Automatic Speech Data Processing with Praat[1]
Lecture Notes

Ingmar Steiner
steiner@coli.uni-sb.de

Winter Semester 2007/2008

[1] www.praat.org

Contents

List of Figures

List of Tables

Listings

Chapter 0

A Short Preview

This chapter will showcase a short test run in Praat, which demonstrates a few of the things yet to come by explaining a simple script and what it does. It requires nothing from the reader except an open mind, and a willingness to postpone full comprehension until later chapters, where everything will be explained from the ground up.

0.1 Automating Praat

We start Praat by executing the `praat` binary (or `praat.exe` under Windows), which brings up the Praat *Object Window*, as well as the *Picture Window*. Both are, for now, empty, and since we don't need the Picture Window yet, we can simply close it (it will open again as required).

To load a `wav` file, we use the command `Read from file...` from the *Read* menu, which opens the usual file selection dialog window. Once we select a file, that file is loaded into the *object list* (unless of course the file is of a type that Praat can't recognize, in which case we get an error message instead). For now, let's assume I want to load a file called `aufnahme_1.wav`.

0.2 The Script Editor

Let's do the same thing again, but by running a script.

Select the command `New Praat script...` from the *Praat* menu to open a fresh *Script Editor* Window. The Script Editor is nothing but a simple text editor, which we will use to develop our scripts. There is a *history mechanism* in Praat that keeps track of all commands issued and objects selected, which is accessible via the `Paste history` command in the *Edit* menu of the Script Editor. Using this command, we see that the two lines correspond exactly to what we just did, i.e. load a `wav` file and open the Script Editor. In fact, the commands now in the script are *precisely* what the commands in the menu of the Object Window are called, all the way to the ... at the end of the file opening command! (This signifies that the `Read from file...` command takes an *argument*, namely the path to the file it opened.)

We can run the script with the `Run` command from the *Run* menu, and *voilà!* it loads the Sound file again and opens another Script Editor (which we close

Figure 0.1: The Praat Object Window in Linux/KDE3, with a Sound loaded

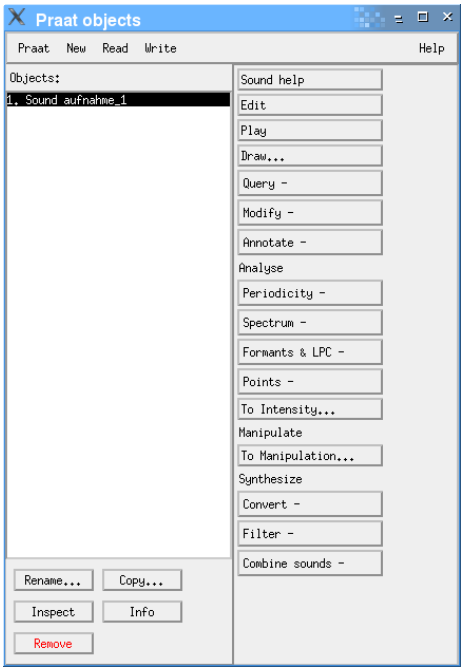
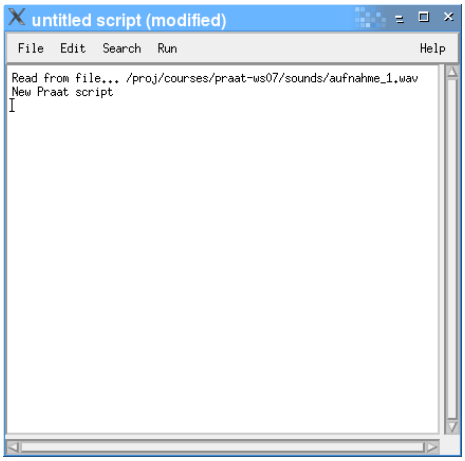


Figure 0.2: The Script Editor window



again, since we already have one).

Let's start a new script by using the `New` command from the *File* menu of the Script Editor (selecting "Discard & New" when prompted). Let's save this script as `testrun.praat` (using the `Save as...` command from the *File* menu), because that will allow us to use relative paths.

0.3 Batch open script

Checking the `sounds` directory, we have four `wav` files that we can load in this fashion. So let's open them all at once (as one "batch"), because having to click on `Read from file...` and selected one file multiple times is just plain annoying.

0.3.1 Repeating commands

We could write a script like this:

Listing 0.1: `batchOpen1.praat`

```
Read from file... sounds/aufnahme_1.wav
Read from file... sounds/aufnahme_2.wav
Read from file... sounds/aufnahme_3.wav
Read from file... sounds/aufnahme_4.wav
```

0.3.2 `for` loop

But that's not really elegant, because we're doing things repeatedly that differ only in a single number. So instead, we could do this:

Listing 0.2: `batchOpen2.praat`

```
for number from 1 to 4
  Read from file... sounds/aufnahme_'number'.wav
endfor
```

This involves a `for` *loop*, which takes a counter variable called `number`, sets it to the value given after the `from` (here, `1`), and does everything until the `endfor` line, at which point it adds `1` to the value of `number` and checks whether `number` is less than or equal to the number we supplied after the `to` (here, `4`). If yes, then it repeats everything between the `for` and `endfor` lines (increasing the value of `number` again), if not, the loop is finished (and the rest of the script is processed).

This means that the `Read from file...` command is actually run four times. As the argument to the command, we've used the variable `number` again, and by enclosing it in 'single quotes', we ensured that its value (first `1`, then `2`, and so on), rather than its name (`"number"`) is used, so that the argument to the `Read from file...` command (the filename) actually *changes* every time we go through the loop.

0.3.3 `Strings` file list

What if we have different names for the files? What if we want to open all `wav` files in a directory, regardless of their names?

Praat has a type of object called `Strings`, which is essentially a list of *strings*, each string being a list of characters (letters, numbers, etc.). There is a command called `Create Strings from file list...`, which looks at the contents of a directory and returns all files matching a given pattern as the strings of a `Strings` object. Once we have a `Strings` object, we can use commands like `Get number of strings` and `Get string...` to print information about the object (and its contents) to the *Info Window*.

Let's write a script like this:

Listing 0.3: `batchOpen3.praat`

```
Create Strings as file list... wavList sounds/*.wav
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings wavList
  filename$ = Get string... 'stringCounter'
  Read from file... sounds/'filename$'
endfor
```

First, we create a `Strings` object called *wavList* (we could just as well call it something else, though), which contains the names of all files in the `sounds` directory ending in `.wav`. Since we can't be sure how many there will be and have to tell the `for` loop how many times we want it to go around, we use the `Get number of strings` command from the *Query* menu of the `Strings` object's *dynamic menu* (to the right of the object list). The output of this query command is redirected into another variable, which we call `numberOfStrings` (again, this could be anything, but we want to use names that make sense).

Then comes the loop. Inside the loop, we'll skip over the first line for now, and look at the `Get string...` command (again, from the *Query* menu). This one takes an argument (remember? that's what the ... means), namely the index of the string we want to know. An argument of `1` returns the first string, `2`, the second, and, `'numberOfStrings'`, the last one (in this script, anyway). Since we want a different string each time the loop goes through, we use `'stringCounter'` as the argument (because `stringCounter` is our loop's counter variable). But again, we redirect this query command's output (the `stringCounter`th string, i.e. filename) into a variable, which we call, for the sake of transparency, `filename$`. The reason there is a `$` at the end of this variable's name is that it is a *string variable*, not a *numeric variable*, which is the type of output of the `Get string...` command. And finally, we use that string variable in the `Read from file...` command as before.

One pitfall we've avoided is that once the first `wav` file is loaded, the selection in the object list changes so that only that `Sound` object is selected. However, the next time the script goes through the loop, the `Get string...` command will cause an error, because that command only works when a `Strings` object is selected. This error can be avoided if we explicitly select the `Strings` object containing our file list in the loop, before we use the `Get string...` command. This is done with the `select` command, which takes either an object's numeric *ID* or, as here, its *class* and *name*. In this case, we know the class (`Strings`), and the name as well (`wavList`), because we just assigned it. Usually however, using the ID number is preferable.

0.3.4 Simple dialog windows

What if we want to use this script for a different directory and open other files there? Wouldn't it be nice to have a way for Praat to ask which directory it should look inside and open all files of a specific type out of?

Let's go through the following script:

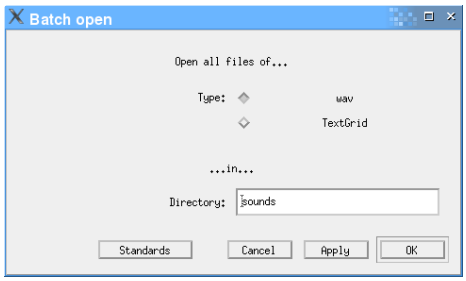
Listing 0.4: `batchOpen4.praat`

```
form Batch open
  comment Open all files of...
  choice Type: 1
    button wav
    button TextGrid
  comment ...in...
  sentence Directory sounds
endform

Create Strings as file list... 'type$'List 'directory$'/*.'type$'
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings 'type$'List
  filename$ = Get string... 'stringCounter'
  Read from file... 'directory$'/'filename$'
endfor
```

The first part of the script consists of something that looks like a `form` "loop", but it actually defines a dialog window that Praat will display when the script is run, which prompts the user for certain arguments to be used during the second part of the script.

See if you can figure out what the lines between the `form` and `endform` do:

Figure 0.3: Dialog window of `batchOpen4.praat`



The `choice` and `sentence` lines are the actual point of this `form`, since they provide variables whose values are filled in by the user. So once the user clicks the "OK" button in the dialog window, the script continues with two new variables, `type$` and `directory$`[1], which contain either `"wav"` or `"TextGrid"`, and `"sounds"` (or whatever the user entered into the text field), respectively. The details of the `form` loop will be explained later.

The second part of the script is basically the same as the script in the previous section, except that references to a "hard-coded" (i.e. fixed) directory `sounds/` have been replaced with `'directory$'`, which is whatever the user entered in the dialog window, and similarly for references to `wav` as the file type.

0.3.5 Good scripting practices

It's generally advisable to make a script as robust as possible, with portability and scalability in mind. This means that we should add a few things to that last script.

For instance, it is quite possible that the user will accidentally input a directory in the dialog window that does not exist or is not readable. In this case, the script will simply terminate with an error generated by Praat directly, which we couldn't do any better.

On the other hand, if the directory exists (and a `Strings` object is successfully created), but contains no files of the selected type, the `Strings` will be empty, and no files will be loaded. It would be nice for the user to receive some information about this, so we'll add a *condition* with `if...endif` and cause an error window of our own to pop up, using `exit`.

And finally, after the script is finished, we no longer need the `Strings` object, so we simply remove it. However, to be really sure we get the right object (in case there happens to be another object of the same class with the same name in the object list), we'll use the `Strings` object's numeric ID, which we get with the `selected()` function, select it (with `select` or `plus`), and use the `Remove` command, which is actually just a button in the Praat Object Window, below the object list.

Just to be explicit, we'll also finish the script by selecting all of the objects it loaded, so that the user knows immediately what happened. For this, we'll

[1] Actually, *three* new variables: the selection of `Type` is additionally stored in the numeric variable `type`, which contains the *number* of the selected `button`, in this case, `1` or `2`.

store all of the objects' IDs in an *array* as they are loaded. This is a tricky, but important part of Praat scripting, but it won't be explained in detail until later.

This is our new script (several *comments* have been inserted to explain the new parts, these are lines starting with a #):

Listing 0.5: batchOpen5.praat

```
form Batch open
  comment Open all files of...
  choice Type: 1
    button wav
    button TextGrid
  comment ...in...
  sentence Directory sounds
endform

Create Strings as file list... 'type$'List 'directory$'/*.'type$'
stringsID = selected ("Strings")
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings 'type$'List
  filename$ = Get string... 'stringCounter'
  Read from file... 'directory$'/'filename$'
  # populate array with object IDs
  file_'stringCounter'_ID = selected()
endfor

# cleanup Strings object
select stringsID
Remove

# check if Strings is empty
if numberOfStrings == 0
    exit No 'type$' files were found in directory 'directory$'!
endif

# select all files loaded by this script
select file_1_ID
for fileNumber from 2 to numberOfStrings
    plus file_'fileNumber'_ID
endfor
```

Chapter 1

Scripting Fundamentals

Before we begin, a note concerning reference: This introduction assumes no familiarity with programming in general or Praat scripting in particular. However, the reader is strongly encouraged to consult the Praat Manual for reference, which is available via the "Help" function within Praat, or online at http://www.fon.hum.uva.nl/praat/manual/Intro.html.

1.1 My first program

Traditionally, the first step in learning any programming language is to cause the words "Hello World!" to appear on the screen. We'll do this using Praat, because that's what this course is about. Since Praat can be considered a scripting language, we need two things for this example to work: the main Praat program (called `praat` under Linux or `praatcon.exe` under Windows) and a text file containing our instructions in a format that Praat can understand.

The text file is what we will refer to as our *script*, and can be created with any text editor. Using our favorite editor, let's create a script file called `helloWorld.praat`. (The `.praat` part at the end, sometimes referred to as the *file extension*, is not necessary and could just as well be something else, such as `.script`, `.psc`, `.txt`, or whatever. It's not important because the file is just a text file, and Praat will check its contents for well-formedness when we tell it to run the script. However, the `.praat` extension is the quasi-official standard.)

This script file should contain only the following line:

Listing 1.1: "Hello World!" in Praat

```
echo Hello World!
```

That's it!

Before we get into explanations, let's run the script (from the command line) and make sure it works:

```
$[1] praat helloWorld.praat
Hello World!
```

[1] I'll use `$` as a generic command prompt, because that's what it usually looks like under Linux. Under Windows, this could be `C:\>` or something similar.

Great! So what just happened? Well, we invoked the `praat` program and gave it the script as an *argument* by typing a space followed by the script filename. This caused Praat to open the script file, and starting from the top, carry out the instructions, line by line.

Our script consists of only a single instruction, which works much in the same way as what we did to run the script. There is one command, `echo`, followed by an argument. The `echo` command takes exactly one argument, so everything after the first space is treated as that argument (even if there is another space before the end of the line), and prints that argument to the output, which is just what we wanted.

To put things into perspective, other programming and scripting languages (the distinction is irrelevant here) can be much more complicated, as the following examples illustrate:[2]

Listing 1.2: "Hello World!" in C++

```
#include <iostream.h>
main()
{
  cout << "Hello World!" << endl;
  return 0;
}
```

Listing 1.3: "Hello World!" in Java

```
import java.io.*;
class HelloWorld{
  static public void main(String args[]){
    System.out.println("Hello World!");
  }
}
```

Listing 1.4: "Hello World!" in Scheme

```
(define helloworld
  (lambda ()
    (display "Hello World!")
    (newline)))
(helloworld)
```

Of course, none of this is relevant here, except to illustrate how simple by comparison the Praat scripting language is!

1.2 Scripting elements

Apart from the `echo` command, there are of course many other commands that we could write into a script file as instructions. However, each instruction must reside on its own line, since Praat will assume everything to the end of the line to belong to one instruction. We can, however, have spaces and/or tabs ("white-space") at the beginning of the line, before the instruction. This means we can make our script code more readable by indenting lines that belong together.

If a line becomes too long, we can break it into more than one line; if we begin each continuation line with a ..., Praat will treat them as a single instruction.

The following three (!) instructions are all well-formed:

[2] These examples are taken from The Hello World Collection and may not compile properly; they are given here only for illustrative purposes.

```
echo Hello World!
               echo Hello World!
echo This is output generated by a line so long that it was
... continued on a second line.
```

1.2.1 Comments

It is not only possible, but considered good form to explain what we are doing in a script by providing *comments*. This not only helps others who might want to understand our code, but also ourself, once we go back to a script we wrote a few weeks ago. Trust me on this...

Comments should be on their own line, and that line should start with a `#`, `;`, or `!` (perhaps after some whitespace). Some commands will also allow us to place a comment after the instruction on the same line, but others will produce unexpected results when we try this, so it's safest to place comments on their own lines. Essentially, everything after this comment symbol is ignored by Praat. This also allows us to quickly disable certain lines when we're developing a script, in case we don't need them at the moment, or we're trying to find the source of an error ("debugging").

```
# This line is a comment.
! So is this one.
; And this one as well.

# The last line was empty, and therefore ignored.
a = 1 + 2 ; we just did math, and this is another comment.

# The following does not work:
echo Hello World! ; this comment should not be printed, but will be!
```

1.3 Variables

Without variables, there could be (almost) no scripting.

A variable is a name by which Praat remembers the output of an instruction, with the purpose of reusing that output at a later time. Let's take a real-world example:

Let's assume that we want to run a pitch analysis, consisting of several steps, on some male voice data, and each of these steps depends on a certain predetermined value for pitch floor and ceiling. We could enter those floor and ceiling values by hand in each step, taking care to use the same values each time. While this would of course work perfectly well, let's imagine we want to run the same analysis on female voice data, where pitch floor and ceiling will be different. We would have to adjust those values in every single analysis step by hand, taking care not to forget to change any "male" values, or else our analysis would become invalid.

It would be far easier to define the floor and ceiling values once, and then use those values throughout the various analysis steps. This is exactly what variables are for.

So instead of using the following pseudo-script:

```
# male voice data
```

```
# pitch floor is 75 Hz
# pitch ceiling is 300 Hz

# analysis step 1, which involves the values 75 and 300
# analysis step 2, which involves the values 75 and 300
# analysis step 3, which involves the values 75 and 300
# analysis step 4, which involves the values 75 and 300

# female voice data

# pitch floor is 100 Hz
# pitch ceiling is 500 Hz

# analysis step 1, which involves the values 100 and 500
# analysis step 2, which involves the values 100 and 500
# analysis step 3, which involves the values 100 and 500
# analysis step 4, which involves the values 100 and 500
```

We could use the following, subtly different one:

```
# male voice data

pitch_floor = 75
pitch_ceiling = 300

# analysis step 1, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 2, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 3, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 4, involving 'pitch_floor' and 'pitch_ceiling'

# female voice data

pitch_floor = 100
pitch_ceiling = 500

# analysis step 1, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 2, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 3, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 4, involving 'pitch_floor' and 'pitch_ceiling'
```

Note how the lines with the actual analysis instructions (which would of course be more complex in a real script) are *exactly the same* for both speaker analyses. This may seem trivial at first, but implies all the power of scripting with variables.

Now, let's look more closely at what the lines that are not comments do. The instruction `pitch_floor = 75` tells Praat to create a variable with the name `pitch_floor` and assign to it a value that is equal to whatever is on the right side of the `=`, in this case, the number `75`. After this instruction has been carried out, we can at any time refer to this number, stored as `pitch_floor` by using the variable name `pitch_floor`. In fact, this is exactly what is done in the analysis steps (except that here, being comments, they don't do anything at all).

Once we get to the part where we look at the female voice data, we no longer need the pitch parameters of the male voice, so we *change* the values of the `pitch_floor` and `pitch_ceiling` variables. This is done simply by redefining them, which causes Praat to forget what their previous values (if any) were.

Before you wonder, once they have been created ("declared" or "instantiated"), variables remain available until the end of the script, even if their values change. There is no way to delete a variable or otherwise remove it from memory during a script's run time, but there should not be a need to, either. Conversely,

all variables created by a script are deleted when the script finishes, so that variables are *only* available during run time.

Now, let's write a short script that instead of chewing through pitch analyses, simply outputs the pitch parameters for the male and female voice data:

Listing 1.5: `outputPitchParameters.praat`

```
#male voice data

pitch_floor = 75
pitch_ceiling = 300

echo Male voice:
echo Pitch floor is 'pitch_floor' Hz
echo Pitch ceiling is 'pitch_ceiling' Hz

# female voice data

pitch_floor = 100
pitch_ceiling = 500

echo Female voice:
echo Pitch floor is 'pitch_floor' Hz
echo Pitch ceiling is 'pitch_ceiling' Hz
```

This script actually *does* something when run:

```
$ praat outputPitchParameters.praat
Male voice:
Pitch floor is 75 Hz
Pitch ceiling is 300 Hz
Female voice:
Pitch floor is 100 Hz
Pitch ceiling is 500 Hz
```

1.3.1 Variable names

There are simple but important rules to follow when choosing names for our variables, namely they must

- start with a lower-case letter;
- contain only letters (upper or lower-case), digits, and underscores;
- *not* contain spaces, dashes, punctuation marks, umlauts, or anything not in the previous point.
- One exception is the leading dot, explained in Section 1.7.2.

So `a`, `fooBar`, `number_1`, and `aEfStSgs3sWLKJW234` are all valid, legal variable names, while `Pitch`, `my-number`, `column[3]`, and `lösung` are not.

Furthermore, it is not entirely impossible to inadvertently choose a variable name that is the same as a function name or a predefined variable.[3] If this happens, Praat will usually give us an error. Don't worry too much about this

[3] A common example is the predefined variable `e`

for now, though; we will soon learn more about function names and predefined variables, so that we can avoid the few that there are.

Finally, a word of advice on naming variables: choose names that are semantically transparent and that we will not confuse with others in our scripts. While we may have to press a few more keys to type `numberOfSelectedSounds` than `ns`, we will certainly know what the variable stands for. Remember, *cryptic code is not prettier!*

1.3.2 Variable types

There are actually two different types of variables in Praat scripts: *numeric variables* and *string variables*. The first type is what we've seen already, but has an important restriction: numeric variables can only contain numbers. So, `4`, `-823764`, `0.03253`, and `6.0225e23` (6×10^{23}; Avogadro's number) are all possible values for a numeric variable, while `abc`, `All this belongs together`, `€ 78.56`,

```
Amplitude:
Minimum: -0.87652892 Pascal
Maximum: 0.83545512 Pascal
Mean: -8.5033717e-07 Pascal
Root-mean-square: 0.36832867 Pascal
```

, and everything else are not. They are *strings*.

Strings can be assigned to string variables. These work exactly like numeric variables, but their names have a `$` at the end. This means that the numeric variable `foo` is *not* the same as the string variable `foo$`, and both may occur side-by-side in the same script.

Whenever a string is to be used in a place where an (unevaluated) string variable is expected, the string must be enclosed in "double quotes", for example when declaring a string variable:

```
stringVariable$ = "the string contents"
```

One reason for the distinction between numeric and string variables will become apparent later, when we learn about operators. For now, let's leave it at this simple explanation: *numeric variables are variables we can do math with, and string variables aren't.*

Predefined variables

Incidentally, Praat provides a number of predefined variables, which will come in handy later on. For now, we should just have a quick look at Table 1.3.2.

Special characters in strings

To create a string containing special characters, such as tabs and line breaks, the apropriate predefined variables should be used. A double quote *within* a string must be doubled:

Listing 1.6: Double quotes in strings

```
quotedString$ = """string"""
echo quotedString$ = 'quotedString$'
```

```
$ praat doubleQuote.praat
quotedString$ = "string"
```

Table 1.1: Predefined variables

Name	*Value*
`pi`	`3.141592653589793`
`e`	`2.718281828459045`
`newline$`	"line break" character
`tab$`	"tab" character
`shellDirectory$`	the current working directory
`preferencesDirectory$`	the directory where Praat stores certain configuration files
`date$()`	current time and date (format example: `Mon Jun 24 17:11:21 2002`)
`environment$(`*key*`)`	*value* of environment variable *key*[a]

Note: `date$()` and `environment$()` are actually *functions*, cf. Section 1.4.

[a] This is specific to the operating system. In Linux, environment variables can be listed with the `env` command; in Windows, the corresponding button is found in the "System Properties".

1.4 Operators and functions

We've already seen one operator, the *assignment operator* `=` that takes whatever is to its right side and assigns it to the variable to its left. There are of course others, but they share the syntax to use them, which is,

```
OPERAND1 operator OPERAND2
```

On the other hand, there are also *functions*, which for scripting purposes do similar things as operators, but tend to involve parentheses. Functions use the following syntax (brackets denoting optionality),

```
function ( ARGUMENT1 [, ARGUMENT2 [, ARGUMENT3 [,...]]] )
```

As we can see, the function takes a number of arguments (the number and individual type of the arguments is specific to the function), separated by commas and enclosed in parentheses.

Spaces around operators, parentheses, and commas are almost always optional, but increase the legibility of script code.

There are quite a number of operators and functions available in Praat, but they are divided into those that work on numbers and numeric variables, and those that work on strings and string variables. The former are commonly used for mathematical operations while the latter are sometimes collectively referred to as "string handling".

1.4.1 Mathematics

A short selection of commonly used mathematical operators and functions, along with some examples, follows:

The full selection of mathematics operators and functions can be found in the Praat Manual, under "Formulas 2. Operators" and "Formulas 4. Mathematical functions", respectively.

Of course, all operators and functions can be nested, i.e. used as *arguments* of others. Parentheses can and should be used to modify the priority as intended. An example:

Table 1.2: Mathematical operators and functions (selection)

		Example	*Outcome*
`+`	addition	`1 + 2`	3
`-`	subtraction	`3 - 2`	1
`*`	multiplication	`2 * 3`	6
`/`	division	`6 / 3`	2
`^`	exponentiation	`2 ^ 3`	8
`div`	division, rounded down	`10 div 3`	3
`mod`	modulo (remainder of `div`)	`10 mod 3`	1
`abs()`	absolute value	`abs(-1)`	1
`sqrt()`	square root	`sqrt(9)`	3
`round()`	nearest integer	`round(0.5)`	1
`floor()`	next-lowest integer	`floor(1.9)`	1
`ceiling()`	next-highest integer	`ceiling(0.1)`	1
`sin()`	sine	`sin(pi)`	0
`cos()`	cosine	`cos(pi)`	-1

```
abs(5 - (1 / (cos(2 * pi) + sqrt(4))) ^ -2) ; outcome: 4
```

Just for fun, the above instruction is the same as $\left|5 - \left(\frac{1}{\cos 2\pi + \sqrt{4}}\right)^{-2}\right|$.

In some situations (such as when working with `while` loops, cf. Section 1.5.2) we will find it convenient to know that there is a shorthand to writing `a = a + n` (where `n` is a number), namely the *increment* operator, which does exactly the same thing, but is written as `a += n`.

Note that there is also a *decrement* operator, `-=`, as well as `*=` and `/=`, which work analogously.

1.4.2 String handling

A string is, in effect, a list of characters, and such a list can be queried and modified. An important concept is that of a *substring*, which is essentially a *part* of a string, or more formally, a contiguous sublist of the list of characters in a string. It sounds more complicated than it really is, as illustrated by these examples:

```
hello$ = "Hello World!"

# substring of hello$ containing the first 5 characters:
# "Hello"

# substring of hello$ containing the last 6 characters:
# "World!"

# substring of hello$ containing characters 3 through 7:
# "llo W"
```

There are a number of handy functions in Praat for doing things with strings, the first three of which do just what the last example implied. Functions with a `$` at the end of their name return a string, the others return a number. Note that the number of arguments, as well as their sequence and type (string or numeric), is important!

Table 1.3: String functions (selection)

	Returns
`left$(string$, length)`	first `length` characters of `string$`
`right$(string$, length)`	last `length` characters of `string$`
`mid$(string$, start, length)`	substring of `length` characters from `string$`, starting with the `start`th character
`index(string$, substring$)`	starting position ("index") of first occurrence of `substring$` in `string$` (`0` if not found)
`rindex(string$, substring$)`	starting position ("index") of last occurrence of `substring$` in `string$` (`0` if not found)
`startsWith(string$, substring$)`	`1` if `string$` starts with `substring$`, `0` otherwise
`endsWith(string$, substring$)`	`1` if `string$` ends with `substring$`, `0` otherwise
`replace$(string$, target$, replacement$, howOften)`	`string$` with the first `howOften` instances of `target$` replaced by `replacement$` (for unlimited replacement, set `howOften` to `0`)
`length(string$)`	number of characters in `string$`
`extractWord$(string$, pattern$)`	substring of `string$` starting *after* the first occurrence of `pattern$` and ending before the next space or `newline$` or at `string$`'s end (returns empty string if `pattern$` is not found in `string$`; empty string as `pattern$` returns the first word)
`extractLine$(string$, pattern$)`	as `extractWord$()`, but returns substring from `pattern$` to end of line or `string$`
`extractNumber(string$, pattern$)`	as `extractWord$()`, but returns number immediately following `pattern$` (returns `--undefined--` if no number after `pattern$` or if `pattern$` not found)

Listing 1.7: String function examples

```
helloWorld$ = "Hello World!"

# first 5 characters
hello$ = left$(helloWorld$, 5)
echo 'hello$'

# last 6 characters
world$ = right$(helloWorld$, 6)
echo 'world$'

# characters 3 through 7, i.e.
llo_W$ = mid$(helloWorld$, 3, 5)
echo 'llo_W$'

# starting position of first "l"
firstL = index(helloWorld$, "l")
echo 'firstL'

# starting position of last "l"
lastL = rindex(helloWorld$, "l")
echo 'lastL'
```

```
# does helloWorld$ start with "H"?
firstCharIsH = startsWith(helloWorld$, "H")
echo 'firstCharIsH'

# does helloWorld$ end with "d"?
lastCharIsD = endsWith(helloWorld$, "d")
echo 'lastCharIsD'

# replace first "Hello" with "Goodbye"
goodbyeWorld$ = replace$(helloWorld$, "Hello", "Goodbye", 1)
echo 'goodbyeWorld$'

# replace all "l"s with "w"s
hewwoWorwd$ = replace$(helloWorld$, "l", "w", 0)
echo 'hewwoWorwd$'

# length of helloWorld$
helloLength = length(helloWorld$)
echo 'helloLength'
```

```
$ praat simpleStringFunctions.praat
Hello
World!
llo W
3
10
1
0
Goodbye World!
Hewwo Worwd!
12
```

It is also quite simple to *concatenate* strings. This is accomplished using the + operator, which works differently with strings than numbers. Observe:

```
helloWorld$ = "Hello" + " " + "World!"

# outcome: "Hello World!"
```

Similarly, the - operator also works on strings, removing a substring from the end of a string ("truncating" the string), but *only if* the string indeed ends with the substring in question:

```
helloWorld$ = "Hello World!"

hello$ = helloWorld$ - "World"

# outcome: "Hello World!"
# why? because helloWorld$ doesn't end in "World", but in "World!"

hello$ = helloWorld$ - "World!"

# outcome: "Hello "
```

As with mathematical functions and operators, string functions can be nested. For instance, to get everything *except* the first 3 characters from a string, we could do this:

```
helloWorld$ = "Hello World!"
```

```
from3$ = right$(helloWorld$, length(helloWorld$) - 3)

# outcome: "lo World!"

# which is the same as

from3$ = mid$(helloWorld$, 4, length(helloWorld$) - 3)
```

1.4.3 Variable evaluation

The crucial part of working with variables is the ability to use either their names or their values. This means that in some situations, we will type the variable's *name*, but we want Praat to interpret it as if we had typed the variable's current *value*. This is called *evaluating* (or "substituting" or "expanding") the variable. In Praat, this is done by enclosing the variable's name in single quotes (as in `'myVariable'`). Figuring out when to evaluate a variable, and when to just use its name is one of the tricky parts of writing Praat scripts.

However, a few examples should shed light on this mystery. We've already used evaluation several times, in combination with the `echo` command. As we saw in our very first script, the `echo` command simply outputs whatever follows it on the same line.

```
echo This is a sentence.

# output: This is a sentence.
```

If we have a variable called `numberOfFiles` and assign it the number `4`, then output this variable using `echo`, we have to use variable evaluation. Observe:

```
numberOfFiles = 4
echo numberOfFiles

# output: numberOfFiles
# however:

echo 'numberOfFiles'

# output: 4

# or, more verbosely:

echo number of files: 'numberOfFiles'

# output: number of files: 4
```

As we've also seen, we can freely mix normal output text and evaluated variables, all as the argument to the `echo` command.

So what happens when a variable is evaluated that has not been instantiated yet? Observe:

```
echo 'noSuchVariable'

# output: 'noSuchVariable'
```

(This may happen to you fairly often as you learn how to write Praat scripts, and is usually caused by mis-typing variable names.)

As a rule of thumb, every variable in single quotes is evaluated before the line itself is interpreted by Praat.[4]

Evaluating string variables works the same way, except that we use the string variable's name (i.e. `echo 'myString$'`).

This raises an intriguing possibility.

Evaluating variables *within* strings

Since variables can be evaluated *anywhere* in a Praat script, we can use this to evaluate a variable *within a string*! This means that the following is possible:

```
a$ = "is"
b$ = "sentence"
c$ = "This 'a$' a 'b$'."

# outcome: "This is a sentence."

# by the way, this is the same as...

c$ = "This " + a$ + " a " + b$ + "."

# ...but slightly more intuitive!
```

In fact, this feature is the basis of Praat's mechanism for arrays (cf. Section 1.6).

Additionally, this is also how we can "convert" a numeric variable into a string, and vice versa:

```
a = 1
a$ = "'a'"

# outcome: "1"

a = 'a$'

# outcome: 1
```

Note that the conversion from string variable to number only works if the contents of `a$` can be interpreted as a number.

1.4.4 Comparison operators

Finally, there are a few comparison operators, which are used almost exclusively in condition statements (cf. Section 1.5.1), which return either "true" or "false". This is called a *truth value* (also referred to as a *Boolean* value). Praat has a healthy, inherently binary, notion of truth values in that "false" is always `0` and "true" is `1` (usually), or more generally, `not 0`.

As usual, these operators can be combined to allow complex conditions such as `(a == 2 and not b <= 10) or c$ != "foo"`. You are strongly encouraged to use parentheses to ensure proper grouping of multiple subconditions.[5]

[4] Cf. Paul Boersma's explanation in the Praat User List.

[5] In contrast to several other programming/scripting languages, Praat leniently treats `=` and `==` as equivalent in comparison expressions. However, to explicitly distinguish the *comparison* operator from the *assignment* operator, I will use the more widespread notation `==` in comparisons (and prefer `!=` over `<>` for good measure).

Table 1.4: Comparison operators

Returns `1` *iff*	
`x`	`x` is not `0`
`not x`	`x` is `0`
`x and y`	`x` and `y` are both not `0`
`x or y`	either `x` or `y` is not `0`
`x = y` (*or* `x == y`)	`x` and `y` are the same
`x <> y` (*or* `x != y`)	`x` and `y` are different (same as `not x = y`)
`x < y`	`x` is smaller than `y`
`x <= y`	`x` is smaller than or equal to `y`
`x > y`	`x` is greater than `y`
`x >= y`	`x` is greater than or equal to `y`
. . . and `0` *otherwise*	

} works for strings, too!

Note that the concepts "smaller" and "greater" are in fact applicable to strings as well as numbers, but refer to alphabetical ordering, i.e. `"a" < "b"` is true. In the same sense, upper-case letters are "smaller" than lower-case letters.[6]

1.5 Flow control

1.5.1 Conditions

Rather often in a script, there are instructions that should only be carried out if certain circumstances are met, and not if they aren't. This is what *conditions* (also referred to as "jumps") are for. Let's look at an example:

Listing 1.8: `if...endif`

```
condition = 0
echo 'condition'

if condition
  echo Condition has been met!
else
  echo Condition has not been met!
endif

condition = 1
echo 'condition'

if condition
  echo Condition has been met!
else
  echo Condition has not been met!
endif
```

```
$ praat ifThenElse.praat
0
Condition has not been met!
```

[6] This is because the values that are actually compared are the values of the *ASCII* codes of the letters. Look it up!

```
1
Condition has been met!
```

Notice how in the first `if...endif` block, only the first instruction was carried out, and in the second, only the second instruction. While the blocks themselves are identical, the value of `condition` changed, which caused the *condition* given after the `if` to evaluate to `0` in the first case, and `1` in the second.

In case we only want to do something if a certain condition is met, but nothing if it isn't, we can omit the `else` part.

On the other hand, if we want to differentiate between several cases if the first condition is not met, we can use the `elsif`[7] command, as in:

```
if not value
  echo Value is 0
elsif value < 0
  echo Value is negative
elsif value <= 10
  echo Value is greater than 0 but no greater than 10
else
  echo value must be greater than 10
endif
```

Only one of the echo commands will be carried out, depending on the value of `value`. Note that if more than one condition evaluates to true, only the first one will be applied.

1.5.2 Loops

The magic key to automating repetitive tasks are *loops*. Loops keep performing instructions until a *break condition* (also referred to as an "exit condition" or "terminating condition") is met. There are three different flavors of loops in Praat, `repeat...until`, `while...endwhile` and `for...endfor` loops. They all share a dangerous pitfall: if the break condition is never, ever met, the script will continue to run until the Praat *task* is ungracefully terminated by hand.[8] This is called an *infinite loop*, and Praat cannot help us detect one in advance. It's our responsibility to avoid these when using loops.

`repeat` loops

In a `repeat...until` loop (which we'll call a `repeat` loop for brevity's sake), all instructions between the `repeat` and `until` lines are carried out *repeatedly until* the break condition, supplied *after* the `until`, evaluates as true. This usually means that we need some sort of conditional variable, whose value is checked by the break condition.

Listing 1.9: `repeat` loop

```
counter = 10

echo Countdown:

repeat
  echo 'counter'...
```

[7] Instead of `elsif`, we can also write `elif`.

[8] In Windows, this is done with the *Task Manager*; in Linux, using e.g. the `kill` command.

```
  counter -= 1
until counter = 0

echo Blastoff!
```

```
$ praat repeatUntil.praat
Countdown:
10...
9...
8...
7...
6...
5...
4...
3...
2...
1...
Blastoff!
```

Note that even if the break condition is true from the start, the `repeat` loop is still performed at least once.

`while` loops

The `while` loop works similarly to the `repeat` loop, except that the break condition is defined at the beginning of the loop, right after the `while`. This means that if the break condition is false from the start, the `while` loop is not performed at all.

Listing 1.10: `while` loop

```
sentence$ = "This is a boring example sentence."
searchChar$ = "e"

echo The sentence...
echo "'sentence$'"

numberFound = 0

while index(sentence$, searchChar$)
  firstPosition = index(sentence$, searchChar$)
  numberFound += 1
  sentence$ = extractLine$(sentence$, searchChar$)
endwhile

echo ...contains 'numberFound' "'searchChar$'"s.
```

```
$ praat whileEndwhile.praat
The sentence...
"This is a boring example sentence."
...contains 5 "e"s.
```

If `searchChar$` is not in `sentence$` at all, `index()` returns `0` and the entire `while` loop will be skipped.

`for` loops

As we will soon come to see, the most common type of loop by far involves a *counter* variable (also referred to as an "iterator"), while the break condition is simply a value this iterator must not exceed.

This could easily be accomplished with a certain type of `while` loop:

Listing 1.11: `for` loop using `while`

```
iterator = 1
while iterator <= 5
  echo 'iterator'
  iterator += 1
endwhile
```

```
$ praat whileFor.praat
1
2
3
4
5
```

However, because it is so common, a streamlined syntax has been provided for this type of loop which implicitly initializes and iterates the counter:

Listing 1.12: `for` loop

```
for iterator from 1 to 5
  echo 'iterator'
endfor
```

```
$ praat forEndfor.praat
1
2
3
4
5
```

The `for` loop takes the counter variable, whose name is provided after the `for`, sets it to the value provided after the `from`, performs all instructions between the `for` and `endfor`, increases the value of the variable by `1`, and repeats, until the value becomes larger than the value provided after the `to`.

In fact, if we want to start from `1` (as is usually the case), we can streamline this syntax even further by omitting the `from 1`, which is then implicitly assumed. And just as with the `while` loop, if the break condition is true from the start (e.g. `for i from 2 to 1` or something similar), the loop won't be executed even once.

1.6 Arrays

Combining `for` loops with what we learned in Section 1.4.3, we have everything we need for another important concept in Praat scripting: *arrays*.

An array is essentially a group of variables that have names with numbers in them. These variables are usually created within a `for` loop, and later used

in another loop. The punchline, however, is that in creating and accessing the variables, the loops' iterators are used *within* the variable names!

So we might have several variables called `value_1`, `value_2`, `value_3`, and so on, and while this in itself is nothing new, it would allow us to do the following:

```
numberOfValues = 3
sumOfValues = 0
for i to numberOfValues
  sumOfValues += value_'i'
endfor
```

So what's going on? In the first iteration of the loop, `sumOfValues` is increased by the value of `value_1`, in the second iteration, by the value of `value_2`, and in the third and final iteration, by the value of `value_3`.

There are two important limitations here. The first is that we need some variable (such as `numberOfValues` in the example) to keep track of how many variables like `value_1` there are. We have to know this, because we need this number in the break condition of the `for` loop. If we were to try and access something like `value_4`, and that variable had not been previously set, we would probably get an error.

The second limitation is not as obvious, but becomes apparent if we try to *output* the respective value within the loop using e.g. `echo`. We have to evaluate the variable in the argument to the `echo` command, but we would have to *nest* one evaluated variable within another. However:

Listing 1.13: Evaluation nesting problem

```
value_1 = 1
value_2 = 2
value_3 = 3

for i to 3
  echo 'value_'i''
endfor

# let's make things interesting:

value_ = 99

for i to 3
  echo 'value_'i''
endfor
```

```
$ praat nestingProblem.praat
'value_1'
'value_2'
'value_3'
99i''
99i''
99i''
```

As we can see, none of this worked as we hoped. The only solution is to assign the desired variable to a "placeholder" variable, which we then output.

In fact, we can easily create and access "multidimensional" arrays by using loops within loops. Observe:

Listing 1.14: A table of products

```
# create the array

for x to 7
  for y to 7
    product_'x'_'y' = x * y
  endfor
endfor

# access the array to build the table

table$ = ""
for x to 7
  for y to 7
    # this is the placeholder:
    thisProduct = product_'x'_'y'
    table$ = "'table$''thisProduct''tab$'"
  endfor
  table$ = table$ + newline$
endfor

# output the table

echo 'table$'
```

```
$ praat tableOfProducts.praat
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

In this script, we additionally see that we can store a long string containing several lines in a single string variable, which is then output with a single `echo` command.[9]

Note that although the variables composing an "array" cannot be addressed as a single entity (unlike in many other programming languages), we will nevertheless uphold the custom of referring to such variables as *elements* of an array, although the array itself is just a handy concept and has no concrete manifestation in the Praat scripting language itself.

1.7 Procedures

Sometimes we will come across a portion of code in a script that occurs several times in the script. It would be desirable to only have to write this code *once* and then refer to it again as needed. This is where *procedures* come in.

A procedure is essentially a block of several instructions that are defined and named, and which can then be *called* whenever needed. A call to a procedure

[9] Alright, I'll concede that we could have computed and output the respective product in a single pass through a double loop, but I was trying to demonstrate array usage, and in the real world, single passes will not always be possible. Just bear with me here!

simply executes all lines of the procedure at the point where the call is made. Observe:

Listing 1.15: Procedures

```
# define an array of squares
for x to 10
  square_'x' = x ^ 2
endfor

# define a procedure to output this array
procedure output_array
  for x to 10
    square = square_'x'
    printline 'square'
  endfor
endproc

# call the procedure simply with
call output_array
```

No matter where in the script the procedure is defined, it can always be called, before or after the definition, which allows us to banish all tedious procedures to the end of the main script, clearing things up considerably. This way, we can "outsource" blocks of code that deal with one aspect of our script into individual procedures and have a very elegant "main" script:

Listing 1.16: More procedures

```
# begin main
call define_array
call output_array
# end main

procedure define_array
  for x to 10
    square_'x' = x ^ 2
  endfor
endproc

procedure output_array
  for x to 10
    square = square_'x'
    printline 'square'
  endfor
endproc
```

1.7.1 Arguments to procedures

Procedures can have *arguments* of their own. They are defined along with the procedure simply by adding them to the `procedure` line. These arguments act as variables in their own right, defined when the procedure is called.

When such call is made, these arguments must be passed to the procedure, and the number and type (number or string) of the arguments must match the procedure definition. Of course the arguments passed can also be variables, but we should realize that they are different from the variables used within the procedure!

Listing 1.17: Procedures with arguments

```
call define_array squares 10
call output_array squares 10

procedure define_array array_name$ array_size
  for x to array_size
    'array_name$'_'x' = x ^ 2
  endfor
endproc

procedure output_array array_name$ array_size
  for x to array_size
    square = 'array_name$'_'x'
    printline 'square'
  endfor
endproc
```

The way string arguments are passed to a procedure may be slightly confusing, especially when string *variables* are passed, but we should keep in mind that string arguments in a procedure *call* expect *strings*, not string *variables*. Hence:

Listing 1.18: Procedures with arguments passed from variables

```
name_of_array$ = "squares"
size_of_array = 10
call define_array 'name_of_array$' size_of_array
call output_array 'name_of_array$' size_of_array

procedure define_array array_name$ array_size
  for x to array_size
    'array_name$'_'x' = x ^ 2
  endfor
endproc

procedure output_array array_name$ array_size
  for x to array_size
    square = 'array_name$'_'x'
    printline 'square'
  endfor
endproc
```

This is equivalent to:

Listing 1.19: Procedures in "plain text"

```
name_of_array$ = "squares"
size_of_array = 10

# this mimics calling the first procedure
array_name$ = "'name_of_array$'"
array_size = size_of_array
for x to array_size
  'array_name$'_'x' = x ^ 2
endfor

# at this point, we have an "array" of 10 variables:
#  squares_1
#  squares_2
#  ...
#  squares_10

# this mimics calling the second procedure
```

```
array_name$ = "'name_of_array$'"
array_size = size_of_array
for x to array_size
  square = 'array_name$'_'x'
  printline 'square'
endfor
```

Quoting string arguments

So what happens if the string arguments contain spaces? Praat makes assumptions about spaces, which are potentially not what we had in mind. Observe:

Listing 1.20: Procedures with string arguments

```
procedure greet greeting$ name$
  printline 'greeting$',
  printline 'name$'!'newline$'
endproc

# This works, but only because the first string contains no space
call greet Hello Mr. President

# This no longer works
call greet Happy birthday Mr. President

# Now with too many double quotes
call greet "Happy birthday" "Mr. President"

# Finally, this works just as intended
call greet "Happy birthday" Mr. President

# now the same with variables, which doesn't work...
happyBirthday$ = "Happy birthday"
mrPresident$ = "Mr. President"
call greet happyBirthday$ mrPresident$

# ...because they must be evaluated
call greet 'happyBirthday$' 'mrPresident$'

# but as before, the first, and not the second, in quotes
call greet "'happyBirthday$'" 'mrPresident$'
```

```
$ praat procedures6.praat
Hello,
Mr. President!

Happy,
birthday Mr. President!

Happy birthday,
"Mr. President"!

Happy birthday,
Mr. President!

happyBirthday$,
mrPresident$!
```

```
Happy,
birthday Mr. President!

Happy birthday,
Mr. President!
```

Numeric arguments

Numeric arguments can be numbers or numeric variables. In both cases, they must not be quoted, otherwise they would be interpreted as strings, which would cause an error.

However, a numeric argument may also be complex expression, a so-called *formula*, containing variables, functions, and the like, so long as the formula returns a number. If the formula contains a space (although most formulas can be written without spaces), it must be quoted, unless it is the last argument, in which case it *must not* be quoted.

Listing 1.21: Passing numeric arguments

```
call printSum 1 1

call printSum pi e

one = 1
two = 2
call printSum one two

call printSum length("abc") -1

call printSum 1+1 2+2

call printSum "1 + 1" 2 + 2

procedure printSum number1 number2
  sum = number1 + number2
  printline 'number1' + 'number2' = 'sum'
endproc
```

```
$ praat numericArguments.praat
1 + 1 = 2
3.1415926535897 + 2.7182818284590 = 5.8598744820488
1 + 2 = 3
3 + -1 = 2
2 + 4 = 6
2 + 4 = 6
```

To summarize, arguments in a procedure call must be wrapped in double quotes if they contain spaces, except for the last argument, which must *not* be quoted. If non-final arguments don't contain spaces, quoting them is not mandatory; *however*, since it's not always foreseeable whether or not a string will contain spaces at run time, quoting non-final string arguments is generally a good idea.

1.7.2 Local variables

Normally, all variables declared within a procedure (starting with the procedure-"internal" variables in the procedure definition) are available in the script, as soon as the procedure has been called for the first time. This works just like with normal variables, and these normal variables are referred to as *global* variables.

Within procedures, however, it is possible to declare and use *local* variables, which means that they can be used only within the procedure. Outside the procedure itself, these variables are unavailable. In Praat, local variables have names that begin with a . (dot).[10]

Listing 1.22: Procedures with local variables

```
name_of_array$ = "squares"
size_of_array = 10
call define_array 'name_of_array$' size_of_array
call output_array 'name_of_array$' size_of_array

procedure define_array .array_name$ .array_size
  for .x to .array_size
    '.array_name$'_'.x' = .x ^ 2
  endfor
endproc

procedure output_array .array_name$ .array_size
  for .x to .array_size
    .square = '.array_name$'_'.x'
    printline '.square'
  endfor
endproc
```

Note that the reverse is also true: local variables declared in the "main" part of a script (i.e. outside of procedure definitions) are not accessible from within procedures. In fact, this entails that a local variable in the main script and a local variable *with the same name* within a procedure will not overwrite each other and could be used side-by-side, as shown here:

Listing 1.23: Mutually "invisible" local variables

```
.foo$ = "foo"
echo '.foo$'
call bar
echo '.foo$'

procedure bar
  .foo$ = "bar"
  echo '.foo$'
endproc
```

```
$ praat procedures8.praat
foo
bar
foo
```

[10] This is the only exception to the rule that variable names in Praat begin with a lower-case letter and consist only of letters, digits and underscores.

1.8 Arguments to scripts (part 1)

Just as procedures can receive arguments, the entire script itself can also take arguments, which are provided from the command line exactly as detailed in the preceding section for procedure calls. This is done with a `form` block.

`form` blocks work slightly differently from the rest of Praat script syntax.[11] The `form` itself must be followed by a space.[12] Between the `form` and `endform` lines, there are a series of argument (a.k.a. "parameter") declarations. Each consists of the type of the argument (`real` or `text`, for numbers or strings, respectively), a space and the name of variable the argument will have in the script. Since the type is defined by the first part of the declaration, the name of a string variable does not end in a `$`. Let's have an example:

Listing 1.24: Script arguments

```
form
  real howMany
  text greeting
  text name
endform

echo 'howMany' 'greeting$'s, 'name$'!
```

```
$ praat form.praat 100 "Happy birthday" Mr. President
100 Happy birthdays, Mr. President!
```

Quotes around string arguments are handled similarly, but not identically, because the arguments are first split according to the operating system's rules for command line arguments, and then passed to the Praat script. This should not create insurmountable problems, though; if in doubt, just try it out.

1.9 External scripts

Apart from using procedures, there are two other ways to re-use code in Praat scripts: *including* another script and *executing* it.

1.9.1 `include`

The `include` command takes as its only argument the name of another script file. This other script file is then "inserted" into the including script at run time, just as if all lines in the included file had been typed into the including script at the point where the `include` command was issued.[13] Of course, it is possible to include multiple scripts. Note that Praat will perform `include` commands before

[11] This is due to the fact that they seem to have been designed primarily as a means to create custom dialog windows in the graphical version of Praat. We will return to this in a later chapter.

[12] . . . followed by the dialog window's title, which is ignored in command line use.

[13] Praat's behavior in the regard goes as far as counting lines in the including script as if all lines of the included script were actually present in the including script. This means that if Praat gives an error message about something that happens in the including script *after* the `include` command, we will have to subtract the number of lines in the included script from the line number of the error to find the actual line number of the offending command in the including script. This behavior can be slightly reduced by placing `include` commands at the end of scripts.

anything else in the script, so we cannot use a variable to provide the filename of the included script.

Global variables in included scripts will count as global variables in the including script, so take care to check which variable names are used in scripts before you include them, or you might inadvertently overwrite variables in the including script...

The most effective way to use the `include` command is to use it with scripts that contain nothing but procedures, thereby *providing* these procedures to the including script without actually doing much at `include` time. Combining this approach with the use of local variables makes it rather safe concerning accidental variable overwriting.

1.9.2 `execute`

Another way to have one script run another is the `execute` command. In contrast to the `include` command, this simply runs the executed script from start to finish, then returns control to the executing script and continues with it. No variables are shared or overwritten.

If the executed script takes any arguments (using `form...endform`), these must be provided along with the `execute` command. Passing these arguments works syntactically exactly as passing them from a procedure call (cf. Section 1.7.1) or from the command line.

1.10 File operations

Praat provides a limited number of functions and commands to query, read and write files. But first, a word about paths.

1.10.1 Paths

If a Praat script is to access any file (even another script) that is not in the same directory as the script itself, we have to supply the *path* to the file, either as an absolute or relative path. The exact format of *absolute* paths depend on the operating system under which we're running the script. Table 1.5 gives a few examples. What these absolute paths have in common is that they are fixed; if we move our script to another directory and run it from there, files given with absolute paths will still be found.

Table 1.5: Examples of absolute paths

Windows	`"C:\Documents and Settings\John Doe\Desktop\praat"`
Linux *or* MacOS X	`/home/jdoe/Desktop/praat` *or* `~jdoe/Desktop/praat`
MacOS ≤9	`"My Disk:Desktop:praat"`

However, it is usually preferable to use *relative* paths. These take the script's directory as the base, and work from there. So if we have our script in a directory, along with a subdirectory called `"Sounds"` containing some sound files (e.g. `abc.wav`) which we want to access with our script, we would simply precede

references to these files with the name of the directory, followed by a forward slash `/` (e.g. `Sounds/abc.wav`).

The main advantage of using relative paths is *portability*. We can move the script and the relevant subdirectories to another location (directory or disk), and everything will work just as before. Also, since relative paths in Praat scripts always use forward slashes, scripts are even portable across different operating systems.[14]

1.10.2 File Input/Output

File input and output ("I/O") is extremely easy in Praat scripts. The only thing we need is a string variable and the relevant I/O operator, `<`, `>`, or `>>`.

Reading a file

To read the entire contents of a text file into a string variable, use the `<` operator.

Listing 1.25: A text file

```
This is a text file containing several "sentences",...

...an empty line, and some numbers, separated by tabs:
        123     456.67  89000
```

Listing 1.26: Praat script to read a text file

```
foo$ < foo.txt

# The following expression is now true:
foo$ == "This is a text file containing several ""sentences"",..."
... + "'newline$''newline$'...an empty line, and some numbers, "
... + "separated by tabs:'newline$''tab$'123'tab$'456.67'tab$'89000"
```

Writing a file

To write the contents of a string variable to a text file, use the `>` operator instead. Be careful; if the file already exists, its contents will be deleted first!

Appending to a file

Appending to a file uses the `>>` operator and works just like writing, with one exception: if the file already exists, the contents of the string variable is *added* at the end of the file.[15]

To append *any* text (not just string variables) to a file, we can use the `fileappend` command. This command is followed by the filename, and everything after that (to the end of the line) is treated as the string to be appended. This works similarly to the `echo` command. If the filename is stored in a string variable, that variable must be evaluated *and enclosed in double quotes*.

[14] Praat is able to use platform-specific path syntax, but this is strongly discouraged, since writing a script e.g. under Windows with backslashes instead of forward slashes will destroy the script's portability to a non-Windows system.

[15] In fact, `<`, `>`, and `>>` behave exactly as the respective standard input/output redirection operators in Windows/DOS and Linux (commonly referred to as STDIN and STDOUT).

```
greet$ = "Hello"
fileappend hello.txt 'greet$' World!
```

```
$ cat hello.txt[16]
Hello World!
```

1.10.3 Deleting files

A file can be deleted simply by using the `filedelete` command, followed by the name of the doomed file. If the file does not exist, the command has no effect. `filedelete` can be useful in combination with `fileappend`, in case we want to write more text than just the contents of a string variable to a file, but don't want that file's previous contents (if any) to survive.

1.10.4 Checking for file availability

Sometimes it is important to know whether a certain file exists. For instance, trying to read a file that isn't there will usually cause an error. In such cases, we can use the `fileReadable` function to have our script check for the file's existence first. The only argument to this function is the filename, and the function returns a boolean (i.e. `1` if the file can be read, `0` otherwise).[17] See Section 1.11.1 for an example.

1.11 Refined output

The `echo` command is not the only way to print text to the screen. There is also the `printline` command, which is essentially equivalent as long as we are using Praat scripts from the command line.

If we don't want to have the automatic line break at the end of an output command, we can use the `print` command. This allows us to print some text to the screen, then do something else, and print some more text *into the same line* as the last text we printed. Hence:

Listing 1.27: Printing with `print`

```
# the line
printline Hello world!

# creates the same output as
print Hello
print  world!
print 'newline$'
```

[16] `cat` is a Linux untility that can print the contents of files to the screen. The equivalent Windows/DOS command is `type`.

[17] As the function's name implies, `fileReadable` will also return `0` if the file exists, but we don't have permission to read it, which can occur e.g. on UNIX and modern Windows (NTFS) filesystems.

1.11.1 Controlled crash with `exit`

If we want to abort the script for any reason, we can issue the `exit` command. Any further text in the same line will be printed to the screen, in addition to Praat's standard error message. This allows us to terminate a script early on, before a more serious error can occur, which can be a good thing e.g. in case a script argument is not what we intended. It also allows us to inform the user about the reason for the `exit` command.

Listing 1.28: Catching an exception with `exit`

```
# filename argument received from command line
form
  text filename
endform

# no filename received?
if filename$ == ""
  exit no input file specified!
# filename reveived, but file not found?
elsif not fileReadable(filename$)
  exit input file "'filename$'" not found!
endif

# read file
filetext$ < 'filename$'

# just print file contents to screen
print 'filetext$'
```

```
$ praat exit.praat
Error: no input file specified!
Script "exit.praat" not completed.
Praat: command file "exit.praat" not completed.

$ praat exit.praat noFile
Error: input file "noFile" not found!
Script "exit.praat" not completed.
Praat: command file "exit.praat noFile" not completed.
```

If all we want to do is make sure the script does not continue unless a certain condition is met, we can use the much shorter command `assert`. This command is followed by a statement, and if that statement is false, Praat will terminate the script with a standard error message. Using `assert` is much quicker than checking for conditions explicitly and using `exit`, but the tradeoff is that we cannot change the format of the error message, which is not necessarily meaningful to a potential scripting-illiterate user:

Listing 1.29: Catching an exception with `assert`

```
form
  text filename
endform

assert filename$ <> ""
assert fileReadable(filename$)
```

```
filetext$ < 'filename$'

print 'filetext$'
```

```
$ praat assert.praat
Error: Script assertion fails in line 5 (false):
   filename$ <> ""
Script "assert.praat" not completed.
Praat: command file "assert.praat" not completed.

$ praat assert.praat noFile
Error: Script assertion fails in line 6 (false):
   fileReadable(filename$)
Script "assert.praat" not completed.
Praat: command file "assert.praat noFile" not completed.
```

1.12 Self-executing Praat scripts

It is possible to have scripts run by themselves without explicitly calling the `praat` command and passing the script as the first argument. Depending on the operating system, the procedure to set this up can vary.

Note that this is essentially a cosmetic feature and intended only for advanced users.

1.12.1 Linux

Under Linux and similar operating systems, we need two steps to make a script self-executing:

1. add a special line at the top of the script[18] containing the path to the `praat` program
2. make the script file executable by modifying its file permissions

Below is an executable version of `helloWorld.praat`:

Listing 1.30: "Hello World!" in Praat, executable

```
#!/path/to/praat
echo Hello World!
```

```
$ chmod -v u+x helloExe.praat
mode of `helloExe.praat' changed to 0700 (rwx------)
$ ./helloExe.praat
Hello World!
```

[18] This must indeed be the first line of the script and consist of a `#!`, followed by the absolute path to the `praat` binary. This works exactly as with bash, perl, python, and similar scripts.

1.12.2 Windows

In Windows, we can make Praat scripts self-executing by configuring the *file association* of "PRAAT Files" (i.e. files whose name ends with `.praat`, the "filename extension") so that they are automatically opened with the `praatcon.exe` program. The exact procedure depends on the version of Windows, as well as several other factors too Windows-specific to be listed here, but usually involves double-clicking a script file and taking it from there.

Note that while we should now be able to run a Praat script simply by double-clicking it, it will open a command prompt window to run the script and close this window again automatically (configuring Windows to keep the window open for review can be tricky.)

However, we *can* now simply enter the script filename on the command line, and Windows will automatically use `praatcon.exe` to run the script:

```
> helloWorld.praat
Hello World!
```

Note that Windows classifies files exclusively by filename extension, so if you use a different extension for Praat script files (such as `.psc` or `.script`), you will have to modify your file type settings accordingly.

1.13 System calls

The following is also relevant only to advanced console jockeys.

It is possible to have Praat make a *system call* to the operating system, executing a command that would normally only be usable on the command line. Since this depends entirely on the operating system under which the Praat script is being executed, the possibilities are far beyond the scope of this introduction. The command for making such system calls is `system`, the rest of the line being interpreted by the operating system. In case a system call could return an error, we can instead use the `system_nocheck` command to keep the Praat script from terminating at that point.

As an afterthought, there is also a way to make Linux-type environment variables available to a Praat script, by using the `environment$()` function, which takes a single string argument, the name of the environment variable, and returns its value. So under Linux, e.g. `environment$("PWD") == shellDirectory$`.

Chapter 2

Praat GUI

While we can theoretically accomplish a lot with command line use of Praat scripts, the full set of Praat features is available only through the Graphical User Interface ("GUI"). Praat is obviously much more than a script interpreter; its main focus lies in phonetic analysis, and for this, we need visualization and editing capabilities. In fact, there are hundreds of Praat commands that only make sense when we work with object selection, which is entirely hidden and non-interactive if we use Praat from the command line. The only way to discover these commands (and their arguments) is to work with Praat graphically, and even if a script is designed to be run from the command line, it is almost always developed graphically first.

We should keep in mind, though, that calling scripts from the command line is more efficient (i.e. faster) when processing large amounts of data or complex computations, and so such "batch processing" scripts should be designed with command-line use in mind.

2.1 Object Window

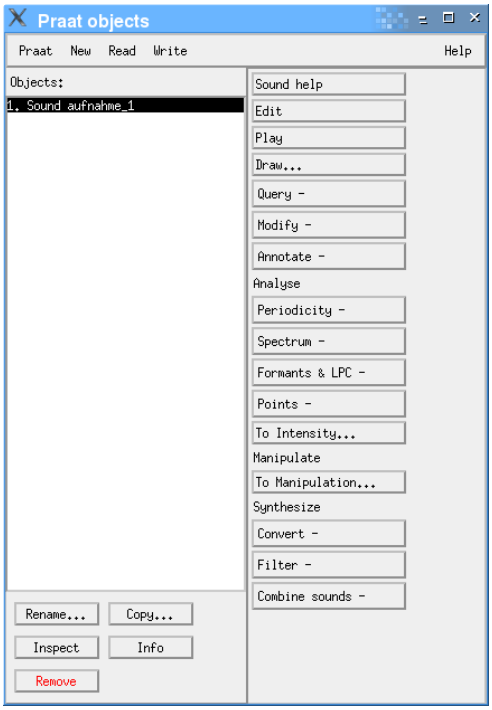
The graphical interface of Praat is started by executing the `praat` program with no argument. Under Windows, it is actually a different program, `praat.exe`, as opposed to the command line only version, `praatcon.exe`.

When Praat starts, we see *two* windows, the *Object Window* ("Praat objects") and the *Picture Window* ("Praat picture"). For now, we will ignore the Picture Window. In fact, we can close that window for now.

There are essentially four areas of the Object Window which demand explanation:

1. the *menu bar* at the top of the Object Window, consisting of the *Praat*, *New*, *Read*, and *Write* menus
2. the *object list*, entitled "Objects", is where objects can be added, selected, and removed
3. the *dynamic menu* to the right of the object list, containing a number of buttons and button menus; its contents changes according to type and number of objects selected in the object list (if none are selected, the dynamic menu will be empty)

Figure 2.1: Praat Object Window in Linux/KDE, with a Sound loaded



4. the area below the object list, which has no proper name, but always contains the buttons `Rename...`, `Info`, `Copy...`, `Remove`, and `Inspect`, which can be applied to all types of objects

2.1.1 Menu bar

The entries in the menu bar are all Praat commands, and mostly *static*. This means that (with the exception of the *Write* menu) they can be used regardless of the contents and state of the object list. Those that cannot be used at a given time will be visible, but disabled ("grayed out").

2.1.2 Objects

All objects in Praat appear in the object list until they are removed or Praat is closed. Each object entry consists of that object's *ID number*, *class*, and *name*. The class of the object can be `Sound` or `TextGrid` or something else. The name can consist of any sequence of letters, digits, and underscores.[1] It is possible, though potentially confusing, to have more than one object with the same name, even when the class is the same.

[1] Unlike scripting variables, object names *can* begin with an uppercase letter, digit, or underscore. As of version 4.6.32, Praat supports unicode characters in object names, but at the time of this writing, this should still be considered experimental.

For this reason, Praat uses unique ID numbers to keep track of objects. The first object placed in the list after Praat has been started gets the ID 1, the second, 2, and so on. If an object is removed, that object's ID is *not* recycled; Praat's internal counter assigning IDs is never reduced.

It is fairly obvious that objects can be renamed with the `Rename...` button and duplicated with the `Copy...` button. What is not so obvious is that the *order* of objects in the list can *never* be modified. This entails that every object will always have a higher ID than objects above it in the list, and a lower ID than objects following it.

Object selection

In the Object Window, objects are *selected* by clicking on them with the mouse. Any previous selection is *deselected*. We can also "drag" the mouse pointer over several objects to select them all. Alternatively, holding the Shift key while clicking an object will select that object, as well as all other objects between that object and the current selection, while holding the Ctrl key and clicking an object will add only the clicked object to the current selection. Holding these keys can of course be combined with dragging the mouse pointer.

All currently selected objects are collectively referred to as the current *selection*.

Removing objects from the object list is done with the `Remove` button, which removes all currently selected objects.

2.1.3 Dynamic menu

The contents of the dynamic menu depends entirely on the current selection. Selecting a single object will show all available commands for that class of object, but selecting multiple objects will usually decrease the number of available commands, in many cases down to none (depending on the objects' classes). Sometimes, however, certain commands will become available only if a specific combination of objects is selected. In Section **??**, we will see how this specification works when we learn how to manipulate the dynamic menu and add custom buttons. If no object is selected, the dynamic menu will also be empty.

2.2 Script Editor

By choosing the command `New Praat script` from the *Praat* menu, we can open a fresh *Script Editor* window. This is where scripts are developed and run in the graphical version of Praat.

The Script Editor is a simple text editor, lacking many of the fancy features present in full-fledged editors but containing a few features specific to Praat.

We can write a new script, save it, or load a previously saved script from a file (using the appropriate command from the *File* menu). The `Where am I?` and `Go to line...` commands in the *Search* menu return the number of the line the cursor is on, or send the cursor to the specified line, respectively.

2.2.1 Running scripts

To have Praat execute the script currently in the Script Editor, select the `Run` command from the *Run* menu. Additionally, we can also select only a portion of the script and use `Run selection` command to have Praat execute only the selected lines of the script, ignoring all others.[2]

2.2.2 Command history

A unique feature of the Script Editor is its access to Praat's internal *command history*. Praat records every click on an object, button or menu entry, and they can all be retrieved with the Script Editor's `Paste history` command, found in the *Edit* menu. Note that the entire command history will be inserted at the current cursor location and usually contains many more commands than we need, many of them selection commands. We can, however, use the `Clear history` at any time to erase all recorded commands and begin afresh.

The history mechanism can be quite useful and instructive to scripting beginners, because it outputs everything as a well-formed script which, if run, does *exactly* what the user did up to the point of the `Paste history` command. The drawback is that the power of such scripts is very limited. The history's contents is simply a batch of commands, one after the other, and makes no use whatsoever of variables, loops, or more advanced techniques. Therefore, a script "written" exclusively with the history mechanism will seldom enhance productivity compared to doing everything manually. On the other hand, if in doubt of the correct syntax for a command with many different arguments, the easiest solution is to use the command once and then note the command history's last entry.

2.3 Output

Since we can no longer receive output on the terminal ("standard out") in the Praat GUI, there are other analogous strategies, and even some new ones, to output information.

2.3.1 Info Window

A window that is initially not visible but that will appear when needed is the *Info Window* ("Praat: Info"). It looks just like another text editor window, and you can even type into it and delete text and so forth, but this window is where Praat directs almost all of its output. Whenever a command is used that returns output, that output will appear in the Info Window. Note that every time this happens, the previous contents of the Info Window will be deleted.

The contents of the Info Window can also be cleared by hand (usind the `Clear` command from the *File* menu), or saved as a text file, or copied, etc. The Info Window can also be closed; it will reappear as required.

In scripts, the Info Window can be cleared with the `clearinfo` command. The Info Window is also where the output of `echo`, `printline`, and `print` will be

[2]Note that any variables declared before the selection start will not be available, so this approach is of limited use.

displayed in the Info Window as well. This is also where the difference between `echo` and the two `print` commands is finally explained; the former will clear the Info Window before writing to it; the latter two will only append to it. This means that

```
echo
```

is equivalent to

```
clearinfo
printline
```

or

```
clearinfo
print 'newline$'
```

Beware of accidentally overwriting your script's output with multiple `echo` commands; this can become the cause of a lengthy and frustrating bug hunt! Conversely, if you use only `print` commands, you may end up not seeing your script's output as it becomes appended below the visible edge of the Info Window. We can avoid this with a single `clearinfo` at the beginning of the script.

The contents of the Info Window can also be appended to a text file with the `fappendinfo` command, which works similarly to the `fileappend` command (cf. Section 1.10.2).

2.3.2 Error messages

Not all output is written to the Info Window. The other way Praat can give us feedback is through *messages*. These appear as small pop-up windows and usually give us some sort of warning or error message. This is how Praat tells us about errors in a script, for instance. If we use the `exit` command (cf. Section 1.11.1) in a script, it will also generate such a message window.

Figure 2.2: Error message about faulty scripting command

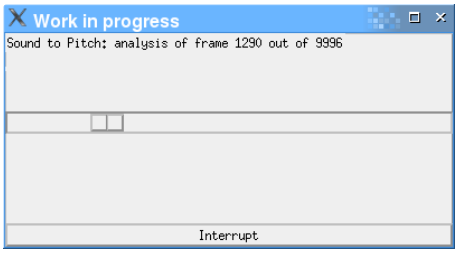
Figure 2.3: Error message about faulty Praat command

2.3.3 Other forms of output

Another way to give feedback to the user during a script is to use the `pause` command, which works similarly to `exit`, but simply displays our text, along with two buttons, "Continue" and "Stop". As expected, the former will let the script continue, the latter will abort. This raises interesting possibilities in script usability design but should not be overused. Note that this command is ignored in command-line use.

Some commands in Praat are expected to take relatively long to complete. For instance, creating a Pitch object from a Sound will take longer, the more samples must be processed. In such cases, Praat will show a *Progress Window* which allows some estimate of how long the command will take to complete. There is also an *Interrupt* button in the Progress Window, which allows us to abort the process (which is useful in case we e.g. want to modify some command parameters to decrease processing complexity).

Figure 2.4: Progress Window showing `To Pitch...` process



2.4 Objects in scripts

A Praat script can select objects and run available commands ("buttons") just as easily as if we used the mouse to do everything by hand, but very much faster! In fact, most scripts will perform such "actions" in the blink of an eye.

2.4.1 Object selection commands

To select an object with a script, we use the `select` command, which is equivalent to clicking on the object. Of course we have to supply an argument to the command specifying which object should be selected. This can either be the object's class *and* name (separated by a space), or its ID. So if we have a Sound object named `My_Recording` in the object list, we can select it in a script with the commands `select Sound My_Recording`. Of course, nothing prohibits another Sound with the same name from existing in the object list, and in cases of ambiguity, the *last* object will always be selected.

For this reason, it is generally preferable to use the `select` command with object *IDs* instead of names, in which case the object class is omitted. So if the `Sound` named `My_Recording` that we want to select has the ID `44` (being the 44^{th} object placed in the object list since program start), we can have the script select it simply with the command `select 44`.

To select more than one object at once, we must *add to* an existing selection, using the command `plus`, which otherwise works just like `select`. If the object happens to be already selected, `plus` does nothing. To *remove* an object from the *selection*, use the `minus` command. Again, if the specified object is not selected anyway, `minus` does nothing. Note that we can use `minus` to deselect the last object in the selection, thereby clearing the selection. Likewise, we can use `plus` even if no object is currently selected.

To simply select all objects in the object list at once, use the command `select all`.

2.4.2 Querying selected objects

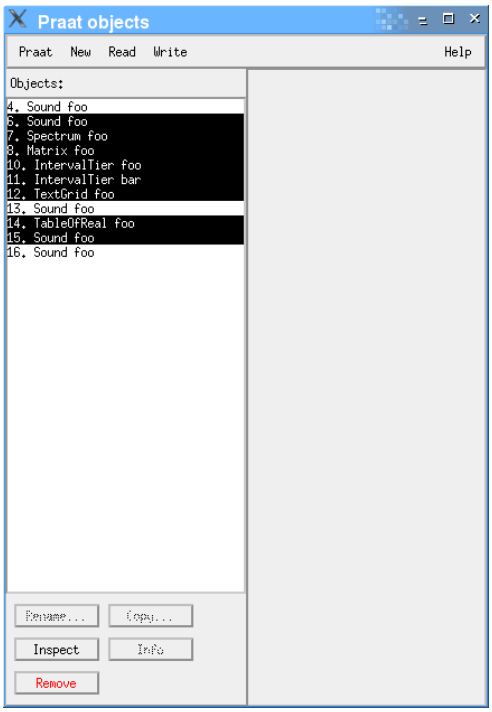
So how do we get the name or ID of a selected object for use in a script? We use one of two functions, `selected$()` or `selected()`. Notice how the first returns a string and the second, a number. These return values will be the selected object's class and name, or ID, respectively.

There's more to these functions, however. If the selection contains more than one object, we can pass either, or both, of two arguments. The first is the class of the object we're interested in (passed to the function as a string), in which case `selected$()` will return only the object's name, and the other is a number. This number n returns the name or ID of the n^{th} object in the selection, starting from the top.[3] If we want to count from the bottom, we simply specify a negative n argument.

To get the number of selected objects, use the function `numberOfSelected()`, and to get only the number of selected objects of a certain class (presumably from a selection also containing objects of other classes), provide this function with the desired class as a string argument.

Time for a few examples (which assume we have a selection corresponding to Figure 2.5):

Figure 2.5: Praat Object Window with various objects selected



[3] In fact, `selected()` is simply shorthand for `selected(1)`.

```
name$ = selected$()
# outcome: "Sound foo"

id = selected()
# outcome: 6

secondObject$ = selected$(2)
# outcome : "Spectrum foo"

secondID = selected(2)
# outcome: 7

secondSoundName$ = selected$("Sound", 2)
# outcome: "foo"

secondSoundID = selected("Sound", 2)
# outcome: 15

lastIntervalTierName$ = selected$("IntervalTier", -1)
# outcome: "bar"

thirdToLastObject$ = selected$(-3)
# outcome: "TextGrid foo"

firstIntervalTierID = selected("IntervalTier")
# outcome: 10

secondToLastIntervalTierID = selected("IntervalTier", -1)
# outcome: 11

seventhObjectClass$ = extractWord$(selected$(7), "")
# outcome: "TableOfReal"

numberOfSelectedObjects = numberOfSelected()
# outcome: 8

numberOfSelectedSounds = numberOfSelected("Sound")
# outcome: 2
```

Applying this to what we already know about arrays, we could easily store the IDs of all selected object in an array, to later recall the initial state of the selection:

Listing 2.1: Store IDs of selected objects in array

```
obj_num = numberOfSelected()
for o to obj_num
  obj_'o'ID = selected(o)
endfor
```

2.5 Praat command syntax

Notice how all menu commands and buttons in the various Praat windows begin with a capital letter or digit. This is the exact opposite of the scripting commands we have seen so far, which all begin with a lower-case letter. In general, the scripting commands are only available in scripts while the Praat commands beginning with a capital letter (or digit) can also be clicked on by hand when using Praat graphically and interactively.

2.5.1 Praat commands in scripts

We can use *all* of Praat's commands in scripts. However, we have to make sure that the command is available (i.e. visible and not grayed out) at the time when it is used in the script. Otherwise we will get an error message about the command's unavailability (cf. Section 2.3.2).

When we use such a command, we have to take special care to type it on its own line in the script, *exactly* as it appears on the button or in the menu. That means we have to pay extra special attention to capitalization, spaces, and other characters (such as parentheses, numbers, etc.). Otherwise, we'll get the error.

Arguments to Praat commands

There are many Praat commands that pop up *dialog windows*, asking for arguments of certain types. These commands invariably end in ... (three dots), which is Praat's indication that arguments must be supplied. When such a command is called in a script, the arguments must be given after the command, in the same line, separated by *single* spaces. This works similarly to arguments to procedures (cf. Section 1.7.1), with a few differences regarding double quotes and variable evaluation:

- Numeric arguments to Praat commands are *formulas* and may, but don't have to be, enclosed in double quotes, with some specialties concerning numeric variables:
  1. If a numeric argument contains a numeric variable, that variable may or may not be evaluated, *however*
  2. If a numeric argument consists only of a numeric variable (and no spaces), and the variable is not evaluated, then the argument may not be quoted (otherwise it would be interpreted as a string!)
- A string arguments to Praat commands may, but doesn't have to be, enclosed in double quotes, except if it contains a space, in which case it *must* be quoted.
- Any variables supplied as string arguments (or parts of string arguments) to Praat commands must always be evaluated.
- The last argument must *never* be quoted.

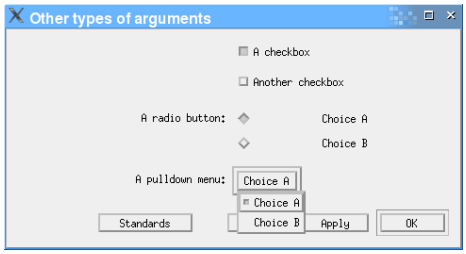
Some Praat commands may require other types of arguments, namely checkboxes, radio buttons, or pulldown menus:

A checkbox is essentially a boolean, either on or off, true or false, and hence, a checkbox argument can be supplied as either `1` or `0`.[4] However, we can also use `yes` and `no` instead, respectively.

Radio buttons and pulldown menus are essentially identical, except in appearance. Their arguments are strings and must be passed exactly as the respective buttons or menu entries are presented in the dialog.

[4]Note that the distinction is not just between `0` and `not 0` as with scripting booleans, but between `0` and `1`; any other numeric value is not allowed here.

Figure 2.6: Example of other argument types



Assuming there were a Praat command called `Other types of arguments...` and Figure 2.6 displayed its dialog and the accompanying arguments, the following example illustrates its syntax in a script:

```
# this works
Other types of arguments... 1 0 "Choice A" Choice B

# this works as well
Other types of arguments... yes no "Choice A" Choice B

# this would NOT work
Other types of arguments... 1 0 "Choice A" "Choice B"
# because there is no pulldown menu item ""Choice B""

# and neither would this
Other types of arguments... 1 0 Choice A Choice B
# because the radio button would receive the string argument
# "Choice" and the pulldown menu "A Choice B"
```

If you have trouble figuring out the correct scripting syntax for a command with complex arguments, remember the Command History (cf. Section 2.2.2)!

Redirecting output into variables

Every Praat command that outputs some form of information to the Info Window can have its output *redirected* and assigned to a variable. This variable will be a string variable, except if the output begins with a number. In this case, it can *instead* be assigned to a numeric variable, but everything after the number (usually a unit of measurement) will be truncated.

```
duration$ = Get total duration
# outcome: "5 seconds"

duration = Get total duration
# outcome: 5
```

Trying to assign non-numeric output to a numeric variable will result in an error.

Redirecting command output into variables is both essential to scripting and the only way to make Praat display the output of several commands at once. Otherwise all e.g *Query* commands will behave like `echo`, erasing any previous Info Window contents.

Suppressing warnings and progress dialogs

Sometimes Praat will display a warning or error message, or a progress window. Assuming we know what we are doing, we may find it undesirable to have this kind of output during execution of a script. If a command might output a warning message, we can prefix the command with the `nowarn` directive. To suppress an error message, use `nocheck`. And to suppress a progress window, use `noprogress`.

```
# hypothetical samples with high amplitude will be clipped when saved
nowarn Write to WAV file... mySoundWhichMightBeClipped.wav

# no progress window regardless of how long this will take
noprogress To Pitch... 0 75 600

# remove even if there is no object selected
nocheck Remove
```

`nocheck` can cause serious problems if used incorrectly. Do not use it unless you can be sure of what will happen, and that the error is something non-critical. Even then, there might be a better solution.

2.6 Editor scripting

The only Praat commands easily available to a script are those in the Object and Picture Windows. This means that initially, all commands in the various Editor Windows are unavailable. However, there is a way for a script to "enter" an Editor Window and use all commands available there. This is accomplished via an `editor` block.

Listing 2.2: Enter and use Sound Editor window

```
# make sure we have exactly one Sound selected
assert numberOfSelected() == 1
assert extractWord$(selected$(),"") == "Sound"

# remember the Sound's name...
soundName$ = selected$("Sound")

# open the Editor Window
Edit

# enter the Editor Window named for the Sound
editor Sound 'soundName$'

  #
  # do things in Editor Window
  #

  # close Editor Window
  Close
endeditor
```

The `editor` statement takes two arguments, the class and name of the object being edited (just like named `select`, etc.). These can be easily seen in the title bar of the Editor Window itself, but for a script to use these dynamically, we have to query the object as described in Section 2.4.2.

Note that while in the `editor` block, *only* the commands in the Editor Window are available for scripting; Praat commands in the Object and Picture Windows are not available again until after the `endeditor` statement.[5] Also note that editor scripting is not possible when running Praat scripts from the command line.

2.6.1 Editor scripts

If we want to write a script that uses the editor windows, we can also create and run a script from *within* the editor. Such a script, referred to as an *editor script*, is already in "editor mode" from the start and does not require the `editor` and `endeditor` statements. It can be created with the `New editor script` command in the Editor window's *File* menu, which opens up a Script Editor window tied to that specific Editor window (as visible in the Script Editor's title bar).

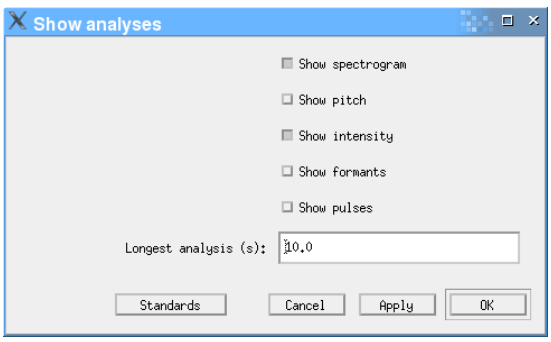
2.6.2 Sound Editors

Commands in Editor Windows that display a Sound's waveform (oscillogram) and (optionally) its spectrogram, intensity, pitch, formants, and glottal pulses can be difficult to use in scripts. This is due to the fact that only the visible analysis components are available to the commands, while the commands usually depend on the current position of the *cursor*. This means that three things play a role here:

Visibility of analysis

To ensure that a certain analysis is visible, we can use the `Show analyses...` command from the *View* menu with appropriate arguments.

Figure 2.7: `Show analyses...` dialog



Additionally, the "Longest analysis (s)" argument determines the maximum length of the analyzed part of the Sound. If the current window shows more than this, *none* of the analyses will be visible, and commands such as `Formant listing` will fail with an error message.

[5] Scripting commands, such as `printline`, `select`, and others beginning with a lower-case letter, *are* available even in "editor mode".

Zoom

To make sure we view an appropriate part ("window") of the Sound (and that the current view is not longer than the "Longest analysis (s)" (cf. previous Section), we can use the `Zoom...` command from the *View* menu, or commands like `Zoom to selection` (cf. next Section). `Zoom in` is probably not specific enough.

Cursor position and selections

We can also move the cursor to a specified position with the `Move cursor to...` command from the *Select* menu, or we can specify a selection with the `Select...` command. There are several relevant commands in the *Select* menu that are useful in this respect. What is important is that we can control the cursor and selection, which determines the output of other commands such as `View spectral slice` or `Get pitch`.

It is important to realize that almost all analyses and extraction commands of an Editor Window are also available as similar commands in the Object Window, usually in *Query* or *Modify* submenus in the dynamic menu. For scripting, it is actually easier and more precise to use the Object window's commands and avoid using the Editor Windows. Such scripts are also more robust and run faster!

2.6.3 Querying the Editors

The "state" of an Editor Window, i.e. the size of the visible window, the cursor position/selection, as well as many configuration details, such as which analyses are visible, can be retrieved with the `Editor info` command. If the string it returns is parsed accordingly, checking specific relevant settings in a script is very easy.

For example, the following editor script measures the pitch in a SoundEditor, making sure that no error occurs if the pitch contour is not visible:

Listing 2.3: Query editor and draw pitch

```
# get editor info
editorInfo$ = Editor info

# make sure pitch is shown
showPitch$ = extractWord$(editorInfo$, "Pitch show: ")
if showPitch$ == "no"
  Show pitch
endif

# measure pitch
pitch = Get pitch

# get selection times
selectionStart = extractNumber(editorInfo$, "Selection start: ")
selectionEnd = extractNumber(editorInfo$, "Selection end: ")

# output depending on whether or not mean was computed:
if selectionStart == selectionEnd
  echo Pitch at cursor ('selectionStart'):'newline$''pitch:5' Hz
else
  echo Mean pitch in interval ('selectionStart' - 'selectionEnd'):
  ...'newline$''pitch:5' Hz
endif
```

```
# restore previous setting
if showPitch$ == "no"
  Show pitch
endif
```

2.7 Picture Window

The *Picture Window* is one of the powerful, but commonly underestimated features of Praat. It allows us to produce graphics and illustrations (usually, but not necessarily, based on Objects), which can be helpful for data analysis, and additionally be exported as vector-based image files for insertion into research papers and reports.

2.7.1 Picture Window basics

The Picture Window is essentially an (initially) empty canvas measuring 12×12 inches (16 yellow squares, each 3 inches on a side, as indicated by the *rulers* at the canvas edges). By default, only the left half and top three quarters of this canvas are visible.

In addition, there is a pink *selection* rectangle, which can be set by dragging the mouse. Note that it is not possible to modify this selection by dragging its edges, so the selection behaves much like a selection in a Sound Editor, albeit in two dimensions.

The selection actually consists of two rectangles, the *outer viewport* and the *inner viewport*. It is the area *between* these two viewports that is filled in pink.[6] The inner viewport is where most of the graphics should be created, while the outer viewport serves as an outer guideline for axis labels, titles and things of the sort. The behavior of the mouse with regards to viewport creation, as well as the obligatory precise commands `Select inner viewport...` and `Select outer viewport...` are found in the *Select* menu.

Before we continue, let's have an example of how the viewport determines what will be drawn in the Picture Window. With the default viewport (6×4 inches), the script

Listing 2.4: Create 1kHz sine and draw its spectrum

```
Create Sound from formula... sine_1kHz Mono 0 1 22050
... 1/2 * sin (2 * pi * 1000 * x )
To Spectrum... no
Draw... 0 0 0 0 yes
```

results in the Picture Window contents shown in Figure 2.10:

The `Draw...` command available for `Spectrum` objects has a number of parameters (cf. Figure 2.8) that determine which portion of the spectrum will be drawn, as well as the scale. The "Garnish" option adds the frame along the inner viewport edge (the "inner box"), as well as the axis labels. Notice how these labels are drawn into the area between the inner and outer viewports.

There are many Drawing and Painting commands available for various object classes, and most of them should be sufficient for normal use. Remember that

[6] The difference in size between the inner and outer viewports depends on the currently selected *font size*, see below.

Figure 2.8: Spectrum: `Draw...` dialog

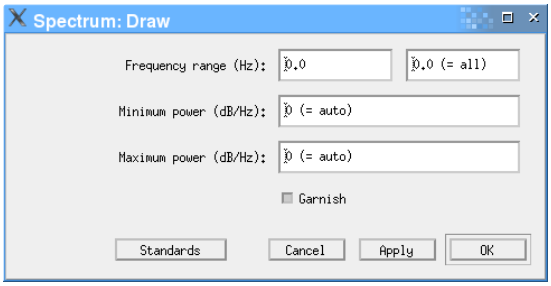
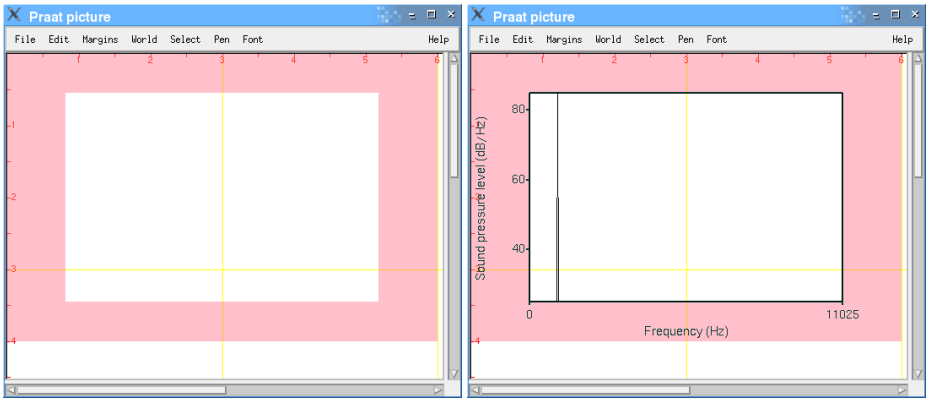


Figure 2.9: Empty Picture Window

Figure 2.10: Result of Listing 2.4



the non-interactivity of the Picture Window is by design; these graphics are not meant to rival the editors, but to allow exporting analysis data in a format perfect for visual analysis and professional publishing.

Importing external graphics

Before you get your hopes up, there is *no way* to get any type of external graphic or image into the Picture Window. The only ways to insert anything there is to use Drawing commands (either by hand or in a script, obviously).

However, it is possible to write the complete current contents of the Picture Window to a file using the Praat Picture file format (`prapic` is the default extension, though we can use what we want). With the `Write to praat picture file...` command, we can create a binary file, which can subsequently be imported into the Picture Window using the `Read from praat picture file...` command. Any contents in the Picture window at the time of import remains beneath. This can also be used to exchange graphics between Praat on two different platforms.[7]

[7]So if we really needed to save an EMF from Praat under Linux, we could use this feature to create a `prapic` file, then read it back into Praat under Windows and write it to an EMF file...

Don't use screenshots!

If you ever want to export anything visual from Praat to be included in a research paper or other publication, *do not use screenshots* of an editor window or anything of the sort. Doing so will create a pixel-based image with a resolution no higher than that of the screen from which it was captured. Print resolution will almost always be *much* higher, so the image will become blocky or blurry, depending on how it was processed, but will never look good.

Also, pixel-based images tend to consume rather large amounts of memory (each pixel is stored individually), unless compression is used. One of the most common types of image compression is JPEG, which, when configured improperly, will introduce artifacts along high-contrast edges. Text processors such as Microsoft Word tend to make the worst of such images when it comes to printing.

Additionally, window decorations (borders, etc.) distract from the analysis you're trying to show with your image, and if you want your readers to know that you used Praat, you should state it in the text. Demonstrating that you were running Praat under e.g. Windows XP with the "Energy Blue" theme is not desirable, and the names of files or objects you analyzed are details that usually irrelevant.[8]

The solution to these issues is to export the contents of the Picture Window to a file that recreates it using *vector graphics*. One such format is *Encapsulated PostScript* (EPS), used with the `Write to EPS file...` and its variants, which can be easily converted to any other vector-based format using appropriate software. Another is Microsoft's *Enhanced Metafile* (EMF) format, which is well suited for insertion into Microsoft Office documents. However, the required commands, `Write to Windows metafile...` or `Copy to clipboard`, are available only in the Windows version of Praat.

Vector images can be scaled arbitrarily without deteriorating edges or introducing artifacts, because their components are essentially *continuous functions*, which are sampled and redisplayed optimally whenever they are rendered.[9] Since these components in most cases take up very little memory, most vector images are also very efficiently stored.

In fact, the contents of the Picture Window displayed in Figure 2.10 could be exported as an EPS file and inserted into a LaTeX document such as this one directly:

```
\begin{figure}
  \includegraphics{spectrum1kHz}
\end{figure}
```

Code like that was used to insert Figure 2.11.

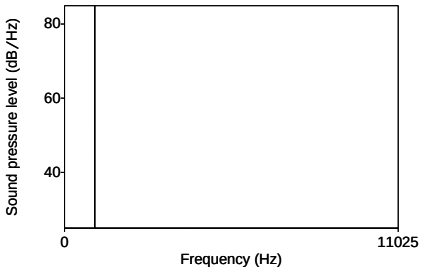
2.7.2 Custom drawing commands

Besides exporting graphics to files for insertion into documents, we can of course draw arbitrary graphics into the Picture Window. There are a number of commands at our disposal, and scripting makes them efficient to use.

[8] I realize that I'm ranting against everything I've done myself in this document, but I'm trying to focus on the interaction with Praat itself, not the data!

[9] Incidentally, this rendering is conceptually quite similar to the digitization of audio signals!

Figure 2.11: Result of Listing 2.4, but exported as EPS



Preliminaries

Similar to the Info Window, drawing commands will not clear the Picture Window, so to start with a blank canvas, we can issue the `Erase all` command in the *Edit* menu.

We can try out various drawing commands by hand, and whenever we make a mistake, we can use the `Undo` command (also in the *Edit* menu), which can come in handy.

Most commands that draw lines or shapes are modified by the current settings in the *Pen* menu. These include the line *type* (solid, dotted or dashed) and *width*, controlled with the commands `Solid line`, `Dotted line`, `Dashed line`, and `Line width...`, respectively.

Likewise, Text printed to the Picture Window can be controlled with respect to font *size* (`Font size...`) and *family*: `Times`, `Helvetica`, `New Century Schoolbook`, `Palatino`, and `Courier`, all in the *Font* menu. Several common font sizes can also be specified directly, with the commands `10`, `12`, `14`, `18`, and `24` (which may look strange in a script, on a line all by themselves, but are nevertheless valid Praat commands).

Furthermore, lines, shapes, and text can be colored with the following palette:

Axes and scale

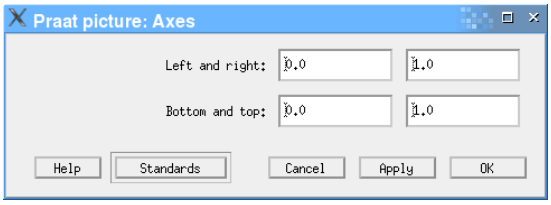
While the rulers along the edges of the Picture Window aid in selecting the viewport's proportions, they have nothing to do with the actual coordinates used to draw objects in the Picture Window. The coordinate system is defined using the command `Axes...` (found both in the *Margins* and *World* menus). This can be arbitrary, and redefined as desired; in fact, the left margin does not necessarily have to be smaller than the right margin, and likewise for top and bottom.

The `Axes...` command takes four numeric arguments, the left, right, bottom, and top values for the coordinate system enclosed by the *inner viewport*. This means that after clicking *OK* in the dialog shown in cf. Figure 2.12, the lower left-hand corner of the inner viewport is the point of origin of a coordinate system spanning to the upper right-hand corner of the inner viewport, which has the position $(1, 1)$.

Table 2.1: Color commands and their colors

Command	*Color (Linux)*	*Color (Windows)*
`Black`		
`White`		
`Red`		
`Green`		
`Blue`		
`Yellow`		
`Cyan`		
`Magenta`		
`Maroon`		
`Lime`		
`Navy`		
`Teal`		
`Purple`		
`Olive`		
`Silver`		
`Grey`		

Figure 2.12: `Axes...` dialog



This can easily be illustrated by executing the following commands, which results in Figure 2.13:[10]

```
Marks left every... 1 0.1 yes yes yes
Marks bottom every... 1 0.1 yes yes yes
Draw inner box
```

Now, a few simple drawing commands could be to paint a blue circle with a diameter of 0.2 right into the center of the viewport, then print the text "Earth" in 18pt Courier in the lower right-hand corner and draw an arrow from the text to the circle:

```
Paint circle... Blue 0.5 0.5 0.1
Font size... 18
Courier
Text... 0.25 Centre 0.25 Half Earth
Draw arrow... 0.3 0.3 0.4 0.4
```

This enriches the Picture Window to look like this:

[10]It would be tedious to explain every drawing command's arguments from here on, so please refer to the Praat program to see what the arguments mean.

Figure 2.13: Coordinate system from $(0, 0)$ to $(1, 1)$

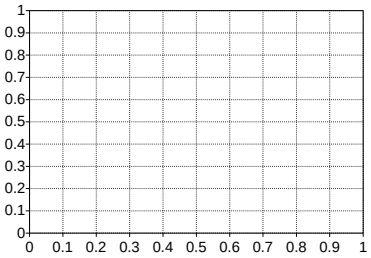
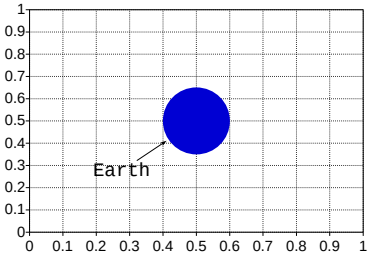


Figure 2.14: A few things drawn in



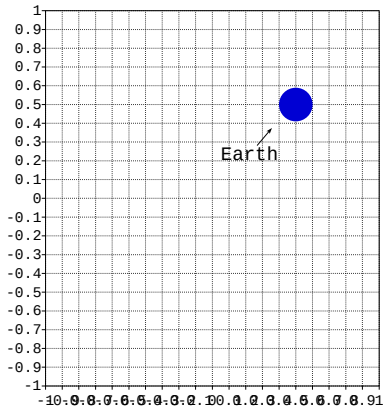
We could just as well select the viewport to have a different aspect ratio and redefine the axes, which results in Figure 2.15:

```
Select outer viewport... 0 6 0 6
Axes... -1 1 -1 1

Marks bottom every... 1 0.1 yes yes yes
Marks left every... 1 0.1 yes yes yes
Draw inner box

Paint circle... Blue 0.5 0.5 0.1
Font size... 18
Courier
Text... 0.25 Centre 0.25 Half Earth
Draw arrow... 0.3 0.3 0.4 0.4
```

The point of being able to define and redefine the axes at will is that various datasets can be drawn without having to first scale the values to some fixed coordinate system.

Note that even though the axes are defined with reference to the inner viewport, things can still be drawn outside of the inner viewport, but tend to look messy. However, the *bounding box* will be set to the *outer* viewport when saving to an EPS or EMF file, which means that most programs will clip everything outside of the outer viewport when rendering the resulting file. Even then, the canvas size will restrict what can be drawn. While it is possible to select the

Figure 2.15: Same as Figure 2.14, but with a different scale



viewport off-canvas, those portions will be lost on export.

So now we know everything we need to put the Picture Window to good use!

2.7.3 Data analysis with the Picture Window

Where Praat's graphical analysis commands don't offer what we want, we can easily script our own.

As an example, we will have Praat draw a histogram with the duration of each interval on the first tier ("Word") of `festintro.TextGrid` (cf. Section **??**).

Listing 2.5: Duration histogram of `festintro.TextGrid`

```
# open TextGrid file (modify as appropriate)
Read from file... festintro.TextGrid

# read interval durations into array
numIntervals = Get number of intervals... 1
for i to numIntervals
  start = Get starting point... 1 i
  end = Get end point... 1 i
  interval_'i'_Duration = end - start
endfor
Remove

# for the vertical dimension, we need to know the maximal duration
maxDuration = 0
for i to numIntervals
  if interval_'i'_Duration > maxDuration
    maxDuration = interval_'i'_Duration
  endif
endfor

Axes... 0 numIntervals 0 maxDuration
for i to numIntervals
  x_left = i - 1
```

```
  x_right = i
  y_bottom = 0
  y_top = interval_'i'_Duration
  Paint rectangle... Red x_left x_right y_bottom y_top
  # to make it look nice, draw an outlined rectangle over that
  Draw rectangle... x_left x_right y_bottom y_top
endfor

# garnish
Draw line... 0 0 0 maxDuration
Marks left every...  1 0.1 yes yes no
Text left... yes Duration (sec)
Text bottom... no Intervals
```

This produces the following Picture Window contents: