Praat scripting manual (workshop) for beginners

v. 1.7 – August, 2014

Mauricio Figueroa
www.mauriciofigueroa.cl

Index

1. Introduction 2
2. Preparatory work: installing software 2
3. Praat environment and objects in Praat 2
    3.1. How does the Praat Objects window work? 3
    3.2. The link between the object window and a script 5
    3.3. How to read and write Praat scripts using Sublime Text 6
4. What's a script and what's a better script 7
    4.1. So, what's a script? 7
    4.2. How to write a good script? 8
5. Variable usage 11
6. Controlling the flow: jumps and loops 14
    6.1. Conditional jumps 15
    6.2. Loops 17
    6.2.1. For loops 17
    6.2.2. While loops 18
    6.2.3. Repeat loops 19
7. Some useful functions 20
    7.1. Mathematical functions 20
    7.2. String functions 22
8. Testing and debugging techniques 24
    8.1. Send stuff (variable values, results) to the Praat Info window 24
    8.2. Pause a script to observe a given state 26
    8.3. Make your script crash if behaved unexpectedly and give feedback to the user 28
9. Navigate the bubble and beyond: objects, files, inputs & interactions 29
    9.1. A bit more to say about navigating the bubble 30
    9.2. Accessing objects outside the script 34
    9.3. Interacting with the user 37
    9.4. Include other scripts and procedures 40
10. Procedures and arrays 42
    10.1. Procedures: writing your own functions 42
    10.2. Arrays: a variable with many coexisting values 46
11. Where to look for more information? 47
Acknowledgements 47

1. Introduction

Welcome to this Praat[1] scripting manual (workshop) for beginners! In the following sections we'll try to cover all the basics that will enable you to read and write Praat scripts to perform typical tasks, particularly those related to object manipulation, querying from objects, saving, controlling the flow of information, etc. I hope that by the end you'll feel that scripts are your friends[2] and that they can make your life easier and not the other way around.

2. Preparatory work: installing software

In order to be able to work with Praat scripts, you have to install the following software:

a. The latest version of Praat:
http://www.fon.hum.uva.nl/praat/download_mac.html (Mac)
http://www.fon.hum.uva.nl/praat/download_win.html (Windows)
http://www.fon.hum.uva.nl/praat/download_linux.html (Linux)

And I recommend you to install the following as well:

b. Sublime Text 2[3]:
http://www.sublimetext.com/2

c. Praat syntax highlighter for Sublime Text:
https://github.com/mauriciofigueroa/praatSublimeSyntax

3. Praat environment and objects in Praat

To begin with, let's open Praat. You'll see that two windows have appeared. One of those, the most important one, is called the *Praat Objects* window, and the other the *Praat Picture* window. Most of the time, unless you're planning on using the Praat Picture window to draw something, I would recommend closing it to keep your window hygiene[4].

The Object window can be quite unfriendly at first sight. It is sort of empty, it almost has no

1 Boersma, Paul & Weenink, David (2014). Praat: doing phonetics by computer [Computer program]. Version 5.3.80, retrieved 29 June 2014 from http://www.praat.org/.

2 Cf.: http://en.wikipedia.org/wiki/Tsubasa_Oozora

3 Although we'll be using a syntax highlighter for Sublime Text, other syntax highlighting systems have been designed for other word processors as well, such as this highlighter for Notepad++ (available for Windows and Linux), this highlighter for Kate (available for Linux) and this other highlighter for TextMate (available for Mac OS X). Sublime Text is available for Windows, Linux and Mac OS X.

4 The drawing capabilities of Praat are quite neat. For starters, the drawing process can be scripted, so you don't have to manually add each element of a drawing each time you redraw. This is particularly useful when you need to dynamically draw something (for example, when the drawing depends on a data set that keeps changing or when the drawing depends on the result of some calculation). Although this workshop does not cover drawing in Praat, check an example in your "files_and_code" folder, called "draw_example" (look for the script called "draw_sound.praat").

buttons because some of the buttons are unavailable for the time being. It shows five menus by default (unless you've added some buttons, which is possible[5]): *Praat*, *New*, *Open*, *Save* and *Help*. Under *Praat*, for example, you can find *Praat > New Praat script* or Praat > *Open Praat script...*.

> *TTGYHD 1*[6] *(difficulty level: **slug**)*: Take a look at some of the options inside the menus and consider which ones might be more useful for you and/or in which situations you could use them in your own research scripting situations.

Some of those buttons available that I've used quite a lot these years are the following (but this is highly dependant on the type of tasks you actually do on Praat!):

New > Record mono Sound...
New > Sound > Create Sound from formula...
New > Tables > Create Table with column names...
New > Create Strings as file list...
New > Create Strings as directory list...
Open > Read from file...
Open > Open long sound file...
Open > Read Table from [tab|comma|whitespace]-separated file...
Save > Save as text file...
Help > Search Praat manual...
Help > About Praat...

It might seem like an odd idea to explore buttons just for the sake of exploring, but you'll be surprised to find yourself learning a lot about the properties of each object and the capabilities of Praat just by taking a look at the buttons available by default and then for each type of object. In the long run you'll be saving time by knowing that some options are available, so that you don't have to program those functions yourself.

3.1. How does the Praat Objects window work?

Having the Praat Objects window in front of you, go to *Open > Read from file...* and open the two files that are located inside the folder "first_objects", within your "files_and_code" folder. These files are called "sound_object.wav" and "textgrid_object.TextGrid".

> *Tip 1*: If you don't like the endless clicking of the menus, press *Control + O* (*Command + O* in Mac) to start the file opening process.

> *Tip 2*: You can open more than 1 object at the same time by pressing *Shift* (for group selection) or *Control* (for individual selection) while doing the clicking.

Once the two files have been opened, you should notice that several things have changed. First

5 Check out how to do this here: http://www.fon.hum.uva.nl/praat/manual/Add_to_dynamic_menu___.html
6 "TTGYHD" stands for "Time To Get Your Hands Dirty".

of all, your Praat Objects window now has two objects: “1. Sound sound_object” and “2. TextGrid textgrid_object” (as seen in *Figure 1*, below). Notice that the objects have been named in a specific format: first, the object has an *ID number*, which is assigned to each new object; second, the name of the type of object; and, thirdly, the name of the file, without the extension (“.wav” and “.TextGrid”, respectively).

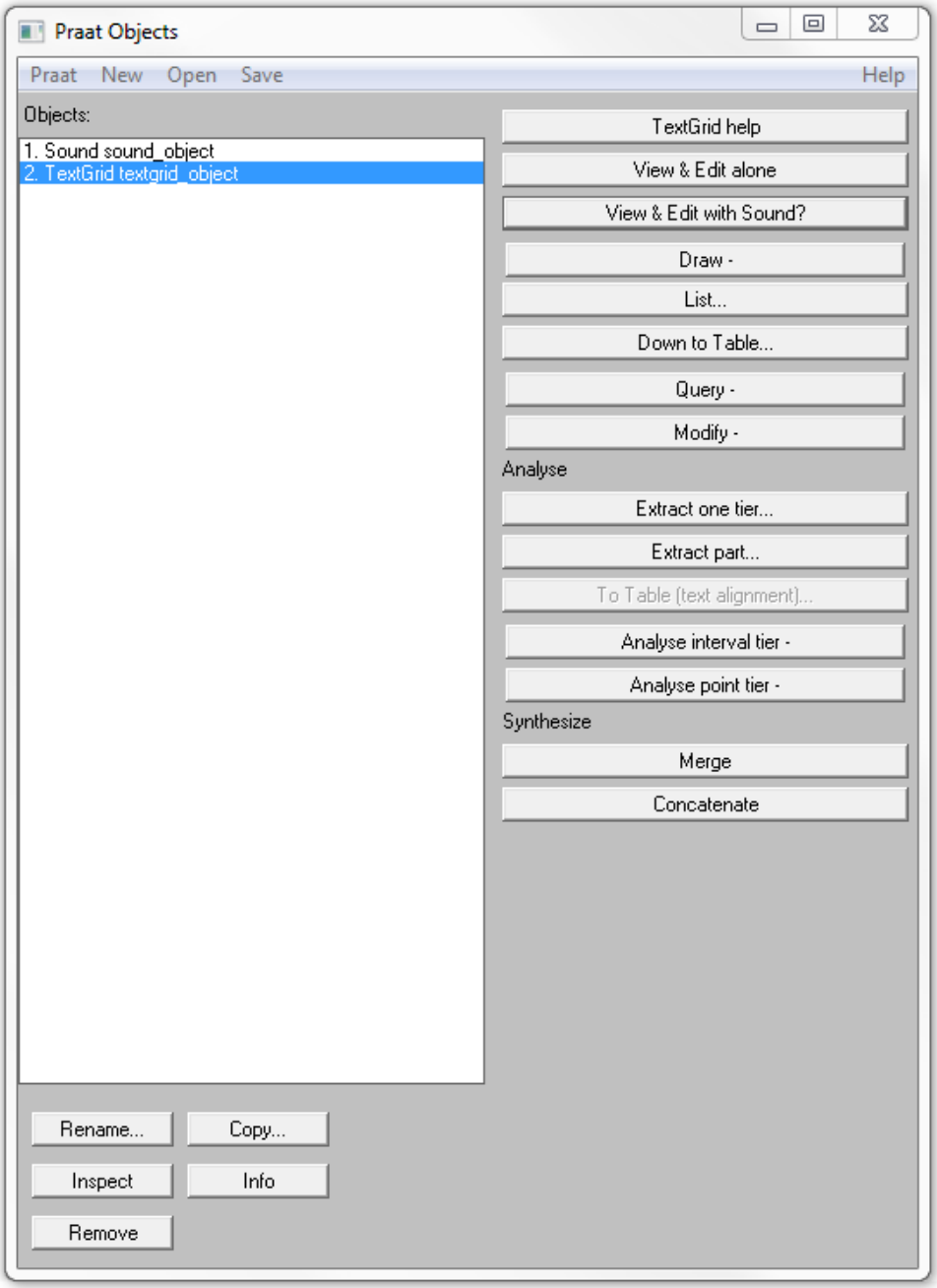

Figure 1: Praat Object window with two objects.

All these elements are going to be important while preparing scripts for Praat, because we will need to select different objects at different points (and we don't want to select wrongly), and also because – as now you can see – the options displayed in the Praat Objects window are object-

dependant.

Go and select the first of those objects and then select the second. Notice how the options available on the right hand side change. Also, select the first object and select the *Save* menu, and then select the second object and do the same. These options also have changed! Select the first object one more time; then press *Shift* or *Control* (*Command*) and select the second object at the same time. The options available have changed again.

This is a dominant feature of Praat and of Praat scripting: this software and its programming language are *object oriented*, because Praat assumes the existence of a group of entities that have predefined characteristics and you are sort of restricted to those features and use scripting to make these objects to interact. Actually, most of the time we will be dealing with the creation, selection, use and removal of Praat objects.

A good way to think about this is by means of an analogy: in a typical human body, for example, all of its constituent elements work together and are linked in several ways, but each one is in charge of only a certain number of tasks (receives certain inputs, produces certain outputs, interacts with other objects, etc.). The same goes for Praat.

> *TTGYHD 2 (difficulty level: **slug**)*: What do you think would be the result of using a command or function for an object from a Praat script when that option is not available for that particular object?

There are a couple of very valuable lessons to take home from this: firstly, what we can do in Praat will depend heavily on the object currently selected; secondly, given that there are so many options per object, it is a good exercise to explore the options available for those objects we often use.

> *TTGYHD 3 (difficulty level: **slug**)*:
> **a**. Select the Sound object, then the TextGrid[7] object and then the two of them combined. Inspect the options for each of these selections and identify one option for each object that you've never seen before and that you think will be useful to you some day.
> **b**. (If you're in a group) Let's talk about that.

3.2. The link between the object window and a script

All the buttons/options that you see in the Praat Objects window are available for use within a script. You have to imagine this as if you had a "virtual pointer" that does all the clicking that you would do by hand, but now, instead of clicking, you have to use the name of that option/object to use it/access it.

> *Tip 3*: Go and press *Praat > New Praat Script*. You should see a new empty window named "untitled script" (that's because it hasn't been saved yet). Press *Edit > Paste history*. Kaboom! All the "clicking" that you've been doing so far has been pasted into the script window using script lingo. Pasting Praat's history can be _very_ useful.

7 For documentation about TextGrids, see: http://www.fon.hum.uva.nl/praat/manual/TextGrid.html

This is what I get after doing the *Edit > Paste history* (you should be getting something similar; I'm using Windows):

```
Read from file: "E:\files_and_code\first_objects\sound_object.wav"
Read from file: "E:\files_and_code\first_objects\textgrid_object.TextGrid"
selectObject: "Sound sound_object"
selectObject: "TextGrid textgrid_object"
selectObject: "Sound sound_object"
plusObject: "TextGrid textgrid_object"
selectObject: "Sound sound_object"
Play
New Praat script
```

You can see now that, from Praat's point of view, the clicking versus the use of those commands and functions in a script are not different at all. When using the mouse, you read the options, decide which one to use and then do clicks to press-those-buttons/select-the-options that you need. What Praat does behind the scene is parse those clicks as if you were writing a script with your mouse (see the code above, if you don't believe me). When writing a script, you use those same commands and/or functions, also by their name, but now you've got to tell Praat by writing out (not clicking) which commands and functions you want to use and which values those commands and functions will take.

3.3. How to read and write Praat scripts using Sublime Text

From now on, whenever you need to read or write a Praat script, use Sublime Text. The reason to do this is that the Praat scripting interface only shows plain text, and doesn't tell you the number of each line or has syntax highlighting. Also (but this is word-editor-dependent), it doesn't help you write script via snippets, it doesn't suggest variable-names that you've already used, it doesn't allow to do efficient searches or comparisons for debugging, amongst other common functionalities of word-editors.

To read a Praat script in Sublime Text, open the script using *File > Open File...* (or by pressing Control *+ O*) and find your way to the file. You can also drag-and-drop files into Sublime Text, which is even quicker.

To write a Praat script and get the syntax highlighter, you have several options: you can start a new file (by using *File > New File* or pressing *Control + N*) and save it with a meaningful name plus the ".praat" extension (for example, "formant_analyses.praat"). Once you save the file with that extension, the highlighter should start working immediately. You can also write a Praat script without saving it and still get the highlighting by selecting *View > Syntax > Praat*.

When you want to test your script in Praat, open it using *Open > Read from file...*. A new window should appear. In that window, select *Run > Run* (or press *Control + R*). That should execute your script. Now, if you want to modify that script, go back to Sublime Text, modify it, _save it_ and then go back to the script window in Praat and select *File > Reopen from disk*, and then run it again.

It is a very good practice to edit your scripts in a proper text editor (such as Sublime Text, Notepad++, Kate, etc.) and then only use Praat's script window to reopen the script and run it.

Tip 4: The process of editing and testing scripts is iterative and potentially time-consuming. Get used to the hotkeys to save time and become less mouse dependant.

*TTGYHD 4 (difficulty level: **slug**)*:
a. Copy, character by character (I mean, without using copy & paste), the script shown below. After writing the first line, save the script in your "files_and_code" folder as "my_first_script.praat", and then finish writing it and save it again.
b. Open the resulting script in Praat and run it.
c. Edit it in Sublime Text to have "number = 2" instead of "number = 1". Remember to save your script.
d. Go to the Praat scripting window and use *File > Reopen from disk*. Then, run your script once more.

```
number = 1
if number == 1
   appendInfoLine: "The number is one."
else
   appendInfoLine: "The number is two."
endif
```

4. What's a script and what's a better script

4.1. So, what's a script?

A script is nothing more than a sequentially ordered set of instructions given to an interpreter (in our case, Praat) to perform one or more tasks. Naturally, these instructions to the interpreter have to be said in the interpreter's language, that is, in Praat scripting syntax.

When the tasks to be done in Praat are simple, or when we are just exploring the commands and functions, the mouse clicking is perfectly fine. However, as soon as you need to do something repetitively or when you need to conduct calculations, or whenever possible – I'd add – a script is a better idea.

Having a script is a great idea for several reasons: (a) once you write it, you just re-run it as many times as needed, and avoid all the clicking that would otherwise be necessary; (b) with a script you assure reproducibility of your analyses; if there is no randomization and your sample stays the same, the results will always be the same; (c) very complex tasks can be modularized and implemented in a script, but would be almost impossible to run "by hand".

Writing a script might seem as a difficult task at first, and at the beginning you might not see the point on doing something that could potentially be done faster by hand. Yes, there is a learning curve, and also Praat scripting might not be the easiest programming language ever invented, but with time, patience and a good deal of Googling, things get easier over time.

4.2. How to write a good script?[8]

Below I've listed some of the principles that I follow when writing scripts (in order of importance). More guidelines exist and the hierarchy that I use might not be the same for other programmers.

(a) *It has one clear goal*: You script most have one very clear goal. The more clearly you know what you need to do, the easiest is to divide that big challenge into simple tasks and then write them using Praat's programming language. If you don't know what you're supposed to be writing, then you're not ready to start scripting. As a general rule of thumb, avoid having more than one script in the same file (unless you're using procedures). Use a filename that is informative and representative of your script, and that will tell you something meaningful if you visit your script in two-month's time.

(b) *It's explicit*: Even if your script doesn't work, even if it's ugly, even if you don't understand half of your own script, be explicit. This means, in more practical terms, to get used to generously comment on your scripts and to use meaningful variable names. If you stick to these rules, you will be helping yourself understand your own script and also, of course, you'll be helping others that might need to use your script.

It happens very often that you go back to a script you wrote 3 months, 1 year or many years before and you have no idea what that script was supposed to do. This is when you thank yourself for having been a good commenter and for using meaningful variable names.

*TTGYHD 5 (difficulty level: **slug**)*: Take a look at the script "explicit_bad.praat", below (you can find it in your "files_and_code" folder). What do you think this script does? Does it strike you as an easy-to-read script?

```
z = randomInteger(2,100)
      d = 0
for i to z
q=    i mod 3 ; What on earth is "mod"?
if q     == 0
d = d +i
endif
endfor
writeInfoLine:d
```

*TTGYHD 6 (difficulty level: **slug**)*: Now, let's consider "explicit_good.praat" (see inside your "files_and_code" folder), below. Notice that the script does _exactly the same_ than the previous one. Why is it easier to understand now?

8 This section is heavily inspired in a section of José Joaquín's tutorial on Praat scripting, referenced at the end of this manual.

```
# Creating a random number.
random_number = randomInteger(2, 100)

# Defining dummy variable to add something to it later.
result = 0

# Iterator to assess numbers between "1" and the random number.
for number_to_test from 1 to random_number

      # Obtain the remainder of number_to_test when divided by "3".
      remainder = number_to_test mod 3

      # Assess if reminder shows divisibility by 3 (it should be "0").
      if remainder == 0

            # If the condition is met, add the number to the dummy.
            result = result + number_to_test
      endif
endfor

#Send final result to screen
writeInfoLine: result
```

Tip 5: As you have probably noticed, comments are declared by using the octothorpe symbol "#" (a.k.a. number sign or hash). You can also create inline comments by using semicolon, as shown below.

```
# This is a comment.
random_number = randomInteger(2, 100) ; This is also a comment.
```

(c) *No-line-unknown rule*: When writing a script do not let yourself move to the next line of code if you don't know exactly what your current line is doing or the value that it's supposed to have at that particular point. Same coin, different side: you should avoid writing a section of a script that comes after a section that you don't understand or that you haven't been able to finish (unless you really know what's going to happen on the first section or if you're writing procedures).

This is less important when reading scripts, particularly if you're a beginner, because some lines of code can be quite complicated and in long scripts it's virtually impossible to follow the flow of information just by eyeballing. However, using someone else's script without understanding what it does (even a single line), can be quite dangerous. Unfortunately this happens a lot!

(d) *It's kept tidy*: A script that is kept tidy is easier to read and maintain than a messy one, that's for sure. Let's take another look to the "explicit_bad.praat" script, but now without spaces where there shouldn't be and with indentation. It gets better, doesn't?

```
z = randomInteger (2, 100)
d = 0
for i to z
      q = i mod 3
      if q == 0
```

```
            d = d + i
      endif
endfor
writeInfoLine: d
```

Avoid using spaces or "tabs" where there shouldn't be. The indentation has to be used meaningfully as well. Normally, a process that happens inside another will require +1 indentation. In the script above, what happens inside the "for" loop has been indented, and then what happens inside the "if" conditional jump has been indented as well.

In programming languages where indentation is not parsed as part of the language's structure, such as Praat scripting, the use of indentation is determined by your scripting-style preferences. However, where and when to use indentation should hopefully be consistent across your script and also meaningful for you and your readers. It's up to you whether you choose to use "tab" or a fixed number of spaces (normally 2 or 4) for your indentation.

*TTGYHD 7 (difficulty level: **slug**)*: Go to the script "indentation.praat" (see "files_and_code") and indent what you consider should be indented and separate the different sections.
a. Do your results look like "indentation_SOLUTION.praat"?
b. How is the indentation showing structure in "indentation_SOLUTION.praat"?

(e) *It's efficient*[9]: Is your script doing the same thing more than once? Are you using the same sequence of commands several times in your script? Are the choices you've made computationally economical?

As a rule of thumb, if you see yourself writing the same lines of code over and over in your script, you probably need either a "for loop" (see section "6.2. Loops") or a "procedure" (see "10.1. Procedures: writing your own functions"). Also, keep in mind that, although modern computers barely complain about running out of memory, they do have limits. Try to find efficient ways to do the same tasks.

*TTGYHD 8 (difficulty level: **slug**)*: Go to the folder called "efficiency", inside your "files_and_code" folder, and open and compare "efficiency_1.praat", "efficiency_2.praat" and "efficiency_3.praat". All these scripts do the same thing (they print to the Praat Info screen the even numbers between 0 and 100.), albeit using different approaches.
a. Rank the scripts regarding efficiency.
b. Justify your ranking to yourself[10] or to someone else, if you happen to have someone around you.

9 This and the next two guidelines are more relevant for advanced scriptors; if you are a beginner, do prioritize the ones above.

10 

Tip 6: If you want, you can use *Shift + Alt + 3* in Sublime Text to divide your screen into 3 cool columns. The same works for 2 or any other number of columns up to 5. It's easier to compare documents by having more than one column available in Sublime Text.

(f) *It's easy to expand*: After you gain some experience, you'll be writing scripts that perform very specific but commonly used tasks, and you'll want those small snippets of code to interact. With that in mind, try to keep your scripts open to possible expansions and contemplate separating some of the tasks in different scripts.

*TTGYHD 9 (difficulty level: **slug**)*: For a good example of what I mean, take a brief look at José Joaquín's "JJATools", here: https://github.com/jjatria/plugin_jjatools.
a. Do you see how some "scripts" (procedures) are used inside other scripts?
b. Why would you want to do that?

(g) *Something good, if short, twice as good*[11]: Well, this is sort of self-explanatory. You can compare again "efficiency_2.praat" to "efficiency_3.praat" in case you find yourself sceptical about this guideline.

5. Variable usage

In programming, a variable is a place in the computer's memory where something is stored. For our purposes, we'll define a variable as the result of the assignment of a certain value (numerical or not) to a certain tag or identifier (its name). It is by assigning content to a tag that the computer's memory is allocated; also, by virtue of this type of link, the content of a variable can be accessed (referenced) whenever needed through its tag.

Although this might sound a bit awkward, for some reason I feel compelled to justify why we should be using variables in a script on the first place; after all why should we bother?

Let's imagine that you, for some obscure and Mephistophelian reason, are planning to prepare a relatively simple script that helps you to do some calculations: it will multiply any number by that very same number minus 1, subtract to the result of than the original number and divide the result by the result of the first multiplication. If you'd like to start with the number "4", those calculations could be written as follows:

```
((4 * (4 - 1)) - 4) / (4 * (4 - 1))
```

Now, if the starting number were to be 16, we could write:

```
((16 * (16 - 1)) - 16) / (16 * (16 - 1))
```

So, no big fuss and everyone's happy, right? Well, what if you had to do this 10, 100 or 500 times, with different numbers? Without variables, you'd have to manually make 5 changes to your script _as many times as the number of times you need to run it_. It certainly doesn't sound so simple anymore. Also, don't you feel that you're rewriting the same stuff over and over (the

11 Conversely: something bad, if long, twice as bad.

numbers “4” and “16”)? Wouldn't it be simpler to have a way to refer to that starting value many times without having to rewrite the entire thing? This is when variables appear and save the day. With variables, you could write a very short script, that would require only one change per iteration of your script (the assignation that occurs on the first line):

```
number = 4
result = ((number * (number - 1)) - number) / (number * (number - 1))
```

As you can see, variables are tremendously useful because they can be referenced several times in a script and also we can decide when to modify its values, even without input from the user. In the following sections and chapters we'll be exploring ways in which variables can be used efficiently in order to make your life easier.

*TTGYHD 10 (difficulty level: **slug**)*:
a. Can you imagine a script that doesn't requires variables?
b. What type of tasks could it actually do in Praat?
c. Can you write one right now? What about copying and running the one-liner scripts from above?

In Praat, variables can be of two classes: *numeric* or *string*. The numeric variables can contain any real number (integers, decimals, negative numbers, etc.[12]). To assign the value “1” to a numeric variable called “numeric_variable”, you'd do the following:

```
numeric_variable = 1
```

Notice that you can assign the content of a numeric variable to another numeric variable, by doing something like this:

```
numeric_variable_1 = 1
numeric_variable_2 = numeric_variable_1
```

The first character of a variable must always start with a lower-case letter from the English alphabet (a-z). From the second character on, you can use upper-case characters (A-Z), numbers (0-9)[13] and underscores (“_”). The following, for example, would be a valid name:

```
numericVariable_0.3_OhYes = 1
```

With numeric variables you can do whatever you can normally do with numbers. To give a simple example, the result of the following script would print to Praat Info the number “3”:

```
variable_a = 1
variable_b = 2
variable_c = variable_1 + variable_2 ; Comment: here we are adding two numbers!
appendInfoLine: variable_3
```

12 For more about this: http://www.fon.hum.uva.nl/praat/manual/Formulas_1_4__Representation_of_numbers.html
13 You can use decimal numbers in variable names, as in “numericVariable_0.3_OhYes = 1”, but I wouldn't recommend that.

*TTGYHD 11 (difficulty level: **slug**)*: Use at least 2 different numeric variables to conduct an arithmetic operation of your preference using Praat scripting. Share the result of your endeavours to your fellow scriptors, if they are around.

String variables, on the other hand, contain text. To assign the text “moon” to a string variable called “string_variable$”, you'd do the following:

```
string_variable$ = "moon"
```

Notice the differences between the notation for the string and numeric variables. The string variables _always_ end in “$”. Also, the text that is assigned to a string variable _always has to be enclosed by double quotation marks_, unless we assign the content of one string variable to another, in which case the name of the variable must be used:

```
string_variable_1$ = "mon"
string_variable_2$ = "soon"
string_variable_3$ = string_variable_1$ + string_variable_2$
```

*TTGYHD 12 (difficulty level: **slug**)*: By the way, what do you think will be the result of adding up two strings like in the previous script? Make your predictions and then run the script in Praat.

Double and single quotation marks ("mon" vs. 'mon') mean different things in Praat. While the former defines the content of a string variable, the second is used for *variable substitution*. Essentially, instead of referring to a string, or to the variable that contains that string, if you were to use single quotation you'd be substituting the reference by the content of the variable (“*moon*”). In our example above, using single quotation marks ('mon') crashes the script. Given that variable substitution should be generally avoided, we won't keep talking about this here. Instead, just stick to double quotation marks, unless you really need variable substitution.

You have to be careful at distinguishing numeric and string variables, and also at knowing when you need to use double quotations or not using quotations at all. In the following script, for example, we will be saving a string into a variable:

```
potato$ = "1"
```

The variable “potato$” contains the string (the sequence of characters) “1”, but not the number (the value) 1. If you were to add the content of “potato$” to a numeric variable, Praat would complain and your script would crash.

The following script also shows a result that might look a bit counter-intuitive, but it works perfectly fine (it's syntactically sound, although it's difficult to imagine a scenario where someone would write this particular script):

```
number = 1
potato$ = "number"
writeInfoLine: potato$
```

*TTGYHD 13 (difficulty level: **slug**)*: What will be printed on the Praat Info window after running the previous script?

Tip 7:
a. The functions *writeInfoLine* and *appendInfoLine* can make the content of a variable visible. The first of these functions (*writeInfoLine*) erases everything previously printed on the Praat Info window and then writes the new line. This is useful when you need a clean slate and you need to print something into the Praat Info windows as well, but you risk to delete information from your Praat Info window that you still need (be careful!).
b. The second function (*appendInfoLine*) doesn't delete anything; it only appends a new line to the existing ones in the Praat Info window.
c. You can combine numeric and string variables in a line by using commas. If you use "+", you can only add variables of the same category: only numeric or only string variables.
d. If you want to clean the Praat Info window without adding a newline, you can use the *clearinfo* command.
The script "adding_lines.praat", located in your "files_and_code" folder, shows an example of some of these tips.

Do remember to use variable names that are meaningful and that you think you're going to remember afterwards. Those names can be as long as you want, but try to keep them short. Also, some people prefer to use underscores to separate words within a variable, while others choose to separate them via starting upper-case characters; you can also find some scriptors that don't show separate words at all (see the snippet of code, below). There is no golden rule here, although I do suggest to show different words in variable names: use the system that best suits you.

```
some_men_just_want_to_watch_the_world_burn$ = "Alfred Pennyworth"
someMenJustWantToWatchTheWorldBurn$ = "Alfred Pennyworth"
somemenjustwantowatchtheworldburn$ = "Alfred Pennyworth"
```

6. Controlling the flow[14]: jumps and loops

Often you'll want to make the behaviour of your script dependent on certain conditions such as "if A happens, do this; if B happens, do this other thing instead". This is when *conditional jumps* become handy. Also, the whole point of writing scripts is to take advantage of the computer capabilities to do many things very fast, instead of you doing them one by one, slowly. This is when *loops* are fundamental.

*TTGYHD 14 (difficulty level: **medium**)*:
a. Where in this handout have we already used conditional jumps and loops?

14 Cf. https://www.youtube.com/watch?v=WhBoR_tgXCI

b. For what purpose?

6.1. Conditional jumps

Conditional jumps allow you to control the flow of information in your script into different alternative directions depending on predefined conditions (hence the name, "conditional jumps"). This is _very_ useful, and it is important that you become accustomed to use them and to using them well.

In Praat scripting, the simplest conditional jump possible starts with an "if" declaration, followed by a variable, number or string, a comparison operator, and then another variable, number or string. The conditional jump ends in a different line with an "endif" declaration. Any lines that you insert between these two declarations ("if" and "endif") will only be executed if the conditions evaluated are *true*. Also, you can only evaluate two values from the same category: numbers with numbers and strings with strings.

```
if 12 == 12
      appendInfoLine: "This is indeed equal"
endif
```

As you can see, the conditional jump begins with and "if" statement, followed by the number "12", which is compared via "==" to the number "12". If the comparison holds *true*, a line will be appended to the Praat Info window. After this line, the conditional jump is closed through the "endif" declaration.

Tip 8: The following comparisons are allowed in the current version of Praat scripting: equals ("==" or "="[15]), unequal ("<>"), less than ("<"), greater than (">"), less than or equal ("<=") and greater or equal (">=").

You can have multiple evaluations within the same declaration, as in the following script:

```
if 12 == 12 and "tomato" <> "potato"
      appendInfoLine: "The numbers are equal, the strings are not."
endif
```

And, naturally, you can have forking paths, so to make your script react in different ways depending on the defined conditions:

Tip 9: The function "randomInteger(min, max)" provides a random integer between the minimum and maximum defined values.

```
# Generating and storing random number
number_to_test = randomInteger(1, 4)
```

15 Although the evaluation within the conditional jump will work anyways whether you use a single or a double equals sign, it is healthy to distinguish comparisons ("if 12 == 12") from assignments ("variable = 12"). I recommend you to always use double equals sign for comparisons.

```
# Conditional jump to assess a random number.
if number_to_test == 1
      appendInfoLine: "The number is 1."
elsif number_to_test == 2
      appendInfoLine: "The number is 2."
elsif number_to_test == 3
      appendInfoLine: "The number is 3."
else
      appendInfoLine: "The number is 4."
endif
```

In this script, we generate a random number between “1” and “4” and save the result of that operation in the numeric variable called “number_to_test”. Then, we declare a conditional jump, which as usual starts with “if”. Then, we include the terms of a comparison; in this case, the content of the variable “number_to_test” and “1”. If this condition is met, a line is appended to the Praat Info window that says “The number is 1.”.

We have added more possible outcomes for our conditional jump, however. In subsequent declarations, still in the same general conditional jump, we evaluate whether “number_to_test” equals “2” or “3”. We do that via the “elsif” declaration, which is short for “else if” (you can have as many “elsif” declarations as you want). Just as in the first “if” comparison, you need to enter _both terms of the comparison_ for the “elsif” declarations. If you forget to do this (if you had written “elsif == 2”), Praat will complain, because: if *what* is compared to “2”?

Finally, there is a final condition, declared via “else”. This “else” declaration can be translated as “anything else that has not been covered in the previous declarations via “if” and “elsif” statements”. Given that it's any other scenario not covered by the previous definitions, it doesn't require a declaration of terms (this can be a double-edged sword, if your “else” statement covers more conditions that those you foresaw). By the way, using “elsif” doesn't force you to use “else” as well, and vice-versa. Both are optional and only the initial “if” statement is mandatory.

*TTGYHD 15 (difficulty level: **slug**)*: Use Sublime Text to write a script (and test it in Praat) called “what_if_if.praat” in which you assess whether a numerical variable called “numeric” is equal to “1”, “2” or “3”. Save the script in your “files_and_code” folder and share your solutions with the group (if you happen to be in one). If you get stuck, check a possible solution in your “files_and_code” folder, called “what_if_if_SOLUTION.praat”.

You can always nest elements inside of other elements in Praat. This means that you can have “if” conditional jumps inside other conditional jumps. You can go as deep as you want, as long as you know what you're doing[16]. In the script below, there is nesting of the sort I'm talking here.

```
# Defining the variable
number = 1313

# Conditional jump: first level.
if number > 10
      new_number = number / 2
      remainder_number = new_number mod 2

      # Conditional jump: second level.
```

16 Cf. https://www.youtube.com/watch?v=66TuSJo4dZM

```
      if remainder_number == 0
            appendInfoLine: "Number is bigger than 10 and divisible by 2."
      else
            appendInfoLine: "Number is bigger than 10, but not divisible by 2."
      endif
else
      appendInfoLine: "Number is smaller than 10."
endif
```

6.2. Loops

A *loop* allows you to execute a task repetitively a certain number of times. How many times exactly will depend on certain conditions that you have to define in the loop's declaration. They are the main tool that you'll use to simplify iterative processes and begin to be able to conduct massive tasks that would otherwise be impossible to finish by hand.

6.2.1. For loops

These loops are the simpler ones to use. They execute a task as many times as indicated in the range that has to be defined after the main declaration. The *for* loop begins with a "for" declaration which is followed by an iterative variable, that will contain the range of values specified immediately afterwards: "from 11 to 20", as in the example. The for loop is closed with an "endfor" declaration. Any lines placed between these two elements ("for" and "endfor") will be executed as many times as specified. See an example (run it!) of a simple for loop below:

```
for changing_variable from 11 to 20
      appendInfoLine: changing_variable
endfor
```

The most difficult part about using a for loop has to do with understanding the iterative variable ("changing_variable" in the example) and using it whenever convenient. Think of this iterator variable as a variable that will contain a different value at each cycle of your for loop. Which value? In our example above, "11" on the first cycle, "12" on the second, … , … , and "20" on the last cycle.

Given that this variable is an iterator, often it's conventionally written as "i". Of course, you can name that variable as you see fit. Also, if your for loop starts from "1", which is the case most of the time, you can skip the "from 1" part in your for loop declaration. The following loop – which prints to the Praat Info window the numbers from 1 to 10 – exemplifies this:

```
for i to 10
      appendInfoLine: i
endfor
```

You can use variables instead of number for the ranges of your loop; also, you can nest for loops. You can access inside your for loop variables that have been defined outside, as in the next example:

```
minimum = 5
maximum = 15
message$ = " is a number."
for i from minimum to maximum
      appendInfoLine: i, message$
endfor
```

Tip 10: You can also intervene the iterator variable from within the script, which can be useful if you need to, for example, skip a particular number in your cycle (e.g., if you need to iterate "from 1 to 10", but skipping the number "6"), or if you want to make your for loop to only use even numbers. Be careful, though! If you do this wrong, you might end with an infinite loop that will make Praat crash (and maybe your computer as well).

*TTGYHD 16 (difficulty level: **slug**)*:
a. Check in your folder "files_and_code" the script "skip_a_beat.praat", where the iterator variable has been intervened to skip a number.
b. Imagine a scenario when this could be useful for your own daily tasks with scripts and Praat analyses.
c. Can you think of an alternative and simpler way to achieve the same result? Clue: "6.1".

6.2.2. While loops

These loops iterate while certain predefined condition (or set of conditions) is (are) *true*. The *while* and *repeat* loops are more difficult to grasp than the for loop. Also, if used improperly, it's quite easy to inadvertently create an infinite loop that will make Praat to crash, so I recommend you to treat these with a bit of respect and think them through before attempting to use them.

Let's imagine that you need to divide a number by 1.25 until it's smaller than 5, but given that you don't know the original number (nor its order of magnitude) until the user inputs it, you don't know beforehand how many times you'll have to divide the number. You can use a while loop here to divide the number until it meets the required condition:

```
input_number = 142857142867
while input_number > 5
      input_number = input_number / 1.25
      appendInfoLine: input_number
endwhile
```

In this script we see that "input_number" receives a value. Then, a while loop is declared, and the comparison to be evaluated is stated as usual. The while loop is closed by using "endwhile". All that lies between these two declarations will be repeated until the above-mentioned comparison is satisfied. In order for the condition defined to be satisfied at some point, we need at least one of the elements of the comparison to be a variable, so that we can modify its value _from within the while loop_. Do you see how this works in the script above? The comparison states "input_number > 5", and we affect "input_number" from within, by dividing that number by "1.25" and then assigning the result of that calculation to "input_number" again. Neat, right?

If the line "input_number = input_number / 1.25" is a bit counter-intuitive for you – because

we are doing something to variable A and then saving the effect of that into A itself – think of it as being the same than doing the following, but in less steps:

```
input_number = 142857142867
while input_number > 5
      result_of_division = input_number / 1.25
      input_number = result_of_division
      appendInfoLine: input_number
endwhile
```

*TTGYHD 17 (difficulty level: **slug**)*:
a. Why is the following script (see below) flawed?
b. What do you think would happen if we run it?

```
input_number = 142857142867
while 142857142867 > 5
      input_number = input_number / 1.25
      appendInfoLine: input_number
endwhile
```

6.2.3. Repeat loops

Repeat loops are similar to while loops in that they somehow repeat something until it reaches a certain condition (which is defined at the end of the whole declaration). How is this different from the "while loop", then? There are two main differences. The most important one is that "while" loops could eventually not be executed even once, if the declared comparison happens to be *false* from the beginning, whereas repeat loops are always going to be executed at least once, because during the first cycle of the loop the condition(s) declared has(have) not been evaluated yet.

*TTGYHD 18 (difficulty level: **slug**)*: Can you imagine a scenario where this difference (repeat: always runs at least once; while: runs only if conditions are met) could be relevant or even crucial?

The second difference, which causes the behaviour described above, has to do with where the assessment is placed in the declaration. In while loops this assessment is declared at the beginning; in repeat loops this assessment is declared at the end. Given that the actual _final_ result from a while and repeat loops are normally the same, some people conceive these two loops as equivalent, but we now know that the number of times a condition is evaluated changes depending on the loop you're using, and if you are also interested in the number of evaluations (besides the final result, I mean), then you need to decide which loop best accommodates your requirements.

Oh, mortals, behold (below) a repeat loop, which does the equivalent of the while loop that we were analysing above:

```
input_number = 142857142867
repeat
```

```
      input_number = input_number / 1.25
      appendInfoLine: input_number
until input_number < 5
```

Besides the declaration being in different places, do you notice any other difference? Hey! Wait a second! Why in the while loop we have “while input_number > 5”, but in the the repeat loop we have “until input_number < 5”? The first one has a “greater than" and the second one a “less than”... Weren't we doing the same thing? Well, not exactly. The while executes a loop while a condition is *true* (“I will keep cycling while this condition is *true*, no matter what; if I happen to find a *false* result for my assessment, I will stop”), whereas a repeat executes a loop until something happens (“I will keep cycling throughout all this *false* assessments, until I find myself a good and nice *true*; only then I shall stop”).

*TTGYHD 19 (difficulty level: **slug**)*: Go to your “files_and_code” folder and open (in Sublime Text) and execute (in Praat) the script “repeat_until_while.praat”.
a. Observe the default behaviour of the two loops and compare the results that are printed into the Praat Info window.
b. Modify the numeric variable “input_number” _for both loops_ so that it's smaller than “5”. How does the behaviour of the loops change?

7. Some useful functions

So, first, what's a function? A function is itself a (normally short) script that takes in arguments and produces an output. The actual script that makes the function work is hidden from us, we only call the function, provide arguments, get the result and do with it whatever we want.

Tip 11: Normally, the first thing you'd like to do with the result of a function is to save it into a variable so that you can access it later.

There is plethora of tools for Praat scripting that fit the rather inclusive definition provided above. For the time being, however, we will only examine some *mathematical functions*[17] and *string functions*[18].

7.1. Mathematical functions

Well, there are lots of mathematical functions in Praat. How many you use and when will obviously depend on the tasks you have at hand. If you do signal processing, for example, you'll find yourself checking the documentation for mathematical functions frequently. If you are more of a measurements type of person, you'll probably only use a few. Some of the more often used ones are listed below.

17 Documentation here: http://www.fon.hum.uva.nl/praat/manual/Formulas_4__Mathematical_functions.html
18 Documentation for this here: http://www.fon.hum.uva.nl/praat/manual/Formulas_5__String_functions.html

(a) *abs(x)*: Provides the absolute value of a number, that is, it ignores if it is a positive or negative number.

```
number = -0.333
absolute_number = abs(number)
writeInfoLine: absolute_number ; Prints "0.333" to the screen.
```

*TTGYHD 20 (difficulty level: **slug**)*: What type of acoustical variable can take positive and negative values?

(b) *round(x)*: It provides the nearest integer.

```
number = -0.333
rounded_number = round(number)
writeInfoLine: rounded_number ; Prints "0" to the screen.
```

(c) *sqrt(x)*: Provides the square root of "x"

```
number = 9
sqrt_number = sqrt(number)
writeInfoLine: sqrt_number ; Prints "3" to the screen.
```

(d) *min(x, …)*: Finds the minimum of a series of numbers.

```
number_1 = 10
number_2 = 30
minimum = min(number_1, number_2, 50)
writeInfoLine: minimum ; Prints "10" to the screen.
```

(e) *max(x, …)*: Well, this one finds the maximum.

```
number_1 = 10
number_2 = 30
maximum = max(number_1, number_2, 50)
writeInfoLine: maximum ; Prints "50" to the screen.
```

(f) *exp(x)*: It exponentiates the constant *e* (2.71828182845...) by the value provided in "x".

```
e_number = exp(3) ; same as e^3
writeInfoLine: e_number ; Prints "20.085536923187668" to the screen.
```

(g) *randomInteger(min, max)*: It provides an integer value between the minimum and maximum values defined in "min" and "max".

```
minimum = 2.54
maximum = 27.00002
random_number = randomInteger(minimum, maximum)
writeInfoLine: random_number ; Prints random _integer_ number to the screen.
```

*TTGYHD 21 (difficulty level: **high**)*: Use the *randomInteger(min, max)* function in a script to find the minimum and maximum values from a pool of 100 randomly generated numbers, whose minimum (*min*) will be the year 0 and the maximum (*max*) the current year of our Lord. You'll need at least one for loop and several conditional jumps to solve this problem. Save your script as "min_max_from_100.praat". You can find one possible solution to this challenge in your "files_and_code" folder, in a script called "min_max_from_100_SOLUTION.praat".

(h) *hertzToBark(x)*: It transforms a value from raw Hertz to Bark-rate (analogous functions for other transformations are: *barkToHertz*, *hertzToMel*, *melToHertz*, *hertzToSemitones* and *semitonesToHertz*).

```
value_in_Hz = 1400
value_in_Bark = hertzToBark(value_in_Hz)
writeInfoLine: value_in_Bark ; Prints the result to screen.
```

Finally, what if the mathematical function you need is not in the list provided in the documentation? You can create your own functions (check "10.1. Procedures: writing your own functions") and then use them in other scripts as you would with any other function (check "9.4. Include procedures and other scripts").

7.2. String functions

If you work with TextGrids, you'll be using these *string functions*. All these have in common that they return a text string as output or have a string as at least one of their arguments. Essentially, they do stuff with strings. Those that return a string as the result of their mingling are followed by a dollar sign.

Again, here it is a partial list of string functions (but do check the documentation to find other very interesting and potentially useful ones):

(a) *length(a$)*: Provides the length of the string[19].

```
string$ = "potatoes"
length_string = length(string$)
appendInfoLine: length_string ; Prints the number "8" to screen.
```

(b) *left$(a$, n)*: Gives back a string that has the first *n* characters of *a$.*

```
string$ = "potatoes"
left_characters$ = left$(string$, 3)
appendInfoLine: left_characters$ ; Prints "pot" to screen.
```

(c) *mid$(a$, i, n)*: Returns a string of length *n* from *a$*, starting from character in position (index) *i*.

19 This is an example of a string function that is not followed by "$" (you have "length_string = length (string$)"), because it returns a number.

```
string$ = "potatoes"
mid_characters$ = mid$(string$, 3, 4)
appendInfoLine: mid_characters$ ; Prints "tato" to screen.
```

(d) *right$(a$, n)*: Gives back a string that contains the rightmost *n* characters of *a$*.

```
string$ = "potatoes"
right_characters$ = right$(string$, 4)
appendInfoLine: right_characters$ ; Prints "toes" to screen.
```

TTGYHD 22 (difficulty level: medium): Let's create a script called "first_last_equals.praat" that parses words written in lower-case and sends a message to the screen if the first and last characters of the word are identical. If you get stuck, check the solution in your "files_and_code" folder, called "first_last_equals_SOLUTION.praat".

(e) *index(a$, b$)*: Gives you the index (the place in the sequence of characters) of the first occurrence of the string *b$* in the string *a$*.

```
string$ = "potatoes"
string_to_look$ = "t"
index_t = index(string$, string_to_look$)
appendInfoLine: index_t ; Prints the number "3" to screen.
```

(f) *rindex(a$, b$)*: Gives the index of the last occurrence of *b$* in *a$*.

```
string$ = "potatoes"
string_to_look$ = "t"
rindex_t = rindex(string$, string_to_look$)
appendInfoLine: rindex_t ; Prints the number "5" to screen.
```

TTGYHD 23 (difficulty level: high): Can you modify "first_last_equals.praat" so that, instead of using *left$(a$, n)* and *right$(a$, n)* it uses *length(a$)*, *left($, n)* and *rindex(a$, b$)* to obtain the first and last character and assess if they are equal, within a conditional jump? Name that script "first_last_equals_modified.praat". You can find a possible solution to this problem in your "files_and_code" folder, in a script called "first_last_equals_modified_SOLUTION.praat".

(g) *string$(number)*: Transforms a number into a string format. It digests exponential notation and percentages!

```
number_1 = 56
number_2 = 5.6
number_3 = 56%
number_4 = 5e6
n1_str$ = string$(number_1)
n2_str$ = string$(number_2)
n3_str$ = string$(number_3)
n4_str$ = string$(number_4)
```

```
# This line prints "56, 5.6, 0.56, 5000000" to the screen.
appendInfoLine: n1_str$, ", ", n2_str$, ", ", n3_str$, ", ", n4_str$
```

(h) *number(a$)*: Interprets a string as a number.

```
number_1$ = "56"
number_2$ = "5.6"
number_3$ = "56%"
number_4$ = "5e6"
n1_str = number(number_1$)
n2_str = number(number_2$)
n3_str = number(number_3$)
n4_str = number(number_4$)

# This line prints "5000062.16" to the screen.
appendInfoLine: n1_str + n2_str + n3_str + n4_str
```

If the string function you need is not provided by default in Praat (check the documentation first, though), you can create and use your own functions (check sections below).

8. Testing and debugging techniques

Writing lines of code produces bugs, and there is nothing we can do to avoid it[20]. The only way to avoid adding bugs into our code is to avoid writing scripts altogether, which implies using someone else's code and assuming that it's bug-free. This also implies, of course, devoting some time to finding out whether the script that you need already exists and to check that it works properly (that is, to de-bug it).

Whether you're going to write your own scripts or you're using someone else's scripts, you'll need to test and debug the scripts as efficiently as you can to make sure that your script is doing exactly what it's supposed to do.

8.1. Send stuff (variable values, results) to the Praat Info window

Although a bit cumbersome and slow, this is one of the best ways you have to test or debug a script, both when reading or writing one. Particularly for beginners, I would _very strongly recommend_ that you get into the habit of checking that you're getting the results you expect from your variables after every line of code that you write, unless your line is very very obviously correct.

Failing to observe the content of your variables after writing a line of code might be very time-consuming afterwards, especially for long scripts and when the bug you introduced only manifests itself as a problem 25 lines of code later.

Tip 12: One very typical bug comes from simply trying to reuse a variable and misspelling it. Then, when you run your script, you get an “Unknown variable” message from Praat

20 Different rates of bugs per line of code (LOC) have been estimated, as shown here: http://programmers.stackexchange.com/questions/185660/is-the-average-number-of-bugs-per-loc-the-same-for-different-programming-languag

and all hell breaks loose. To prevent this, in Sublime Text, get into the habit of double clicking the variable that you just rewrote, and check whether the previous instance of that variable gets highlighted. If it does, all is good to go. For an example, copy the code below in Sublime Text.
a. Does *sugestive_number* match with the first instance of the same variable?
b. What about the variable called *remainder_2*?
Naturally, the longer and more obscure your variable names are, the higher the risk for introducing these bugs.

```
suggestive_number = 1313
remainder_2 = sugestive_number mod 2
if remainder_2 < 2
      # Do something
endif
```

The two main ways you have to print to the screen are using “writeInfoLine” and “appendInfoLine”, as explained before (see *Tip 7*). Be careful when using the former because you know that everything that has already been printed into your Praat Info window will get deleted.

Normally, you'd want to print to screen the current state of a variable or group of variables, mostly when it forms part of a loop or when its value depends on conditional jumps[21].

Tip 13: Use “tab$” to separate elements to be printed in a table-looking fashion. This is particularly useful for inspecting loops. Check the code below in Praat to get a better idea of its use (before that, copy it to Sublime Text to check the syntax).

```
clearinfo
input_number = 142857142867
counter = 1
appendInfoLine: "NUMBER", tab$, "VALUE"
while input_number > 5
      appendInfoLine: counter, tab$, input_number ; Check the "$tab" here!
      counter = counter + 1 ; This is the same as writing "counter += 1"
      input_number = input_number / 1.25
endwhile
```

TTGYHD 24 (difficulty level: medium): Go to your “files_and_code” folder and open the script “what's_wrong_with_you_1.praat” in Sublime Text, and then run it using Praat. The script will crash. Praat will give you a report, including the number of the line where there is a problem. To make things a little bit harder, the script isn't commented or indented.
a. Use “writeInfoLine” and/or “appendInfoLine” and your common sense to find the bugs.
b. Fix the script until it works.

21 Printing to the screen consumes computer-time. When you print a line or two, your computer will manage to do that very, very fast and you probably won't notice the time involved in printing that line. However, when you want to send hundreds or thousands of lines to Praat's Info window, you'll see that, often, your script uses most of its time on printing those lines instead of doing whatever else your script is supposed to do. Once you've debugged a section of code containing a loop by printing into Praat's Info window, I'd suggest that you delete or, even better, comment-out the line of code that prints the current values of variables for that loop.

c. Leave an inline comment in your script (using ";") to remind you of each bug you found. If you get stuck, check one possible solution in your "files_and_code" folder, called "what's_wrong_with_you_1_SOLUTION.praat".

You can also print information into the Praat Info window which can be useful for tracking which processes have been completed in your script. If your script is rather large or it's going to deal with a large data set, it's worth telling the user what the script is doing and/or when it has finished doing a certain sub-task. The following code provides an example of what I mean. Try it (first, copy the code in Sublime Text and select Praat highlighting to read it properly)!

```
# First for loop.
appendInfoLine: "Going through loop 1..." ; FEEDBACK LINE (FL)
variable = 0
for i from 1 to 400000
	random_number = randomInteger(1, 100)
	variable = variable + random_number
endfor
appendInfoLine: "Loop 1 finished: variable equals ", variable, "." ; FL

# Second for loop.
appendInfoLine: "Going through loop 2..." ; FL
for i from 1 to 400000
	random_number = randomInteger(1, 100)
	variable = variable - random_number
endfor
appendInfoLine: "Loop 2 finished: variable equals ", variable, "." ; FL

# Comparing the results
if variable > 0
	appendInfoLine: "Loop 1 wins (more numbers added)!"
else
	appendInfoLine: "Loop 2 wins (more numbers subtracted)!"
endif
```

Tip 14: Provide feedback to your user whenever you know that a task will take a long time. In the script above, for example, each loop iterates 400,000 times. Depending on the speed of your computer, completing this can take whatever from a fraction of a second to several seconds. As you can see, _before starting each loop and not inside the loop_, the script sends a feedback line to the user via Praat's Info window ("Going through loop 1..." and "Going through loop 2..."), so that the user knows that the script is doing something, even if it's taking a long time. You'd normally want to do this for huge loops and also when your script creates big objects, such as Intensity, Formant or Pitch objects from long sound files[22].

8.2. Pause a script to observe a given state

Another tool that can be used to debug or test a script is to use pauses. The function

22 Interested in knowing how long does a script take to do whatever it does? Use *stopwatch*! Documentation here: http://www.fon.hum.uva.nl/praat/manual/Scripting_6_5__Calling_system_commands.html

pauseScript[23] allows you to momentarily stop your script and print stuff to a message window that will pop up. See the code below for an example.

```
variable = 1313
pauseScript: "At this point variable equals ", string$(variable), ".", newline$
variable = variable - (variable / 2)
pauseScript: "Now, the variable equals ", string$(variable), ".", newline$
```

*TTGYHD 25 (difficulty level: **slug**)*: Copy this script into Praat's script window and run it. Observe how the two pauses that are defined behave.

Tip 15: We use "newline$" to insert a new line character after our line or text, so that the message to the user is more readable. The result of adding this line becomes more evident when using the *exitScript* function (see next subsection).

Notice that, whenever *pauseScript* is used, the script prompts the pop-up and the message that we define after the *pauseScript* function. Also, notice how the use *string$(...)* allows us to send numerical information to the pause pop-up. It is as if we were printing to the screen, but pausing the script at the same time.

An additional application of the *pauseScript* function is to be able to observe which object is selected at the time that the pause is executed. This can be tremendously important to test if your script needs to select more than one object at different stages of the task. A very typical bug in a Praat script is trying to execute a command for an object different than the one that is currently selected. If that's the case, then your script might crash, although if the same command is available for more than one object (two Sound objects, for example), bugs could get way more difficult to detect and fix.

*TTGYHD 26 (difficulty level: **slug**)*: Go to you "files_and_code" folder and there access the folder named "pause_practicing". Inside, you'll find the script called "using_pause.praat".
a. Open that script in Sublime Text and try to understand it without running it.
b. Once you think you understand the script, run it, and observe how the use of *pauseScript* allows us to check which object is selected at a given stage in the Praat Objects window[24].
c. Use *appendInfoLine* to investigate the content of those weird variables named "sound_a_ID", "sound_b_ID", "sound_c_ID" and "sound_d_ID". Why are they there? Clue: we use those variables later in the script to do something.

23 The function *pauseScript* only became available after version 5.3.82 (before that, *pause* was used instead). If you have and older version of Praat, *pauseScript* will crash your script.

24 For more on how to select objects and object management, see: "9.1. A bit more to say about navigating the bubble".

8.3. Make your script crash if behaved unexpectedly and give feedback to the user

Let's imagine that you're conducting an automatized production experiment[25] to compare jitter[26] and shimmer[27] values from three age-groups: young adults (18-30), adults (31-55), older adults (56-80). You prepare a form (see section "9.3. Interacting with the user") and ask your users to enter their age, and then, within a larger script, you include the following lines of code:

```
age_entered = 16
if age_entered < 18
      exitScript: "You need to be over 18 (your age: ", string$(age_entered),
      ").", newline$
endif
```

If you run these lines of code, the script will terminate if the condition stated in the conditional jump is met. We're using *exitScript* to make this happen. Notice that you can add a message after the *exitScript* function. Just as we did before for pauses, you can also use *string$(...)* to introduce the value of numerical variables defined before (in this case, "age_entered") into that string.

*TTGYHD 27 (difficulty level: **slug**)*: Imagine a situation, related with the type of things you do with Praat, where using *exitScript* to terminate the script would be useful. Let's talk about that.

Using conditional jumps you can test many properties of the information that's flowing through your variables, or that is entering via the user's intervention or via interaction with other files, the outer world and the universe beyond. When it comes to numbers, for example, you can test whether they are positive or negative, their size with respect to another variables or themselves, if they have changed as you were expecting or not, if they are actually numbers (and not disguised strings acting like numbers: "variable$ = "1""), etc. If the content of a variable fails your conditional jump test, then you terminate your script with *exitScript* and give feedback to the user regarding the nature of the problem.

It goes without saying that the same can be done about the content of string variables: you can compare them, query their length, check whether they have changed as you want, if they contain sequences of characters that you need (capital characters, spaces) or, maybe more importantly, that they don't have some characters that would mess with your script, etc.

*TTGYHD 28 (difficulty level: **high**)*: Write a script called "what's_wrong_with_you_2.praat" that parses the content of a numeric variable called "my_number" and a string variable called "my_string" to detect the following:
a. The number has to be an integer, positive, bigger than 18 and lower than 120.
b. The string can only start with a lowercase character from a-z, it must finish in a vowel, and it most be shorter than 10 characters.

25 Yes, you can run experiments using Praat scripting. Check the following sites for more information on this:
http://www.fon.hum.uva.nl/praat/manual/ExperimentMFC.html
http://www.fon.hum.uva.nl/praat/manual/Demo_window.html
26 About jitter in Praat: http://www.fon.hum.uva.nl/praat/manual/Voice_2__Jitter.html
27 About shimmer in Praat: http://www.fon.hum.uva.nl/praat/manual/Voice_3__Shimmer.html

c. Make sure you give adequate feedback to your user both if all the conditions are met or if any condition is violated and you need to terminate your script.
Only after you finish your own version, check two possible solutions to the problem in your "files_and_code" folder. One of these proposed solution scripts (called "what's_wrong_with_you_2_SOLUTION_1.praat") uses brute-force string parsing to test the strings, while the other one ("what's_wrong_with_you_2_SOLUTION_2.praat") uses *regular expressions*[28].

9. Navigate the bubble and beyond: objects, files, inputs & interactions

One nice thing about Praat is that it can interact with the outer computer-world in quite useful ways. Do you remember the first section, when we were talking about exploring available buttons and stuff? Well, four of the common buttons/options that we saw up there happen to be about Praat interacting with the computer world around it:

Open > Read from file...
Open > Open long sound file...
Open > Read Table from [tab|comma|whitespace]-separated file...
    Open > Read Table from tab-separated file...
    Open > Read Table from comma-separated file...
    Open > Read Table from whitespace-separated file...
Save > Save as text file...

It's a bit obvious here: the first ones have to do with opening files in order to create objects in Praat (e.g., sounds or tables), the last one has to do with Praat saving an object into the outer world. Notice that some of these functions are quite general (the first and last one), and seem to apply to several types of objects that Praat can read or write as text. Others seem more object-specific, such as the ones related to tables.

Tip 16: Apropos saving: there is no such thing as an "autosave" function in Praat. Praat will never save something unless explicitly instructed to do so. This means that you could have been working all day, say, segmenting and labelling a TextGrid and then, God forbid, Praat crashes (or there is a power fail, or you happen to close Praat, etc.). Well, all those precious hours of work will be lost, and you'll be cursing and hating yourself.
Get yourself into the habit of saving those objects that are being edited as often as your OCD demands. Saving often takes way less time than redoing the work!

In the following subsections, we'll explore some of the ways that we can use to administer the way in which Praat navigates between objects and how Praat can access and obtain input from outside.

28 The actual functions that allow you to use regular expressions are *index_regex (a$, b$)*, *rindex_regex (a$, b$)* and *replace_regex$ (a$, b$, c$, n)*. The use of these functions is explained in the string functions documentation. Regarding regular expressions in Praat, see: http://www.fon.hum.uva.nl/praat/manual/Regular_expressions.html

9.1. A bit more to say about navigating the bubble

We've been talking about this quite a bit already. First, we already know that Praat works by creating objects that we can then access and do stuff with. A fundamental part of our capability to write scripts for Praat will have to do with our capacity to navigate objects with precision.

Firstly, there are various ways by which you can refer to a given object. Remember that objects in Praat contain several types of information (ID number, type of object and name), so we can use these parameters to select an object.

*TTGYHD 29 (difficulty level: **slug**)*: Before continuing, restart Praat. Then, open the four WAV files that you will find in the folder "selection_tests", within your "files_and_code" folder. Once you do this, you should see the four objects in your Praat Objects window, in alphabetical order, with ID numbers from 1 to 4, and the last one should be selected.

Tip 17: Whenever you open or create a new object, Praat will select it by default. If you open or create several objects, the last one will always be selected by default. Keep this in mind when writing your scripts!

There are two ways to select objects. In order to select an object using its ID number, we can do:

```
selectObject: 1
```

Obviously, you need to know that object's ID number before being able to write a line like this. This is a bit of a problem (Problem 1), because you already know that each object has its unique ID, so if you were to remove those four objects from the Praat Objects window and you open them again (without closing Praat), their IDs will change and you'll have objects going from 5 to 8, so if you write the above line in your script Praat will complain and say: "No object with number 1".

You can also imagine a similar problem (Problem 2): imagine that you write a script that (a) opens two files; (b) does something with those objects; (c) selects the first one with a line like the one above ("selectObject: 1"); (d) removes that first object from Praat's Objects window. If you send this script to your Praat party-friends and they run it immediately after opening Praat, all should work smoothly. However, if your friend has opened a TextGrid first and has edited it profusely without saving and then runs your script, your script will open the two objects, do stuff with them, select the first object via its ID (object ID number 1 = the TextGrid object!) and remove it, along with your friendship.

You can also select an object by it's name, which is a combination of the type of object plus the name of the file opened or the name that you provided when creating the object. This type of selection would look like this (don't forget the double quotation marks when doing this):

```
selectObject: "Sound c"
```

To use this method, just like with the IDs, you'll need to know the full name of your object in order to select it, which can be a bit of a hassle. Also, this method suffers from a critical problem

(Problem 3): if you were to reopen the same files, for example, you'd see that the new four objects that have been opened have the same names than the previous four objects. If you have these eight objects open, and the names are repeated, the line of code above would only select the second "Sound c" object, and never the first one. In short, if two objects have the same name and you select them by using a line as the one above, only the second one can be selected.

So far, it looks as if selecting objects is a real pain, because: (a) we need to know for sure the ID or name of an object in order to select it, and to do that, we'll need to look at the Praat Objects window; and (b) if you reopen the same object, it will have a different ID and you can't access the previous same object using its name.

In order to circumvent all these problems, we will use (now and forever) a simple trick available in Praat: whenever you open a file or create an object in your script, you'll save that object's ID in a variable with a meaningful name. Given that each object has a unique ID, there is no risk of selecting the wrong file afterwards, as long as you use the correct variable. This would look like this in your script:

```
# When reading a file (the path to the file is relative):
sound_ID = Read from file: "..\selection_tests\a.wav"

# When creating an object:
intensity_ID = To Intensity: 100, 0.0, "yes" ; See Tip 18 below.

# Selecting the objects one after the other:
selectObject: sound_ID
selectObject: intensity_ID
```

In the script above, we are opening a WAV file (via "Read from file: ...") and we're assigning the ID of the resulting object to a numeric variable, called "sound_ID". The name of that variable, of course, can be anything you like, but I suggest keeping those names informative and relevant. Then, the script creates an Intensity object via "To Intensity: ...", and assigns the ID of that resulting object to a variable called "intensity_ID". Afterwards, when the script needs to select those objects, we use the variables "sound_ID" and "intensity_ID" to make the selections.

This elegant technique solves all the problems listed above, because (a) you don't need to know, find out or remember the names or IDs of the objects; (b) it doesn't matter if you already have more objects opened in the Praat Objects window; (c) it doesn't matter whether the names of the objects are repeated, because IDs are unique.

To summarise, then, whenever you do something in a Praat script that results in a new object in your Praat Objects window (when opening a file, when creating an object, etc.), this technique can be used to assign the ID of that new object to a variable, and then you can use the variable to select the objects you need.

Tip 18: You might be wondering how we know what to write when we're creating an object in Praat, like we did in the example script above in the 4th line:

```
> intensity_ID = To Intensity: 100, 0.0, "yes"
```

It seems as if the *To Intensity* function requires some arguments to work (just like the

other functions we've seen before). To know which arguments are required, we'll have to explore that function, but before, a little important detour.

If you go to the Praat Objects window and select a sound, you'll see that some of the buttons that appear end in nothing (*Play*), others end with a hyphen (*Draw* -) and others end with three dots (*To Intensity...*). The first type of command (*Play*), can be accessed in a script just by writing the name of the function (Praat is case sensitive, so mind your uppercases and lowercases), as in the following line (without the ">"; that's just to show that that's a line of code here):

```
> Play
```

The buttons that end in a hyphen (*Draw* -) aren't really commands, but collections of commands, so you can't access them from a script. You can only access them via clicking, and the result of that is just that you see more buttons to appear. The commands that end in three dots, such as *To Intensity...*, are functions that will require arguments to work. In our script above, this functions requires 3 arguments (end of the detour).

To know what those arguments are, you can either go to the documentation in Praat, or go to the Praat Objects window, press that button and take notice of the arguments required. If you were to do that now (do it!), a window will appear and you'd see that the first argument corresponds to the "Minimum pitch (Hz)" value, the second to the "Time step (s)" and the third one to whether or not we want to subtract the mean. This third option is different because it's a Boolean argument.
Numeric arguments should be written in a script as such, Boolean values ("yes" / "no" options) should be entered as "yes" or "no" between double quotation marks (they are mandatory options; you have to provide an argument for every required Boolean argument), string values should be inputted between quotation marks, and non-Boolean lists of options (alternatives) via the name of the option that you want written between double quotation marks. You have to use commas to separate each argument.

*TTGYHD 30 (difficulty level: **slug**)*:
a. Go to your Praat Objects window and select a sound object. Then, using your mouse, select *Analyse periodicity - > To PointProcess (extrema)...*. A window will open asking for arguments. How would the default selected options need to be written in a Praat script? Test your line to see if it works.
b. Still with a sound selected, select *Annotate - > To TextGrid...*. A window will open telling you the required arguments for this function. How would the default options need to be written within a Praat script? Test your line to see if it works.
c. Go to *New > Tables > Create Table with column names...*. Write a Praat scripting line that creates a Table object named "awesome_table", with 5 rows and the column names by default. Test your line of code to see if it works.

Save your script inside the folder named "arguments", and name it "arguments.praat". Check the solution for this exercise in "arguments_SOLUTION.praat", which can be found

in the “arguments” folder, within “files_and_code”. In the solution, the path to the sound file is relative.

What were we talking about before these huge but fruitfully distracting excursions? Oh, yes, about selections. There are some other ways to obtain the ID number or name of a given object in Praat. As long as the object you're interested in is selected, you can use *selected(“type_of_object”)* and *selected$(“type_of_object”)*. The first function allows you to get the ID number of the selected object, but you have to specify the type of object of the current selection[29]. The second function provides the name of the object being queried, which can be saved into a string variable; again, the type of object has to be specified. Examples are shown below.

```
# Creating dummy sound object from formula:
Create Sound from formula: "sineWithNoise", 1, 0, 1, 44100, "1/2 *
   sin(2*pi*377*x) + randomGauss(0,0.1)"

# Using functions to obtain the object's ID and name and assign to variables:
id_number = selected("Sound")
name$ = selected$("Sound")

# Append line with the content of the previous variables:
appendInfoLine: name$, tab$, id_number
```

If you use these functions and ask for an object type which is not included in the current selection that you have, Praat will complain and terminate your script. Also, if you have more than one object of the same category selected, Praat will only provide the ID number or name of the first one selected.

Now that we know how to select a unique object through its ID (or name), we can do more stuff. You can select more than one object, for example, by using *plusObject*. This has to be written like this:

```
# Option A (don't use this method)
selectObject: 1
plusObject: 4
plusObject: 7

# Option B (this is how you should do it!)
selectObject: sound_ID
plusObject: intensity_ID
plusObject: table_ID
```

You can use this function to select as many objects as you need, as long as you're able to provide their unique IDs. Once all these objects have been selected, you can do to all of them whatever you want and is available as an option for that particular combination of objects (for example: *Remove*).

You can also deselect a particular object, by using *minusObject* and the name or ID of the

29 There are many type of objects in Praat. Some of the most common ones are: Sound, TextGrid, List, Table, Formant, Pitch, Spectrum and Pulses, but there are many, many more. For a full list, go to Praat and select *Praat > Technical > List readable types of objects*.

object, as in the following mock script:

```
selectObject: sound_ID
plusObject: intensity_ID
minusObject: intensity_ID
```

Finally, you can use the *select all* function to, well, select all the objects presently in your Praat Objects window. Use this function wisely! You don't want to write a couple of lines of code like the following unless you like to live in danger:

```
select all
Remove
```

Tip 19: A good practice when writing scripts is to remove from your Praat Objects window those objects that you won't use any longer. To do this, however, you must be sure that: (a) you're removing the right object; and (b) that you won't need that object anymore. Depending on the capabilities of your computer, it is perfectly possible for Praat to deal with a huge number of objects in the Praat Objects window – by default, up to 10.000 –, but you probably don't want that cap to be reached, for several reasons:
(a) You might run out of RAM memory before running out of available objects;
(b) If you surpass the 10.000 cap, Praat will complain and crash;
(c) Are you _really_ using those 10.000 objects at the same time? Isn't there even a couple thousand that you could live without for the time being?

9.2. Accessing objects outside the script

We've been doing this all this time: we've been opening files a lot. The result of that has been that new objects start to appear in our Praat Objects window and then we can do things with them. In order to open a file via the *Read from file...* function, that file has to be one of the types of file supported by Praat[30]. If the file is not one of the files that Praat can read, it will complain with an error message.

Remember that there are other methods to open files in Praat, such as *Read Table from comma-separated file...*, so some of the files that can't be opened with the *Read from file...* function, such as ".csv" files, can be accessed with alternative and more specific functions.

Tip 20: You can assess whether a file is readable by using the *fileReadable (fileName$)* function, which provides 1 (*true*) if the file exists and can be read and 0 (*false*) if not.

You can use absolute and relative (to the script) pathnames to define the location of a file you want to access from Praat:

```
# Absolute path file:
table_ID_abs = Read Table from comma-separated file: "F:\04._2014_06_Praat-
   Workshop_Antwerp\files_and_code\opening_files\data\just_data.csv"
```

30 By the way, Praat completely ignores the extensions of the files while accessing them (an extension is the ".wav", ".mp3", ".doc" bit that specifies the type of file).

```
# Relative (to the script) path file:
table_ID_rel = Read Table from comma-separated file: "data/just_data.csv"
```

*TTGYHD 31 (difficulty level: **slug**)*: You can try this script by going to the "opening_files" folder, within your "files_and_code" folder, and opening "opening_table.praat" in Praat and then running it. Remember: open the script in Praat (not copying; opening) and then run it. The absolute path should make the script crash unless you have a path exactly like that on your computer.

Tip 21: Did you notice the different types of slashes ("slash" vs. "backslash") in our script above? Back in time, the type of slash used for the paths were dependent on whether you were using Windows, Mac OS or a distribution or Linux. Now, Praat handles both types of slash in pathnames, which is rather nice.

Using absolute path files has the advantage of making it more difficult to open the wrong file, but it makes your script more difficult to move to another computer, because that pathname probably doesn't exist in other computers. Using relative path files makes your script more portable, which is very much desirable, but it also makes opening the wrong file more likely. While some people are strong-minded about this, I would recommend you to use the paths that better suit your requirements.

Take into account that relative paths only work if Praat knows the "relative to what" part. Let's say you close Praat now and open it again all fresh and nice. Go to *Praat > New Praat script* and then copy the line of code below and run that script.

```
table_ID = Read Table from comma-separated file: "data/just_data.csv"
```

Unless you're very weird, Praat should give you an error message saying, essentially, "I couldn't find the path that leads to that file, so I couldn't open it". That's because you haven't told Praat the "relative to what" part, and Praat is assuming it should apply that relative path to its default folder location, which is the folder where you keep the programme's "Praat.exe" file.

In order to tell Praat the "relative to what" part of the path, you can either provide the full path in our script (which is not really a solution) or open the script from within a folder where that path makes sense. This is simpler that it sounds: the script "opening_table.praat" is saved in a folder called "opening_files", and "just_data.csv" is saved in a folder called "data", inside the "opening_files" folder. If you open "opening_table.praat" in Praat (not copying and pasting, but opening), then Praat will reset its working folder as the folder where your script is located, and all of a sudden the relative path makes sense, because relative to the working folder, the path works. As a matter of fact, if the script and the data file were kept in the same folder, and you open your script in Praat (again: opening, not copying and pasting) you could just write:

```
table_ID = Read Table from comma-separated file: "just_data.csv"
```

Tip 22: Having your scripts and the files it requires within the same folder is a good idea for beginners. For complex situations, with many files being opened and many others

being saved at different stages, relative paths to different folders are more effective, although you have to be careful to write your relative paths correct.

Besides reading all sorts of files and being able to save all its objects, Praat is able to perform other interesting tasks with files[31]:

(a) *writeFileLine*: If the file exists, it will delete all the lines that are already in that file and then it will write the line that you have specified after the function. If the file doesn't exist, it will try to create it and then add the line that you specified. In the script below, the file to be modified (or created) is "myFile.txt", and the line to be added is specified afterwards.

```
writeFileLine: "myFile_A.txt", "Cute line to be added."
```

(b) *writeFile*: It does exactly the same as *writeFileLine*, but it doesn't add a newline symbol at the end of the line. The newline symbol is what commoners would call "an Enter" (they mean, the result of pressing "Enter" in a word processor).

```
writeFile: "myFile_B.txt", "Gorgeous line to be added."
```

(c) *appendFileLine*: It appends the specified line of text to the end of an already-existent file.

```
appendFileLine: "myFile_B.txt", "Another gorgeous line to be added."
```

(d) *appendFile*: Does the same as *appendFileLine*, but without adding the newline symbol.

```
appendFile: "myFile_B.txt", "And yet another gorgeous line to be added."
```

(e) *createDirectory*: It creates a directory with the specified name and in the specified path, if the directory doesn't exist. Otherwise it does nothing.

```
directory_name$ = "hyper_praat"
createDirectory: directory_name$
```

(f) *deleteFile*: It deletes the specified file or directory, if it exists.

```
deleteFile: "hyper_praat"
deleteFile: "myFile_A.txt"
deleteFile: "myFile_B.txt"
```

*TTGYHD 32 (difficulty level: **slug**)*: In your "files_and_code" folder, there is a folder called "creating_modifying". Write a Praat script called "creativity.praat" that is able to:
a. Create a folder inside "creating_modifying" called "im_creative".
b. Create a text file called "to_be_edited.txt".
c. Appends a list of the even numbers between 0 and 50. The script will have to add the numbers one by one, each in a new line.

31 For more about files, see: http://www.fon.hum.uva.nl/praat/manual/Scripting_6_4__Files.html

d. It tells the user when the script has finished doing all its deeds.
If you get stuck (but you have to try!), check the solution in "creating_modifying", called "creativity_SOLUTION.praat".

9.3. Interacting with the user

The most efficient way to interact with your user is to create a *form*. In Praat, a form is a pop-up that occurs at the beginning of a running script (irrespective of where in your script the form was actually placed) that is capable of asking for information from the user and then assigning that information to numerical and string variables for later use.

Whenever you need something from the user, use a form. This will happen often in your life if you write scripts, because you may want to ask several things from users: which files they want to analyse/modify/create, all sorts of parameters of interest, sociolinguistic information, etc. Also, using a form can prevent some common problems such as the difficulty of using relative paths to access files: if you make the users specify the path, then it's their fault if they choose the wrong one.

To write a form[32], you declare it by using *form* and you close it by using *endform*. The displayed form can have a title, which has to be written (without quotation marks) after *form*. Everything between the opening and closing statement, should be specified as a field type. The field types need to be declared at the start of a line, then you should specify the name with which they will appear in the pop-up window (which is going to be used as the variable name as well), and then you can optionally include a default value for the user. For a simple example, see the following form:

```
form User's input
   comment Please enter the following information:
   sentence Name:
   integer Age:
   choice Sex: 1
       button Female
       button Male
   real Smoothing_factor: 100 (= average)
endform
```

This form will pop-up as shown in *Figure 2*, below. You can see that the pop-up is titled "Run script: User's input". Then, several field types appear. First, there is a comment, which is displayed as text in the form. If you write a line that is too long, it won't be automatically trimmed to fit the size of the form, so you have to try to keep these comments short or use several *comment* field types, although that doesn't look very good.

Next, there is a field type called *sentence*, after which the users are expected to write their name; then the age, which has to be entered into an *integer* field type. After that, there is a *choice* field type, whose variable name is *Sex*. Two choices are provided ("Female" and "Male", and the form by default selects the first one). Finally, there is a *real* field type, that takes any real number.

Once the user has entered his/her responses to this form, you can access that information via the variables that you've used, although you have to modify the names a bit for it to work. The variables for this particular form, are: *name$*, *age*, *sex* or *sex$* (the first one returns the number of

32 Documentation here: http://www.fon.hum.uva.nl/praat/manual/Scripting_6_1__Arguments_to_the_script.html

the option that has been chosen – 1 or 2, in this case; the second one returns the string of the answer – "Female" or "Male") and *smoothing_factor*.

```
writeInfoLine: name$
appendInfoLine: age
appendInfoLine: sex, tab$, sex$
appendInfoLine: smoothing_factor
```

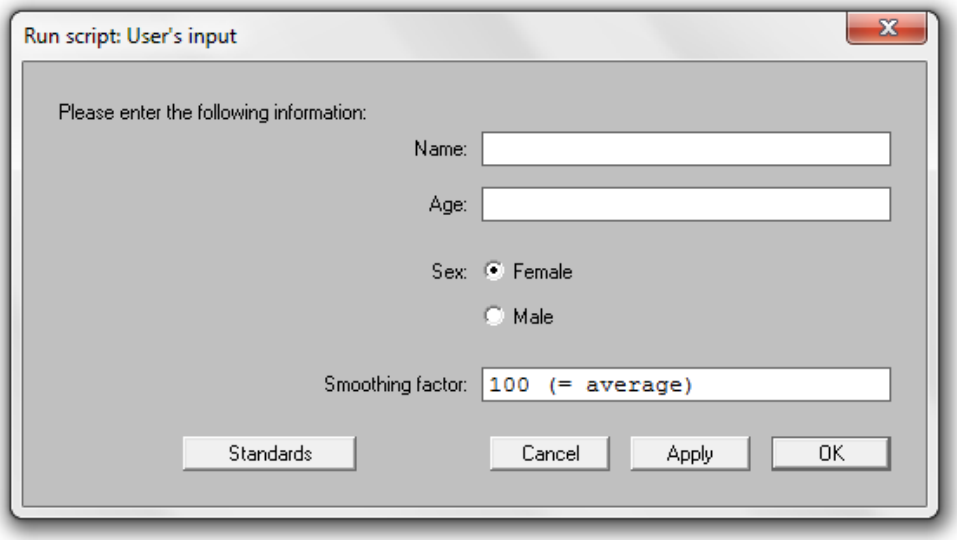

Figure 2: Example of a Praat form window with four field-types.

Notice several things: first, all the capitals that were visible on the script and form are gone when we call those variables; the same can be said of the colons that were written after each variable name in the script and that were visible in the form. Also, a variable that was written in the script as "Smoothing_factor:" appears on the screen as "Smoothing factor:" and then is accessed as "smoothing_factor". Finally, notice that you can add a short comment after an option (the "(= average)" in the Smoothing factor field type). This is ignored by Praat when parsing the responses from the form, but is useful to guide the user.

The complete list of field types is the following: *real* (for real numbers), *positive* (for positive numbers), *integer* (well, for integers), *natural* (for positive integer numbers), *word* (for strings without spaces; "yes indeed" becomes "yes"), *sentence* (for short strings), *text* (for long strings), *boolean* (for check boxes where 0 is *off* and 1 is *on*), *choice* (which shows a radio box; has to be followed by...), *button* (two or more buttons for the radio box declared in *choice*), and *comment* (for a line of text). Instead of *choice* + *button*, you can also use *optionmenu* and *option*, which work in the same way but use less space in the form.

TTGYHD 33 (difficulty level: medium): Let's create a form!
a. Create a Praat script called "my_first_form.praat" containing a form that asks the user to fill in three field types. The form should be able to obtain a string (without spaces), an integer and the answer to a Boolean or multiple choice question.
b. Write more code so that somehow your script does something depending on the

content of the inputted variables.
c. Provide feedback to the user regarding the result of whatever you did with the aforementioned variables.
You can check for a possible solution to this problem in your "files_and_code" folder, named "my_first_form_SOLUTION.praat".

If you need your user to choose a file or a directory (this happens a lot), so that you can later access those paths, you can always ask him/her to write the absolute or relative path through a *sentence* field type and then use that string as your path. This, however, sounds like a bit too much effort for your user. It's easier if you make your script prompt a file-opener pop-up via *chooseReadFile$*, as shown below:

```
file_name$ = chooseReadFile$: "Open a table file"
if file_name$ <> ""
   table = Read Table from comma-separated file: file_name$
endif
```

As you can see from the first line, the result from the function *chooseReadFile$* needs to be assigned to a string variable. You can add a message to the user after the colon. This message gets displayed in the top of the pop-up window.

When this line of code is read, a file selector window will appear, with (in this example) "Open a table file" as the title. If the user clicks *Open*, the variable "file_name$" will contain the name of the file that the user selected; if the user clicks *Cancel*, the variable "file_name$" will contain an empty string.

If you want your user to choose where to save a file (from an object that exists in the Praat Objects window), you can use:

```
selectObject: sound_ID
file_name$ = chooseWriteFile$: "Save as a WAV file", "sound_ID.wav"
if file_name$ <> ""
   Save as WAV file: file_name$
endif
```

This code will select an object from the list of objects and then it will assign to a string variable the result of the selection from the user (the user will be able to modify the "sound_ID.wav" name, which is prompted in the dialogue). As with the previous function, it's perfectly possible to create a conditional jump to assess whether the user selected something and pressed *Open*, in which case "file_name$" will contain a string, or *Cancel*, in which case the string will contain nothing.

TTGYHD 34 (difficulty level: medium): Create a script inside "outside_the_bubble", which is in the folder "files_and_code", to practice *chooseReadFile$* and *chooseWriteFile$*. Your script, which you should call "outside_the_bubble.praat", must do the following:
a. Make the user select a Sound and a TextGrid file. Use a message to suggest to the user to select those files from the folder "outside_the_bubble". If the user selects no files, make terminate the script and send a message to the user.
b. Make the script select both objects in the Praat Objects window and to prompt a *View & Edit* window. As soon as the prompt occurs, insert a *pauseScript* including a message to

the user telling him/her to modify the TextGrid. Once the modification has been made, the user must press *Continue*.
c. Make the script ask the user to choose a location and name to save the modified TextGrid. Also, suggest that the user modifies the name of the original file. The original name in this case is "textgrid_object.TextGrid", so the modified version you suggest could be "textgrid_object_mod.TextGrid". If the user selects no location or name for the new file, make the script terminate and send a message to the user.
d. Tell the user when the script has finished.
If you, after trying decently hard, can't come up with a solution for this script, take a look to "outside_the_bubble_SOLUTION.praat".

Finally, you can make your user select a directory, using the function *chooseDirectory$*, which works similarly to the previous functions:

```
directory_name$ = chooseDirectory$: "Choose a directory to save your stuff."
if directory_name$ <> ""
   Save as WAV file: directory_name$ + "/sound.wav"
endif
```

9.4. Include other scripts and procedures

Another way to access information from outside the script is to include other scripts and procedures into your script. Procedures? We haven't talked about procedures yet (we will in "10.1. Procedures: writing your own functions"). For the time being, imagine procedures as being pieces of code that might take arguments to carry a (normally very) specific task. You can think of them as small scripts that you can use inside your script whenever there is a task that you have to do several times (not in the sense of a loop, but in the sense of a task you find yourself doing frequently in your script).

Regarding scripts being included in other scripts, let's first look at the snippets of code below. The first code contains a short script with a form that requires the user to enter his/her age (in years) and select his/her sex. Once the form has been completed, the script clears Praat's Info window and then appends a line of text in Praat's Info window which includes the content of the variables defined in the form: age and sex (the last one as a string).

```
# FIRST PIECE OF CODE ("script_being_called_1.praat"):
# Form to obtain information from the user:
form Enter your age and sex
   comment Please fill the following field types:
   integer Age: 00 (= in years)
   choice Sex: 1
         option Male
         option Female
endform
clearinfo
appendInfoLine: "Your age is: ", age, ". Your sex is: ", sex$, "."
```

This next script does only one thing: it calls and runs the previous script (called

"script_being_called_1.praat") and another script called "script_being_called_2.praat". You can see that there are two uncommented lines containing the function *runScript*. This function requires only one mandatory argument, which is the name of the Praat script to be called. Besides this argument, you should enter all the values required if the script being called contains a form. In the case of the scripts being called in the code below, the first script requires two arguments: age (in years) and sex (as string). The second script being called doesn't require any arguments (you can inspect this second script inside your "files_and_code" folder, inside "script_in_script").

```
# SECOND PIECE OF CODE ("script_in_script.praat"):
runScript: "script_being_called_1.praat", 29, "Male"
runScript: "script_being_called_2.praat"
```

Unfortunately, you can't access the content of variables that exist in the script being called (such as "age$" from "# FIRST PIECE OF CODE […]") in the script that is doing the call (see "# SECOND PIECE OF CODE […]"). If you really need to access the content of those variables, what you need is a procedure.

*TTGYHD 35 (difficulty level: **slug**)*: Open in Praat the script "script_in_script.praat", which is saved in the folder "script_in_script", inside your "files_and_code" folder.
a. Can you anticipate what should happen before running the script?
b. Play with the scripts until you feel comfortable calling scripts from within other scripts.

Using a procedure inside a script is quite easy. You only have to use *include* and then provide the name of the procedure that you want to include (be careful so that the procedure and the script are in the same folder; otherwise, provide the correct relative or absolute path). The following piece of code is a procedure called "years_in_months". This procedure has been saved in a file named "procedure_to_be_included.proc", which you can find inside "include_procedure", which is inside your "files_and_code" folder.

```
# Declaring procedure "years_in_months", which takes only one argument: "years".
procedure years_in_months: years
   # What the procedure does.
   total_months = years * 12
   appendInfoLine: "Your age in months is: ", total_months, " months."
endproc
```

This procedure takes an argument from the user (in a variable called "years") and then does calculations with that variable and sends the result of that calculation to the Praat Info Window. The next piece of code shows how to include this procedure (which is located in a different file) in a script called "script_which_includes_procedure.praat", which has been saved in the same folder as the previous file.

```
# Clearing Praat Info window.
clearinfo

# Including the procedure in this script:
include procedure_to_be_included.proc
```

```
# Using the procedure four times:
@years_in_months: 11
@years_in_months: 22
@years_in_months: 29
@years_in_months: 105
```

These lines of code clean the Praat Info window, then include the procedure using *include* and the name of the file where the procedure is, and then use that procedure four times.

> *TTGYHD 36 (difficulty level: **slug**)*: Open in Praat the script "script_which_includes_procedure.praat", which is saved in the folder "include_procedure", inside your "files_and_code" folder.
> **a**. Can you anticipate what should happen, before running the script?
> **b**. Play with the scripts until you feel comfortable calling procedures from within other scripts.

10. Procedures and arrays

In this last section, we'll be dealing with procedures and arrays, which are tools that the common mortal users of Praat don't normally employ, but that are nonetheless quite easy to use and to understand, and are very useful. Let's start with procedures, about which we've been talking a bit in the previous sections.

10.1. Procedures: writing your own functions

As we said above, procedures[33] are scripts inside your scripts, which are particularly useful when there are some lines of code that you find yourself using repeatedly in your script (or that you think you'll be using again another time)[34]. Let us think of an example. Imagine that you're writing a script in which you need to process some strings in different ways, and you need to do this several times, but in different parts of your script (so that a loop doesn't really solve the problem). In that case, you should write a procedure.

Just as a loop or conditional jump, a procedure declaration has to be opened and closed, and then the content of the procedure should be written between these two entities. All the lines of code that you write inside the procedure will be run by your script whenever you call the procedure. Also, and this is quite important, procedures can take arguments.

Let's consider an example:

```
clearinfo
# Calling the procedure several times.
@dealing_with_strings: "crocodile", 5
@dealing_with_strings: "hippopotamus", 10
@dealing_with_strings: "immateriality", 3

# Defining the procedure.
procedure dealing_with_strings: string_to_parse$, how_short
```

33 Procedures are similar (although not equal) to what in other languages such as Python and R is called "function".
34 Documentation here: http://www.fon.hum.uva.nl/praat/manual/Scripting_5_5__Procedures.html

```
   # Parsing the string
   string_to_evaluate$ = string_to_parse$
   short_string$ = left$(string_to_evaluate$, how_short)

   # Reporting to the user:
   appendInfoLine: "The short version of the string ", string_to_evaluate$, "
   is: ", short_string$, "."
endproc
```

In this script, after clearing the Praat Info window, a procedure called “dealing_with_strings” is summoned three times[35]. To call the procedure, an *at* symbol (“@”) must be used, and all the arguments required by the procedure must be provided as well. This procedure requires two arguments: a string and then a number.

This is only clearly visible when the actual procedure is analysed. The procedure is declared with *procedure*. Then, the name of the procedure is defined (in this case, “dealing_with_strings”). All the names of the variables that the procedure will use (the arguments) have to be declared after the colon (in this case, “string_to_parse$” and “how_short”). Notice that you must specify whether your arguments will take string or numeric values.

The procedure is closed via *endproc* and everything between “procedure” and “endproc” will be run every time the procedure is called. Notice that the arguments that were provided at the beginning of the procedure are indeed used inside the procedure (that's the whole idea!).

*TTGYHD 37 (difficulty level: **high**)*: Modify the script from above so that:
a. The procedure also evaluates whether a given one-character-long string matches the first character of *string_to_parse$* and whether another given one-character-long string matches the last character of *string_to_parse$*.
b. The procedure should give feedback to the user regarding these matches.
c. Name your script “procedure_example.praat” and save it in your “files_and_code” folder.
If you get stuck after trying, examine the script “procedure_example_SOLUTION.praat”.

Once the procedure has been run at least once, you can access from outside those variables used inside the procedure. For example, you could write a new line of code after the procedure with the following:

```
appendInfoLine: short_string$
```

Notice that this line of code contains a variable (“short_string$”) that has never been used outside the procedure yet, but given that the procedure already ran, it can be accessed now because this variable has been defined and it contains a value. The value contained by this variable will correspond to the value that has been assigned to it inside the procedure _the last time that the procedure was called_.

In the lines of code below, the seventh line will print to Praat's Info window “imm”, because we

35 Yes! Procedures can be summoned “before” being defined in your script. What actually happens is that Praat scans your whole script to see if it contains procedures or if procedures are being included (see: “9.4. Include other scripts and procedures”) and starts parsing the whole script again from top to bottom only after it has acknowledged their existence (if they exist).

are calling the variable “short_string$” after the second time we called the procedure.

```
# Clearing Praat Info window.
clearinfo

# Calling the procedure several times:
@dealing_with_strings: "hippopotamus", 10
@dealing_with_strings: "immateriality", 3
appendInfoLine: short_string$ ; <-- CHECK THIS LINE AND MOVE IT AROUND.

# Defining the procedure
procedure dealing_with_strings: string_to_parse$, how_short
   # Parsing the string
   string_to_evaluate$ = string_to_parse$
   short_string$ = left$(string_to_evaluate$, how_short)
endproc
```

*TTGYHD 38 (difficulty level: **slug**)*: What would be printed to the screen if we were to swap the fifth and sixth lines of code in this script?

Those variables that can be used outside a procedure are called *global* variables. The fact that variables that are used inside a procedure can be used outside can be very useful (and, thus, desirable) or it can be a hindrance, depending on what you want to do and how tidy you code.

If the names of your variables are recycled and you are using the same variable names inside and outside your procedure (don't!), the fact that the variables declared inside the procedure can be used outside of it can become a big problem and a source for bugs. Also, let's imagine that you send a procedure in a “.proc” file to your Praat-loving friends for them to include in their scripts, and some of the variable names are unknowingly the same in your procedure and your friends' script. Things can become very confusing and dangerous.

In order to restrict those variables used inside the procedure to the procedure itself, we can use variables whose scope is *local*. This is a way to play safe, so I would recommend you to use local variables for all those variables that only fulfil a role inside the procedure and reserve global variables for those _very few_ that you clearly need to be using outside your procedure (this is still not the safest way to go, though! Do check *Tip 23* for the ultimate safe trick to access local variables from procedures).

To define a local variable you need to include a full stop symbol before the name of the variable, as shown below:

```
clearinfo
# Calling the procedure several times:
@dealing_with_strings: "hippopotamus", 10
@dealing_with_strings: "immateriality", 3

# Defining the procedure
procedure dealing_with_strings: .string_to_parse$, .how_short
   # Parsing the string
   .string_to_evaluate$ = .string_to_parse$
   .short_string$ = left$(.string_to_evaluate$, .how_short)

   # Reporting to the user:
   appendInfoLine: "The short version of the string ", .string_to_evaluate$, "
```

```
   is: ", .short_string$, "."
endproc
```

These variables now have a local scope, which means that if we try to use them outside the procedure, as in the line below, Praat will give us an "Unknown variable" error message.

```
appendInfoLine: .short_string$
```

Tip 23: You can access the content of local variables outside your procedure if you call the variables as follows (this line only applies to the script from above, obviously):

> appendInfoLine: dealing_with_strings.short_string$

This line will only work if the procedure has been called once, and it will have the same value as the last time the procedure used the variable "short_string$". This is by far the safest way to access variables from inside a procedure, because (a) all the variables inside the procedure can remain local, and (b) it forces the scriptor to be very explicit when calling a local variable by using the name of the procedure.

*TTGYHD 39 (difficulty: **high**)*: Since Praat's version 5.3.44, Praat's syntax changed a bit to, amongst other things, become variable-substitution-free. Several transformations have taken place since then and some old scripts might not run in newer versions of Praat if some new commands and/or functions are used[36]. The changes in the syntax became stable after version 5.3.65, so any version of Praat _older_ than 5.3.65 should deal well with the new syntax.
a. Write a stand-alone procedure (a Praat script ending in ".proc") called "version_check.proc" and save it inside the folder "version_check", inside your "files_and_code" folder.
b. Your procedure should aim to obtain the current version of Praat and terminate the script if the version is older than 5.3.65.
c. You'll need one of Praat's predefined variables[37]: *praatVersion* or *praatVersion$* (I recommend you to use the former).
d. Create another short script with any dummy lines of code (print something to the Praat Info window, for example), but make it call your procedure as the first thing it does (check: "9.4. Include procedures and other scripts"). Save this script as "include_variable_check_proc.Praat" inside your "version_check" folder, which is inside your "files_and_code" folder.
e. Make sure that your procedure works by modifying the conditional jump inside the procedure to ask for different versions of Praat (for example, make your procedure terminate the script if the version is too new, just for the sake of testing your procedure and script).
If you get stuck, check some proposed solutions inside your "version_check" folder, in "include_variable_check_proc_SOLUTION.praat" and in "version_check_SOLUTION.proc".

36 Older scripts should always run in newer versions of Praat, but newer scripts will not always run in older versions.
37 Documentation here: http://www.fon.hum.uva.nl/praat/manual/Scripting_5_1__Variables.html

10.2. Arrays: a variable with many coexisting values

An array is a data type that contains a collection of elements (values), which can be accessed via indices. Praat has numeric and string arrays[38]. Arrays are useful whenever you need the same variable to contain several values (either numerical or string) at the same time. For example, arrays are a good way to store results of iterative processes, so that you can access all the steps of an iterative process even after it has finished and also from outside the process declaration.

In order to use an array, you have to add square brackets after your variable name. Inside those square brackets, specify the index where you'll assign a value and then assign the value. If we wanted to save the numbers 10, 15 and 20 into the indices 1, 2 and 3 of an array called "my_numbers", we could do the following:

```
my_number [1] = 10
my_number [2] = 15
my_number [3] = 20
```

In order to use these variables later, you have to refer to the variable including its index. To see the content of the three values of the "my_number" array, you could write something like this:

```
for i from 1 to 3
   appendInfoLine: my_number [i]
endfor
```

String arrays work in the same way. Naturally, you can use these arrays inside other functions and loops, as shown below:

```
for i from 1 to 5
   string_array$ [i] = string$ (i) + "_now_as_string"
   appendInfoLine: string_array$ [i]
endfor
```

*TTGYHD 40 (difficulty: **high**)*: Inside your "files_and_code" folder, go to "array_training" and create a script called "array_training.praat" that does the following:
a. It asks the user to find and open the file called "numbers_1-99_words.txt" – which is located in your "array_training" folder – as a String object. You'll need to use the *Read Strings from raw text file...* command.
b. It goes through the list of strings (clue: you'll need something from *Query -*) and saves each string inside a specific index within a Praat string array.
c. It calculates the length of each string and saves those results inside consecutive indexes of a numeric array.
d. It prints to the screen, separated by tabs, a counter or iterator that goes from 0 to 99, the string (extracted from your string array), and the length of the string (taken from your numeric array).
If you get stuck, take a look at "array_training_SOLUTION.praat" to see an example of how this can be done.

38 Documentation here: http://www.fon.hum.uva.nl/praat/manual/Scripting_5_6__Arrays.html

11. Where to look for more information?

To begin with, try to use Praat's documentation as much as you can. Although sometimes it's not very easy to follow, it contains all you could ever need to know regarding Praat's usage and scripting. Given that all the documentation can be found online, sometimes is faster to search for something online than using the manual that's included in Praat (start your search with something like “praat script ...”).

I can also recommend José Joaquín Atria's tutorial on Praat scripting, which can be found in his website (http://www.ucl.ac.uk/~ucjt465/assets/presentation/atria.psp.pdf), and there are more manuals for beginners here: http://www.fon.hum.uva.nl/praat/manualsByOthers.html.

There is a quite useful Praat User List (https://uk.groups.yahoo.com/neo/groups/praat-users/info), where many questions, both simple and complex, have already been asked, so you might find what you're looking for there. If you join this group, you can subscribe to receive aggregated summaries of all the new questions/answers made within a certain time range in the form of a digest. Reading these discussions is very useful to broaden your knowledge of Praat's capabilities and uses; also, you get to know current trends in the development of the software and, hopefully, you get to contribute as well.

Acknowledgements

I'd like to thank the following people who have contributed to create and improve this manual: thanks to Michèle Pettinato and Steven Gillis, for inviting me to teach this Praat scripting workshop in the Department of Linguistics of the University of Antwerp; many thanks to Albert Lee and Sonia Granlund for proofreading the entire document (v. 1.2), and to Ilke De Clerck and again to Michèle for spotting some writing problems. Also thanks to JJ Atria and Paul Boersma for helping me to correct/update some small problems present in version 1.4. of this manual.