Session 1 (09:00 – 13:00, Wed.): Praat scripting 101

- Preliminaries: checking installed software; how and where to edit code.
- Writing a script: overview, “hello world”, variables (string and numerical), conditional jumps, loops (for, while, repeat), functions and pseudo-arrays.
- Scripting challenges: small scripting challenges to practice typical programming challenges, with increasing difficulty.

1.1. Preliminaries: salutations

- Welcoming remarks.
- Presentations: name and what do you do.
- Previous experience relevant to this workshop: what type of studies do you normally conduct or want to conduct, your programming background, etc.
- Name one expected outcome for this workshop relevant to your research; try to be specific and to give some details.

1.2. Preliminaries: Checking technical requirements

- Computers on and running.
  - Access to the internet working.

- Latest version of Praat (v. 6.0.05) installed and running.
  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)

- Sublime Text 2 or Sublime Text 3 installed and running.
  http://www.sublimetext.com/3

- Download and install Praat's syntax highlighter for Sublime Text:
  https://github.com/mauriciofigueroa/praatSublimeSyntax

- Download materials for each session here:
  http://www.mauriciofigueroa.cl/workshop_kent/

- You'll need headphones for the second and third day.

1.3. Typical work-flow and first approach to Praat.

- Let's open Praat!
- When you open Praat, you see two windows appearing.
  - Picture window:
    - It shows you a canvas where pictures can be drawn.
    - We'll cover some of the capabilities of this window later.
    - Normally you'd just close this window to keep your windows hygiene.

- Object's window:
  - It looks rather empty and unfriendly: it is.
  - Its name gives away its purpose and a key component of Praat's functionality:
    - **Praat handles objects**.
    - It doesn't do much unless you create or load an object.
    - Let's explore the menus as they are.
  - Let's load a couple of files from our “companion_folder_session_1”. In particular, these files: “00_test_1.wav ” and “00_test_2.TextGrid”.
    - How to open files? You can save some time by using “Ctrl. + O” or “Command + O” in Mac.
  - Notice how the Objects window now has some, ehrm..., objects.
  - Notice that these Objects have an **ID number** and a name.
  - Notice how the menus and buttons available in the Objects window change depending on the selected object or group of objects.
    - Notice that selecting more than one object is also possible and that it enables some options that are otherwise not available.
    - **Important lesson**: You can only access to the commands and functions for a given object if it is selected (both using the Objects Window or from within a script). Selection management is a central part of Praat's optimal usage.
  - Let's explore for some minutes the options for each object.
    - You can learn a lot about what Praat does and what can do by just spending some time exploring the options for each object.
      - Notice how some commands and functions look very useful!
      - Let's talk about the difference between nothing, “`...`” and “ `-` ”.
    - Actually, you'll need to become good at this if you want to access the full capabilities of Praat programming.

1.4. Writing a script

1.4.1. Overview

- We can imagine a Praat script as a series of instructions to Praat, that could all actually be conducted manually by clicking and entering values as arguments into commands.
- For example: We could write a (rather useless) script to open our sample files one by one and then display them together.
- In a way, a script is nothing else than a “**virtual pointer**” (I believe this term was coined by JJ) so that the computer does all the clicking for us.

- Behind the scenes, Praat actually works with script-like instructions, which you can access by pasting the **history** into a new script.
  - This can be done by creating a new script: `Praat > New Praat Script` and then choosing `Edit > Paste history`.

- But then, **why bother writing a script on the first place**? Amongst other reasons:
  - Reproducibility:
    - You can just re-run your script and get exactly the same results (and also so your peers). In science, you need to ensure the full reproducibility of your results.
    - You can re-run your analyses many months or years later, when you forgot most of what you were doing. You'll be happy to have a script.
  - Efficiency:
    - A script can execute instructions many many many times faster than a human.
    - Repetitive steps can be automatized.
    - You can adapt old scripts to new tasks, saving a lot of time in the process.
    - You can break enormous and difficult tasks into small manageable pieces and go at them one by one.
  - Accessing Praat's full potential:
    - Many tools and functions become truly available only through scripts (e.g., string or mathematical functions).

1.4.2. "Hello world"

- How are we going to write scripts on this workshop:
  - Let's take a look at our first unofficial script.
  - We only have a **plain text interface**, with several problems:
    - It lacks line numbering (sigh).
    - There is no syntax highlighting whatsoever.
    - There are no writing aids (e.g., autocomplete).
    - There are no selection aids or hints.
    - There are no advanced editing tools (e.g., search by regular expressions).
  - We won't use this interface to write scripts. We'll only use it to load scripts and update them.
- Let's go to **Sublime Text** and let's write the following code into it:

```
appendInfoLine: "Hello world!"
```
(1.1)

- So far, it doesn't look tremendously different from the plain text interface, but we have line numbering and all the tools from the text editor itself.
- Save that script into a location where you can find it later easily, and make sure you save the script using a name with **no spaces** and with the ".Praat" extension (e.g., "hello_world.Praat").
- Magic! Once you saved the file with an extension, Sublime Text recognizes the **syntax highlighter** that has to use and the plain text comes to life.
  - You can also select the syntax highlighter manually by using your mouse.

- Now let's go to Praat and let's select `Praat > Open Praat Script...`, and browse and find your script, and see it there all open and nice.
- Select `Run > Run` to execute it, or just press "Ctrl. + R" (or "Command + R" in Mac). **Voilà**, you've just written and executed your first script (if you hadn't done that before).

    - What does that particular script do? I just prints whatever is enclosed within quotation marks into the Info window.
- This is how we're going to operate henceforth: We'll write (and edit) our scripts in **Sublime Text**, always, and then we'll open them in Praat and run them.
    - You won't believe the number of people that write and edit scripts in plain text editors...
- If you need to change your script, you do so in Sublime Text, you save your script, and then, in Praat, you just go to your Script window and select `File > Reopen from disk`.
    - After this your script is reloaded and updated, and you can run it again as usual.

1.4.3. Variables

- In computer programming, a variable is a place in the computer's memory where something is stored.
- Here, we'll understand a variable as the result from an **assignment** of a certain value (numerical or not) to a tag or identifier (its name).
- We can then access the content of the variable by calling the variable's name, as many times as we wish.

- Some restrictions for Praat variables:
    - The first character of any variable name has to start with a lower-case character from the English alphabet.
    - From the second character onwards you can also use upper-case characters, numbers, dots and underscores (try not to use dots; it'll become clear why later).

- **Numeric variables**: Contain any real number (integers, decimals, negatives, octals, etc.). To assign “1” to a numeric variable we can do:

```
this_variable = 1
```
(1.2)

- You can also assign the content of a variable to another variable (you'll do this a lot):

```
this_variable_a = 1
this_variable_b = this_variable_a
```
(1.3)

- And you can do with numeric variables whatever you'd do with numbers:

```
hidrogen = 1
oxigen   = 1
water    = (hidrogen * 2) + oxigen
writeInfoLine: water ; why not "appendInfoLine"?
```
(1.4)

- **Time to have some fun 1.1**: Play with numeric variables for a while. Get used to assign numbers of all sorts to variables and to manipulate these variables as you would with numbers.

- **String variables**: These variables contain text. To assign the text “moon” to a string variable you can write:

```
my_variable$ = ”moon”
```
(1.5)

- Notice that there is a **dollar sign** after the variable's name. This indicates to Praat that this variable is a string variable and not a numeric one. Also, the text to be assigned, if entered as such, has to be enclosed in **double quotation marks**.
  - Single quotation marks serve a specific purpose in Praat (they enable variable substitution), which is no longer recommended and should be avoided whenever possible. You'll see a lot of variable substitution in old scripts, but run away from it.

- You can do all sorts of interesting stuff with string variables:

```
tomato$ = ”the”
potato$ = ”ultimately”
zapato$ = ”42”
appendInfoLine: tomato$ + ” ” + (potato$ - ”ly”) + ” = ” + zapato$
```
(1.6)

- Be careful when distinguishing numeric and string variables. It can get confusing and lead to bugs:

```
number$ = ”1”
number  = 2
appendInfoLine: number$ + number
```
(1.7)

- Choose your **variable names** carefully. Make sure that they are as short as possible while meaningful at the same time.
- Some programmers prefer variable names with several words separated by underscores as in `this_is_a_variable_name` and others prefer using camel case as in `thisIsAVariableName`.
  - I use underscores for variable names and camel case for procedure names, just to make them visually different.

- **Time to have some fun 1.2**: Play with string variables now. Again, assign some short strings to string variables (don't forget the dollar sign), and see what operations are tolerated by strings. For instance, what happens when you subtract one string variable from another? Is this allowed in some cases only?

1.4.4. Conditional jumps

- Conditional jumps are analogous to **gates** that allow you to control the direction that your script will take. If certain predefined conditions are met, one door opens and another one closes, forcing your script into a certain route.
- Let's analyse the structure of this example until we fully understand how it works:

```
key = 5
if key < 3
  appendInfoLine: "Your number is tiny."
else
  appendInfoLine: "Your number is not tiny."
endif
```
(1.8)

- Notice that a conditional jump has got a predefined and mandatory structure, which includes starting with `if` and closing with `endif`. After the opening `if` you have to state your evaluation (the condition that hast to be met for that gate to open = become `TRUE`).

- You can use boolean flags instead of evaluations as well (the value “0” stands for `FALSE`, and any other real number stands for `TRUE`:

```
if 1
  appendInfoLine: "This is TRUE."
endif
```
(1.9)

- As in any other programming language, you can have **more than one evaluation** per conditional jump:

```
if 42 == 42 and "tomato" <> "potato"
  appendInfoLine: "Conditions have been met."
endif
```
(1.10)

- You can have as **many forking paths** as desired:

```
variable = 12
if variable == 12
  appendInfoLine: "I like this number."
elsif variable < 12
  appendInfoLine: "I don't like this number that much; too small."
else
  appendInfoLine: "I really like this number a lot."
endif
```
(1.11)

- Notice that you have to **declare** the evaluation each time, with the exception of the “else” statement, which encompasses all other cases not declared elsewhere.
- You can **nest** conditional jumps:

```
variable = 1234
# First conditional jump.
if variable > 1000
  appendInfoLine: "This number is higher than 1000."
  # Nested conditional jump.
  if variable < 1500
    appendInfoLine: "This number is also lower than 1500."
  endif
endif
```
(1.12)

- **Time to have some fun 1.3**: You have some minutes to create a short script (in Sublime Text) that uses at least one numeric variable, one string variable, a conditional jump, and prints something to the Info window.

  - Let your imagination roam free and just give it a go; at this stage is more interesting to see why a script failed to having a working script.
  - Let's tak a look at your scripts!

1.4.5. Loops

- Loops execute a task a certain number of times. How many times exactly has to be defined in the declaration.
- **For loops**: They execute a task as many times as specified in the declaration. Let's examine an example:

```
for i from 1 to 11
  appendInfoLine: 11 - i
endfor
```
(1.13)

  - These loops have the following structure:
    - They are declared by writing `for`, then the name of the iterator variable (in this case `i`) then the beginning of an iteration range (in this case `1`), and finally the end of the iteration range (in this case `11`).
    - They are closed with `endfor`.
    - Whatever is in between is going to be executed as many times as defined in the iteration range.
  - Some details:
    - The variable `i` is conventionally used for iterators, but you can choose whichever variable name you prefer.
    - If your range starts from 1, you don't need the `from 1` part.

  - The most difficult part in understanding *for loops* is to fully comprehend what the **iterator** variable is and what it does.
    - The iterator is a numeric variable that changes at each loop to contain the corresponding value from the range defined in the declaration.
    - As any other numeric variable, you can use it inside the loop, just as we did in our small example.
  - Instead of any numbers, you can use variables for your declarations:

```
start = 1
end   = 11
for i from start to end
  appendInfoLine: end - i
endfor
```
(1.14)

- **Time to have some fun 1.4**: You have something some minutes to create a *for loop* that counts from 0 to your current age in years minus 1 and that prints a line into the Info window for each iteration, that should look something like the lines you'll be shown.
  - You'll need commas to separate elements in your `appendInfoLine` command.
  - Bonus points if you manage to add a conditional jump (and “if” evaluation) that allows you to write “year” instead of “years” just for the 1st year.

```
I used to be 0 years old.
I used to be 1 year old.
I used to be 2 years old.
...
```
(1.15)

```
clearinfo

my_age = 31
for i from 0 to my_age - 1
  number_year$ = if i == 1 then "year" else "years" fi
  appendInfoLine: "I used to be ", i, " ", number_year$, " old."
endfor
```
(1.16)

- An **inline conditional jump** was used in the example above.
  - They are used to assign the result of an evaluation to a variable, which can be a numeric or string variable. Notice that, after the variable's name, we assign the result of the evaluation with an equals sign. Then, the declaration starts with `if`, then the condition is made explicit (`i == 1`), then we use `then`, the result that we want to assign to the variable if that evaluation is TRUE, then `else`, which is the result of the evaluation when it turns out to be FALSE, and finally `fi` to close the structure.
- Inline conditional jumps can also be nested!

- **While loops**: These loops iterate as long as the conditions defined in the declaration remain TRUE. This evaluation occurs at each iteration.
  - They are very useful when you need to iterate through something about which the number of iterations is not know beforehand.
  - As an example, let's divide a number by 1.25 (that is, not by much) until it becomes smaller than 5:

```
clearinfo

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

# How can we add a counter that shows us the number of iterations? :D
```
(1.17)

  - Notice that the last number that appeared in the Info window is indeed smaller than 5, and that when this condition was met, the loop stopped.
  - Also, notice that, for a while loop to work, one of the variables being assessed in your while loop's declaration has to be modified from **inside the loop**.
    - Where does this happen in the previous script?
    - This might be somehow counter-intuitive, but you'll find yourself doing this a lot when you write scripts.
  - It is very easy to inadvertently create an **infinite** loop when using *while loops*, so use them carefully and test them thoroughly.

- **Repeat loops**: These loops iterate until the condition, which is stated at the end of the structure, becomes FALSE.
  - The main difference between “while” and “repeat” loops is that *while loops* might eventually not be executed even once (if the initial condition is never TRUE), whether repeat loops will always be executed at least once, give that the evaluation occurs at the end.
  - Let's replicate the example which used the while loop to do the same, but now with the repeat loop. Let's take a good look at its structure and differences.

```
input_number = 142857142867
repeat
  input_number = input_number / 1.25
  appendInfoLine: input_number
until input_number < 5
```
(1.18)

  - Besides the declaration being made at the end, do you notice any other difference? What about the direction of the signs in the declaration?
    - Why is this?

- **Time to have some fun 1.5**: Let's imagine a real script where the difference between while and repeat loops could be crucial. Can you think of an example from your own research?

- **Time to have some fun 1.6**: You have some minutes to adapt your previous “for loop” script to now work using a *while loop* structure and then adapt it again to work with a *repeat loop* structure.
  - Remember that your script show print the ages from 0 and upwards.
  - Save your work from time to time so that, if you crash Praat, you can restart it and access your script again quickly.
  - We'll choose some of the successful and unsuccessful scripts as examples and we'll discuss them as a group.

```
my_age = 0
while my_age < 33
  number_year$ = if my_age == 1 then "year" else "years" fi
  appendInfoLine: "I used to be ", my_age, " ", number_year$, " old."
  my_age += 1
endwhile
```
(1.19)

```
my_age = 0
repeat
  number_year$ = if my_age == 1 then "year" else "years" fi
  appendInfoLine: "I used to be ", my_age, " ", number_year$, " old."
  my_age += 1
until my_age > 32
```
(1.20)

1.4.6. Functions

- A function in Praat is a short script that takes in arguments and produces an output. There are tons of functions already built for Praat.
- Normally, you'd want to save the result of a function into a variable so that you can use the result later.

- Let's see some examples of some **mathematical functions** (but there are many more):
    - `abs(x)`: Provides the absolute value from a number.
    - `round(x)`: Provides nearest integer.
    - `sqrt(x)`: Provides the square root.
    - `min(x, ...)`: Provides the minimum number from a list.
    - `max(x, ...)`: Provides the maximum number from a list.
    - `exp(x)`: It exponentiates *e* by `x`.
    - `randomInteger(min, max)`: It provides a random integer value between the defined range.
    - `hertzToBark(x)`: It transforms a value from raw Hertz to Bark-rate (analogous functions for other transformations are: `barkToHertz`, `hertzToMel`, `melToHertz`, `hertzToSemitones` and `semitonesToHertz`).
    - Let's take a look at the full list online!

- Let's see some examples of **string functions** (but there are many more too):
    - `length(a$)`: Provides the length of the string.
        - Did you notice?
    - `left$(a$, n)`: Gives back a string that has the first n characters of `a$`.
    - `mid$(a$, i, n)`: Returns a string of length `n` from `a$`, starting from character in position (index) `i`.
    - `right$(a$, n)`: Gives back a string that contains the rightmost `n` characters of `a$`.
    - `index(a$, b$)`: Gives you the index (the place in the sequence of characters) of the first occurrence of the string `b$` in the string `a$`.
        - Did you notice now?
    - `rindex(a$, b$)`: Gives the index of the last occurrence of `b$` in `a$`.
        - What about now?
    - `string$(number)`: Transforms a number into a string format. It digests exponential notation and percentages!
    - `number(a$)`: Interprets a string as a number.
        - Maybe now?
    - `fixed$(n, p)`: Formats `n` as a string with a decimal precision defined by `p`.
    - Let's take a look at the full list online!

- What if the function you're looking for doesn't exist? You can always create your own using procedures, which we'll cover later.

- **Time to have some fun 1.7**: Write a new script from scratch that uses string and mathematical functions to determine whether the number contained in the string “my_sweet_17” plus a randomly generated integer between 1 and 1313 is an odd or even number.
    - Before start writing, spend some time thinking how this big task can be broken down into smaller pieces.
    - Your script has to start with the line: `input$ = "my_sweet_17"`.
    - Your script has to report its results to Praat's Info window.
    - You'll need conditional jumps and the integer division operator `mod` to make things work, besides functions mentioned and explained above.

```
clearinfo

input$    = "my_sweet_17"
number    = number(right$(input$, 2))
my_random = randomInteger(1, 1313)
remainder = (number + my_random) mod 2
result$   = if remainder == 1 then "odd" else "even" fi
appendInfoLine: string$(number) + " + " + string$(my_random) + " = " +
  ... string$(number + my_random) + " = " + result$ + "."
```
(1.21)

1.4.7. Pseudo-arrays

- An **array** is a data type that contains a collection of elements (values), which can be accessed via indices. Praat has got numeric and string arrays.
  - Arrays are a good option when you need to store the results from iterative processes.

```
my_number[1] = 10
my_number[2] = 15
my_number[3] = 30
```
(1.22)

- Can you imagine a scenario in which using arrays might be useful? Let's think about this for a bit.

1.5. Scripting challenges

- **Time to have some fun 1.8**: Choose any of the following scripting challenges (taken from “https://adriann.github.io/programming_problems.html”, sometimes with slight modifications) and give it a go by using Praat scripting exclusively. Try as hard as you can and then let's discuss your results.
  - Write a script in whose first line a number `n` is assigned into a numeric variable, and prints the sum of the numbers `1` to `n`.
  - Write a script that tests whether a string is a palindrome.
  - Write a script that takes a number and returns a list of its digits.