displays a Save Panel so the user may specify the name of a file or folder to create, and returns the full path name of the file in the variable `it`.

When to Use It

Use the `ask file` or `ask folder` command when you want the user to specify the name and location of a file or folder that will be created or overwritten by your script.

The full path name of the file name specified by the user will be returned in the variable `it`. If no file is selected (that is, the user clicks the “Cancel” button), the variable `it` will be empty following the `ask file` command, and the `result` function will return “Cancel”.

Note that, as with all standard Save panels, if the user selects an existing file, an alert panel will be displayed, asking whether the file should be replaced. If the user clicks the “Replace” button in this panel, the selected file name will be returned in the variable `it`. The existing file will not be removed or changed in any way, however. It is up to your script to remove the existing file, if possible, before creating a new one with the same name. You can check whether a file exists by using the `there is a file` operator, as in `if there is a file` fileName `then ...`.

Examples

```
ask file "Enter output file name:"
ask file "Temporary file to create:" with "temp1"
ask file "Log Changes" of type ".mylog" in folder "logs"
ask folder "Enter new folder name:" in folder homeFolder
```

Tech Talk

Syntax: ask [file | folder | directory] {*Options***}**

Options: **{prompt}** *promptExpr*
[title | titled] *titleExpr*
of type *requiredExtension*
with *defaultFile*
in [folder | directory] *defaultFolder*
allow multiple
{with} {button} label *buttonLabel*

The simplest form of the ask file command is merely `ask file`. A number of additional options may be specified as part of this command. None of these options is required, and they may be specified in any order, although no option may be specified more than once.

If the **prompt** or **title** options are specified, the expression given will be used as the title to be displayed at the top of the Save panel. If not given, the word "Save" will be used. If both are specified, only the promptExpr will be used.

If the **of type** requiredExtension option is used, the Save panel will use the specified file extension as the type of file to be saved, and will ensure that the file name returned has that extension (a file extension is the part of the file name following the last period in the name). File extensions may be specified with or without the period (e.g., "eps" and ".eps" are both acceptable).

If **with** *defaultFile* is specified, the Save panel will come up with that name already entered in its file name field. If the value includes a folder as well as a file name, that will be the initial folder shown in the Save panel.

The **in folder** option specifies the initial folder whose contents will be shown in the Save panel.

If the **label** option is specified, buttonLabel is used as the label of the default button in the Save panel. If not specified, the button's label will be "Save".

In the case of the `ask folder` command, the value returned will end with a slash, unless `the folderNamesEndWithSlash` global property is set to false. The slash at the end makes it easy to create a full path name by simply appending a file name.

File, Socket, Process, and Stream Input and Output

There are several commands, functions, and global properties for working with files, sockets, processes, and streams in your scripts:

```
open file / open socket / open process
close file / close all / close socket / close process
read
seek
write
openFiles() / openSockets() / openProcesses()
the defaultStringEncoding
```

```
the readTimeout
the umask
```

The file input and output commands (`open file`, `close file`, `read from file`, `seek in file`, and `write … to file`) are for creating and accessing text or binary files on your system. Use them to read and write data that is stored in the file system.

In addition to using these commands, you can access a file directly as a container within your script. Accessing a file as a container provides the simplest means of reading or manipulating its contents, but provides less control and is somewhat less efficient when performing multiple operations on the same file than the commands described here.

The socket input and output commands (`open socket`, `close socket`, `read from socket`, and `write … to socket`) perform similar operations on sockets, permitting you to open a connection to a socket provided by another process and read and write data through that connection. You cannot seek to a specified location on a socket, because a socket is just an open communication channel between two processes.

Similarly, the process input and output commands (`open process`, `close process`, `read from process`, and `write … to process`) allow you to launch an external process and communicate with it by writing to the standard input and reading from the standard output of that process.

The `read` and `write` commands can also be used to read from standard input and to write to the standard output or standard error streams (`read from input`, `write … to output`, `write … to error`).

The `openFiles()` and `openSockets()` functions return lists of all of the currently open files or sockets, respectively.

`The defaultStringEncoding` global property controls the encoding format that is used when reading or writing text from a file or socket. `The umask` affects the file permissions of a file or folder when it is created by SenseTalk.

◊ open file

What it Does

Opens or creates a file for reading or writing or both.

When to Use It

The `open file` command must be used to open a file before anything can be read from or written to that file using the `read` or `write` commands. When you are finished with a file, it should be closed using the `close file` or `close all` commands.

Examples

```
open file myFile
open file "/etc/passwd" for reading
open file "~/.gamescores" for appending
```

Tech Talk

Syntax: open file *fileName* **{ for [reading | writing | readwrite | appending | updating] }**

When you open a file, you may optionally specify the manner in which the file will be accessed. The file may be opened for reading only, for writing only, or for both reading and writing. The default mode is **updating**, if you open a file without specifying the manner of access.

- **reading** read only (file must exist already, will not create a file)
- **writing** write only (replaces existing file, if there was one)
- **readwrite** reading and writing, starting at beginning of existing file
- **appending** reading and writing, starting at end of existing file
- **updating** the same as “readwrite” except file is never truncated

All of the modes except for reading will create the file (including the full path to it) if it doesn’t already exist. If you want to open a file only if it already exists, you can check for its existence using the unary operator file fileName exists or there is a file fileName as shown in this example:

```
if there is a file myFile then
   open file myFile for updating
else
   answer "File " & myFile & " doesn't exist!"
   exit all
end if
```

The **readwrite**, **appending**, and **updating** modes all open the file for both reading and writing. However, a file opened in **readwrite** mode will be truncated following the last (highest) character position in the file that is written to. If nothing is written to the file it will be left unchanged.

The **appending** mode is simply a convenience that automatically seeks to the end of the file as it is opened (rather than starting at the beginning of an existing file) so that additional text can be written at the end without overwriting any text that is already in the file.

The following table summarizes the differences between the various modes:

Mode	Can Read	Write / Create	Starts At	Existing File
Reading	yes	no	beginning	unchanged
Writing	no	yes	beginning	replaces existing file
ReadWrite	yes	yes	beginning	truncates after highest write
Appending	yes	yes	end of file	never truncated; may grow
Updating	yes	yes	beginning	never truncated; may grow

Use the `openFiles()` function to get a list of all files which are currently open.

◊ open socket

What it Does

Opens a socket connection for bidirectional communication (reading and writing data) to another process.

When to Use It

The `open socket` command must be used to open a socket before anything can be read from or written to that socket using the `read` or `write` commands. When you are finished with a socket, it should be closed using the `close socket` command.

`Open socket` establishes a TCP socket connection to another process (program). The other process may be running on the same computer, or on some other computer on the network. It must already be running, and have registered a socket to which SenseTalk can connect. Once the connection is established, data may be transmitted in either direction in whatever manner both sides understand.

Examples

```
open socket remoteListener
open socket "192.168.1.4:22"
open socket "localhost:5900#2"
```

Tech Talk

`Syntax: open socket` *socketIdentifier*

The *socketIdentifier* must be of the form *host:port* where *host* is the name or IP address of the machine, and *port* is the port number on that machine of the socket to be connected to.

The *socketIdentifier* may optionally end with a pound sign "#" (or a vertical bar "|") character followed by an arbitrary number or identifier string. This serves the purpose of allowing you to create multiple identifiers to establish more than one connection to the same host and port, and identify each connection uniquely – just use the appropriate *socketIdentifier* with the `read`, `write`, and `close` commands to identify the correct connection.

If the socket connection cannot be established within the time specified by the `readTimeout` global property, an exception will be thrown. Use the `openSockets()` function to get a list of all sockets which are currently open.

◊ open process

What it Does

Launches an external process and opens a connection through which the script may interact with that other process.

When to Use It

Use the `open process` command whenever a script needs to launch and interact with an external process. If all that is needed is to run an external process and receive any output from that process when it completes, the `shell()` function provides a much simpler way to achieve that. The open process mechanism, on the other hand, provides much greater flexibility, allowing the script to conduct complex interactions with another process, or to start a lengthy operation without blocking the script and retrieve the results of that operation at a later point in the script.

`Open process` launches another process (program) which may be (and most commonly is) a shell through which still other programs may be executed. Once the other process is launched, a connection is established and text may be transmitted in either direction – the script may write to the standard input and read from the standard output of the other process.

Examples

```
open process preferredShell & "#myshell"
open process "/bin/sh" with options myOptions
open process "/usr/local/bin/mysql#2"
```

Tech Talk

Syntax: open process *processIdentifier* **{with options** *options***}**

The *processIdentifier* should be in the form *processPath#identifier* where *processPath* is the full path of the process to run. If *processPath* is omitted, a shell process will be launched (as specified by `the shellCommand` global property). The *#identifier* portion is also optional – it merely serves as a way to make a *processIdentifier* unique, so that a script can open and interact with multiple processes at once that use the same *processPath*.

If *options* is used, it should be a property list that may include any of these properties:

parameters	a list of values to be passed as parameters to the process when it is launched
folder or directory	the current directory where the process will be run
environment	a property list specifying environment variables and their values

If the process cannot be launched, the `result()` function will be set to an exception (or the exception will be thrown, if `the throwExceptionResults` global property is set to true). The `openProcesses()` function can be used to get a list of all processes which are currently open.

◊ close file, close socket, close process, close all

What it Does

Closes an open file, socket, or process, or all open files, sockets, or processes.

When to Use It

Use the `close file` command when your script has finished accessing a file. Use the `close socket` command to close an open socket when you are done with it. Use the `close process` command to close the script's connection to an open process and terminate that process.

Use the `close all files` command to close all of the currently open files when your script is done working with all of them. Similarly, the `close all sockets` and `close all processes` commands can be used to close all of the currently open sockets or processes. SenseTalk automatically closes all open files, sockets, and processes whenever it stops executing your scripts, but it is good practice to for your script to close them when it is done working with them.

Examples

```
close file "/etc/passwd"
close all files  -- close all open text files
close socket "localhost:5900"
close process "#9"
```

Tech Talk

```
Syntax: close file fileName
        close socket socketIdentifier
        close process processIdentifier
        close all { files }
        close all sockets
        close all processes
```

The `close all files` command closes all currently open files, regardless of which script or handler opened the file. This could be potentially problematic if files have been opened by other scripts, and are still in use. Use the `openFiles()` function to get a list of all open files. The same applies to the close all sockets and close all processes commands.

The `close file` command closes an open file that was previously opened with the `open file` command. *FileName* should be an expression which gives the name of the file to be closed. As with the `open file`, `read` and `write` commands, the file name should be either the full path name of the file, or should be given relative to the current working folder (as returned by `the folder`).

The `close socket` command closes a socket that was previously opened with the `open socket` command. The *socketIdentifier* should be identical to the identifier used when opening the socket.

The `close process` command closes a process that was previously opened with the `open process` command. The *processIdentifier* should be identical to the identifier used when opening the process.

◊ read

What it Does

Reads data (text or numbers) from an open file, an open socket, an open process, or from the standard input stream.

When to Use It

Use the `read` command to read data from a file, socket, process, or standard input. Data is read into the variable `it` or into a destination container if specified (using an `into` clause). When there is no more data to read, the destination container will be empty. Any time less data is read than was requested, the `result` function will contain a value giving the reason (such as “time out”, or “eof” if the end of file is reached).

The `read` command can read a specified number of characters, words, items, or lines; can read until a specified delimiter is found; or can read a list of one of several different types of numeric values. Reading begins at the current position, or, in the case of a file, a starting position may be specified (using an `at` clause). The syntax of the read command is flexible, allowing the various options to be specified in any convenient order.

Examples

```
read from file myFile for 20 -- read the next 20 chars into it
read from input until return -- read text typed by user
read from process mysql until end -- read available text
```

```
read into numList from socket inStream for 3 unsigned integers
read 2 lines into couplet from file sonnet
read into inQueue 5 items from file dataFile at 100
read 6 unsigned 8-bit integers from socket rfb into unitSales
read 10 chars from socket "192.168.1.4:22" in 15 seconds
```

Here is an example showing one way to read and process an entire file:

```
open file "/tmp/abcd"
repeat forever
   read from file "/tmp/abcd" until return -- read one line
   if it is empty then exit repeat -- we've reached the end of the file
   put it -- or do other processing with 'it' here
end repeat
close file "/tmp/abcd"
```

Tech Talk

Syntax: read {*Options*}

Options:

from file *fileName*
from socket *socketIdentifier*
from process *processIdentifier*
from [**input** | **stdin**]
at *startpos*
into *container*
until [*terminator* | **eof** | **end**]
{**for**} **quantity** {*dataType*}
in *timeLimit*

One of the three `from` options is required, to specify the source from which to read. All other options are optional, but only one of each type may be specified. If neither a `for` nor `until` option is given, a single character is read.

When reading from a file, the *fileName* expression must yield the name of a file that was previously opened with an `open file` command. It must have been opened in a mode that permits reading (that is, not "for writing" only). You do not need to open the standard input stream – it is always open (you may refer to it as stdin instead of input if you prefer).

When reading from a socket, the *socketIdentifier* expression must yield the exact identifier used when a socket was previously opened with the `open socket` command.

When reading from a process, the *processIdentifier* expression must yield the exact identifier used when a process was previously opened with the `open process` command. The value that is read corresponds to the standard output from the process.

If at *startpos* is specified, reading from the file will begin at that character position within the file (if *startpos* is negative, it specifies the number of characters back from the end of the file where reading will begin). Otherwise, the first character to be read will be the one at the current position in the file, as determined by the most recent prior `read`, `write`, or `seek` command in that file. If at startpos is specified when reading from a socket, process, or the standard input, it is simply ignored, since those sources cannot seek to a location.

Tech Talk

If into *container* is specified, the data that is read will be put into the given container. If an into option is not specified, the data will be read into the special `it` variable.

If until *terminator* is specified, all of the characters from the starting position until the next occurrence of the specified character or string will be read. This is useful for reading one line at a time from the source (by using `return` as the terminating character), or to read just until some other delimiting character (such as a `tab`). The *terminator* may be more than one character in length, and will be returned as part of the value that was read. Specifying `until eof` or `until end` will read all the way to the end of the file, or to the end of input from a socket or stream. The standard input stream indicates it is at the end after a Control-D character is received. For sockets and processes, the `until eof` or `until end` option will wait until either some input is available, or the duration of the readTimeout or **in** *timeLimit* (see below) has elapsed. This greatly simplifies reading when some input is expected.

If for *quantity dataType* is used, the number of characters or other data elements specified by *quantity* are read from the file. If *dataType* is a text chunk type (`characters`, `words`, `items`, or `lines`), text is read until the requested amount is available. The final delimiter (if any) is not included with the text that is read. If no *dataType* is given, characters are assumed (and the word `for` is required in this case).

If you specify a numeric *dataType* instead of a text chunk type, the value stored into `it` or *container* by the read will be a *list* of the data values that were read. The following numeric data types may be used:

DataType	Value
`int1` or `8-bit integer`	an 8-bit (or 1 byte) signed integer
`uint1` or `unsigned 8-bit integer`	an 8-bit (or 1 byte) unsigned integer
`int2` or `16-bit integer` or `short integer`	a 16-bit (or 2 byte) signed integer
`uint2` or `unsigned 16-bit integer`	a 16-bit (or 2 byte) unsigned integer
`int4` or `32-bit integer` or `integer`	a 32-bit (or 4 byte) signed integer
`uint4` or `unsigned 32-bit integer`	a 32-bit (or 4 byte) unsigned integer
`int8` or `64-bit integer`	a 64-bit (or 8 byte) signed integer
`uint8` or `unsigned 64-bit integer`	a 64-bit (or 8 byte) unsigned integer
`real4` or `32-bit real` or `float`	a 32-bit (single-precision) floating-point number
`real8` or `64-bit real` or `double`	a 64-bit (double-precision) floating-point number

The in *timeLimit* option gives the maximum time the `read` command will wait for the requested data to become available. If a time is not specified, the value of the `readTimeout` global property will be used instead. If the requested data is not read within the time specified by timeLimit or readTimeout, whatever has been read will be returned and `the result` function will be set to indicate “time out”.

◊ seek

What it Does

Sets the position within a file where the next `read` or `write` command will occur.

When to Use It

Use the `seek` command to set the current position in a file prior to performing a `read` or `write` in that file. Although the `read` and `write` commands both provide "**at** startpos" options to specify the starting position for that operation, the `seek` command provides additional flexibility in that the position in the file can be specified relative to the current location, as well as from the beginning or end of the file.

Examples

```
seek in file myFile to -10 from the current position
seek in file "~/.gamescores" to the end
```

Tech Talk

Syntax: `seek in file` *fileName* **[at | to]** *position* **{ from {the} [start | beginning | current position | end] }**

The *fileName* expression must yield the name of a file that was previously opened with an `open` command.

Position is a numeric expression which specifies the location to seek to in the file. If one of the "**from ...**" options is not used to specify the origin of the seek, then a positive *position* indicates the number of characters from the beginning of the file, and a negative *position* indicates the number of characters back from the end of the file. Instead of a number or numeric expression, *position* can also be "**the end**", which means the same as

```
seek in file ... to 0 from the end
```

If "**from ...**" is included, it specifies whether position is relative to the beginning or end of the file or from the current position.

◊ write

What it Does

Writes data into a file, to a socket or process, or to the standard output or standard error stream.

When to Use It

Use the `write` command to store data in a file on disk. The data can then be read from the file again at a later time by your scripts, or by another application entirely. You can also use this command to write data to a socket or process or to the standard output or error streams.

Examples

```
write line 1 of accounts & return to file myFile
write "ls -l" & return to process "#4"
write highScore & tab to file "~/.gamescores" at eof
write "Please enter your account id: " to output
write numberList as 16-bit integers to file bData
write (2,3,5,9) as 8-bit integers to socket msock
write 24.672 as float to file fn at 20
```

Tech Talk

Syntax: write *data* **{as** *dataType***} to file** *fileName* **{at [***startpos* **| end | eof]}**
write *data* **{as** *dataType***} to socket** *socketIdentifier*
write *data* **{as** *dataType***} to process** *processIdentifier*
write *data* **{as** *dataType***} to [output | stdout | error | stderr]**

Data can be any valid SenseTalk expression. If *dataType* is not specified, the value of the *data* expression is treated as a string of characters, which is written out to the specified file, socket, or stream.

The *fileName* expression must yield the name of a file that was previously opened with an `open file` command. It must have been opened in a mode that permits writing (that is, not "for reading" only). When you are finished accessing a file, it should be closed with the `close file` command to ensure that all of the data written out is saved properly to the disk. The standard output stream (designated by `output` or `stdout`) and the standard error stream (`error` or `stderr`) do not need to be opened or closed.

The *socketIdentifier* expression must yield the identifier of a socket that was previously opened with the **open socket** command.

The *processIdentifier* expression must yield the identifier of a process that was previously opened with the **open process** command. The data that is written will be sent to the standard input of the process.

If **at** *startpos* is specified, writing to the file will begin at that character position within the file (if *startpos* is negative, it specifies the number of characters back from the end of the file where writing will begin). Using the **at end** or **at eof** option will tell SenseTalk to write the *data* at the end of the file, following any other text already in the file. Otherwise, if no location is specified, data will be written beginning at the current position in the file, as determined by the most recent prior `read`, `write`, or `seek` command in that file.

Writing into a file at a position that already contains data will cause that data to be overwritten. To insert text into the middle of an existing file, you must read all of the text in the file from that point to the end and store it in a container. Then the text to be inserted can be written out, followed by the stored text.

If an existing file was opened in **readwrite** or **appending** mode then writing to the file will cause data to be dropped from the file beyond the highest position in the file which was written. To avoid this file truncation, open the file in **updating** mode (see the `open file` command for more information about these modes).

If as *dataType* is specified, the data is converted to that binary format before being written. In this case, *data* may be a list of numeric values, which are all converted to the same data type. See the `read` command for a list of the valid data types.

◊ openFiles function

What it Does

Returns a list of the files which are currently open as a result of the `open file` command.

When to Use It

Use the `openFiles()` function to obtain a list of all files which are currently open. For instance, the following example shows how you might use this information to close all open files whose names end in ".dat".

Examples

```
repeat with each item of the openFiles
   if it ends with ".dat" then close file it
end repeat
```

Tech Talk

```
Syntax: the openFiles
        openFiles()
```

◊ openSockets function

What it Does

Returns a list of all sockets which are currently open as a result of the `open socket` command.

When to Use It

Use the `openSockets` function to obtain a list of all sockets which are currently open. For instance, the following example shows how you might use this information to close all sockets that are currently open to host 192.168.1.12:

Examples

```
repeat with each item of the openSockets
   if it begins with "192.168.1.12:" then close socket it
end repeat
```

Tech Talk

```
Syntax: openSockets()
        the openSockets
```

◊ openProcesses function

What it Does

Returns a list of all processes which are currently open and available for interaction as a result of the `open process` command.

When to Use It

Use the `openProcesses` function to obtain a list of all processes which are currently open. For instance, the following example shows how you might use this information to send a logout command to all ssh processes that are currently open:

Examples

```
repeat with each item of the openProcesses
   if it begins with "/usr/bin/ssh" then
        write "logout" & return to process it
   end if
end repeat
```

Tech Talk

```
Syntax: openProcesses()
        the openProcesses
```

◊ the defaultStringEncoding global property

What it Does

`The defaultStringEncoding` specifies how text strings are encoded whenever they are read from or written to a file, socket, or URL. This setting is used by the `read` and `write` commands, and also when treating either a file or a URL as a container.

When to Use It

Use `the defaultStringEncoding` to control the way text is interpreted when it is being read or written. Within SenseTalk, text characters are simply what they are: the letter “A” is the letter “A”, and the letter “é” is the letter “é” (an e with an acute accent). Externally, though, there are a number of different ways that the same characters might be represented as sequences of bits and bytes. Each different text representation is called an “encoding”. The standard encoding used by SenseTalk is UTF8, which is a widely-used 8-bit system for encoding Unicode characters.

Examples

```
put the defaultStringEncoding into origEncoding
set the defaultStringEncoding to "Unicode" -- full 16-bit unicode text
put myInternationalText into file "/tmp/twoByteText"
set the defaultStringEncoding to origEncoding -- restore original value
```

Tech Talk

```
Syntax: the defaultStringEncoding
```

Acceptable values for `the defaultStringEncoding` include: UTF8, Unicode, ASCII, and many others. Use the availableStringEncodings() function for a full list of encodings.

◊ the availableStringEncodings function

What it Does

Returns a list of the names of all the available string encoding formats. In some cases there may be more than one name for the same encoding.

When to Use It

Use the `availableStringEncodings` function to learn the names of the encodings that are available to use when setting `the defaultStringEncoding` global property.

Examples

```
put the availableStringEncodings
```

Tech Talk

```
Syntax: the availableStringEncodings
        availableStringEncodings()
```

◊ the umask global property

What it Does

Sets or retrieves the posix permissions mask for newly created files.

When to Use It

Use `the umask` property to control the access permissions for any file or folder created by SenseTalk, either directly or indirectly. The mask is a 3-digit number: the digits control file permissions for the user, the group, and others, respectively. Each digit is a number from 0 to 7 indicating the permissions that will be *blocked*, with the value 4 used to block read permission, 2 to block write permission, and 1 to block execute permission. The values may be added together to block more than one permission.

Examples

```
put the umask into origMask
```

```
set the umask to 247 -- set so user can't write, group members can't read, and
others have all permissions blocked
create file "/tmp/permissionsTest"
set the umask to origMask -- restore it to its original value
```

Tech Talk

Syntax: the umask

Working with URLs and the Internet

This section describes how SenseTalk can be used to access resources on the internet through their URLs (Uniform Resource Locators). Through this mechanism, SenseTalk can access the contents of files and websites across the internet and communicate with remote servers using standard protocols.

A number of supporting functions are available to make it easy to convert information between standard URL formats.

Referring to URL Resources in a Script

To refer to a file or other internet resource accessed through a URL in a script, just use the word `url` followed by an expression that evaluates to the URL of the file or resource.

Examples

```
put url "http://www.somewhere.com/somepage.html" into htmlContents
put htmlContents into url "file://localhost/localCopy.html"
put (scheme:"http", host:"www.apple.com") into applePage
```

Tech Talk

Syntax: `url` *urlString* **`{with headers`** *headerPropertyList***`}`**
`url` *urlPropertyList* **`{with headers`** *headerPropertyList***`}`**

The *urlString* may be a “file:”, “http:”, or “ftp:” type of URL. URLs of type “file:” can be treated as containers — that is, you can store into or change their contents, provided you have write access to the file. If a *urlPropertyList* is used, it may contain “scheme”, “host”, “path”, “query” and other keys as defined by the `makeURL()` function (described later in this section)

When retrieving an http resource from the internet using this syntax, an http GET operation is used. The *urlString* or *urlPropertyList* may include query data to simulate submitting a form. If you use a string, the `makeURL()` function can be used to easily construct a string containing a properly formatted query.

If with headers *headerPropertyList* is specified, *headerPropertyList* should be a property list containing any custom HTTP headers to be used in accessing that URL. The custom header information is passed along with any standard headers for both GET (when accessing a URL) and POST (when using the `post` command) operations.

After accessing a URL, you may check the status of the remote operation by calling the `result()` function on the next line of the script. This will return the status value of the request if it is less than the minimum level treated as an error (as specified by `the URLErrorLevel` global property) or an exception if the status is at that level or greater and `the throwExceptionResults` is set to false.

The data retrieved from the URL is interpreted as text according to the current setting of `the defaultStringEncoding` global property. Set this property to the correct encoding (often "UTF8") before fetching the URL data.

Configuring URL Behavior

◊ the URLCacheEnabled global property

`The URLCacheEnabled` global property controls whether the contents retrieved from a URL resource may be cached. When set to `true` (the default), URL contents may be cached by the system and retrieved from the cache rather than being fetched remotely each time. To force a fresh copy of the URL to be loaded, set `the URLCacheEnabled` property to `false` before accessing a URL, as in this example:

```
set the URLCacheEnabled to false
put url "http://www.apple.com" into appleHomePage -- get a fresh copy
set the URLCacheEnabled to true -- restore caching
```

◊ the URLTimeout global property

When fetching a URL resource, `the URLTimeout` global property specifies the maximum time allowed before the request will time out. If this time limit is exceeded or there is no internet connection available, or if some other error occurs, `the result` function will be set to return an exception object identifying the error, which can be accessed on the next line of the script.

◊ the URLErrorLevel global property

The value of `the URLErrorLevel` global property is an integer indicating the lowest URL status value that is treated as an error when fetching the contents of a URL. The default value is 400, so a returned status value of 400 or above will either throw an exception or set the `result()` function to an exception (depending on the current setting of `the throwExceptionResults` global property). You may set `the URLErrorLevel` to zero (or to any sufficiently large number) to prevent this type of exception from being generated.

Internet and URL Commands and Functions

SenseTalk provides several commands and functions for performing transactions over the internet and manipulating URLs and data in standard formats.

The `post` command posts data to a URL and retrieves the results. The `makeURL()` function helps in constructing a properly-formed URL string from a given set of components, and its companion `extractURL()` function extracts the components from a URL string. The URLEncode and URLDecode functions convert text to and from the special encoded format commonly used in URL queries and data passed across the network.

◊ post command

What it Does

Posts data to a URL address on the web (using an http POST command) and retrieves the results.

When to Use It

Use the `post` command to conduct a transaction with an online web-based service, simulate the posting of a web form, or similar interactions with a web server. Results returned by the web server are stored in the variable `it`.

Examples

```
post queryData to url "http://someservice.net" with \ headers ("Content-
Type": "text/xml")
post URLEncode(dataToPost) to url makeURL(urlComponents)
put it into postResults
```

Tech Talk

Syntax: post *data* to **url** *theURL*

The *data* can be any text string, or it may be a property list, in which case it will be formatted automatically in the standard query format. The `post` command waits up to the maximum time specified by `the URLTimeout` global property to receive a response back from the web server before proceeding with the next line of the script. Whatever response is received is stored in the variable `it`. If a response is not received within the allotted time or some other error occurs, `it` will be empty, and `the result` will be set to an exception object indicating the error.

◊ makeURL function

What it Does

Creates a URL string from a supplied set of component parts.

When to Use It

Use the `makeURL` function to create a properly-formatted URL string using a property list containing components of the desired URL. The `makeURL` function will automatically encode each part of the URL as appropriate. This function is called implicitly whenever a URL expression with a property list instead of a string is used

Examples

```
get makeURL(scheme:"http", host:"myserver.net", path:"slinky.html")
put url makeURL(host:"google.com", path:"search", \ query:("q":"EggPlant",
"ie":"UTF-8") ) into searchResults
```

Tech Talk

Syntax: makeURL(*componentPropertyList***)**
the makeURL of *componentPropertyList*

The *componentPropertyList* should be a property list containing the components that will be used to construct the resulting URL string. Any of the following properties may be specified; other properties will be ignored.

`scheme`	the scheme or protocol, such as “http”, “ftp”, or “file” (if not specified, http will be assumed)
`host`	the host machine (web server) the URL refers to

path	the path to the resource on the host web server
port	the port number to connect to
user	the user name to log in on the server (used sometimes for ftp URLs)
password	the password used to log in on the server (used along with **user** for some ftp URLs)
parameters	additional parameters to include (such as a “type” parameter for an ftp URL)
fragment	the fragment identifies a particular anchor point within a resource
query	a property list containing keys and values to be passed to the web server, such as would be supplied by a form on a web page

See Also: the `extractURL` and `URLEncode` functions, below.

◊ extractURL function

What it Does

Extracts the component parts from a URL string, returning them as a property list.

When to Use It

Use the `extractURL` function to easily separate a URL string into a property list containing its component parts. The `extractURL` function will automatically decode each part of the URL as appropriate.

Examples

```
put extractURL("http://myserver.net/slinky.html") into urlParts
```

Tech Talk

Syntax: extractURL(*urlString***)**
the extractURL of *urlString*

The *urlString* should be an expression that evalutes to a string in the standard URL format. The value returned by the extractURL function will be a property list containing the component parts that were extracted from that URL string. If *urlString* is empty, the returned property list will be empty, otherwise it will contain one or more of the following keys, depending on what the URL string contained:

scheme	the scheme or protocol, such as “http”, “ftp”, or “file”
host	the host machine (web server) the URL refers to
path	the path to the resource on the host web server
port	the port number to connect to
user	the user name to log in on the server (used sometimes for ftp URLs)

password	the password used to log in on the server (used along with user for some ftp URLs)
parameters	additional parameters to include (such as a "type" parameter for an ftp URL)
fragment	the fragment identifies a particular anchor point within a resource
query	the value of this key will be a property list containing keys and values present in the urlString that would be passed to the web server, such as would be supplied by a form on a web page

See Also: the `makeURL()` function, earlier in this section.

◊ URLEncode function

What it Does

Substitutes '+' for each space in a string, and replaces other non-alphanumeric characters with '%xx' encoded values as used in URL strings. If the parameter is a property list, it is encoded as a URL query string in the same way that a query is encoded by the `makeURL()` function.

When to Use It

Use the `URLEncode` function to encode a string in a manner suitable for passing data as part of a URL string used to get an internet resource.

Examples

```
put urlEncode(myData) into encodedData
```

Tech Talk

Syntax: URLEncode(*string***)**
the URLEncode of *string*

The `URLEncode` function replaces each space in string with a '+' character, and each non-alphanumeric character other than space with a three-character code of the form '%xx', where xx is a two-digit hexadecimal number representing the replaced character's ASCII value.

See Also: the `URLDecode` function, below, and the `makeURL` function, above.

◊ URLDecode function

What it Does

Converts characters encoded as '%xx' back into their normal text form, and '+' characters into spaces.

When to Use It

Use the `URLDecode` function to decode data from a URL string. This is the inverse of the `urlEncode` function.

Examples

```
put urlDecode(encodedData) into myData
```

Tech Talk

Syntax: URLDecode(*encodedString***)**
the URLDecode of *encodedString*

The URLDecode function replaces each triplet from encodedString of the form '%xx' (where xx is a two-digit hexadecimal number) with the character having that ASCII value. Each '+' character in encodedString is replaced by a single space character.

Working with Trees and XML

SenseTalk's "tree" structure provides the ability to easily read data in XML format, access and manipulate that data within a tree, and produce XML from it. A tree is a hierarchical data structure which behaves as both a list and a property list (with some restrictions). As a list, a tree contains items – sometimes called "nodes" – which are also trees. As a property list, a tree has properties which correspond to "attributes" of a node in XML terminology.

The tree capabilities in SenseTalk are provided by the STTreeNode XModule. Typically, this XModule is loaded automatically on launch. Whenever this external module has been loaded, the features described in this section are available for working with trees and their contents.

Note: `loadedModules()` function

To verify what modules are loaded, you can use the `loadedModules()` function, described in Other Commands and Functions. The tree module appears as "ST_Tree" in the list.

Trees and Nodes

A tree is a hierarchy that consists of a "root node" that may have any number of "child" nodes. Each child node may itself be a tree that may have any number of child nodes and so forth to any depth. Each node (except the root node) has a "parent" node, which is the tree (or subtree) that contains that node as a child.

In addition to having a parent node and zero or more child nodes, each node may also have a number of properties or attributes, including a tag name.

The basic structure of a tree is like this:

```
Root
|
|------ child1
|   |
|   |—— grandchild1
|   |
|   |—— grandchild2
|
|------ child2
|
|------ child3
|   |
|   |—— grandchild3
|         |
|         |—— great-grandchild1
|------ child4
```

Because every node in a tree can have its own child nodes, each node (together with its children and later descendants) is itself a tree.

Trees and XML

While a tree can be useful for storing various types of hierarchical data, the tree structure in SenseTalk is specifically designed for working with XML documents or XML-based data. XML (which stands for "eXtensible Markup Language") is a rich, flexible, and complex language that is used as the underlying foundation for a huge variety of data formats in use today.

SenseTalk's tree structure simplifies working with XML documents and data structures, making it easy to access individual values, while providing full access to all parts of a document when needed.

To fully understand trees and their relationship to XML it will be helpful to have at least a basic understanding of the XML structure and some of its terminology. Consider the following example XML document:

```
<?xml version="1.0"?>
<order id="001">
<customer name="Janet Brown"/>
<product code="prod345" size="6">
<quantity>3</quantity>
<amount>23.45</amount>
</product>
</order>
```

Here, the first line identifies this as a version 1.0 XML document. The second and last lines wrap the rest of the content in "order" tags. This entire section constitutes either a "document" node (as in this case) or an "element" node (if the xml version info was not present) that contains two other elements: customer and product. The customer element has a name "attribute" but no additional content. The product element has code and size attributes, and also contains two more elements: quantity and amount. Both the quantity and amount elements contain enclosed text, known as text nodes.

We might represent this information in a tree like this:

```
order (document with 'id' attribute)
|
|── customer (element with 'name' attribute)
|
|── product (element with 'code' and 'size' attributes)
    |
    |── quantity (element)
    |     |
    |     |── 3 (text)
    |
    |── amount (element)
          |
          |── 23.45 (text)
```

The information in parentheses indicates the type of node, and the attributes, if any, that are present in that node. In tree form, we can see that the order node has two children (customer and product) and that the product node in turn also has two children (quantity and amount). The quantity and amount nodes each have one child: a text node holding the corresponding value.

Looked at in this way, we can see that XML data consists mainly of elements (the document node can be treated as a special type of element). Each element has a tag ("order", "customer", "product", etc.), may have attributes ("id", "name", "code", etc.), and may also have children. An element's children may be elements, too, or may be simple text values ("3", "23.45"). There are other node types as well, including processing instructions and comments, but they are less common and aren't shown in this example.

The order example presented here is fairly typical of many XML formats in use today. By combining elements, attributes, and text in a nested structure, XML allows for a wide variety of different formats, and variations within them.

Tree = List + Property List

SenseTalk 's approach to working with trees leverages the capabilities inherent in the language for dealing with lists and property lists, by treating a tree as a hybrid of both container types. The children of a tree can be accessed just like the items in a list; and its attributes can be accessed like the property values in a property list.

Some details are different for trees (as discussed below), but on the whole if you're familiar with working with lists and property lists, you already know most of what you need to work with trees as well. See Lists and Property Lists for details.

Because trees have these characteristics, they may be useful even in situations that have nothing to do with XML, as a hybrid container type that behaves as both a list and a property list.

Working With Trees

Creating a Tree from XML

To load the contents of a URL containing XML data into an internal tree structure within a script, simply use the "as tree" operator:

```
put url "http://some.site/data.xml" as a tree into myTree
```

This statement accesses the indicated URL and reads its contents. The "as a tree" operator tells SenseTalk to treat that data as an XML document and convert it into a tree structure which is then stored in the variable `myTree`.

Sometimes it may be more convenient to start with XML contained directly in a script. The "as tree" operator works equally well for this:

```
set XMLSource to {{
<order id="001">
<customer name="Janet Brown"/>
<product code="prod345" size="6">
<quantity>3</quantity>
<amount>23.45</amount>
</product>
</order>
}}
set order to XMLSource as tree
```

We will refer to this order tree in other examples below.

Creating XML from a Tree

Producing text in XML format from the data in a tree is even easier than creating a tree from XML text. Whenever a tree structure is used as text, it automatically creates an XML representation of the tree's contents. So all that is required to make an XML file from a tree, for example, is a simple command like this:

```
put myTree into file "/path/to/aFile.xml"
```

Accessing Tree Content

The children of a tree can be accessed just like the items of a list:

```
put item 1 of order -- <customer name="Janet Brown"></customer>
```

The attributes of a tree can be accessed just like the properties of a property list:

```
put order.id -- 001
```

Combining items and properties provides access to more deeply nested data:

```
put the code of item 2 of order -- prod345
```

Accessing Tree Nodes Using XPath Expressions

XPath is a standard mechanism for accessing content in XML documents. It provides a way to describe the node or set of nodes that you are interested in, and extracting the desired information for you. SenseTalk supports this powerful mechanism through `node` expressions, which allow you to access content within a tree by tag name:

```
put node "customer" of order -- <customer name="Janet Brown"></customer>
```

Node expressions can describe a path to a nested node of a tree:

```
put node "product/amount" of order -- <amount>23.45</amount>
```

A special "text" property helps to extract just the content of a tree or node:

```
put the text of node "product/amount" of order -- 23.45
```

Use `all nodes` or `every node` to return a list of every node of a tree that matches an XPath expression:

```
put all nodes "product/*" of order -- (<quantity>3</quantity>,<amount>23.45</
amount>)
put every node "*/amount" of order -- (<amount>23.45</amount>)
```

The `nodePath` function will return an XPath expression for a particular node within a tree:

```
put nodepath of item 2 of item 2 of order -- order/product[1]/amount[1]
```

A `node` expression (but not `all nodes`) can also be used as a container that can be stored into to alter the contents of the tree:

```
put 7 into node "product/quantity" of order
```

XPath expressions include many different options for accessing specific nodes, only a few of which were shown here. For more details on using XPath, see http://en.wikipedia.org/wiki/XPath or the full specification at http://www.w3.org/TR/xpath

Three Special Properties: _tag, _children, _attributes

There are a number of property names (all beginning with an underscore character) which have special meaning in a tree. The three most important ones are:

The "_tag" property refers to the tag name of an element.

The "_children" property refers to the children of a tree. Its value is a normal list containing all of the child trees.

The "_attributes" property refers to the attributes of a tree. Its value is a normal property list whose keys and values

are the names and values of all of the tree's attributes. Through the _attributes property it is possible to access any attributes of a tree, including those that have the same name as one of the special tree properties.

In addition to the _tag, _children, and _attributes properties, a Document node may also have an _xmlinfo property, described below under Converting a Tree to Text.

Creating an Empty Tree

To create a tree entirely within a script rather than starting from an existing XML document, start with an empty tree to which content can be added:

```
put an empty tree into order
```

The tree produced by this statement is ready to accept children or attributes. It does not have a tag name, so a recommended second step would be to set its "_tag" property:

```
set order's _tag to "order"
```

Setting XML Attributes of a Tree

A tree's properties correspond to the "attributes" of an XML element. They are containers, and can be set just like the properties of a property list are set:

```
set order's id to "001"
```

The only limitations on setting a tree's properties are that values are always converted to text when they are set; and property names must conform to the rules for standard XML identifiers (which are the same as for identifiers in SenseTalk: they must begin with a letter or underscore, and contain only letters, underscores, and digits).

Adding Children to a Tree

The children of a tree are accessed like items in a list. To add a new child, use the `insert` command:

```
insert << <customer name="Janet Brown"/> >> into order
```

Children must be trees, or values that can be converted into a tree. Values are converted automatically when they are added to a tree, using the same rules as the `tree` function, described later in this section. Only nodes that have a nodeType of Document, Element or DTD can have children. Other types of nodes do not have children and do not behave like lists.

Converting a Tree to Text

When a tree is accessed as text (such as when it is displayed by a `put` command), SenseTalk converts it automatically to a text representation in XML format. By setting `the treeFormat's prettyPrint` to true or false you can control whether or not the XML will be formatted on multiple lines with indentation for easier reading by a person. By default that property is set to true. The `standardFormat()` function may also be used to format a tree as text.

If the tree has document-level information (as defined by the _xmlinfo property) it will be used in generating the text representation of the tree. The _xmlinfo property can only be set at the top level of a tree (not a sub-tree), so inserting a tree as a sub-tree of another will discard its document-level information. The _xmlinfo property is a property list that can include the following document-level properties:

- `CharacterEncoding` – if set, this should be the name of a valid XML encoding (see http://www.iana.org/assignments/character-sets for a list of valid encoding names -- these are not the same as SenseTalk's string

encoding types).

- `DocumentContentType` – must be one of XML, XHTML, HTML, or Text. This controls some aspects of the text representation that will be generated for that tree.
- `MIMEType` – should be set to a valid MIME type (see http://www.iana.org/assignments/media-types/index.html).
- `URI` – the Uniform Resource Identifier (usually a URL) associated with that document.
- `Version` – should be either "1.0" or "1.1" to indicate the XML version.

In addition, the _xmlinfo can also include two lists of tree nodes representing comments or processing instructions which precede or follow the root element of the document:

- `Head` – a list of comments and processing instructions that precede the root element
- `Tail` – a list of comments and processing instructions that follow the root element

Creating a Tree from a Property List

Sometimes it may be convenient to represent information in a script in the form of a property list, then convert it to a tree in order to produce XML output. SenseTalk's `tree` function (or `asTree()` or `as a tree` operator) supports property lists in several formats to make this convenient.

In the full standard format, the property list may include these special properties (and values): "_tag" or "_element" (tag name of an element); "_attributes" (property list of attributes of an element node); "_children" (list of child nodes); "_text" (contents of a text node); "_comment" or "--" (contents of a comment node); "_processingInstruction" or "_pi" (contents of a processing instruction node); "?" followed by processing instruction name (body of a processing instruction node); "_XMLinfo" (property list of special XML document attributes). Here is a simple example using this approach:

```
put ( _tag:book, _children:"The Rose" ) as tree -- <book>The Rose</book>
```

For situations where XML attributes are not needed, a simplified format can be used:

```
put (book:"The Rose") as tree -- <book>The Rose</book>
```

Some XML formats use attributes but no content, which can be done like this:

```
put tree(_tag:"pg",_attributes:(id:43)) -- <pg id="43"></pg>
```

A simplified format can also be used in this case:

```
put tree(_tag:"pg", id:43) -- <pg id="43"></pg>
```

The rules for converting a property list to a tree can be summed up in this way: If there is only a single property, and it's not one of the special properties, then that property name is taken to be the name of an element, and its value represents that element's children. If a property list has a "_tag" or "_element" property, it will produce an element node. In this case, if there is no "_attributes" property then other properties that don't have special meaning are assumed to be attributes.

Creating a Tree from a List

It is also possible to convert a list to a tree using the `tree` function (or `asTree()` or `as a tree` operator). When converted in this way, a list becomes an unnamed tree (with an empty tag). This also applies to nested lists or lists

within property lists that are being converted.

Converting a Tree to a Property List

A tree can also be converted to a property list, by using the `as` operator. For example:

```
put "<zip>80521</zip>" as tree as object -- (zip:(80521))
```

SenseTalk will use a simplified form for the property list if it can. To produce a standard format in all cases, set `the treeFormat's useStandardFormat` property to true:

```
set the treeFormat's useStandardFormat to true
put "<zip>80521</zip>" as tree as object -- (_children:((_text:"80521")), _
tag:"zip")
```

Tree Comparisons

When two values are compared for equality (using the `is` or = operator), they are ordinarily compared as text. Only when both values are trees (in tree format, not a property list or XML text representation of a tree) are they compared as trees. You can force comparison as trees by specifying `as tree` for any non-tree value.

When one tree is compared to another, the two trees will be regarded as equal if they have identical contents, including identical children and properties. However, if two trees are nearly identical such that the only difference between them is that one tree has a version or characterEncoding property with the default value and the other tree lacks such a property, then the two trees will be treated as equal.

Working with Node Types

Each node within a tree has a *node type*. The `nodeType` property of a node will return a node's type:

```
put order's nodeType -- Document
put the nodeType of item 1 of order -- Element
```

The types of nodes that may be present in a tree include Document, Element, Text, DTD, ProcessingInstruction, and Comment. A node's type cannot be changed. To test whether a node is a particular type, the `is a` operator may also be used:

```
put order is a Document -- true
```

Only Document, Element, and DTD nodes may have children. Attempting to add a child node to any other type of node will result in an error.

Global Properties

There are two global properties which govern certain aspects of tree behavior.

`The treeFormat` – a property list with properties `prettyPrint` and `useStandardFormat`. `PrettyPrint` defines whether trees are displayed nicely formatted (the default). Set this value to false to turn off pretty formatting of trees when they are displayed as XML text. If the treeFormat's `useStandardFormat` property is set to true, trees will always be converted to property lists using a standard format, with `_tag`, `_attributes`, and `_children` properties. When set to false (the default) converting a tree into a property list (using "as property list" or "as object") will result in a simplified property list format being used for nodes that have a tag that isn't a

reserved property name and/or don't have any attributes that are reserved property names.

```
set the treeFormat's prettyPrint to false -- turn off tree indenting
```

The `treeInputFormat` – a property list with property `alwaysJoinText` which defines whether text nodes are combined automatically whenever a change is made to a tree that results in two text nodes adjacent to each other. Setting it to false allows sequential text nodes to be present in trees which may be useful for some applications, but may prevent Xpath node access from working properly. The default setting is true which is needed to ensure that node access (using a node or all nodes expression works reliably.

```
set the treeInputFormat's alwaysJoinText to false
```

Tree Functions

◊ tree, asTree function

What it Does

The `tree` or `asTree` function (called by the `as a tree` operator) returns the value of its parameter converted to a tree.

Examples

```
put file "configuration.xml" as a tree into config
put asTree("<a>Contents</a>") -- <a>Contents</a>
put ( _tag:book, _children:"The Rose" ) as tree -- <book>The Rose</book>
put tree(_tag:"page", num:8) -- <page num="8"></page>
```

Tech Talk

Syntax: **{the} tree of** *factor*
tree(*expr***)**
{the} asTree of *factor*
asTree(*expr***)**

When the `tree` function is called with a parameter that is a property list (object) that has an "asTree" property, the value of that property is used. If the object has an "asTreeExpression" property, the value of that property is evaluated as an expression (equivalent to calling the `treeFromXML()` function) to obtain the tree value. If the object has neither of these properties, an "asTree" function message is sent exclusively to the object and its helpers to obtain the tree value.

If the parameter is an object but doesn't supply a tree representation of itself in any of the above ways, it is taken to be a direct property list representation of a tree structure or a node. The property list may include these special properties (and values): "_tag" or "_element" (tag name of an element); "_attributes" (property list of attributes of an element node); "_children" (list of child nodes); "_text" (contents of a text node); "_comment" or "--" (contents of a comment node); "_processingInstruction" or "_pi" (contents of a processing instruction node); "?" followed by processing instruction name (body of a processing instruction node); "_XMLinfo" (property list of special XML document attributes). See the section "Creating a Tree from a Property List" above for more explanation.

If the parameter is not an object and it is not already a tree, its string value is evaluated as an XML expression (equivalent to calling the `treeFromXML()` function) to obtain the tree value.

If the parameter includes a "version" property the resulting tree object will be a Document type node, otherwise it will be an Element node.

See Also: the discussion of "Conversion of Values" and the `as` operator in Expressions.

◊ treeFromXML, treeFromHTML functions

What it Does

The `treeFromXML` function evaluates a text value as XML and returns a tree. The `treeFromHTML` function evaluates a text value as HTML and returns a tree representation of that HTML content.

Examples

```
put treeFromXML(xmlText) into aTree
put treeFromHTML(htmlText) into htmlTree
```

Tech Talk

Syntax: **{the} treeFromXML of** *factor*
treeFromXML(*expr***)**
{the} treeFromHTML of *factor*
treeFromHTML(*expr***)**

The `treeFromXML` function tries to evaluate its parameter as XML text. If the text is valid XML, it is parsed and the resulting tree returned. The tree returned will be a Document node if document-level information such as the XML version is present in the text, or an Element node otherwise. If the text is not valid XML, the returned tree will represent an XML text node rather than an element or document, and `the result` will be set to a warning message.

Similarly, the `treeFromHTML` function tries to evaluate its parameter as HTML text. If the text is valid HTML, including a valid fragment (rather than a full document) it is parsed and the resulting tree returned. If the text is not valid HTML, an exception will be thrown.

◊ documentTreeFromXML, documentTreeFromHTML functions

What it Does

The `documentTreeFromXML` and `documentTreeFromHTML` functions evaluate a text value as either XML or HTML respectively and return a tree representation of that content. The returned value will always be a Document node rather than an Element node (assuming there are no errors).

Examples

```
put documentTreeFromXML(xmlText) into docTree
put documentTreeFromHTML(htmlText) into htmlDocTree
```

Tech Talk

Syntax: **{the} documentTreeFromXML of** *factor*
documentTreeFromXML(*expr***)**
{the} documentTreeFromHTML of *factor*
documentTreeFromHTML(*expr***)**

The `documentTreeFromXML` function tries to evaluate its parameter as XML text. If the text is valid XML, it is parsed and the resulting tree returned. The tree returned will be a Document node regardless of whether document-level information such as the XML version is present in the text. If the text is not valid XML, the returned tree will contain the text as a text node, and `the result` will be set to a warning message.

Similarly, the `documentTreeFromXML` function tries to evaluate its parameter as HTML text. If the text is valid HTML, including a valid fragment (rather than a full document) it is parsed and the resulting Document tree returned. If the text is not valid HTML, an exception will be thrown.

◊ STTreeVersion function

What it Does

The `STTreeVersion` function returns the current version number of the STTreeNode xmodule.

Examples

```
put STTreeVersion()
```

Tech Talk

Syntax: **the STTreeVersion**
STTreeVersion()

Working with Color

SenseTalk's color-handling capabilities are provided by the STColor XModule. Typically, host environments which make use of color will load the STColor XModule automatically on launch. Whenever this external module has been loaded, the features described in this section are available for working with color values.

Note: `Loadedmodules()` function

To verify what modules are loaded, you can use the `loadedModules()` function, described in Other Commands and Functions.

Color Values

A color can be represented in SenseTalk as a combination of component values in any of several different formats. The recognized formats are listed in the following table:

Color Format	Example Value	Description
`Basic`	128,0,64	Red, green, and blue values from 0 to 255
`Alpha`	128,0,64,255	Red, green, blue, and alpha values from 0 to 255
`HTML`	#800040	Red, green, and blue values in hexadecimal form as used in web pages
`W`	W, 0.241	White value from 0 (black) to 1 (white)k
`WA`	WA, 0.241, 1.000	White and alpha values from 0 to 1
`RGB`	RGB, 0.5, 0, 0.25	Red, green, and blue values from 0 to 1
`RGBA`	RGBA, 0.5, 0.0, 0.25, 1.0	Red, green, blue, and alpha values from 0 to 1
`HSB`	HSB, 0.917, 1.0, 0.5	Hue, saturation, and brightness values from 0 to 1
`HSBA`	HSBA, 0.917, 1.0, 0.5, 1.0	Hue, saturation, brightness and alpha values from 0 to 1
`CMYK`	CMYK, 0.143, 0.942, 0.225, 0.274	Cyan, magenta, yellow, and black values from 0 to 1
`CMYKA`	CMYKA, 0.143, 0.942, 0.225, 0.274, 1.000	Cyan, magenta, yellow, black, and alpha values from 0 to 1

In addition, a number of color names are recognized, as defined by `the namedColors` global property.

The default color format, known as the "Basic" format, is a list of three numbers with values in the range 0 to 255, indicating the amounts of red, green, and blue that compose the color. The "Alpha" format adds a fourth number, known as the alpha value, that represents the opacity of the value, for systems that can work with partially-transparent colors. An alpha value of zero represents clear.

The HTML format is a single value consisting of a pound sign (#) followed by two hexadecimal digits each for the red, green, and blue components of the color. Each of the other formats is a list beginning with a format code, followed by

numeric values in the range 0.0 to 1.0 representing the amount of each component in a particular color model.

When evaluating a value as a color, SenseTalk can accept either a list of appropriate values, or a text list (a text string consisting of values separated by commas).

◊ the colorFormat global property

What it Does

The colorFormat is a global property that specifies the format to use when a value represented internally as a color is converted to a textual representation.

Examples

```
set the colorFormat to "Basic" -- 3 numbers separated by commas
put color("red") -- displays "255,0,0"
set the colorFormat to "HSB"
put color("green") -- displays "HSB, 0.333, 1.000, 1.000"
set the colorFormat to "HTML"
put color("blue") -- displays "#0000FF"
```

Tech Talk

Syntax: set the colorFormat to *formatExpression*
get the colorFormat

The *formatExpression* is an expression that evaluates to one of the valid color format names. It can be set to Basic, Alpha, HTML, W, WA, RGB, RGBA, HSB, HSBA, CMYK, or CMYKA to specify the format in which color values will be displayed. Basic is a list of 3 integers from 0 to 255 representing the levels of red, green, and blue in the color. The Alpha format adds a fourth integer from 0 to 255 representing the opacity of the color (with 0 being completely transparent).

The HTML format presents a color as a hexadecimal number preceded by "#" (such as "#FF0000" for red). The other formats are text lists beginning with the format identifier, followed by one or more component values from 0.0 to 1.0 indicating the level of each color component. The colorFormat is initially set to Basic.

◊ the namedColors global property

What it Does

The namedColors property is a property list whose keys are color names and whose values are colors. It defines the colors that can be specified by name in a script.

Examples

```
put the keys of the namedColors -- list all available color names"
set the namedColors.pink to color("RGB,1.0,0.5,0.5") -- define pink
set the namedColors.dawn to color("pink")
put color of "dawn" -- shows "255,128,128" (if colorFormat is Basic)
```

Tech Talk

Syntax: set the namedColors.*colorName* **to** *colorExpression*
get the namedColors

The *colorExpression* is an expression that evaluates to a valid color using any of the SenseTalk color formats (but not a color name). *ColorName* is the name of a new or existing color. The namedColors property has a default initial value that defines the following colors by name: Aqua, Black, Blue, Brown, Clear, Cyan, DarkBlue, DarkGray, DarkGreen, DarkGrey, DarkRed, Fuchsia, Gray, Green, Grey, LightGray, LightGreen, LightGrey, Lime, Magenta, Maroon, Navy, Olive, Orange, Purple, Red, Silver, Teal, White, and Yellow.

◊ color, asColor function

What it Does

The `color()` function returns a color value from a string or list in one of the recognized color formats. Any of the 11 formats described for the colorFormat property are recognized, as either text lists (with items separated by commas) or as true lists of values. In addition, color names can be used, as defined by the namedColors global property.

When to Use It

Use the `color()` function whenever a color is needed from a value that may not already be represented as a color. One common use of the `color()` function is to compare two colors that may be in different formats to see if they represent the same color.

Examples

```
put color("#00FFFF") is the color of "aqua" -- shows 'true'
if color of stripe is color("red") then ...
```

Tech Talk

Syntax: color(*stringOrList***)**
the color of *stringOrList*

The `color` function converts a value to its internal representation as a color. This can also be called using the synonymous `asColor` function, or the `as color` operator.

See Also: the `red` function and related component functions, below, and the `colorFormat` property, above.

◊ red, green, blue, hue, saturation, brightness, cyan, magenta, yellow, black, white, and alpha functions

What they do

These functions each return one component value of a color interpreted using a particular color space. The value returned is always in the range from 0 (the color does not contain any of that component) to 1 (the color contains the maximum possible amount of that component).

When to Use Them

Use the `red()` , `green()`, and `blue()` functions to obtain the individual components of a color as represented using the red/green/blue color space. Use the `hue()` , `saturation()`, and `brightness()` functions to get the individual components of a color as represented using the hue, saturation, and brightness color space. Use the `cyan()` , `magenta()` , `yellow()`, and `black()` functions to obtain the individual components of a color as represented using the cyan/magenta/yellow/black color space that is commonly used when printing. Use the `white()` function to evaluate a color using a greyscale color space and return the level of white from 0 (black) to 1 (white). Use the `alpha()` function to return the opacity of a color value, from 0 (transparent) to 1 (opaque).

Examples

```
put the red of "purple" -- 0.5
put blue("#00FFFF") -- 1
put hue("purple") -- 0.833333
put yellow("#00FFFF") -- 0.133043
put white of "purple" -- 0.268208
```

Tech Talk

Syntax: red(*stringOrList***) , blue(***stringOrList***) , ...**
the red of *stringOrList* **, the blue of** *stringOrList* **, ...**

Each of these functions evaluates *stringOrList* as a color and returns a value in the range 0 to 1 representing the amount of the requested color component that is present in that color in the appropriate color space. Note that the `black()` and `white()` functions are not opposites, as they evaluate the color in different color spaces.

See Also: the `color` function and `the colorFormat` property, above.

◊ is a color operator

What it Does

The `is a color` operator is an extension provided by the STColor Xmodule to the standard `is a` operator that tests whether a given value can be interpreted as a color.

When to Use It

Use the `is a color` operator whenever your script needs to determine if a value is valid as a color.

Examples

```
put "#00FFFF" is a color -- shows 'true'
if shade is a color then ...
```

Tech Talk

Syntax: *value* **is a color**

If *value* is a string or list in one of the recognized color formats, or is one of the color names defined by `the namedColors` global property, the `is a color` operator will evaluate to "true".

◊ STColorVersion function

What it Does

Returns the version number of the STColor Xmodule.

When to Use It

You will rarely need the `STColorVersion()` function, but might use it if there is a need to check the version of the color module in use for compatibility reasons.

Examples

```
if the STColorVersion < 1.02 then ...
```

Tech Talk

Syntax: STColorVersion()
the STColorVersion

The `STColorVersion` function returns a number.

Working with Binary Data

Most scripts work with data in the form of text and numbers, and sometimes other types of values such as dates or colors. When needed, SenseTalk can also deal with data in its binary form -- the raw bits and bytes that are stored on a computer.

Data Values

Raw data can be represented directly in a script using a pair of hexadecimal digits for each byte of the data, enclosed in angle brackets, < and >.

```
put <00> into nullByte -- a single byte, with a value of zero
put <48656c6c6f> into secretMessage -- five bytes of data
```

The `put before` and `put after` forms of the put command can be used to insert additional binary data before or after an existing value.

```
put <20467269 656e6421> after secretMessage -- append more data
```

When two known binary data values are compared for equality, they are compared byte for byte to see that they have exactly the same binary contents.

```
put secretMessage is <48656c6c6f20467269656e6421> -- true
```

Tech Talk

Syntax: < *hexadecimalData* >

The *hexadecimalData* must consist of an even number of hexadecimal digits 0 through 9 and A through F. Spaces may be used to break the sequence up for readability.

◊ asData function, as data operator

What it Does

The `asData()` function, most often called using the `as data` operator, converts any value to its binary representation.

When to Use It

Use the `asData()` function or `as data` operator when you want to tell SenseTalk to treat a value as binary data. This is especially useful for reading or writing a file or URL in its raw binary form (as described later in this chapter), but can also be used at any time to work with or display a value in its binary form.

When two known binary data values are compared for equality, they are compared byte for byte to see that they have exactly the same binary contents. Use `as data` to ensure that such a binary comparison is made.

Examples

```
put "abcdefg" as data -- <61626364 656667>
put file "picture.jpg" as data into rawImageData -- read file contents as data
if file "monet.png" as data is equal to oldData as data then ...
```

Tech Talk

Syntax: asData(*aValue*)
aValue as data

The as data operator is usually more readable and natural to use than the asData function, but is otherwise identical in functionally.

Byte Chunks

The byte chunk type extends SenseTalk's chunk expressions to provide all of the flexibility offered by chunk expressions to working with binary data. The byte chunk type can be used to access a single byte or a range of bytes within a data value:

```
put <010203040506> into myData
put byte 2 of myData -- <02>
put bytes 3 to 4 of myData -- <0304>
put the last 3 bytes of myData -- <040506>
```

As with other chunk types, a byte chunk is a container, so it can be used to change the data:

```
put <010203040506> into myData -- <010203040506>
put <AABB> into bytes 2 to 5 of myData -- <01AABB06>
put <77> after byte 2 of myData -- <01AA77BB06>
delete the first 2 bytes of myData -- <77BB06>
```

Tech Talk

Syntax: **byte** *byteNumber* **of** *dataSource*
bytes *firstByte* **to** *lastByte* **of** *dataSource*

A byte chunk expression is always treated as data in its immediate context (so there is no need to specify as data with it). The *dataSource* doesn't need to be specified as data. A non-data value will be converted to data automatically before the requested bytes are extracted.

Binary Data Files

One of the most important uses of binary data in scripts is when reading and writing data in binary (non-text) files. There are several ways to work with binary data files, depending on your needs.

Simple Data File Access

The easiest way to access a text file is to treat the file directly as a container. The same approach will work for binary data files, by simply using the `as data` operator to indicate that the bytes of the file should be read directly:

```
put file "horse.tiff" as data into tiffData -- read the entire file at once
```

Writing data to a file can be done in the same way:

```
put rawBudgetData as data into file "budget.dat" -- write a data file
```

Remote URL File Access

Accessing a remote file through a URL works exactly the same as a local file. Simply specify `URL` instead of `file`, provide the URL instead of the file path, and use `as data`:

```
put URL "http://some.company.com/horse.jpg" as data into jpgData
put file "budget.dat" as data into URL remoteBudgetFileURL
```

Full Binary File Access

When more sophisticated processing is needed, the standard set of file commands including `open file`, `read from file`, `write to file`, `seek in file`, and `close file` can be used. The `read` and `write` commands have special options available for reading and writing numbers in binary data in a variety of formats. See the description of the `read` and `write` commands in Working with Files and File Systems.

In addition to those numeric data types, the `byte` chunk type can be used with the `read` command to read any given number of bytes as data:

```
read 20 bytes from file "singer.tiff" into formatData
```

To write binary data into an open file at the current location, just specify `as data`:

```
write orbitalCoordinates as data to file jupiter
```

The `as data` operator can be omitted if the value being written is specifically data already, such as when writing selected bytes from a data value:

```
write bytes 1 to 16 of temperatureRecord to file saturn
```

Data Conversions

Binary data values are automatically converted to text whenever needed. There are many contexts in which this may happen, including when writing a value to a file or when a value is displayed. To force a value to be temporarily treated as data and avoid this conversion, use the `as data` operator:

```
put encryptedPassword as data into file "key" -- write binary data to a file
put "Secret" as data -- display in binary format: <53656372 6574>
```

Whenever a value is converted from text to data, the current setting of `the defaultStringEncoding` global property is used to control how each character is encoded into the binary data. Depending on the encoding that is used, and the particular characters that appear in the text, the resulting data may have exactly one byte for each character in the original text, or there may be two or more bytes for some or all of the characters.

Other Commands and Functions

This section describes miscellaneous commands and functions that provide for interaction with the user to display or request information or play a sound; or interaction with the system, such as to start some other program or process.

User Interaction

These commands allow your script to interact with the user by displaying and/or requesting information.

◊ answer

What it Does

Displays a simple modal panel containing a message and up to three buttons which the user may click. Returns the title of the selected button in the variable `it`.

When to Use It

Use the `answer` command when you need to display short to medium-length messages in a modal panel, or to solicit input from the user in the form of a choice between two or three different alternatives.

The `answer` command is ideal for situations where the user must make a yes/no or continue/cancel type of decision. For more complex choices, use the `ask` command.

Examples

```
answer "Great!  You got " & pctRight & "% correct!"
answer "Go on?" with "Yes" or "No" title "DECIDE"
answer "How do you feel?" with "Great!" or \
        "Okay" or "Really Lousy" title "Greetings!"
```

Tech Talk

Syntax: answer {*panelType*} {*Options*}

Options:

{prompt | body | message} *prompt*
with *reply1* **{or** *reply2* **{or** *reply3* **{or** *reply4***}}}**
[title | titled] *titleExpr*
[timeout | time out] {after | in} *duration*
icon *iconName*

An optional *panelType* of `information`, `question`, `warning`, or `error` may be specified. The exact implementation of each panel type is up to the host application, but typically shows a related icon in the panel.

The *prompt* is an expression whose value will be displayed as the primary text in the panel. *TitleExpr* is the title which will be displayed in a larger font at the top of the panel.

Tech Talk

Reply1, reply2, reply3, and *reply4* are the labels that will be shown on the buttons at the bottom of the panel, with *reply1* being the default (rightmost) button. The number of buttons shown will match the number of reply strings specified in the command. If no replies are specified, a single button labelled “OK” will be included.

If the timeout option is used, *duration* specifies the length of time (in seconds, if not explicitly specified) that the panel will be displayed waiting for user input. If the user doesn't respond to the panel within that time, it will be dismissed automatically. In that case, the variable `it` will be empty, and the result function will be set to a value indicating that the panel timed out.

The icon option may be used to supply a different icon to be shown in the panel. If it is used, *iconName* may be either the name of a known system icon, or the full path to an icon file.

The order in which the prompt, replies, title, timeout, and icon are specified is flexible, and all of them are optional (except that at least one option must be supplied).

After execution of the `answer` command, the variable `it` will contain the title of the button that was clicked (or empty if the panel's timeout elapsed).

◊ ask

What it Does

Displays a modal panel containing a prompt string and a text entry field in which the user may type a response. A default response may be supplied. Returns the user’s answer in the variable `it`.

When to Use It

Use the `ask` command when an answer is required from the user which is not from a predetermined list of choices. Use the `ask password` command to request input that is hidden while it is being typed.

Examples

```
ask "Please enter your name:"
ask "How do you feel?" title "Greetings!"
ask query with defaultAnswer title "Please Answer"
ask password "Enter the secret code"
```

Tech Talk

Syntax: ask {password} {*Options***}**

Options:

{prompt | question} *question*
[title | titled] *titleExpr*
[with {answer} | answer] *presetAnswer*
[hint | placeholder | place holder] *placeholder*
[message | body] {text} *message*
{with} [buttons | button {label{s}}] *label1* **{and** *label2***}**
[timeout | time out] {after | in} *duration*
icon *iconName*

The text value of *question* is displayed as the prompt above the input field in the panel. *TitleExpr* is the title, which will be displayed in a larger font at the top of the panel. The string value of the *presetAnswer* expression is displayed in the answer field as the default response. If *placeholder* is given, the placeholder value will be displayed in the answer field when it is empty and not selected. The *message* text will be displayed in the panel below the title.

If *label1* (and optionally *label2*) are specified, they define the buttons on the panel, otherwise two buttons are presented at the bottom of the panel: “Cancel” and “OK”. If the user clicks the OK button (or presses return) the value in the answer field is put into the variable `it`. If the user clicks the Cancel button the value of the variable `it` is set to empty, and `the result` function returns “Cancel”.

If the timeout option is used, *duration* specifies the length of time (in seconds, if not explicitly specified) that the panel will be displayed waiting for user input. If the user doesn't respond to the panel within that time, it will be dismissed automatically. In that case, the variable `it` will be empty, and the result function will be set to a value indicating that the panel timed out.

The icon option may be used to supply a different icon to be shown in the panel. If it is used, *iconName* may be either the name of a known system icon, or the full path to an icon file.

The order in which the options are specified is flexible, and all of them are optional (except that at least one option must be supplied).

After execution of the `ask` command, the variable `it` will contain the value entered in the field (or empty if the panel's timeout elapsed).

◊ put

What it Does

When no destination container is specified, the `put` command displays text in the standard output.

When to Use It

Use the `put` command in this way when you want to display status information during a script without putting up a modal panel like the `answer` command would. The `put` command is also very popular during the development of a script for displaying the current value of variables at different points in the script.

Examples

```
put "Successfully loaded file " & fileName
put n
put "The area is: " & pi * radius^2
```

Tech Talk

Syntax: put {*expr* { , *expr...*}}

Expr can be any valid SenseTalk expression. If multiple expressions are given, separated by commas, each is displayed on a separate line.

Only the simplest form of the `put` command is described here. For other uses of the `put` command, see Containers.

◊ beep

What it Does

Plays the system beep sound.

When to Use It

Use the `beep` command to get the user's attention or alert them to some occurrence.

Examples

```
beep
beep 5 -- really get their attention!
```

Tech Talk

Syntax: beep {*expr*}

Expr can be any valid SenseTalk expression, yielding a number of times to beep.

◊ play

What it Does

Plays a sound file.

When to Use It

Use the `play` command to begin playing a system sound or other sound or music file.

Examples

```
play "Glass" -- without full path, plays a system sound
play "~/Music/iTunes/iTunes Music/Billy Joel/Piano Man.mp3"
play stop -- stops the music
```

Tech Talk

Syntax: play *soundFile*

SoundFile can be any valid SenseTalk expression, yielding the name of a sound to play, or one of the special commands “pause”, “resume” or “stop”.

◊ sound function

What it Does

Returns the name of the currently playing sound, or “done”.

When to Use It

Use the `sound` function to monitor the sound being played.

Examples

```
wait until the sound is "done"
```

Tech Talk

```
Syntax: the sound
        sound()
```

System Interaction

These commands and functions allow your script to interact with the system where the script is running, to launch other programs, open files in other programs, or run system commands through the UNIX shell.

◊ open command

What it Does

Launches another program, or opens a file with another program on the machine where your script is running.

When to Use It

Use the open command when you want to open a file with another program or launch a particular program on the machine where your script is running. The most common use of this command is probably to open a file generated by the script, such as opening a text file in a text editor, or a tab-delimited file in a spreadsheet program, so the user can make further changes to it, print it out, etc.

Examples

```
open "iTunes" -- launch iTunes on this machine
open "/tmp/myFile" with "TextEdit"
```

Tech Talk

Syntax: **open** *application*
open *file* **with** *application*

If the requested application is already running on the machine where the script is executing, it will be brought to the front, otherwise it will be launched.

Note: Opening files

To open a file for reading or writing its contents within a script, use the open file command, documented in Working with Files and File Systems.

◊ shell command and function

What it Does

Executes a command in the command-line shell on the local machine and returns the output.

When to Use It

Use the shell command when you want to launch a command-line program, or the shell function when you also want to obtain the output that it generates.

Examples

```
shell "rm /tmp/testfile"
put shell("cd /; ls -l") into rootDirectoryList
```

Tech Talk

Syntax: **shell** *command* **{,** *optionalParameters***}...**
get shell(*command* **{,** *optionalParameters***}...)**

Tech Talk

The `shell` command and function both set `the result` to the exit code of the *command* that was executed. Typically the exit code is 0 when a command runs successfully and some other value when there is a problem, but the exact meaning of the exit code depends on the command that was run.

The *command* is executed on the machine where the script is running. To execute a shell command on another machine, you can use one of the shell commands that provides remote access, such as rsh or ssh.

`The shellCommand` global property controls which shell program is used to run the command. By default, on Mac and Linux `the shellCommand` is set to “/bin/sh” (the Bourne shell), and on Windows it is set to "ShellExecute" (see below). To execute commands in a different shell, set `the shellCommand` to the full path of the shell you want to use before calling the shell function, or to "ShellExecute" or empty.

When `the shellCommand` is empty, `shell` treats its first parameter as the command to execute and any additional parameters as parameters to be passed to that command. If only a single parameter is passed, it is split by any spaces that are present in the parameter to derive the command and its parameters.

```
set the shellCommand to empty
put shell("ls -l") -- runs the 'ls' command with '-l' parameter
put shell("ls", "-l") -- does the same thing
```

On Windows the default setting of `the shellCommand` global property is "ShellExecute". This causes the Windows ShellExecute() function to be used to run the specified file:

```
set the shellCommand to "ShellExecute"
shell "example.bat" -- run the indicated batch file
```

When using ShellExecute on Windows, several additional parameters besides the command or file may optionally be passed. The second parameter, if given, specifies any parameters to be passed to the command being run. The third parameter specifies the default working directory for the action. The fourth parameter should be a number specifying any optional flags to pass to the underlying ShellExecute() function. Finally, the fifth parameter, if given, specifies an explicit verb to use (such as "open", "explore", "edit", "find", or "print"). Otherwise the default verb defined in the registry (or "open") will be used. If an error occurs, `the result` will be set to a number, otherwise it will be empty.

When `the shellCommand` is not empty and is not set to "ShellExecute" (the usual case on Mac and Linux, where it is set to "/bin/sh" by default), if multiple parameters are passed to the shell function they are treated as separate commands to each be executed in sequence within a single shell context.

System Information

These functions provide information about various aspects of the system where the SenseTalk script is running.

◊ hostAddresses function

What it Does

The `hostAddresses` function returns a list of all of the IP addresses for the host computer where SenseTalk is running.

When to Use It

Call `hostAddresses` to find all of the network IP addresses for the local host.

Examples

```
put item 1 of the hostAddresses into myIPAddress
```

Tech Talk

```
Syntax: the hostAddresses
        hostAddresses()
```

◊ hostName, hostNames functions

What it Does

The `hostName` function returns the standard host name of the machine on which the script is running. The `hostNames` function returns a list of all of the known host names.

When to Use It

Call `hostName` or `hostNames` to find names by which the local host may be known on the network.

Examples

```
put the hostName into localName
```

Tech Talk

```
Syntax: the hostName
        hostName()
        the hostNames
        hostNames()
```

◊ machine function

What it Does

Returns the type of machine hardware of the computer where SenseTalk is running.

When to Use It

Call `machine` to determine the machine hardware platform.

Examples

```
if the machine is "i386" then put "Intel" into hardwareType
```

Tech Talk

```
Syntax: the machine
        machine()
```

◊ OSInfo function

What it Does

The `OSInfo` function returns a property list containing various items of information about the operating system of the machine where SenseTalk is running.

When to Use It

Call `OSInfo` to learn information about the host operating system.

Examples

```
put the OSInfo
```

Tech Talk

```
Syntax: the OSInfo
        OSInfo()
```

◊ platform function

What it Does

The `platform` function returns the name of the host operating system where SenseTalk is running, such as "MacOS", "Linux", etc.

When to Use It

Call `platform` to make decisions based on SenseTalk's host platform.

Examples

```
if the platform is "MacOS" then exit script
```

Tech Talk

```
Syntax: the platform
        platform()
```

◊ processInfo function

What it Does

The `processInfo` function returns a property list containing information about the process within which the script is running, including its name, parameters, and process id.

When to Use It

Call `processInfo` to obtain information about the identity of the process running the script.

Examples

```
put the processInfo
```

Tech Talk

```
Syntax: the processInfo
        processInfo()
```

◊ processor function

What it Does

The `processor` function returns the type of processor of the host computer where SenseTalk is running, such as "x86", "Power Macintosh", etc.

When to Use It

Call `processor` to determine the CPU type for the local host.

Examples

```
put the processor into CPUType
```

Tech Talk

```
Syntax: the processor
        processor()
```

◊ specialFolderPath function

What it Does

The `specialFolderPath` function returns the file system path to any of a number of special folders on the host computer where SenseTalk is running.

When to Use It

Call `specialFolderPath` with the name of a special folder to get the path to that folder. Folders supported include: "home", "system", "library", "applications", "demo applications", "developer applications", "admin applications", "developer", "users", "documentation", "documents", "core services", "desktop", "caches", "application support", "fonts", "preferences", "temporary" (or "temp"), and "root". Call it with a tilde ("~") followed by the login name of a user to get the path to that user's home folder.

An optional second parameter may be given to specify the "domain" for the folder. The domain may be one of: "user", "local", "network", "system", or "all" (when a domain of "all" is given, more than one path may be returned, as a list). If no domain is specified, a domain appropriate for the folder being requested will be assumed.

Examples

```
put specialFolderPath("applications") -- "/Applications"
put specialFolderPath("~brenda") -- "/Users/brenda"
put specialFolderPath("library", "local") -- "/Library"
```

Tech Talk

```
Syntax: specialFolderPath(folderName {, domainName})
```

◊ systemInfo function

What it Does

The `systemInfo` function returns a property list containing various pieces of information about the system where SenseTalk is running.

When to Use It

Call systemInfo to learn information such as the amount of memory installed or the processor byte order for the local host.

Examples

```
put the systeminfo's memorySize / 1 GB into gigaBytes
if systemInfo().NativeByteOrder is "Big-Endian" then swap
```

Tech Talk

```
Syntax: the systemInfo
        systemInfo()
```

◊ systemVersion function

What it Does

The systemVersion function returns the version number of the operating system on the host computer where SenseTalk is running.

When to Use It

Use systemVersion to check which OS version a script is running on.

Examples

```
if systemVersion() begins with "10.5" then put "Leopard!"
```

Tech Talk

```
Syntax: the systemVersion
        systemVersion()
```

◊ userInfo function

What it Does

The userInfo function returns a property list containing various pieces of information about the user account where SenseTalk is running.

When to Use It

Call userInfo to get information about the logged-in user, such as their FullName, ShortName, and HomeDirectory.

Examples

```
put the userInfo's FullName into fullName
set the folder to userInfo().HomeDirectory
```

Tech Talk

Syntax: the userInfo
userInfo()

Miscellaneous Commands and Functions

These commands and functions provide miscellaneous services and information.

◊ breakpoint

What it Does

The `breakpoint` command, if executed in the context of a debugger, will cause execution of the script to be suspended and control transferred to the debugger.

When to Use It

Call `breakpoint` at any point in your script where you want it to stop for debugging purposes.

Examples

```
if count > upperLimit then breakpoint
```

Tech Talk

Syntax: breakpoint

The `breakpoint` command has no effect if the script is running in an environment without a debugger. Breakpoints may also be disabled by setting the `breakpointsEnabled` global property to false. Setting this property to true again (its default value) will reenable breakpoints.

◊ callStack function

What it Does

The `callStack` function returns a list of SenseTalkFrame objects providing information about the current execution frames.

When to Use It

Call `callStack()` to obtain information about how the current handler was called.

Examples

```
put property "Line" of the last item of the callStack into currentLine
```

Tech Talk

```
Syntax: the callStack
        callStack()
```

◊ env function

What it Does

The `env` function provides access to the environment variables supplied by the environment in which SenseTalk is running.

When to Use It

Call `env()` with 1 parameter that is the name of a particular environment variable to retrieve the value of that variable. Call `env()` with no parameters to get a property list containing all of the environment variables.

Examples

```
put env("Home") into homeFolder
put env() -- display all available information
```

Tech Talk

```
Syntax: the env {of varName}
        env(varName)
```

◊ loadedModules function

What it Does

The `loadedModules` function returns a list of all of SenseTalk's internal back-end modules plus any external modules (XModules) that have been loaded into the system.

When to Use It

The `loadedModules` function can be used to determine if a particular set of functionality is available. For example,

the STColor XModule shows up in the returned list as “ST_Color”, so you can test whether the color functions are available as shown in the example below.

Examples

```
if "ST_Color" is in the loadedModules then ...
```

Tech Talk

Syntax: the loadedModules
loadedModules()

◊ loadModule command

What it Does

The `loadModule` command loads one or more external modules (XModules).

When to Use It

Use `loadModule` to load a module and make its functionality available within your scripts. If a module has already been loaded, this command will do nothing.

Examples

```
loadModule "/Users/doug/Library/stats.xmodule"
```

Tech Talk

Syntax: loadModule *modulePath* **{**, *modulePath* ...**}**

◊ version, long version, senseTalkVersion, buildNumber functions

What it Does

These functions provide the current version number of the application (a number), a long form of the version (a string containing more information), the version of the SenseTalk engine being used (a number), and the current build number (a whole number) respectively.

When to Use It

Use these functions to identify what version of an application or of SenseTalk is in use. This might be useful, for example, to determine if a specific feature is available and avoid using it otherwise.

Examples

```
if version() >= 2.0 then useAdvancedFeatures
put the long version
if the senseTalkVersion < requiredVersion then exit all
put buildNumber() into latestBuildUsed
```

Tech Talk

```
Syntax : the {long} version
         version()
         the senseTalkVersion
         senseTalkVersion()
         the buildNumber
         buildNumber()
```

Appendices

Appendix A – Restricted Words – provides lists of the words whose use is restricted (either they can't be used as command names, or as variable names) in order for SenseTalk to be able to understand your scripts.

Appendix B – All About "It" – describes the important role played by `it` in SenseTalk and lists the commands that manipulate `it`.

Appendix A – Restricted Words

The SenseTalk language includes a large number of words (more than 800, in fact) which have special meaning in one context or another. Only a few of these words are reserved exclusively for their special meaning, however. The rest can also be used as the names of variables or of your own commands and functions.

Care has been taken to ensure that the number of restricted words is as small as possible, to give you the maximum freedom to use names that are meaningful in your scripts. In order for the SenseTalk engine to properly understand your script, there are some words whose use must be confined to their special meaning only. Attempting to use one of these restricted words in a way that is not allowed will result in a syntax error.

Restricted Command and Function Names

The following words may **not** be used as command or function names:

catch	function	on	setProp
constant	getProp	pass	the
do	global	properties	**then**
else	**if**	**repeat**	to
end	local	return	try
exit	next	send	universal

Restricted Variable Names

The following words may **not** be used as variable (container) names:

down	false	**if**	**then**
else	field	**repeat**	true
empty	fld	return	universal
end	global	target	up

Predefined Variables

A number of other words have predefined values, and should generally only be used as variables in situations where their standard meanings are not required. To avoid confusion, it is best to avoid using these words as variable names, but their use is not prohibited, and in some cases it may be useful to store different values in them. Predefined variables are described in detail in Values. Here are some that should probably be avoided as variable names:

- Alternative boolean values: `yes`, `no`, `on`, `off`
- Special characters: `colon`, `comma`, `cr`, `creturn`, `lf`, `linefeed`, `crlf`, `formfeed`, `newline`, `quote`, `space`, `tab`, `slash`, `backslash`, `null`
- Numbers by name: `pi`, `zero`, `one`, `two`, `three`, ...

- The current date and time: `today`, `now`, `date`, `time`

In addition, the host application and any loaded external modules may define other predefined variables.

Unquoted Literals

All other words that are allowed as variable names (i.e. not in the restricted list given above, or having a predefined special value) evaluate to their name until something else is stored in them. In other words, they can be used as unquoted literals. As soon as such a name is used as a variable, its value is treated as empty. Unquoted literals are always capitalized according to their first use in the handler — a subsequent use of the same name (varying only by case) in the same handler will be treated the same as the first occurrence (as illustrated by the first 3 lines of the examples below).

> **Note**
>
> If `the strictVariables` global property is set to true, it prevents the use of uninitialized variables as unquoted literals.

Examples

```
put BIG -- "BIG"
put little -- "little"
put big && LITTLE -- "BIG little"
set the strictVariables to true -- prevent use of unquoted literals
put big -- this will throw an exception
```

Appendix B – All About "It"

The variable `it` plays a special role in SenseTalk. There are many situations where a value is provided or used but no specific variable name is specified. In these situations, SenseTalk automatically makes use of the local variable `it` as the place to store a value. Most of the time this is very convenient, and leads to natural, English-like scripts. However, because quite a few different commands can change `it`'s value, care must be taken not to assume that `it` will hold the same value for very long.

To help, here are the commands and constructs that will change the value of `it`. The following commands *always* set the value of `it`, as their normal way of returning a value:

```
answer
ask
get
post
read
```

The following commands *sometimes* set the value of `it`, depending on how they are used. Generally, if the value passed to one of these commands is a container (such as a variable), the contents of that container are altered directly. If the value passed in is not a container (such as a literal value or an expression), the result is returned in `it`:

```
convert
join
replace
sort
split
```

In addition, the "repeat with each" command uses `it` in its simplest form when a specific loop variable is not supplied:

```
repeat with each ... (when no variable is provided)
```

When any of the commands listed here implicitly set the value of `it`, they set it to be an ordinary variable, not a reference. If `it` was previously a reference, it is unlinked from the container it referred to before being set.