Scripting 1. Your first scripts

This page tells you how to create, run and save a script. To get a feel for how it works, you are advised to try out all the steps.

1. A minimal script

Suppose that you want to create a script that allows you to play a selected Sound object twice. You first create an empty script, by choosing **New Praat script** from the **Praat** menu in the **Praat Objects** window. A **ScriptEditor** window will appear on your screen:

In this window, you type

```
Play
Play
```

Now select a Sound in the **Praat Objects** window. As you expect from selecting a Sound, a **Play** button will appear in the dynamic menu. If you now choose **Run** from the **Run** menu in the ScriptEditor, Praat will play the sound twice. This works because **Play** is a command that becomes available in the dynamic menu when you select a Sound.

2. Some more commands

In the above example, you could use the **Play** command because that was the text on a button currently available in the dynamic menu. Apart from these selection-dependent (dynamic) commands, you can also use all fixed commands from the menus of the **Object window** and the **Picture window**. For instance, try the following script:

```
Erase all
Draw inner box
Play
Play
Erase all
```

When you run this script, you'll see a rectangle appear in the **Praat Picture** window (that's what the command **Draw inner box** in the **Margins** menu does), then you'll hear the Sound play tiwce, then you'll see the rectangle disappear from the Picture window (that's what the command **Erase all** from the **Edit** menu does).

Here we see that the Praat scripting language is an example of a *procedural programming language*, which means that the five *statements* are executed in the order in which they appear in the script, i.e. first **Erase all**, then **Draw inner box**, then **Play** twice, and finally **Erase all**.

3. Experimenting with your script

You don't have to be afraid of making mistakes. Here are a couple that you can try to make.

First, try to run the script when a Sound is not selected (e.g. you create a Pitch object from it and keep that selected, or you throw away the Sound). You will notice that Praat gives you an error message saying **The command "Play" is not available for the current selection**. Indeed, if you select a Pitch or if you select nothing, then no command **Play** appears in the dynamic menu, so the script cannot execute it. Note that the commands **Erase all** and **Draw inner box** are still available, because they continue to be present in the menus of the Picture window; therefore, the script will execute the first two lines (`Erase all` and `Draw inner box`) and stop running at the third line, i.e. at your first `Play`. The result is that the "box" will stay visible in the Picture window, because the fifth line of the script, which should erase the box, is never executed.

Second, try to mistype a command (there's a good chance you already did it by accident), e.g. write `PLay` instead of `Play`, or `Draw inner bocks` or whatever. Again, you are likely to get a message saying that that command is not available. Such messages are the most common messages that you'll see when writing scripts; now you know that they mean either that you mistyped something or that you made the wrong selection.

4. Saving your script

The **File** menu of the ScriptEditor has a command **Save**, with which you can save your script as a file on disk, for instance under the name `test.praat`.

Please try this with the five-line script you just typed. After saving the script, the name of the script file will appear in the window title:

After you save your script, you can close the ScriptEditor window without losing the script: you can reopen the script file by using **Open Praat script...** from the **Praat** menu, or by choosing **New Praat script** again, followed by **Open...** from the ScriptEditor's **File** menu.

It advisable to use `.praat` as the extension for script file names. On the Mac, if you double-click a `.praat` file, Praat will automatically start up and show the script. On the Mac and on Windows, if you drag a `.praat` file on the Praat icon, Praat will also start up and show the script.

Links to this page

Scripting
Scripting 2. How to script settings windows
Scripting 3. Simple language elements

Scripting 2. How to script settings windows

Not all menu commands are as simple as those on the **previous page**, which act immediately once you choose them from a menu (e.g. **Play**, **Erase all**). Most commands in Praat require the user to supply additional information; these are the commands whose title ends in "...".

For instance, when you select a Sound, the command **Draw...** will appear in the **Draw** menu, and when you click it, Praat will present you with a *settings window*, which asks you to supply six pieces of additional information, i.e. six so-called *settings* (or in programming jargon: *arguments*):

In this example, all the settings have their standard values: you want to draw the whole time domain of the Sound, you want to have autoscaling vertically, you want to see garnishings around the picture (a box, labelled axes, and numbers), and you want the waveform to be drawn as a curve. Pressing the OK button in the above window is equivalent to executing the following script line:

```
Draw... 0 0 0 0 yes Curve
```

You see that in a script, all of the arguments are supplied on the same line as the command, in the same order as in the settings window, counted from top to bottom (and, within a line, from left to right). The texts "(= all)" and "(= auto)" above are just Praat's explanations of what it means to type a zero in those fields (namely "draw all times" and "use vertical autoscaling", respectively); in a script they are superfluous and you shouldn't write them.

If you want to draw the sound with different settings, say from 1 to 3.2 seconds, scaled between -1 and +1 instead of automatically, with garnishings off, and with the waveform drawn as poles, you would have the following settings window:

In a script this would look like

```
Draw... 1.0 3.2 -1 1 no Poles
```

1. Numeric arguments

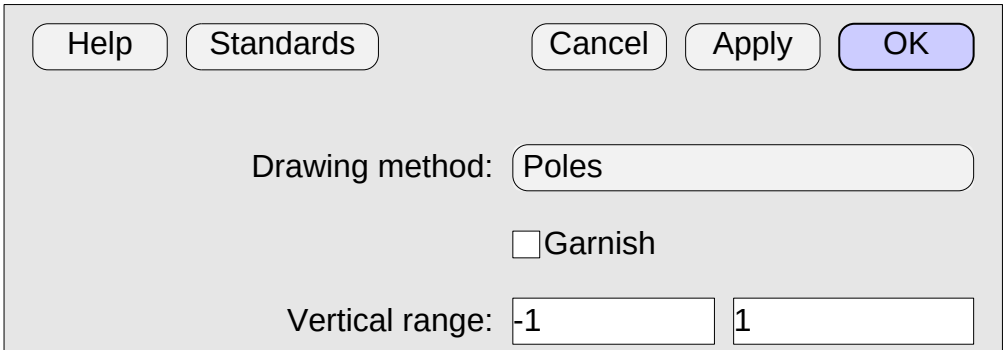
The first four arguments in the above examples are *numeric arguments*: they are (real or integer) numbers. You just write them in the script as you would write them into the settings window.

2. Boolean (yes/no) arguments

The fifth argument in the above examples (**Garnish**) is a *boolean argument* (yes/no choice) and is represented by a *check button*. In the script you write it as `yes` or `no` (or as 1 or 0).

3. Multiple-choice arguments

The sixth argument in the above examples (**Drawing method**) is a *multiple-choice argument* and is represented by an *option menu*. In the script you write the text of the choice, i.e. `Curve` or `Poles` in the examples.



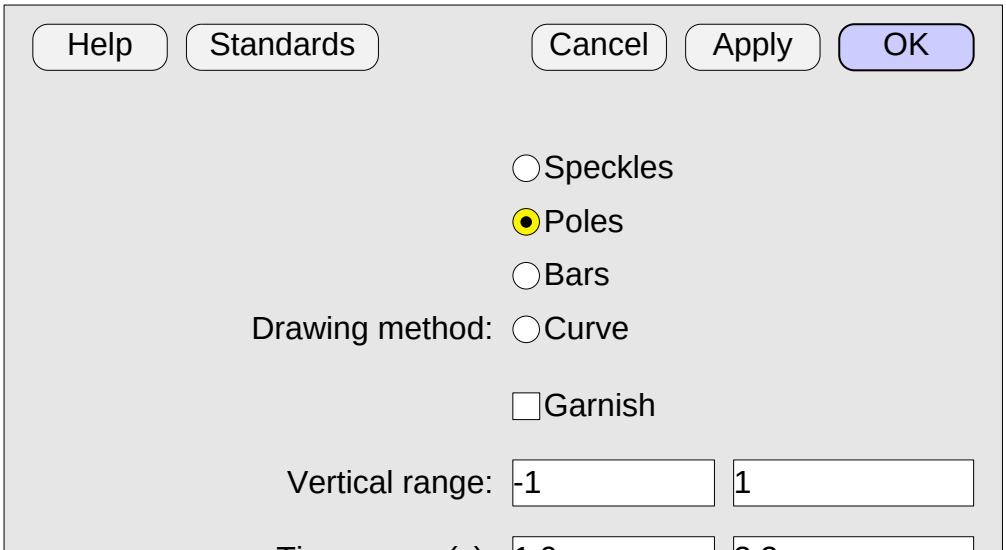
A multiple choice argument is sometimes represented by a *radio box* instead of by an option menu. For instance, the last example above could equally well have looked like

In supplying arguments to a command in a script, there is no difference between an option menu and a radio box. This last example will therefore again look like the following in a script:

```
Draw... 1.0 3.2 -1 1 no Poles
```

4. Text arguments

Consider another frequently used menu command, namely **Create Sound from formula...** in the **New** menu:



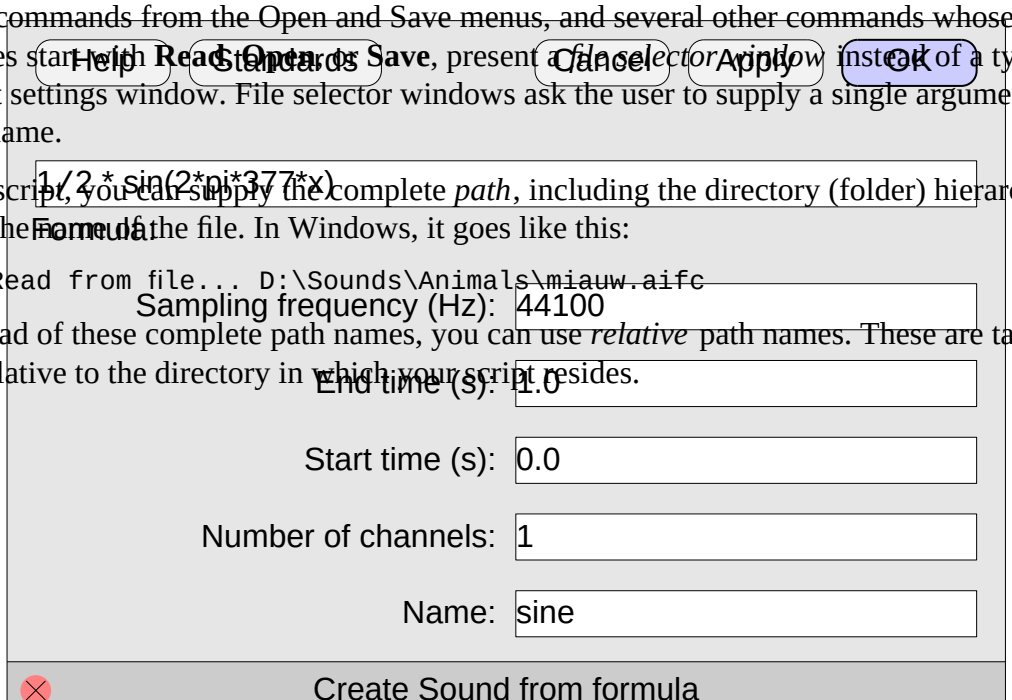
In a script this would look like:

```
Create Sound from formula... sine 1 0.0 1.0 44100 1/2 *
      sin(2*pi*377*x)
```

Both the first argument (**Name**) and the sixth argument (**Formula**) are *text arguments*. They are written in a script just as you would type them into the settings window. Well, *mostly* (see 6 and 7 below)...

5. File arguments

The commands from the Open and Save menus, and several other commands whose names start with **Read**, **Open**, or **Save**, present a *file selector window* instead of a typical Praat settings window. File selector windows ask the user to supply a single argument: the file name.



In a script, you can supply the complete *path*, including the directory (folder) hierarchy and the name of the file. In Windows, it goes like this:

```
Read from file... D:\Sounds\Animals\miauw.aifc
```

Instead of these complete path names, you can use *relative* path names. These are taken as relative to the directory in which your script resides.

In Windows, a relative path name starts without a backslash. So if your script is C:\Sounds\Analysis.praat, the sound file is read by

```
Read from file... Animals\miauw.aifc
```

Finally, your script may not be in a directory *above* the directory from which you like to read, but in a directory on the side, like D:\Scripts. The commands would then read

```
Read from file... ..\Animals\miauw.aifc
```

6. Space paranoia

The thing that separates the arguments in a script line is the *space* character (“ ”). This can become problematic if the argument itself also contains spaces, as can happen in text arguments such as in **Formula** above under 4 (because “1/2 * sin(2*pi*377*x)” contains two spaces). If the text argument with spaces is the last argument (as **Formula** is above), then there’s actually no problem: Praat knows that the **Create Sound from formula...** command takes six arguments; the sixth argument is simply everything that follows the first five arguments, so Praat can figure out that those last two spaces aren’t meant to separate arguments.

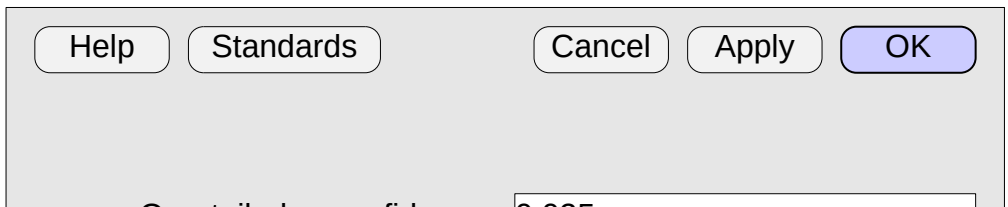
For this reason, most texts in Praat’s settings windows appear in the last (bottom) field. When they don’t, you have to use double quotes around the text argument with spaces. For instance, consider the command **Report difference (Student t)...** (for Table objects):

This command performs a paired-samples t-test between the columns `F0 before` and `F0 after`. In a script this would look like:

```
Report difference (Student t)... “F0 before” “F0 after” 0.025
```

The quotes around the first argument tell Praat that the space between `F0` and `before` is part of the argument, i.e. part of the first column name.

The last, and most confusing, strange thing that can happen is if a (non-last) text argument contains quotes itself. In such a case, you enclose the argument between quotes, and you double each quote that appears inside the argument. For example, the following window



is scripted as

```
Report difference (Student t)... "F0 ""before""" "F0 ""after""" 0.025
```

7. How to supply arguments automatically

Now you know all the ways to write the arguments of commands in a script line, including some weird cases. If you dislike manually copying arguments from settings windows into your script, or if you are still not sure how a complicated argument should be written in a script, you can use the **history mechanism**: choose **Clear history** from the **Edit** menu in your **ScriptEditor**, click your command button, edit the arguments, and click **OK**. The command will be executed. Then choose **Paste history**, and the command line, including the arguments (with correct quotes, for example), will appear in the ScriptEditor at the position of the text cursor. You can build whole new scripts on the basis of this mechanism.

Links to this page

Scripting
Scripting 3. Simple language elements

Scripting 3.1. Hello world

Many manuals of computer programming languages start with their answer on the following question:

How do I write the text “Hello world” on the screen?

For the Praat scripting language, there are two answers.

1. “Hello world” in the Info window

The simplest answer is that you open the ScriptEditor window with **New Praat script** from the **Praat** menu, then type the following line into the ScriptEditor window:

```
echo Hello world
```

and finally choose **Run** from the **Run** menu.

When you try this, the result should be that the Info window comes to the front, and that it shows the text `Hello world`:

Now suppose that you to write two lines instead of just one, so you try a script with two lines:

```
echo Hello world
echo How do you do?
```

This turns out not to do what you want: it seems to write only the text `How do you do?`. This happens because the **echo** command first erases the Info window, then writes the line of text. So the first line of the script did write the text `Hello world`, but the second line wiped it out and wrote `How do you do?` instead. The script that does what you want is

```
echo Hello world
printline How do you do?
```

Now the result will be

This works because **printline** write a line without erasing the Info window first.

Finally, try the following script:

```
printline Another try
printline Goodbye
```

The result could be

In other words, **printline** writes lines into the Info window without erasing it, even if you run a script anew. This is why many Praat scripts that write into the Info window do an **echo** first, and follow it with a series of **printlines**.

For more information on these commands, see
Scripting 6.2. Writing to the Info window.

2. “Hello world” in the Picture window.

You can also show text in the Picture window. If you are an experienced Praat user, you have probably used the comamnd **Text top...** before. You can use it do draw a text at the top of the current *viewport*, which is the part of the Picture window where the next drawing will occur and which is marked by the pink *margins*. Thus, when you select the top 4×3 inches of the Picture window (with the mouse), set the font size to 12 (with the **Pen** menu), and run the script

```
Text top... yes Hello world
```

then you’ll see

So this works the same as when you choose **Text top...** from the **Margins** menu by hand, with **Far** switched on.

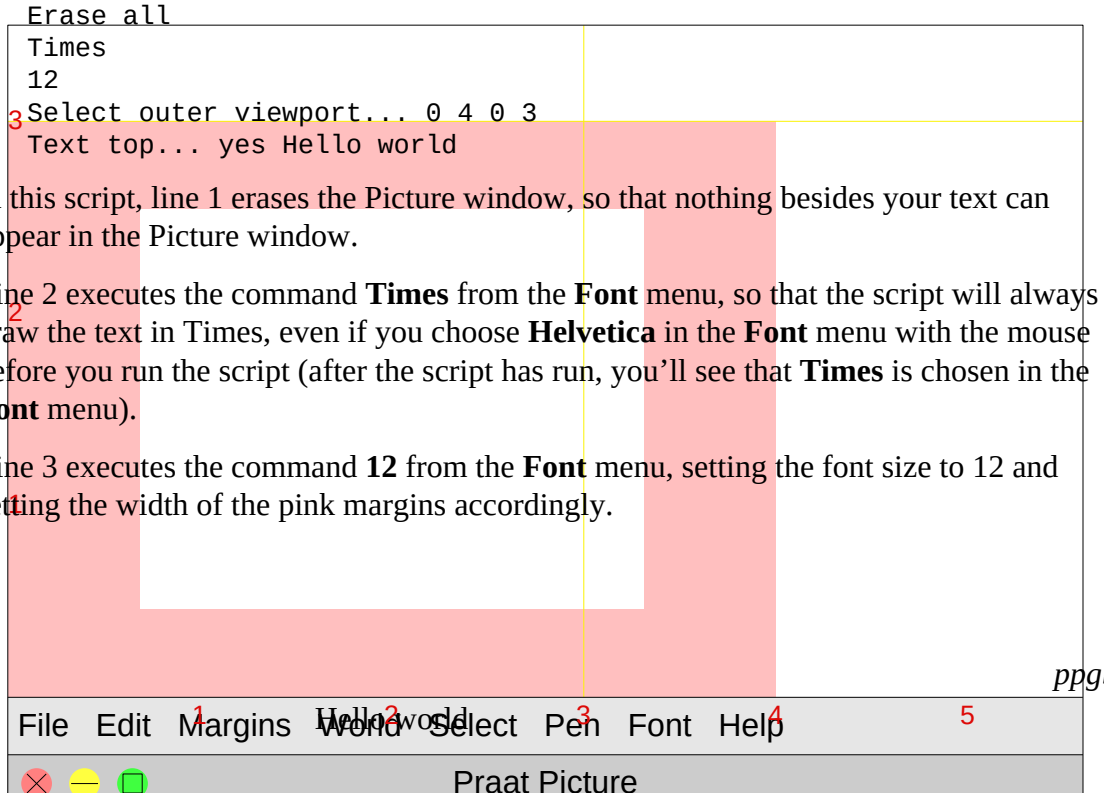
If you want your script to always show the same text at the same position, with nothing else in the picture, then you can make your script a bit more extensive:

```
Erase all
Times
12
Select outer viewport... 0 4 0 3
Text top... yes Hello world
```

In this script, line 1 erases the Picture window, so that nothing besides your text can appear in the Picture window.

Line 2 executes the command **Times** from the **Font** menu, so that the script will always draw the text in Times, even if you choose **Helvetica** in the **Font** menu with the mouse before you run the script (after the script has run, you'll see that **Times** is chosen in the **Font** menu).

Line 3 executes the command **12** from the **Font** menu, setting the font size to 12 and setting the width of the pink margins accordingly.



Line 4 executes the command **Select outer viewport...** from the **Select** menu. This performs an action that you would normally do by dragging the mouse, namely selecting the part of the Picture window that runs from 0 to 4 inches horizontally and from 0 to 3 inches vertically. After running the script, the *viewport* is indeed $[0, 4] \times [0, 3]$, as you can clearly see from the pink margins above.

Line 5 finally writes the text.

For more information on these commands, see **Picture window**.

Links to this page

Scripting
Scripting 3. Simple language elements

Scripting 3.2. Numeric variables

In any general procedural programming language you can work with *variables*, which are places in your computer's memory where you can store a number or anything else.

For instance, you could put the number 3.1 into the variable `b` in the following way:

```
b = 3.1
```

This statement is called as *assignment*, i.e., you *assign* the *value* 3.1 to the *variable* `b`. We read this statement aloud as "b becomes 3.1". What this means is that after this statement, the memory location `b` *contains* the numeric value (number) 3.1.

You can regard a variable as a box: you put the value 3.1 into the box named `b`. Or you can regard a variable as a house: the house is called `b` and now the family "3.1" is living there. Or you can regard it as any other storage location.

To see what value a variable contains (what's in the box, or who lives in the house), you can use the **echo** command:

```
b = 3.1
echo The value is 'b'.
```

This will put the text `The value is 3.1.` into the Info window, as you are invited to verify.

A variable is called a variable because it is *variable*, i.e. its value can change. Try the script

```
b = 3.1
b = 5.8
echo The value is 'b'.
```

You will see that `b` ends up having the value 5.8. The first line puts the value 3.1 there, but the second line replaces it with 5.8. It's like taking the 3.1 out of the box and putting the 5.8 in its stead. Or the family 3.1 moves from the house, and the family called 5.8 moves in.

In an assignment, the part to the right of the "becomes" sign (the "=" sign) doesn't have to be a number; it can be any *formula* that *evaluates* to a number. For instance, the script

```
b = 3.1 * 2
echo The value is 'b'.
```

puts the text `The value is 6.2.` into the Info window. This works because Praat handles the first line in the following way:

1. the formula `3.1 * 2` is *evaluated* (i.e. its value is computed), and the result is 6.2.
2. the value 6.2 is subsequently stored in the variable `b`.

After line 1 has been executed, the variable `b` just contains the value 6.2, nothing more; the variable `b` doesn't remember that that value has been computed by multiplying 3.1 with 2.

Formulas can contain more things than numbers: they can also contain other variables:

```
b = 3.1
c = b * 2
echo The value of b is 'b', and the value of c is 'c'.
```

In the first line, `b` gets the value 3.1. In the second line, the formula `b * 2` first has to be evaluated. Praat looks up the value of `b` (which is 3.1), so that it knows that the formula actually means `3.1 * 2`. Praat evaluates this formula and stores the result (namely the value 6.2) into the variable `c`, which will then contain nothing else than the value 6.2. The Info window thus reports `The value of b is 3.1, and the value of c is 6.2.`.

After these explanations, consider the following script:

```
b = 3.1
c = b * 2
b = 5.8
echo The value of c is 'c'.
```

Can you figure out what the Info will report? If you think it will report `The value of c is 6.2.`, then you are correct: after the first line, `b` contains the value 3.1; after the second line, the value of `c` is therefore 6.2, and nothing more; after line 3, the value of `b` has changed to 5.8, but the value of `c` hasn't changed and is still 6.2.

If you thought that `c` would end up having the value 11.6, then you're thinking in terms of a non-procedural language such as Prolog; you may have thought that the thing assigned to `c` in the second line is the whole *formula* `b * 2`, so that `c` changes when `b` changes. But this is not the case: the thing stored in `c` is just the *value* of the formula `b * 2` at that moment, which is 6.2, and `c` doesn't remember how it got that value. If you have trouble understanding this, consult anybody who writes programs.

Links to this page

Scripting
Scripting 3. Simple language elements
Scripting 3.4. String variables

Scripting 3.3. Numeric queries

Now that you know how to script a menu command, and you know how variables work, you are ready to combine the two.

Suppose you have selected a Sound in the object list. One of the commands available in the **Query** menu is **Get power...**. To get the mean power of the whole Sound, you can try the script

```
Get power... 0 0
```

This works, and it puts the text `0.1350605005239421 Pa2` or so into the Info window.

Often though, you will want to use this value in the script itself, perhaps for doing computations with it or for reporting it with a nice text around it. This is how you do it:

```
power = Get power... 0 0
echo The power of this sound is 'power' Pascal-squared.
```

The first line of this script executes the menu command **Get power...**, but puts the value 0.1350605005239421 into the variable `power` instead of into the Info window (the variable can have any name you like, as long as it starts with a lower-case letter and consists of letters and digits; see **Scripting 5.1. Variables**).

The second line then reports the value in the Info window, this time with a nice text around it.

Links to this page

Scripting
Scripting 3. Simple language elements
Scripting 3.5. String queries

Scripting 3.4. String variables

Just as you can store **numeric variables**, you can store *string variables*, which contain text instead of numbers. Here is an example:

```
word1$ = “Hello”
word2$ = “world”
sentence$ = word1$ + “ ” + word2$
echo The whole sentence is: ’sentence$’
```

Yes, this is another way to get the sentence `Hello world` into the Info window. It’s a more linguistically valid way to do it, and here is how it works:

1. In line 1, the value “Hello”, which is a text (as we can see by its use of quotes), is stored into the variable `word1$`, which is a string variable (as we can see because its name ends in a dollar sign).
2. In line 2, the text value “world” is stored into the string variable `word2$`.
3. In line 3, we have the formula `word1$ + “ ” + word2$`, which contains two variables, namely `word1$` and `word2$`.
4. The values of the two variables are “Hello” and “world”, respectively, so what the formula actually says is “Hello” + “ ” + “world”.
5. The pluses in the formula mean “concatenate”, so we concatenate the three strings “Hello”, “ ”, and “world”, giving the longer string “Hello world”.
6. Still in line 3, the string value “Hello world” is assigned to the string variable `sentence$`.
7. Line 4 reports in the Info window: `The whole sentence is: Hello world`

Links to this page

Scripting
Scripting 3. Simple language elements

Scripting 3.5. String queries

Just as you can use menu commands (usually in a **Query** menu) to query **numbers**, you can query texts as well.

For instance, if you want to know the label of the third interval in the first tier of a selected **TextGrid**, you can try the following script:

```
Get label of interval... 1 3
```

This works because when you select a Textgrid, the **Query** menu will contain the command **Get label of interval...**, which takes two numeric arguments, namely **Tier number** and **Interval number**:

The script will write the text of the interval, say `hello`, into the Info window. To get the text into a variable instead, write a script like this:

```
text$ = Get label of interval... 1 3
echo The text in interval 3 of tier 1 is: 'text$'
```

After you run this script, the Info window will look like this:

Hey, yet another way to implement “Hello world” with the Praat scripting language!

Links to this page

Scripting
Scripting 3. Simple language elements
Scripting 3.6. “For” loops

Scripting 3.6. “For” loops

The power of a procedural programming language is most easily illustrated with the *for-loop*.

Take the example of the **previous page**, whereas you wanted to know the text in the third interval of the first tier of a selected TextGrid. It’s easy to imagine that you actually want the texts of *all the first five* intervals. With knowledge from the previous sections, you could write it like this:

```
echo The texts in the first five intervals:
text$ = Get label of interval... 1 1
printline Interval 1: 'text$'
text$ = Get label of interval... 1 2
printline Interval 2: 'text$'
text$ = Get label of interval... 1 3
printline Interval 3: 'text$'
text$ = Get label of interval... 1 4
printline Interval 4: 'text$'
text$ = Get label of interval... 1 5
printline Interval 5: 'text$'
```

The result will be something like

This can be done nicer. the first step is to realize that the sentences starting with `text$` are similar to each other, and the sentence starting with `printline` are also similar to each other. They only differ in the interval number, and can therefore be made *identical* by using a variable for the interval number, like this:

```
echo The texts in the first five intervals:
intervalNumber = 1
```

```
text$ = Get label of interval... 1 intervalNumber
printline Interval 'intervalNumber': 'text$'
intervalNumber = 2
text$ = Get label of interval... 1 intervalNumber
printline Interval 'intervalNumber': 'text$'
intervalNumber = 3
text$ = Get label of interval... 1 intervalNumber
printline Interval 'intervalNumber': 'text$'
intervalNumber = 4
text$ = Get label of interval... 1 intervalNumber
printline Interval 'intervalNumber': 'text$'
intervalNumber = 5
text$ = Get label of interval... 1 intervalNumber
printline Interval 'intervalNumber': 'text$'
```

A new trick that you see here is that as a numeric argument (**Interval number**, the second argument to **Get label of interval...**), you can use not only a number (as in all previous examples), but also a variable (`intervalNumber`). The rest of the script should be known stuff by now.

The script above is long, but it can be made much shorter with the use of a *for-loop*:

```
echo The texts in the first five intervals:
for intervalNumber from 1 to 5
    text$ = Get label of interval... 1 intervalNumber
    printline Interval 'intervalNumber': 'text$'
endfor
```

The two lines that were repeated five times in the previous version now show up with indentation between a *for* line and its corresponding *endfor*. Those two lines (the `text$` and the `printline` line) are executed five times: for `intervalNumber` equal to 1, for `intervalNumber` equal to 2, for `intervalNumber` equal to 3, for `intervalNumber` equal to 4, and for `intervalNumber` equal to 5, in that order.

In the above example, using a loop does not do much more than save eight lines, at the cost of adding two new lines. But imagine the case in which you want to list *all* the texts in the intervals: the version without the loop is no longer possible. By contrast, the version *with* the loop is still possible, because we have the command **Get number of intervals...**, which gives us the number of intervals in the specified tier (here, tier 1). So you do:

```
numberOfIntervals = Get number of intervals... 1
echo The texts in all 'numberOfIntervals' intervals:
for intervalNumber from 1 to numberOfIntervals
    text$ = Get label of interval... 1 intervalNumber
    printline Interval 'intervalNumber': 'text$'
endfor
```

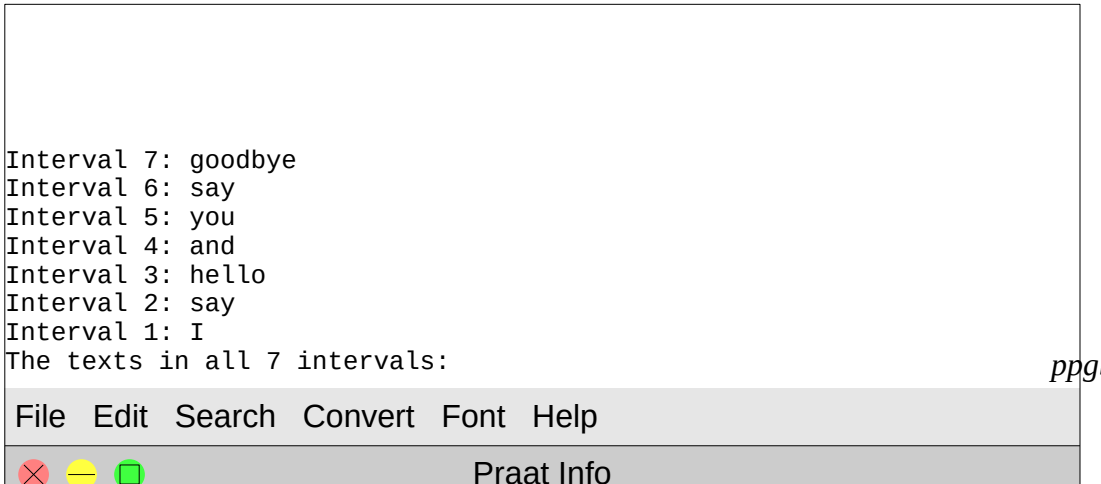
This may yield something like

This is the first script in this tutorial that is useful in itself. On the basis of it you can create all kinds of ways to list the texts in intervals. Here is how you would also list the durations of those intervals:

```
numberOfIntervals = Get number of intervals... 1
echo The durations and texts in all 'numberOfIntervals' intervals:
for intervalNumber from 1 to numberOfIntervals
    startTime = Get start point... 1 intervalNumber
    endTime = Get end point... 1 intervalNumber
    duration = endTime - startTime
    text$ = Get label of interval... 1 intervalNumber
    printline Interval 'intervalNumber' is 'duration' seconds long and
          contains the text: 'text$'
endfor
```

Links to this page

Scripting
Scripting 3. Simple language elements
Scripting 3.7. Layout



ppgb

Scripting 3.7. Layout

This chapter handles the way you use white space, comments, and continuation lines in a Praat script.

White space

Praat ignores all white space (spaces and tabs) that you put at the beginning of lines. The indentation that you saw on the **previous page** was therefore used solely for readability. You are advised to use indenting, though, with three or four spaces for each level, as in the following example, which loops over all tiers and intervals of a TextGrid:

```
echo The texts in all tiers and intervals:
numberOfTiers = Get number of tiers
for tierNumber from 1 to numberOfTiers
    numberOfIntervals = Get number of intervals... tierNumber
    for intervalNumber from 1 to numberOfIntervals
        text$ = Get label of interval... tierNumber intervalNumber
        printline Tier 'tierNumber', interval 'intervalNumber':
              'text$'
    endfor
endfor
```

Praat also ignores lines that are empty or consist solely of white space, so you use those to structure your script visually.

Comments

Comments are lines that start with “#” or “;”. Praat ignores these lines when your script is running:

```
# Create 1 second of a sine wave with a frequency of 100 Hertz,
# sampled at 44100 Hz:
Create Sound from formula... sine 1 0 1 44100 sin (2*pi*100*x)
```

Because of its visibility, you are advised to use “#” for comments that structure your script, and “;” perhaps only for “commenting out” a statement, i.e. to temporarily put it before a line that you don’t want to execute.

Continuation lines

There is normally one line per statement, and one statement per line. But some statements are very long, such as this one on a previous page:

```
printline Interval 'intervalNumber' is 'duration' seconds long and
      contains the text: 'text$'
```

By making the current window wider, you can see that I really put this whole statement on a single line. I could have distributed it over two lines in the following way, by using three dots (an *ellipsis*):

```
printline Interval 'intervalNumber' is 'duration' seconds long
... and contains the text: 'text$'
```

Here is another common type of example:

```
Create Sound from formula... windowedSine 1 0 1 44100
... 0.5 * sin(2*pi*1000*x) * exp(-0.5*((x-0.5)/0.1)^2)
```

You will normally want to follow such an ellipsis with a space, unless you want to concatenate the parts of a long word:

```
Select outer viewport... 0 10 0 4
Text top... yes It's a long way to Llanfairpwllgwyngyll
...gogerychwyrndrobwllllantysiliogogogoch,
... unless you start from Tyddyn-y-felin.
```

Links to this page

Scripting
Scripting 3. Simple language elements

Scripting 4. Object selection

This chapter is about how to select objects from your script, and how to find out what objects are currently selected.

Selecting objects

To simulate the mouse-clicked and dragged selection in the list of objects, you have the following commands:

select *object*

selects one object, and deselects all others. If there are more objects with the same name, the most recently created one (i.e., the one nearest to the bottom of the list of objects) is selected:

```
select Sound hallo
Play
```

plus *object*

adds one object to the current selection.

minus *object*

removes one object from the current selection.

select all

selects all objects (please try not to use this, because it will remove even the objects that your script did not create!):

```
select all
Remove
```

In the Praat shell, newly created objects are automatically selected. This is also true in scripts:

```
! Generate a sine wave, play it, and draw its spectrum.
Create Sound from formula... sine377 1 0 1 44100
... 0.9 * sin (2 * pi * 377 * x)
Play
To Spectrum... yes
! Draw the Spectrum:
Draw... 0 5000 20 80 yes
! Remove the created Spectrum and Sound:
plus Sound sine377
Remove
```

Instead of by name, you can also select objects by their sequential ID:

```
select 43
```

This selects the 43rd object that you created since you started the program (see below).

Querying selected objects

You can get the name of a selected object into a string variable. For instance, the following reads the name of the second selected Sound (as counted from the top of the list of objects) into the variable *name*$:

```
name$ = selected$ ("Sound", 2)
```

If the Sound was called "Sound hallo", the variable *name*$ will contain the string "hallo". To get the name of the topmost selected Sound object, you can leave out the number:

```
name$ = selected$ ("Sound")
```

To get the full name (type + name) of the third selected object, you do:

```
fullName$ = selected$ (3)
```

To get the full name of the topmost selected object, you do:

```
fullName$ = selected$ ()
```

To get the type and name out of the full name, you do:

```
type$ = extractWord$ (fullName$, "")
name$ = extractLine$ (fullName$, " ")
```

Negative numbers count from the bottom. Thus, to get the name of the bottom-most selected Sound object, you say

```
name$ = selected$ ("Sound", -1)
```

You would use **selected$** for drawing the object name in a picture:

```
Draw... 0 0 0 0 yes
name$ = selected$ ("Sound")
Text top... no This is sound 'name$'
```

For identifying previously selected objects, this method is not very suitable, since there may be multiple objects with the same name:

```
# The following two lines are OK:
soundName$ = selected$ ("Sound", -1)
pitchName$ = selected$ ("Pitch")
# But the following line is questionable, since it doesn't
# necessarily select the previously selected Sound again:
select Pitch 'pitchName$'
```

Instead of this error-prone approach, you should get the object's unique ID. The correct version of our example becomes:

```
sound = selected ("Sound", -1)
pitch = selected ("Pitch")
# Correct:
```

```
select pitch
```

To get the number of selected Sound objects into a variable, use

```
numberOfSelectedSounds = numberOfSelected ("Sound")
```

To get the number of selected objects into a variable, use

```
numberOfSelectedObjects = numberOfSelected ()
```

Example: doing something to every selected Sound

```
n = numberOfSelected ("Sound")
for i to n
    sound'i' = selected ("Sound", i)
endfor
# Median pitches of all selected sounds:
for i to n
    select sound'i'
    To Pitch... 0.0 75 600
    f0 = Get quantile... 0 0 0.50 Hertz
    Remove
endfor
# Restore selection:
if n >= 1
    select sound1
    for i from 2 to n
        plus sound'i'
    endfor
endif
```

A shortcut

Instead of

```
Create Sound from formula... sine 1 0 1 44100
... 0.5 * sin(2*pi*1000*x)
sound = selected ("Sound")
```

you can just write

```
sound = Create Sound from formula... sine 1 0 1 44100
... 0.5 * sin(2*pi*1000*x)
```

and instead of

```
To Pitch... 0.0 75 600
pitch = selected ("Pitch")
```

you can write

```
pitch = To Pitch... 0.0 75 600
```

This only works if the command creates a single object.

Links to this page

Scripting
What's new?

Scripting 5.1. Variables

In a Praat script, you can use numeric variables as well as string variables.

Numeric variables

Numeric variables contain integer numbers between -1,000,000,000,000,000 and +1,000,000,000,000,000 or real numbers between -10^{308} and $+10^{308}$. The smallest numbers lie near -10^{-308} and $+10^{-308}$.

You can use *numeric variables* in your script:

variable = formula

evaluates a numeric formula and assign the result to a variable.

Example:

```
length = 10
Draw line... 0 length 1 1
```

Names of numeric variables must start with a lower-case letter, optionally followed by a sequence of letters, digits, and underscores.

String variables

You can also use *string variables*, which contain text:

```
title$ = "Dutch nasal place assimilation"
```

As in the programming language Basic, the names of string variables end in a dollar sign.

Variable substitution

Existing variables are substituted when put between quotes:

```
x = 99
x2 = x * x
echo The square of 'x' is 'x2'.
```

This will write the following text to the Info window:

```
The square of 99 is 9801.
```

You can reduce the number of digits after the decimal point by use of the colon:

```
root = sqrt (2)
echo The square root of 2 is approximately 'root:3'.
```

This will write the following text to the Info window:

```
The square root of 2 is approximately 1.414.
```

By using ":0", you round to whole values:

```
root = sqrt (2)
echo The square root of 2 is very approximately 'root:0'.
```

This will write the following text to the Info window:

```
The square root of 2 is very approximately 1.
```

By using ":3%", you give the result in a percent format:

```
jitter = 0.0156789
echo The jitter is 'jitter:3%'.
```

This will write the following text to the Info window:

```
The jitter is 1.568%.
```

The number 0, however, will always be written as 0, and for small numbers the number of significant digits will never be less than 1:

```
jitter = 0.000000156789
echo The jitter is 'jitter:3%'.
```

This will write the following text to the Info window:

```
The jitter is 0.00002%.
```

Some **predefined numeric variables** are `macintosh`, `windows`, and `unix`, which are 1 if the script is running on a Macintosh, Windows, or Unix platform (respectively), and which are otherwise zero. Another one is `praatVersion`, which is e.g. 5339 for the current version of Praat.

Some **predefined string variables** are `newline$`, `tab$`, and `shellDirectory$`. The last one specifies the directory that was the default directory when Praat started up; you can use it in scripts that run from the Unix or DOS command line. Likewise, there exist the predefined string variables `homeDirectory$`, `preferencesDirectory$`, and `temporaryDirectory$`. These three refer to your home directory (which is where you log in), the Praat **preferences directory**, and a directory for saving temporary files; if you want to know what they are on your computer, try to **echo** them in a script window. The variable `defaultDirectory$` is available for formulas in scripts; it is the directory that contains the script file. Finally, we have `praatVersion$`, which is "5.3.39" for the current version of Praat.

To check whether a variable exists, you can use the function

```
variableExists (variableName$)
```

Links to this page

Formulas 1.9. Formulas in scripts
Scripting

Scripting 3.3. Numeric queries
Scripting 5. Language elements reference

Scripting 5.2. Formulas

In a Praat script, you can use numeric expressions as well as string expressions.

Numeric expressions

You can use a large variety of **Formulas** in your script:

```
length = 10
height = length/2
area = length * height
echo The area is 'area'.
```

You can use numeric variables and formulas in numeric arguments to commands:

```
Draw line... 0 length 0 length/2
```

Of course, all arguments except the last should either not contain spaces, or be enclosed in double quotes. So you would write either

```
Draw line... 0 height*2 0 height
```

or

```
Draw line... 0 "height * 2" 0 height
```

You can use numeric expressions in assignments (as above), or after **if**, **elsif**, **while**, **until**, and twice after **for**.

String expressions

You can use a large variety of **Formulas** in your script:

```
addressee$ = "Silke"
greeting$ = "Hi " + addressee$ + "!"
echo The greeting is: 'greeting$'
```

You can use string variables and formulas in *numeric* arguments to commands:

```
Draw line... 0 length(greeting$) 0 100
Draw line... 0 if(answer$="yes")then(20)else(30)fi 0 100
```

Assignments fromQuery commands

On how to get information from commands that normally write to the Info window, see **Scripting 6.3. Query commands**.

Links to this page

Scripting
Scripting 5. Language elements reference

Scripting 5.3. Jumps

You can use conditional jumps in your script:

if *expression*

elsif *expression*

if the expression evaluates to zero or *false*, the execution of the script jumps to the next **elsif** or after the next **else** or **endif** at the same depth..

The following script computes the preferred length of a bed for a person 'age' years of age:

```
if age <= 3
   length = 1.20
elsif age <= 8
   length = 1.60
else
   length = 2.00
endif
```

A variant spelling for **elsif** is **elif**.

Links to this page

Scripting
Scripting 5. Language elements reference

Scripting 5.4. Loops

“For” loops

for *variable* **from** *expression*$_1$ **to** *expression*$_2$

for *variable* **to** *expression*

the statements between the **for** line and the matching **endfor** will be executed while a variable takes on values between two expressions, with an increment of 1 on each turn of the loop. The default starting value of the loop variable is 1.

The following script plays nine sine waves, with frequencies of 200, 300, ..., 1000 Hz:

```
for i from 2 to 10
   frequency = i * 100
   Create Sound from formula... tone Mono 0 0.3 44100
      0.9*sin(2*pi*frequency*x)
   Play
   Remove
endfor
```

The stop value of the **for** loop is evaluated on each turn. If the second expression is already less than the first expression to begin with, the statements between **for** and **endfor** are not executed even once.

“Repeat” loops

until *expression*

the statements between the matching preceding **repeat** and the **until** line will be executed again if the expression evaluates to zero or *false*.

The following script measures the number of trials it takes me to throw 12 with two dice:

```
throws = 0
repeat
    eyes = randomInteger (1, 6) + randomInteger (1, 6)
    throws = throws + 1
until eyes = 12
echo It took me 'throws' trials to throw 12 with two dice.
```

The statements in the **repeat/until** loop are executed at least once.

“While” loops

while *expression*

if the expression evaluates to zero or *false*, the execution of the script jumps after the matching **endwhile**.

endwhile

execution jumps back to the matching preceding **while** line, which is then evaluated again.

The following script forces the number *x* into the range [0; 2π):

```
while x < 0
    x = x + 2 * pi
endwhile
while x >= 2 * pi
    x = x - 2 * pi
endwhile
```

If the expression evaluates to zero or *false* to begin with, the statements between **while** and **endwhile** are not executed even once.

Links to this page

Scripting
Scripting 5. Language elements reference
Scripting 5.5. Procedures

Scripting 5.5. Procedures

Sometimes in a Praat script, you will want to perform the same thing more than once. In **§5.4** we saw how *loops* can help there. In this section we will see how *procedures* (also called *subroutines*) can help us.

Imagine that you want to play a musical note with a frequency of 440 Hz (an “A”) followed by a note that is one ocatve higher, i.e. has a frequency of 880 Hz (an “a”). You could achieve this with the following script:

```
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * 440 * x)
Play
Formula... 0.4 * sin (2 * pi * 880 * x)
Play
Remove
```

This script creates a sound with a sine wave with an amplitude of 0.4 and a frequency of 440 Hz, then plays this sound, then changes the sound into a sine wave with a frequency of 880 Hz, then plays this changed sound, and then removes the Sound object from the object list.

This script is perfect if all you want to do is to play those two notes and nothing more. But now imagine that you want to play such an octave jump not only for a note of 440 Hz, but also for a note of 400 Hz and for a note of 500 Hz. You could use the following script:

```
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * 440 * x)
Play
Formula... 0.4 * sin (2 * pi * 880 * x)
Play
Remove
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * 400 * x)
Play
Formula... 0.4 * sin (2 * pi * 800 * x)
Play
Remove
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * 500 * x)
Play
Formula... 0.4 * sin (2 * pi * 1000 * x)
Play
Remove
```

This script works but is no longer perfect. It contains many similar lines, and is difficult to read.

Here is where *procedures* come in handy. With procedures, you can re-use similar pieces of code. To make the three parts of the above script more similar, I'll rewrite it using two variables (*frequency* and *octaveHigher*):

```
frequency = 440
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * frequency * x)
Play
octaveHigher = 2 * frequency
Formula... 0.4 * sin (2 * pi * octaveHigher * x)
Play
Remove
frequency = 400
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * frequency * x)
Play
octaveHigher = 2 * frequency
Formula... 0.4 * sin (2 * pi * octaveHigher * x)
Play
Remove
frequency = 500
Create Sound from formula... note Mono 0 0.3 44100
... 0.4 * sin (2 * pi * frequency * x)
Play
octaveHigher = 2 * frequency
Formula... 0.4 * sin (2 * pi * octaveHigher * x)
Play
Remove
```

You can now see that seven lines of the script appear identically three times. I'll put those seven lines into a *procedure* that I name "playOctave":

```
procedure playOctave
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * frequency * x)
    Play
    octaveHigher = 2 * frequency
    Formula... 0.4 * sin (2 * pi * octaveHigher * x)
    Play
    Remove
endproc
```

As you see, a *procedure definition* in Praat consists of three parts:

1. a line with the word **procedure** followed by the name of the procedure;
2. the *body* of the procedure (here: seven lines);
3. a line with the word **endproc**.

You can put a procedure definition anywhere in your script; the beginning or end of the script are common places.

The bodies of procedures are executed only if you *call* the procedure explicitly, which you can do anywhere in the rest of your script:

```
frequency = 440
call playOctave
frequency = 400
call playOctave
frequency = 500
call playOctave
procedure playOctave
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * frequency * x)
    Play
    octaveHigher = 2 * frequency
    Formula... 0.4 * sin (2 * pi * octaveHigher * x)
    Play
    Remove
endproc
```

This script works as follows. First, the number 440 is assigned to the variable *frequency* in line 1. Then, execution of the script arrives at the **call** statement of line 2. Praat then knows that it has to jump to the procedure called *playOctave*, which is found on line 7. The execution of the script then proceeds with the first line of the procedure body, where a Sound is created. Then, the other lines of the procedure body are also executed, ending with the removal of the Sound. Then, the execution of the script arrives at the **endproc** statement. Here, Praat knows that it has to jump back to the line after the line that the procedure was called from; since the procedure was called from line 2, the execution proceeds at line 3 of the script. There, the number 400 is assigned to the variable *frequency*. In line 4, execution will jump to the procedure again, and with the next **endproc** the execution will jump back to line 5. There, 500 is assigned to *frequency*, followed by the third jump to the procedure. the third **endproc** jumps back to the line after the third **call**, i.e. to line 7. Here the execution of the script will stop, because there are no more executable commands (the procedure definition at the end is not executed again).

Arguments

The above example contains something awkward. The procedure *playOctave* requires that the variable *frequency* is set to an appropriate value, so before calling *playOctave* you always have to insert a line like

```
frequency = 440
```

This can be improved upon. In the following version of the script, the procedure *playOctave* requires an explicit *argument*:

```
call playOctave 440
call playOctave 400
call playOctave 500
```

```
procedure playOctave  frequency
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * frequency * x)
    Play
    octaveHigher = 2 * frequency
    Formula... 0.4 * sin (2 * pi * octaveHigher * x)
    Play
    Remove
endproc
```

This works as follows. The first line of the procedure now not only contains the name (*playOctave*), but also a list of variables (here only one: *frequency*). In the first line of the script, the procedure *playOctave* is called with the *argument list* "440". Execution then jumps to the procedure, where the argument 440 is assigned to the variable *frequency*, which is then used in the body of the procedure.

Encapsulation and local variables

Although the size of the script has now been reduced to 12 lines, which cannot be further improved upon, there is still something wrong with it. Imagine the following script:

```
frequency = 300
call playOctave 440
call playOctave 400
call playOctave 500
echo 'frequency'
procedure playOctave  frequency
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * frequency * x)
    Play
    octaveHigher = 2 * frequency
    Formula... 0.4 * sin (2 * pi * octaveHigher * x)
    Play
    Remove
endproc
```

You might have thought that this script will write "300" to the Info window, because that is what you expect if you look at the first five lines. However, the procedure will assign the values 440, 400, and 500 to the variable *frequency*, so that the script will actually write "500" to the Info window, because 500 is last (fourth!) value that was assigned to the variable *frequency*.

What you would want is that variables that are used inside procedures, such as *frequency* and *octaveHigher* here, could somehow be made not to "clash" with variable names outside the procedure. A trick that works would be to include the procedure name into the names of these variables:

```
frequency = 300
call playOctave 440
call playOctave 400
```

```
call playOctave 500
echo 'frequency'
procedure playOctave  playOctave.frequency
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * playOctave.frequency * x)
    Play
    playOctave.octaveHigher = 2 * playOctave.frequency
    Formula... 0.4 * sin (2 * pi * playOctave.octaveHigher * x)
    Play
    Remove
endproc
```

This works. The six tones will be played, and "300" will be written to the Info window. But the formulation is a bit wordy, isn't it?

Fortunately, Praat allows an abbreviated version of these long names: just leave "playOctave" off from the names of the variables, but keep the period (.):

```
frequency = 300
call playOctave 440
call playOctave 400
call playOctave 500
echo 'frequency'
procedure playOctave  .frequency
    Create Sound from formula... note Mono 0 0.3 44100
    ... 0.4 * sin (2 * pi * .frequency * x)
    Play
    .octaveHigher = 2 * .frequency
    Formula... 0.4 * sin (2 * pi * .octaveHigher * x)
    Play
    Remove
endproc
```

This is the final version of the script. It works because Praat knows that you are using the variable names *.frequency* and *.octaveHigher* in the context of the procedure *playOctave*, so that Praat knows that by these variable names you actually mean to refer to *playOctave.frequency* and *playOctave.octaveHigher*.

It is advisable that you use such "local" variable names for all *parameters* of a procedure, i.e. for the variables listed after the **procedure** word (e.g. *.frequency*), as well as for all variables that you create in the procedure body (e.g. *.octaveHigher*). In this way, you make sure that you don't inadvertently use a variable name that is also used outside the procedure and thereby perhaps inadvertently change the value of a variable that you expect to be constant across a procedure call.

Numeric and string arguments

For numeric arguments you can use numeric expressions:

```
call playOctave 400+100
```

For string arguments you can only use literal text:

```
call listSpeaker Bart 38
call listSpeaker Katja 24
procedure listSpeaker  .name$ .age
    printline Speaker '.name$' is '.age' years old.
endproc
```

For string arguments that contain spaces, you use double quotes, except for the last argument, which is the rest of the line:

```
call conjugateVerb  be "I am" "you are" she is
procedure conjugateVerb  .verb$ .first$ .second$ .third$
    echo Conjugation of 'to '.verb$'':
    printline 1sg '.first$'
    printline 2sg '.second$'
    printline 3sg '.third$'
endproc
```

Arguments (except for the last) that contain double quotes should also be put between double quotes, and the double quotes should be doubled:

```
procedure texts .top$ .bottom$
    Text top... yes '.top$'
    Text bottom... yes '.bottom$'
endproc
call texts """hello"" at the top" "goodbye" at the bottom
```

Functions

The Praat scripting language does not have the concept of a "function" like some other scripting languages do. A function is a procedure that returns a number or a string. For instance, you can imagine the function `squareNumber` which takes a number (e.g. 5) as an argument and returns the square of that number (e.g. 25). Here is an example of how you can do that, using the global availability of local variables:

```
call squareNumber 5
echo The square of 5 is 'squareNumber.result'.
procedure squareNumber  .number
    .result = .number ^ 2
endproc
```

Another way to emulate functions is to use a variable name as an argument:

```
call squareNumber 5 square5
echo The square of 5 is 'square5'.
procedure squareNumber  .number .squareVariableName$
    '.squareVariableName$' = .number ^ 2
endproc
```

Links to this page

Scripting
Scripting 5. Language elements reference

Scripting 5.6. Arrays

You can use arrays of numeric and string variables:

```
for i from 1 to 5
    square [i] = i * i
    text$ [i] = mid$ (“hello”, i)
endfor
```

After this, the variables *square* [1], *square* [2], *square* [3], *square* [4], *square* [5], *text$* [1], *text$* [2], *text$* [3], *text$* [4], and *text$* [5] contain the values 1, 4, 9, 16, 25, “h”, “e”, “l”, “l”, and “o”, respectively.

You can use any number of variables in a script, but you can also use Matrix or Sound objects for arrays.

You can substitute variables with the usual single quotes, as in `'square[3]'`, without spaces. If the index is also a variable, however, you may need a dummy variable:

```
echo Some squares:
for i from 1 to 5
   hop = square [i]
   printline The square of 'i' is 'hop'
endfor
```

Links to this page

Scripting
Scripting 5. Language elements reference
What’s new?

Scripting 5.7. Including other scripts

You can include other scripts within your script:

```
a = 5
include square.praat
echo 'a'
```

The Info window will show the result 25 if the file square.praat is as follows:

```
a = a * a
```

The inclusion is done before any other part of the script is considered, so you can use the **form** statement and all variables in it. Usually, however, you will put some procedure definitions in the include file, that is what it seems to be most useful for. Watch out, however, for using variable names in the include file: the example above shows that there is no such thing as a separate name space.

Since including other scripts is the first thing Praat will do when considering a script, you cannot use variable substitution. For instance, the following will not work:

```
scriptName$ = "myscript.praat"
#This will *not* work:
include 'scriptName$'
#That did *not* work!!!
```

You can use full or relative file names. For instance, the file square.praat is expected to be in the same directory as the script that says *include square.praat*. If you use the ScriptEditor, you will first have to save the script that you are editing before any relative file names become meaningful (this is the same as with other uses of relative file names in scripts).

You can *nest* include files, i.e., included scripts can include other scripts. However, relative file names are always evaluated relative to the directory of the outermost script.

The **include** statement can only be at the start of a line: you cannot put any spaces in front of it.

Links to this page

Scripting
Scripting 5. Language elements reference
What's new?

Scripting 5.8. Quitting

Usually, the execution of your script ends when the interpreter has executed the last line that is not within a procedure definition. However, you can also explicitly stop the script:

exit

stops the execution of the script in the normal way, i.e. without any messages to the user. Any settings window is removed from the screen.

exit *error-message*

stops the execution of the script while sending an error message to the user. Any settings window will stay on the screen.

For an example, see **Scripting 6.8. Messages to the user**.

Links to this page

Scripting
Scripting 5. Language elements reference

Scripting 6.1. Arguments to the script

You can cause a Praat script to prompt for arguments. The file `playSine.praat` may contain the following:

```
form Play a sine wave
   positive Sine_frequency_(Hz) 377
   positive Gain_(0..1) 0.3 (= not too loud)
endform
Create Sound from formula... sine'sine_frequency' Mono 0 1 44100
      'gain' * sin (2*pi*'sine_frequency'*x)
Play
Remove
```

When running this script, the interpreter puts a settings window (*form*) on your screen, entitled “Play a sine wave”, with two fields, titled “Sine frequency (Hz)” and “Gain”, that have been provided with the standard values “377” and “0.3 (= not too loud)”, which you can change before clicking **OK**.

As you see, the underscores have been replaced with spaces: that looks better in the form. Inside the script, the field names can be accessed as variables: these do contain the underscores, since they must not contain spaces, but the parentheses (Hz) have been chopped off. Note that the first letter of these variables is converted to lower case, so that you can assign to them in your script.

Inside the script, the value “0.3 (= not too loud)” will be known as “0.3”, because this is a numeric field.

You can use the following field types in your forms:

real *variable initialValue*

for real numbers.

positive *variable initialValue*

for positive real numbers: the form issues an error message if the number that you enter is negative or zero; further on in the script, the number may take on any value.

integer *variable initialValue*

for whole numbers: the form reads the number as an integer; further on in the script, the number may take on any real value.

natural *variable initialValue*

for positive whole numbers: the form issues an error message if the number that you enter is negative or zero; further on in the script, the number may take on any real value.

word *variable initialValue*

for a string without spaces: the form only reads up to the first space (“oh yes” becomes “oh”); further on in the script, the string may contain spaces.

sentence *variable initialValue*

for any short string.

text *variable initialValue*

for any possibly long string (the variable name will not be shown in the form).

boolean *variable initialValue*

a check box will be shown; the value is 0 if off, 1 if on.

choice *variable initialValue*

a radio box will be shown; the value is 1 or higher. This is followed by a series of:

button *text*

a button in a radio box.

comment *text*

a line with any text.

Inside the script, strings are known as string variables, numbers as numeric variables. Consider the following form:

```
form Sink it
    sentence Name_of_the_ship Titanic
    real Distance_to_the_iceberg_(m) 500.0
    natural Number_of_people 1800
    natural Number_of_boats 10
endform
```

In the script following this form, the variables will be known as *name_of_the_ship$*, *distance_to_the_iceberg*, *number_of_people*, and *number_of_boats*.

The variable associated with a radio box will get a numeric as well as a string value:

```
form Fill attributes
    comment Choose any colour and texture for your paintings
    choice Colour: 5
        button Dark red
        button Sea green
        button Navy blue
        button Canary yellow
        button Black
        button White
    choice Texture: 1
        button Smooth
        button Rough
        button With holes
endform
```

```
echo You chose the colour 'colour$' and texture 'texture$'.
```

This shows two radio boxes. In the Colour box, the fifth button (Black) is the standard value here. If you click on "Navy blue" and then ***OK***, the variable *colour* will have the value "3", and the variable *colour$* will have the value "Navy blue". Note that the trailing colon is chopped off, and that the button and comment texts may contain spaces. So you can test the value of the Colour box in either of the following ways:

```
if colour = 4
```

or

```
if colour$ = "Canary yellow"
```

The field types **optionmenu** and **option** are completely analogous to **choice** and **button**, but use up much less space on the screen:

```
form Fill attributes
    comment Choose any colour and texture for your paintings
    optionmenu Colour: 5
        option Dark red
        option Sea green
        option Navy blue
        option Canary yellow
        option Black
        option White
    optionmenu Texture: 1
        option Smooth
        option Rough
        option With holes
endform
echo You chose the colour 'colour$' and texture 'texture$'.
```

You can combine two short fields into one by using *left* and *right*:

```
form Get duration
    natural left_Year_range 1940
    natural right_Year_range 1945
endform
duration = right_Year_range - left_Year_range
echo The duration is 'duration' years.
```

The interpreter will only show the single text "Year range", followed by two small text fields.

Calling a script from another script

Scripts can be nested: the file *doremi.praat* may contain the following:

```
execute playSine.praat 550 0.9
execute playSine.praat 615 0.9
execute playSine.praat 687 0.9
```

With the **execute** command, Praat will not display a form window, but simply execute the script with the two arguments that you supply on the same line (e.g. 550 and 0.9).

Arguments (except for the last) that contain spaces must be put between double quotes, and values for **choice** must be passed as strings:

```
execute "fill attributes.praat" "Navy blue" With holes
```

You can pass values for **boolean** either as "yes" and "no" or 1 and 0.

Links to this page

Scripting
Scripting 6. Communication outside the script
Scripting 6.6. Controlling the user
Scripting 6.9. Calling from the command line

Scripting 6.2. Writing to the Info window

With the **Info** button and several commands in the **Query** menus, you write to the **Info window** (if your program is run from the command line, the text goes to the console window or to *stdout* instead; see **§6.9).**

The following commands allow you to write to the Info window from a script only:

echo *text*

clears the Info window and writes some text to it:

```
echo Starting simulation...
```

clearinfo

clears the Info window.

print *text*

appends some text to the Info window, without clearing it and without going to a new line.

printtab

appends a *tab* character to the Info window. This allows you to create table files that can be read by some spreadsheet programs.

printline [*text*]

causes the following text in the Info window to begin at a new line. You can add text, just like with **print**.

The following script builds a table with statistics about a pitch contour:

```
clearinfo
printline  Minimum   Maximum
Create Sound from formula... sin Mono 0 0.1 44100 sin(2*pi*377*x)
To Pitch... 0.01 75 600
minimum = Get minimum... 0 0 Hertz Parabolic
print 'minimum'
printtab
maximum = Get maximum... Hertz
print 'maximum'
printline
```

You could combine the last four print statements into:

```
printline 'minimum''tab$''maximum'
```

or:

```
print 'minimum''tab$''maximum''newline$'
```

Links to this page

Scripting
Scripting 3.1. Hello world
Scripting 6. Communication outside the script

Scripting 6.3. Query commands

If you click the “Get mean...” command for a Pitch object, the Info window will contain a text like “150 Hz” as a result. In a script, you would rather have this result in a variable instead of in the Info window. The solution is simple:

```
mean = Get mean... 0 0 Hertz Parabolic
```

The numeric variable “mean” now contains the number 150. When assigning to a numeric variable, the interpreter converts the part of the text before the first space into a number.

You can also assign to string variables:

```
mean$ = Get mean... 0 0 Hertz Parabolic
```

The string variable “mean$” now contains the entire string “150 Hz”.

This works for every command that would otherwise write into the Info window.

Links to this page

Query
Scripting
Scripting 5.2. Formulas
Scripting 6. Communication outside the script

Scripting 6.4. Files

You can read from and write to text files from a Praat script.

Reading a file

You can check the availability of a file for reading with the function

```
fileReadable (fileName$)
```

which returns 1 (true) if the file exists and can be read, and 0 (false) otherwise. Note that *fileName$* is taken relatively to the directory where the script is saved; for instance, if your script is in the directory **Paolo/project1**, then the file name "hello.wav" refers to **Paolo/project1/hello.wav**, the file name "yesterday/hello.wav" refers to **Paolo/project1/yesterday/hello.wav**, and the file name "../project2/hello.wav" refers to **Paola/project2/hello.wav** (".." goes one directory up). You can also use full path names such as "C:/Documents and Settings/Paolo/project1/hello.wav" on Windows and "/Users/Paolo/project1/hello.wav" on the Mac.

To read the contents of an existing text file into a string variable, you use

```
text$ < fileName
```

where `text$` is any string variable and `fileName` is an unquoted string. If the file does not exist, the script terminates with an error message.

Example: reading a settings file

Suppose that the file **height.inf** may contain an appropriate value for a numeric variable called `height`, which we need to use in our script. We would like to read it with

```
height$ < height.inf
height = 'height$'
```

However, this script will fail if the file **height.inf** does not exist. To guard against this situation, we could check the existence of the file, and supply a default value in case the file does not exist:

```
fileName$ = "height.inf"
if fileReadable (fileName$)
    height$ < 'fileName$'
    height = 'height$'
else
    height = 180
endif
```

Writing a file

To write the contents of an existing string into a new text file, you use

```
text$ > fileName
```

where `text$` is any string variable and `fileName` is an unquoted string. If the file cannot be created, the script terminates with an error message.

To append the contents of an existing string at the end of an existing text file, you use

```
text$ >> fileName
```

If the file does not yet exist, it is created first.

You can create a directory with

```
createDirectory (directoryName$)
```

where, as with file names, *directoryName$* can be relative to the directory of the script (e.g. “data”, or “yesterday/data”, or “../project2/yesterday/data”) or an absolute path (e.g. “C:/Documents and Settings/Paolo/project1/yesterday/data” on Windows or “/Users/Paolo/project1/yesterday/data” on the Mac). If the directory already exists, this command does nothing.

You can delete an existing file with the function

```
deleteFile (fileName$)
```

or with the directive

```
filedelete fileName
```

If the file does not exist, these commands do nothing.

The simplest way to append text to a file is by using fileappend:

```
fileappend out.txt Hello world!
```

Example: writing a table of squares

Suppose that we want to create a file with the following text:

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
...
The square of 100 is 10000
```

We can do this by collecting each line in a variable:

```
deleteFile (“squares.txt”)
for i to 100
    square = i * i
```

```
    fileappend squares.txt The square of 'i' is 'square''newline$'
endfor
```

Note that we delete the file before appending to it, in order that we do not append to an already existing file.

If you put the name of the file into a variable, make sure to surround it with double quotes when using fileappend, since the file name may contain spaces and is not at the end of the line:

```
name$ = "C:/Documents and Settings/Paul Boersma/Desktop/squares.text"
filedelete 'name$'
for i to 100
    square = i * i
    fileappend "'name$'" The square of 'i' is 'square''newline$'
endfor
```

Finally, you can append the contents of the Info window to a file with

```
fappendinfo fileName
```

Directory listings

To get the names of the files if a certain type in a certain directory, use **Create Strings as file list...**.

Links to this page

Scripting
Scripting 6. Communication outside the script

Create Strings as file list...

A command in the **New menu** to create a **Strings** object containing a list of files in a given directory.

Settings

Name

the name of the resulting Strings object, usually “fileList”.

Path

the directory name, with an optional wildcard for selecting files.

Behaviour

The resulting Strings object will contain an alphabetical list of file names, without the preceding path through the directory structures. If there are not files that match *path*, the Strings object will contain no strings.

Usage

There are two ways to specify the path.

One way is to specify a directory name only. On Unix, you could type **/usr/people/miep/sounds** or **/usr/people/miep/sounds/**, for instance. On Windows, **C:\Document and Settings\Miep\Sounds** or **C:\Document and Settings\Miep\Sounds**. On Macintosh, **/Users/miep/Sounds** or **/Users/miep/Sounds/**. Any of these return a list of all the files in the specified directory.

The other way is to specify a wildcard (a single asterisk) for the file names. To get a list of all the files whose names start with “hal” and end in “.wav”, type **/usr/people/miep/sounds/hal*.wav**, **C:\Document and Settings\Miep\Sounds\hal*.wav**, or **/Users/miep/Sounds/hal*.wav**.

Script usage

In a script, you can use this command to cycle through the files in a directory. For instance, to read in all the sound files in a specified directory, you could use the following script:

```
directory$ = “/usr/people/miep/sounds”
Create Strings as file list... list ’directory$’/*.wav
numberOfFiles = Get number of strings
```

```
for ifile to numberOfFiles
   select Strings list
   fileName$ = Get string... ifile
   Read from file... 'directory$'/'fileName$'
endfor
```

If the script has been saved to a script file, you can use paths that are relative to the directory where you saved the script. Thus, with

```
Create Strings as file list... list *.wav
```

you get a list of all the .wav files that are in the same directory as the script that contains this line. And to get a list of all the .wav files in the directory Sounds that resides in the same directory as your script, you can do

```
Create Strings as file list... list Sounds/*.wav
```

As is usual in Praat scripting, the forward slash ("`/`") in this example can be used on all platforms, including Windows. This makes your script portable across platforms.

See also

To get a list of directories instead of files, use **Create Strings as directory list...**.

Links to this page

FAQ: Scripts
Scripting 6.4. Files
What's new?

Create Strings as directory list...

A command in the **New menu** to create a **Strings** object containing a list of directories in a given parent directory. It works completely analogously to **Create Strings as file list...**.

Links to this page

What's new?

Scripting 6.5. Calling system commands

From a Praat script you can call system commands. These are the same commands that you would normally type into a terminal window or into the Window command line prompt.

system *command*

executes a system command.

Some system commands are identical on all platforms (Macintosh, Windows, Unix):

```
system mkdir sounds
```

which creates a new directory **sounds** in the directory of the script. Some other system commands are different on different platforms. For instance, to throw away all WAV files in the script's directory, you would write

```
system del *.wav
```

on Windows, but

```
system rm *.wav
```

on Macintosh and Unix.

The script will stop running if a system command returns an error. For instance,

```
system mkdir sounds
```

will stop the script if the directory **sounds** already exists. In order to prevent this, you can tell Praat to ignore the return value of the system command:

system_nocheck *command*

executes a system command, ignoring any errors.

Thus, to make sure that the directory **sounds** exists, you would write

```
system_nocheck mkdir sounds
```

environment$ (*symbol-string***)**

returns the value of an environment variable, e.g.

```
homeDirectory$ = environment$ ("HOME")
```

stopwatch

returns the time that has elapsed since the previous **stopwatch**.

Here is a Praat script that measures how long it takes to do a million assignments:

```
stopwatch
for i to 1000000
    a = 1.23456789e123
endfor
```

```
time = stopwatch
echo 'a' 'time:3'
```

Links to this page

Scripting
Scripting 6. Communication outside the script

Scripting 6.6. Controlling the user

You can temporarily halt a Praat script:

pause *text*

suspends execution of the script, and allows the user to interrupt it. A message window will appear with the *text* and the buttons Stop and Continue:

```
pause The next file will be beerbeet.TextGrid
```

In the pause window you can include the same kinds of arguments as in a **form**. Here is an extensive example:

```
echo script
compression = 1
number_of_channels = 2
worth = 3
for i to 5
    beginPause (“Hi”)
        comment (“Type a lot of nonsense below.”)
        natural (“Number of people”, 10)
        real (“Worth”, worth+1)
        positive (“Sampling frequency (Hz)”, “44100.0 (= CD quality)”)
        word (“hi”, “hhh”)
        sentence (“lo”, “two words”)
        text (“ko”, “jkgkjhkj g gdfg dfg”)
        boolean (“You like it?”, 1)
        if worth < 6
            choice (“Compression”, compression)
                option (“lossless (FLAC)”)
                option (“MP3”)
                option (“Ogg”)
        endif
        optionMenu (“Number of channels”, number_of_channels)
            option (“mono”)
            option (“stereo”)
            option (“quadro”)
        comment (“Then click Stop or one of the continuation
              buttons.”)
    clicked = endPause (“Continue”, “Next”, “Proceed”, 2)
    printline ’number_of_people’ ’worth’ ’sampling_frequency’
          ’clicked’
    printline Compression: ’compression’ (’compression$’)
    printline Number of channels: ’number_of_channels$’
endfor
```

This example uses several tricks. A useful one is seen with *number_of_channels*: this is at the same time the value that is passed to **optionMenu** (and therefore determines the setting of the “Number of channels” menu when the window appears) and the name of the variable in which the user’s chosen value of “Number of channels” is stored (because “number_of_channels” is what you get by replacing the spaces in “Number of channels” with underscores and turning its first letter to lower case.

Your own pause windows are not likely to be as rich as the above example. For instance, the example has three continuation buttons (the second of these is the default button, i.e. the button that you can “click” by pressing the Enter or Return key). You will often use only one continuation button, for instance

```
endPause (“Continue”, 1)
```

or

```
endPause (“Finish”, 1)
```

or

```
endPause (“OK”, 1)
```

If your script shows multiple different pause windows, then it is in fact a *wizard*, and it becomes useful to have

```
endPause (“Next”, 1)
```

for most of them, and

```
endPause (“Finish”, 1)
```

for the last one.

The possibility of multiple continuation buttons can save the user a mouse click. The following script, for instance, requires two mouse clicks per sound:

```
for i to 20
    Read from file... sound'i'.wav
    Play
    Remove
    beginPause (“Rate the quality”)
        comment (“How good is the sound on a scale from 1 to 7?”)
        choice (“Quality”, 4)
            option (“1”)
            option (“2”)
            option (“3”)
            option (“4”)
            option (“5”)
            option (“6”)
            option (“7”)
    endPause (if i = 20 then “Finish” else “Next” fi, 1)
    printline 'quality'
endfor
```

The following script works faster:

```
for i to 20
    Read from file... sound'i'.wav
    Play
    Remove
    beginPause ("Rate the quality")
        comment ("How good is the sound on a scale from 1 to 7?")
    quality = endPause ("1", "2", "3", "4", "5", "6", "7", 0)
    printline 'quality'
endfor
```

In this example, the 0 at the end of **endPause** means that there is no default button.

File selection

If you want the user to choose a file name for reading (opening), do

```
fileName$ = chooseReadFile$ ("Open a table file")
if fileName$ <> ""
    table = Read Table from tab-separated file... 'fileName$'
endif
```

A file selector window will appear, with (in this example) "Open a table file" as the title. If the user clicks **OK**, the variable `fileName$` will contain the name of the file that the user selected; if the user clicks **Cancel**, the variable `fileName$` will contain the empty string ("").

If you want the user to choose a file name for writing (saving), do

```
select mySound
fileName$ = chooseWriteFile$ ("Save as a WAV file", "mySound.wav")
if fileName$ <> ""
    Save as WAV file... 'fileName$'
endif
```

A file selector window will appear, with (in this example) "Save as a WAV file" as the title and "mySound.wav" as the suggested file name (which the user can change). If the user clicks **OK**, the form will ask for confirmation if the file name that the user typed already exists. If the user clicks **OK** with a new file name, or clicks **OK** in the confirmation window, the variable `fileName$` will contain the file name that the user typed; if the user clicks **Cancel** at any point, the variable `fileName$` will contain the empty string ("").

If you want the user to choose a directory (folder) name, do

```
directoryName$ = chooseDirectory$ ("Choose a directory to save all
      the new files in")
if directoryName$ <> ""
    for i to numberOfSelectedSounds
        select sound [i]
```

```
        Save as WAV file... 'directoryName$'/sound'i'.wav
    endfor
endif
```

A directory selector window will appear, with (in this example) "Choose a directory to save all the new files in" as the title. If the user clicks **OK**, the variable `directoryName$` will contain the name of the directory that the user selected; if the user clicks **Cancel**, the variable `directoryName$` will contain the empty string ("").

A non-pausing pause window without a Stop button

Especially if you use the pause window within the **Demo window**, you may not want to give the user the capability of ending the script by hitting **Stop** or closing the pause window. In that case, you can add an extra argument to **endPause** that denotes the cancel button:

```
beginPause ("Learning settings")
    positive ("Learning rate", "0.01")
    choice ("Directions", 3)
        option ("Forward")
        option ("Backward")
        option ("Bidirectional")
clicked = endPause ("Cancel", "OK", 2, 1)
if clicked = 2
    learningRate = learning_rate
    includeForward = directions = 1 or directions = 3
    includeBackward = directions = 2 or directions = 3
endif
```

In this example, the default button is 2 (i.e. **OK**), and the cancel button is 1 (i.e. **Cancel**). The form will now contain no **Stop** button, and if the user closes the window, this will be the same as clicking **Cancel**, namely that `clicked` will be 1 (because the Cancel button is the first button) and the variables `learning_rate`, `directions` and `directions$` will not be changed (i.e. they might remain undefined).

Links to this page

Scripting
Scripting 6. Communication outside the script
What's new?

Scripting 6.7. Sending a message to another program

To send messages to running programs that use the Praat shell, use `sendpraat` (see **Scripting 8. Controlling Praat from another program**).

To send a message to another running program that listens to a socket, you can use the `sendsocket` directive. This works on Unix and Windows only.

Example

Suppose we are in the Praat-shell program **Praat**, which is a system for doing phonetics by computer. From this program, we can send a message to the *non*-Praat-shell program **MovieEdit**, which does know how to display a sound file:

```
Save as file... hallo.wav
sendsocket fonsg19.hum.uva.nl:6667 display hallo.wav
```

In this example, `fonsg19.hum.uva.nl` is the computer on which MovieEdit is running; you can specify any valid Internet address instead, as long as that computer allows you to send messages to it. If MovieEdit is running on the same computer as Praat, you can specify `localhost` instead of the full Internet address.

The number 6667 is the port number on which MovieEdit is listening. Other programs will use different port numbers.

Links to this page

Scripting
Scripting 6. Communication outside the script
What's new?

Scripting 6.8. Messages to the user

If the user makes a mistake (e.g. types conflicting settings into your form window), you can use the **exit** directive (**§5.8**) to stop the execution of the script with an error message:

```
form My analysis
    real Starting_time_(s) 0.0
    real Finishing_time_(s) 1.0
endform
if finishing_time <= starting_time
    exit The finishing time should exceed 'starting_time' seconds.
endif
# Proceed with the analysis...
```

For things that should not normally go wrong, you can use the **assert** directive:

```
power = Get power
assert power > 0
```

This is the same as:

```
if (power > 0) = undefined
    exit Assertion failed in line xx (undefined): power > 0
elsif not (power > 0)
    exit Assertion failed in line xx (false): power > 0
endif
```

You can prevent Praat from issuing warning messages:

```
nowarn Save as WAV file... hello.wav
```

This prevents warning messages about clipped samples, for instance.

You can also prevent Praat from showing a progress window:

```
noprogress To Pitch... 0 75 500
```

This prevents the progress window from popping up during lengthy operations. Use this only if you want to prevent the user from stopping the execution of the script.

Finally, you can make Praat ignore error messages:

```
nocheck Remove
```

This would cause the script to continue even if there is nothing to remove.

Links to this page

Scripting
Scripting 6. Communication outside the script

Scripting 6.9. Calling from the command line

Previous sections of this tutorial have shown you how to run a Praat script from the Script window. However, you can also call a Praat script from the command line (text console) instead. Information that would normally show up in the Info window, then goes to *stdout*, and error messages go to *stderr*. You cannot use commands like **View & Edit**.

Command lines on Unix and Macintosh

On Unix or MacOS X, you call Praat scripts from the command line like this:

```
> /people/mietta/praat doit.praat 50 hallo
```

or

```
> /Applications/Praat.app/Contents/MacOS/Praat doit.praat 50 hallo
```

This opens Praat, runs the script **doit.praat** with arguments “50” and “hallo”, and closes Praat.

You also have the possibility of running the program interactively from the command line:

```
> /people/mietta/praat -
```

You can then type in any of the fixed and dynamic commands, and commands that handle object selection, like **select**. This method also works in pipes:

```
> echo “Statistics...” | /people/mietta/praat -
```

Command lines on Windows

On Windows, you call Praat scripts from the command line like this:

```
e:\praatcon.exe e:\doit.praat 50 hallo
```

Note that you use the program **praatcon.exe** instead of **praat.exe**. The script will write to the console output in UTF-16 Little Endian encoding. If you want to use ISO Latin-1 encoding instead, or if you want to use praatcon’s output in a pipe or redirect it to a file, use **praatcon -a** instead.

How to get arguments into the script