Session 2 (14:00 – 18:00, Wed.): Bursting the script bubble properly

- Outside the script bubble: navigating, creating and querying Praat objects; forms, working folders and paths, selecting and creating files and directories, writing procedures, and using other Praat scripts within scripts.
- Quality control: good scripting practices; testing and debugging (printing, pausing and exiting); good practices for long term and big projects (modularity).
- Scripting challenges: writing simple scripts and procedures within scripts to manipulate and query Praat objects; predicting the user's behaviour and act accordingly.

2.1. Outside the script bubble

- Let's load a couple of files from our “companion_folder_session_2”. In particular, these files: “00_test.wav ” and “00_test.TextGrid”.

2.1.1. Navigating Praat

- When using Praat, you'll always be working with objects and is paramount that you're able to select exactly the object that you need for a given task and not any other object.
  - When using the mouse is relatively easy (for small projects) to keep track of objects and select exactly what you need.
  - You won't be able to see the Objects window when working from a script. How to do it then?
- Each object in Praat can be individualized via its **ID number**, the **object type** or by its **name**.
  - To select an object via its **ID**, you can use the command `selectObject`, which requires as an argument the ID of the object or objects you want to select. This would look like this:

```
selectObject: 1
```
(2.1)

  - To select an object via its **name**, you can use a similar line:

```
selectObject: "Sound 00_test"
```
(2.2)

  - The lines of code that we just saw work fine for our examples, but they are useless in a real scripting scenario, and they are prone to cause problems.
    - Firstly, in those lines, you need to, somehow, know the ID number of your object or its name from before so that you can write it on your script.
    - Also, if you have two objects with the exact same object type and name, Praat will only be able to select the last one opened. Names are not necessarily unique (the ID is!).
  - To solve many problems, from now onwards, **assign the ID number to a variable** and then refer to that variable to select your objects.
    - Open the script “01_saving_ids.Praat”, which is located in your “companion_folder_session_2” folder, by using `Praat > Open Praat script...`.

```
# Opening files and storing ID into variables.
wav_id = Read from file: "00_test.wav"
tgd_id = Read from file: "00_test.TextGrid"

pauseScript: "Inspect the Objects window."
selectObject: wav_id
pauseScript: "Inspect it again."
plusObject:   tgd_id
```
(2.3)

- In Praat, **new objects are selected by default**. Whenever you open or create a new object, Praat will select it.
  - If you open or create several objects, the last one will always be selected. Keep this in mind when writing your scripts!
- You can also access the ID number and name of an object by using the following functions, which only require, besides the correct object being selected, the object's type:

```
id_number = selected("Sound")
name$     = selected$("Sound")
```
(2.4)

- Storing the name of an object is very useful when you need to use that name later on, for example, when you are saving stimuli from an original sound, or to create a folder with a name, etc.
- Useful functions to select and remove objects by their ID are:
  - `selectObject`: selects an object or several objects.
  - `plusObject`: adds objects to the current selection.
  - `minusObject`: removes objects from the current selection.
  - `removeObject`: removes objects from the Object window.

- **Tip 1**: It's a good practice to **remove objects** that you finished using from the Objects window, to keep everything tidy and also because you're just wasting your RAM memory if you keep objects open just because (particularly if they are large).
  - There is a limit of 10,000 for the number of objects that Praat can open at the same time.
- **Tip 2**: Praat never ever saves anything unless you explicitly tell it to do so. If you're editing a TextGrid and spend 2 hours on that and your computer crashes, all that work will be lost forever. **Save often**.

2.1.2. Creating and querying objects

- You can create an object in Praat by accessing a file (opening it), by creating it from scratch or by creating it from another object.
  - **Opening files**: We've been doing this already. Let's explore again the default options for opening objects in Praat by looking at the Objects window, particularly under `Open`. For a scripting example, see the following:

```
wav_id = Read from file: "00_test.wav"
```
(2.5)

- **Creating it from scratch**: The objects that you can create from scratch are clustered under `New` in the Objects window. For example, you can create a table with column names and no rows (see first the function and then show script):

```
table_id = Create Table with column names: "my_table", 0, "A B C"
```
(2.6)

  - Notice how you have to provide the arguments that you would otherwise enter manually separated by commas.
    - Let's see how this would go manually!
  - Strings are entered between double quotation marks, numbers without them. We'll see more advanced examples of this later.

- **Creating it from another object**: Some objects can only be created from another object. For example:

```
wav_id   = Read from file: "00_test.wav"
pitch_id = To Pitch: 0.0, 75.0, 600
```
(2.7)

- In this case, the object Pitch can only be created from a Sound object. Notice that the Sound object was selected by default when opened, which which allowed Praat to create a Pitch object.

- Once you have any type of object in your Objects window, you can do whatever that objects allows you to do, including **querying it** or **modifying it**.
  - For example, for a Sound object, under `Query -`, you can obtain its total duration by using `Get total duration`.
    - By the way, this isn't the safest way to query the total duration of a sound; prefer subtracting the start to the end. Why?
  - For a Pitch object, you can obtain the mean pitch by using `Get mean`, which requires you to specify the time range and the type of unit of measurement.

  - I insist that it is a very good idea to spend some time taking a look at the options that each Praat object gives you, particularly for those that you use often. You'll learn an awful lot about them by doing this and you'll also save yourself a lot of time by not writing functions and procedures that already exist.
    - Case study: Chi-square.

- **Time to have some fun 2.1**:
  - Write a script that:
    - (a) Opens the sample sound "00_test.wav" and TextGrid "00_test.TextGrid" that are located in your "companion_folder_session_2" folder.
    - (b) Creates a Formant Object for the sound object using the default settings for the Burg method
    - (c) Queries the TextGrid to find the start and end point of the word /'ka.da/ and stores these values into variables.
    - (d) Measures the mean F1 and F2 values for that word in particular.
    - (e) Reports these results to Praat's Info window.
  - You'll need to open the objects and take a look at the options that Praat gives

you to find the tools you'll need to create the Formant (Burg) object and to query the objects.
    - Some clues:
        - The TextGrid that you have has only 1 tier and that tier is an interval tier.
        - Formant objects are nothing else than spectra shown as a function of time. “Spectra” is the plural of “spectrum”.

```
Read from file: "00_test.TextGrid"
start = Get start point: 1, 4
end   = Get end point: 1, 4
Read from file: "00_test.wav"
To Formant (burg): 0.0, 5, 5500, 0.025, 50
mean_f1 = Get mean: 1, start, end, "Hertz"
mean_f2 = Get mean: 2, start, end, "Hertz"
appendInfoLine: "F1 = ", mean_f1
appendInfoLine: "F2 = ", fixed$(mean_f2, 3)
```
(2.8)

2.1.3. Forms

- Oftentimes the scripts you'll be writing will only be used by you, and if the script serves a rather specific purpose or it isn't going to move much from hand to hand, then it is easy to modify a couple of lines each time you want to use it again (for example, if paths to files need to be modified).
- Many times, however, you want your script to be able to receive **arguments** from users without them having to edit the script. Then is when **forms** become very handy (and they'll prove even more useful later).
- A form is a **pop-up** that prompts before the script actually runs, and it will show up at the beginning regardless or where in the script you actually wrote the form.
    - Conventionally, scriptors tend to write the code for forms at the beginning of scripts, to reflect the fact that they show up at the beginning.
    - Forms are capable of having text fields and other sort of input fields and buttons so that the user can enter stuff.
- Forms can receive the following arguments (documentation here): `real`, `positive`, `integer`, `natural`, `word`, `sentence`, `text`, `boolean`, `choice` & `buttons`, and `comments`. Forms look like this:

```
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
```
(2.9)

- The form starts and ends with the form declaration `form` and `endform`. All the lines in between correspond to either comments or fields that receive arguments. Firstly, the type of field is declared (e.g., `integer`) and then a variable name is declared (e.g., `Name`).
    - Notice that for you can enter **default values**, as we did for `real`, where by default a value of 100 was entered. Also, a help for the user was included in brackets.

  - The brackets from clues and the colons after the variable declarations disappear when you then use the variables.
  - Also, you can use capitals in the form for the variable names (unlike anywhere else in Praat scripts), but then to access the variables you need lowercase.
- Variables can be accessed later, like this:

```
appendInfoLine: name$
appendInfoLine: age
appendInfoLine: sex, tab$, sex$
appendInfoLine: smoothing_factor
```
(2.10)

2.1.4. Working folders and paths

- When Praat starts, it has got a **default directory**, which will most likely not be the place where you have stored your script or the files that your script will try to access.
  - The location of this default directory can be revealed by calling a predefined Praat function called `shellDirectory$`.
- If your script has been saved and you open it in Praat using `Praat > Open Praat script...`, Praat will reset your **working folder** to the folder that contains the script.
  - This has important consequences: if your script needs to open a specific file by name, as we have been doing above, then the file will need to be located in the same folder than the script.
  - Alternatively, you'll have to specify a path relative to the script's location, but this relative path needs to be known too, and it will only work if the script is loaded properly and not just copied and pasted to a new script window.
- To know the location of the new default directory, once you read a script, you can call the predefined function: `defaultDirectory$` (you don't really need to do this very often).
  - Other interesting similar predefined functions are: `homeDirectory$`, `preferencesDirectory$` and `temporaryDirectory$`.
- One **solution** to this mess would certainly be to (a), provide the script along with the files that it will have to modify; (b) ask the user to edit the script and provide a full path to the files the script is supposed to read; or (c) to ask the user to enter the full path into a form.
  - None of these solutions are optimal, though. The first one affects the portability of the script. The second one requires a user that knows how to edit scripts and do it properly. And the third option is better, but a bit tiresome.

- **Time to have some fun 2.2**: Write a script that (a) contains a form that (b) asks the user, separately, for a full path to the WAV file we've been using and the filename; (c) uses that path and filename to load the WAV file into a Sound object; (d) Plays the sound once loaded.

```
form User settings.
  comment Please enter the following arguments:
  sentence Path: /YOUR/PATH/HERE/
  word Filename: 00_test.wav
endform
Read from file: path$ + filename$
Play
```
(2.11)

- The difference between **fixed or hard paths** and **relative paths** is quite simple: the former direct to a specific and fixed place in your computer's memory and the latter directs to a place in the memory relative to the current working directory.
  - It is generally recommended to avoid fixed paths because it is very unlikely for two computers to have the same file structures. A full path (in my computer) to a script that we'll use in this session is:

```
Open Praat script: "/home/mauricio/talks_lectures_workshops/2016_07_06-
  ...08_workshop_Kent/companion_folder_session_2/efficiency/efficiency_1.praat"
```
(2.12)

  - Relative paths should always be favoured. If relative paths cannot be used, then it is recommended to ask the user to enter a path by using a form or by making users choose a folder or file manually.
  - A path, relative to the folder where “efficiency_1.praat” is located (which became the default folder when we loaded the previous script), could be:

```
Read from file: "../00_test.wav"
```
(2.13)

  - Notice that the “relative” bit was made explicit by using `../` for each level that Praat has to climb or go back in order for the path to make sense. So, it climbed a folder from the folder “efficiency”, ending up in “companion_folder_session_2”, where the file “00_test.wav” exists and thus can be accessed.
    - Just, you can climb how many folders as you want, as long as you use a sequence of “`../`”.

2.1.5. Selecting and creating files and directories

- Another alternative is to ask the user to **choose** a file or a directory directly, by using `chooseReadFile$` or `chooseDirectory$`. These commands will prompt a file or directory opening window and the user will be asked to navigate to a file or directory.
- Using these commands is not complicated:

```
file_name$ = chooseReadFile$: "Open a CSV file"
if file_name$ <> ""
  table = Read Table from comma-separated file: file_name$
endif
```
(2.14)

- All you need to do is to assign the result of the action of choosing a file (or a directory, if that was the case) to a string variable. That variable will contain the full path to the file and the filename too. Then, you can do with that full path and name whatever you want to do with it, normally opening it.
- Another useful function is `chooseWriteFile$`, which will prompt a **saving** window and requires the user to choose where to save the object that is currently selected.
  - Warning: It is your duty to save the object with an appropriate extension if you'll be using the file outside of Praat. Also, if you select more than one object at the same time, Praat may complain or (what's worse) will save the batch as a “Collection” object, which can only be read by Praat.

```
file_name$ = chooseWriteFile$: "Save as a WAV file", "sound_ID.wav"
if file_name$ <> ""
  Save as WAV file: file_name$
endif
```
(2.15)

- Each Praat object type (there are many) can be saved in different ways. Some objects which are fairly common outside Praat (e.g., sounds and tables) can be **saved in standard formats** such as WAV or AIFF, and as CSV, respectively.
  - When reading or writing a file, Praat completely ignores the extension of the file (e.g., ".wav"). But is a good idea to include them if the file will need to be read by an external programme.
  - Let's take a look at some of the options for some common objects: Sound, TextGrid, Table, Pitch, Formant, Intensity, Spectrum, Spectrogram and KlattGrid.

- Besides opening files and saving objects, Praat can alter them.
  - `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 specify 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", "This line will be the very first one."
```
(2.16)

  - `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".

```
writeFile: "myFile_B.txt", "This line will also be the very first one."
```
(2.17)

  - `appendFileLine`: It **appends** the specified line to the end of an already-existent file.

```
appendFileLine: "myFile_B.txt", "This line will be appended at the end."
```
(2.18)

  - `appendFile`: Does the same as appendFileLine, but without adding the newline symbol.

```
appendFile: "myFile_B.txt", "And yet another line will be appended at the end."
```
(2.19)

  - `createDirectory`: It **creates a directory** with the specified name and in the specified path. If the directory already exists, it does nothing.

```
directory_name$ = "hyper_praat"
createDirectory: directory_name$
```
(2.20)

  - `deleteFile`: It **deletes** the specified file or directory, if it exists.

```
deleteFile: "hyper_praat"
deleteFile: "myFile_A.txt"
```
(2.21)

```
deleteFile: "myFile_B.txt"
```

2.1.6. Writing procedures

- Procedures are **scripts within scripts**. They are very similar to what in other programming languages would be called a “function”. They can be very useful.
- The best way I've been able to come up with to explain why procedures are very useful is to make you imagine what would you do if a function like `appendInfoLine` **didn't exist** in Praat, but you need it anyways.
  - So, let's imagine that: You'd like to print stuff to the Info window, but there is no function to do that in Praat. Let's assume that you're also gifted in programming and that you manage to write a very short script that, in 10 lines, manages to do what you need: it takes a string and it prints it into the Info window.
  - So now **you have your own function**, written by yourself, and you use it a lot, because in several places in your script you want to, say, check the current state of some variables.
    - So there you find yourself copying those 10 lines of code every single time you need to use your script, and quickly your script becomes repetitive, unnecessarily big and perhaps **difficult to read**.
  - Wouldn't it be nice if you could somehow declare those 10 lines once and then just call it with one line whenever you need it (as you do when using `appendInfoLine`)? **Procedures** allow you to do just that.
- Let's imagine a procedure that takes two arguments, a string and a number, and that its duty is simply to shorten the length of your string by the number provided as second argument and print that string to the Info window. A procedure for that could look like this:

```
@dealingWithStrings: "laughlines", 5
@dealingWithStrings: "outguessing", 3
@dealingWithStrings: "loudspeaker", 4

procedure dealingWithStrings: .string_to_parse$, .how_short
  .short_string$ = left$(.string_to_parse$, .how_short)
  appendInfoLine: .short_string$
endproc
```
(2.22)

- The procedure is defined in the last four lines of the script. It starts with the definition `procedure` and then it receives a name (`dealingWithStrings`) and the name of the variables that will receive as arguments. Then you find what the procedure does and it closes with `endproc`.
- Notice that the procedure is called three times in the first three lines, and that each call requires only one line: You write the procedure once, but you call it as many times as you want.
- Also notice that the variables used inside the procedure were defined as (pseudo) **local variables** (as opposed to **global variables**). This means that we can't access those variables from outside the procedure, unless we use the following hack, in which the name of the procedure is used before the local variable:

```
dealingWithStrings.short_string$
```
(2.23)

- Using local variables is a very good idea. If you don't use local variables, you might end up getting yourself in trouble.

- **Time to have some fun 2.3**: The script copied here does something rather silly. Its aim is to loop through a series of numbers, decompose each of those numbers into individual digits (e.g., 12 is decomposed into 1 and 2) and then find out whether the individual digits can be divided without remainder by a number that is defined at the beginning.
  - As you can see, some of these tasks are carried out in a procedure, separate from the main script. That procedure is called `areTheyDivisible`.
  - Your task: Read the script that is copied below and – without running it! – think carefully why this script will make Praat (and perhaps your computer) **crash**.
  - Once you think you know what's the problem, fix the script and run it. Be ready to explain what changes you made and why do you think they solve the problem.

```
start   = 14
end     = 30
divisor = 2
for i from start to end
  @areTheyDivisible: i, divisor
endfor

procedure areTheyDivisible: number_to_parse, divide_by
  number_as_string$ = string$(number_to_parse)
  length_string     = length(number_as_string$)
  for i from 1 to length_string
    this_digit = number(mid$(number_as_string$, i, 1))
    remainder  = this_digit mod divide_by
    result$    = if remainder == 0 then "yes" else "no" fi
    appendInfoLine: i, tab$, i, tab$, this_digit, tab$, remainder, tab$,
      ... result$
  endfor
endproc
```
(2.24)

  - Now that we know what we need to do to fix this script, let's dissect it line by line until we fully understand it.

2.1.7. Using other Praat scripts within scripts

- **Including** a Praat script into another Praat script (without copying one into another, of course) is quite simple. You basically have two options: using `runScript` or `include`. Let's start with `runScript`.
- Let's suppose that you have the following script:

```
# FIRST PIECE OF CODE ("03_script_being_called.Praat"):
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
writeInfoLine: "Your age is: ", age, ". Your sex is: ", sex$, "."
```
(2.25)

- This script has a simple form that asks for the age and sex of the user. Then, it clears the Info window and writes a string with the arguments entered by the user.
- You can run that script from another script using **runScript**, as shown below. This separate script runs the first one, all that is needed (in this case) is the name of the script and the arguments, if the first script has a form.
  - If the script being ran from another script hasn't got a form, you don't need to provide the arguments after calling the script.
  - Mind your paths! Only if you have opened the second script using `Praat > Open Praat Script...` and if the two scripts are stored in the same place these scripts will run well. Otherwise, you need to ensure that the paths are right.

```
# SECOND PIECE OF CODE ("03_script_that_calls.Praat"):
runScript: "03_script_being_called.Praat", 29, "Male"
```
(2.26)

- Running a script using `runScript` is similar to calling a **procedure with local variables**: Any variable assignment made in the script being called won't be available in the script making the call.
  - This is because a script being called by `runScript` is executed as an entirely different and independent process.

- Now, if you need a whole script and its assignations to become available from a calling script, you need to use **include**.
  - The result of using `include` is exactly the same than if you were to copy and paste the script or procedure that you want to include into your calling script.
  - The script or procedure that has been **included** is ran in the same process than the main script; they are, for all purposes, only one script.
  - You can see a simple example in the following two snippets of code:

```
# Procedure to be called ("04_procedure_which_is_included.Proc").
procedure howLong: .string$
  .this_long = length(.string$)
  writeInfoLine: .this_long
endproc
```
(2.27)

```
# Including procedure.
include 04_procedure_which_is_included.Proc

# Executing procedure.
@howLong: "pneumonoultramicroscopicsilicovolcanoconiosis"
```
(2.28)

- **Time to have some fun 2.4**: Make the snippets of code from above work in your computer. The files that you'll need are stored in your folder “companion_folder_session_2”.
  - Once you've managed to get these snippets of code to run, try to understand line by line what they are doing.

2.2. Quality control

2.2.1. Good scripting practices

- When it comes to writing scripts, there are good scripts and bad scripts. You definitely want to write scripts that are clear, well written, that you understand well, etc. If you don't keep attention to these details at the beginning, then you'll find yourself paying the price once you script starts to move from hand to hand, or when it grows or when you need to make it interact with other scripts.
- You'll see now the guidelines that I follow when writing scripts, which have been adapted from other similar lists, although the ordering and interpretation are mine.

- **It has one clear goal**: You script most have one very clear goal. The more clearly you know what you need to do, the easier it is to divide that big challenge into simple tasks and then write them using Praat's scripting language. If you don't know what you're supposed to be writing, then you're not ready to start scripting.
    - 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.

- **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.
    - It happens very often that you go back to a script that 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.
    - Let's illustrate this with a couple of examples:

```
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
```

(2.29)

(2.30)

```
# 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 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
```

- **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 values that variables are supposed to have at that particular point.
  - 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 very dangerous. Unfortunately this happens a lot!

- **It's kept tidy**: A script that is kept tidy is easier to read and maintain than a messy one. Let's take another look to the bad script from above, but now without spaces where there shouldn't be and with indentation. Still bad, but much better.

```
z = randomInteger (2, 100)
d = 0
for i to z
  q = i mod 3
  if q == 0
    d = d + i
  endif
endfor
writeInfoLine: d
```
(2.31)

  - Indentation has to be used meaningfully. Normally, a process that happens inside another will require +1 indentation. In the script above, what happens inside the loop has been indented, and then what happens inside the conditional jump has been indented again.
  - 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. I prefer using 2 spaces and I suggest avoiding tabs.

- **It's efficient**: 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 or a procedure. 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.
  - **Time to have some fun 2.5**: Let's take a look at three scripts stored in the folder "efficiency", inside your "companion_folder_session_2" folder. All of them manage to output the exact same result, however, some do it way more efficiently than others. Can you rank them according to efficiency?

- **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.
- **Something good, if short, twice as good**[1]: Well, this is sort of self-explanatory. You can compare the efficiency scripts again in case you find yourself sceptical about this.

2.2.2. Testing and debugging

- Writing lines of code produces **bugs** and there is nothing you can do about that. If we understand that bugs are going to happen regardless of our level of expertise, then the next step is to try to minimize their impact and try to detect them and fix them before they wreak havoc with your script.
- Some ideas that we've discussed before are already helpful to prevent bugs: use meaningful variable names (be explicit), understand your script and keep your scripting style tidy.

- **Send stuff to Praat's Info window**: When writing a script, one of the most simple ways to test your script's current state is to send stuff to Praat's Info window.
  - I would recommend that you do this often, particularly if you're a beginner.
  - Failing to check the content of your variables can have catastrophic consequences for long scripts where an error doesn't manifest until 25 lines of code later.
  - By the way, one typical source of bugs is failing to rewrite a variable name correctly, as in the example below:

```
suggestive_number = 1313
remainder_2 = sugestive_number mod 2
if remainder_2 < 2
  # Do something
endif
```
(2.32)

  - To send information to the **Info window** you can use the functions `appendInfoLine` and `writeInfoLine`, which are identical with the exception that the latter clears the Info window before adding the line.

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

- Normally, you send to the Info window the current state of an iterative process, or the result from conditional jumps, as in the script we played with when talking about the *while loop*:

```
clearinfo
input_number = 142857142867
counter = 1
appendInfoLine: "NUMBER", tab$, "VALUE"
while input_number > 5
  appendInfoLine: counter, tab$, input_number
  counter += 1
  input_number = input_number / 1.25
endwhile
```
(2.33)

- **Time to have some fun 2.6**: Go to your “companion_folder_session_2” folder and open the script “05_something_smells_fishy.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.
  - Use writeInfoLine and/or appendInfoLine and your common sense to find the bugs.
  - Fix the script until it works.
  - Leave an inline comment in your script (using “;") to remind you of each bug you found.

```
# This script creates 100 random integers and adds them together one
# after the other. It also divides the result of that addition by the
# randomly generated number. At each cycle of the for loop, a string
# is sent to the Praat Info window, which contains: (a) the iterator
# value, (b) the random value generated, (c) the partial sum, (d) the
# result of the division. Each line should look something like this:
# > 2   47     95     2.021276595744681

maximum_value = 100
dummy_variable = .0
for i from I to maximum_value
random_value = randomInteger(0,50)
dummy_variable = summy_variable + random_value
division_result = dummy_variable / random_value
appendInfoLine: i, tab$, random_value, tab $, dummy_variable, tab$,
  ... division_results
endfor
```
(2.34)

```
maximum_value = 100
dummy_variable = 0 ; Deleted "." before digit.
for i from 1 to maximum_value ; Corrected "from I".
    # All below indented.
    random_value = randomInteger(0,50)
    dummy_variable = dummy_variable + random_value ; Corrected "summy...".
    division_result = dummy_variable / random_value
    appendInfoLine: i, tab$, random_value, tab$, dummy_variable, tab$,
      ... division_result ; corrected "tab $" and "division_results"
endfor
```
(2.35)

- **Pause a script to observe a given state**: Another testing tool, particularly useful to explore the current selection in your Objects window, but also to assert the current stage of development of your script, is to use pauses.
  - The function you'll need is `pauseScript` (you might also see `pause` in old scripts). This function pauses the execution of your script momentarily and prompts a message which is defined with the function.

```
variable = 1313
pauseScript: "Variable equals ", string$(variable), ".", newline$
variable = variable - (variable / 2)
pauseScript: "Now, variable equals ", string$(variable), ".", newline$
```
(2.36)

  - A very very typical bug in Praat is to try to execute a command for an object different from the one selected in the Objects window. To prevent this problem, inspect your script thoroughly using pauses so that you know exactly what is selected at what point.

- **Make your script crash if behaved unexpectedly**: Whenever your script receives arguments from outside the script itself (be it from a user or from a file, etc.), there is a risk that the information that the script receives is different from the information it expects.
  - For example, you might have a script that requires the user to select a file. Users can choose to select a file or not, so it is perfectly possible that they choose none.
  - If that happens, instead of having your script crashing embarrassingly and with a lot of noise and error messages, it would be better to take that possibility into account and have your script commit suicide gracefully by using `exitScript` and a message to the user explaining the problem:

```
file_name$ = chooseReadFile$: "Open a CSV file"
if file_name$ <> ""
  table = Read Table from comma-separated file: file_name$
else
  exitScript: "No file was selected.", newline$
endif
```
(2.37)

  - You can do the same with variables from forms. For example, if you need your user to enter a value within a range, it is convenient for you to check that that condition has been met before running the script. You can evaluate that information and terminate the script if there is a problem, and you can let the user know what the problem was in the message contained in the `exitScript` function.

2.2.3. Good practices for long term and big projects

- It is difficult to give specific guidelines for long term and big projects, particularly because they can be very different from each other. For example, to which extent will scripts and files be shared can vary widely. Still, here I go, in no particular order:

- **Plan carefully**:
  - Visualize your gigantic task and plan how to divide it into smaller manageable pieces.
  - Make your pieces follow a logic structure and a work flow. For example, separate extracting data from analysing data.

- **Adopt modularity**:
  - Beginners tend to be afraid of modularity, because it demands an additional level of abstraction when writing a script and because it feels as if you're loosing some control over your scripts.
  - Modularity, however, is essential if you plan to undertake a large project such as the data extraction and analysis for a production study. If you need too many words to describe what your script does, then you definitely can split it into several pieces.
  - Use several scripts or external procedures to break your script apart. If you're using the same lines of code more than once, then they should constitute a procedure and you'll reduce those lines to 1 call.
  - Eventually, when you get really good at this, you'll start developing tools that you'll use all the time. You can see several examples of this here: http://cpran.net/plugins/

- **Define scripting standards and stick to them**:
  - Whether you'll be working alone or with more people, define scripting standards and stick to them.
  - Some people prefer indenting with tabs, other with spaces. Some prefer longer but clearer variable names, other people prefer shorter but more obscure names. Some programmers prefer underscores between words, other camel case.

- **Use abundant commenting**: Get used to comment your own scripts thoroughly. You shouldn't be surprised if you have to come back to your own scripts many times during long periods of time (months or even years). You'll be very thankful to your old self if you add good comments that make it easier to understand how your script works.
  - Use comments to head sections and subsections.
  - Comment those places of your script that are difficult to understand.
  - Keep your comments short and sweet (clarity is the goal).
  - Use line comments (#) and inline comments (;).

- **Have a tidy file structure**:
  - Take into account that paths can be a problem for scripts. Try to write flexible scripts that allow you to work efficiently.
  - File structure changes are typical in big projects: Make it easier to change paths and avoid hard paths (prefer relative paths).
  - Keep your scripts separate from other type of data.
  - Make your file structure to reflect your project's structure (use numbering in your folder names; do the same with scripts that deal with different stages of a process).

- **Back-up often**:
    - Not only of your current files, but also of previous versions of scripts and other documents that are iteratively modified.

- **Time to have some fun 2.7**: Think during some minutes about your next big study that will involve using Praat. If you haven't got one, come up with one that you'd be interested in conducting.
    - Divide that study into stages and be explicit about which type of scripts you'd need for each stage.
    - Think about how you could have a modular approach to programming that project.
    - Also, think about a good file structure that would allow you to keep things tidy but also that will help you to handle hard and relative paths.
    - Let's talk about that.

2.3. Scripting challenges

- **Time to have some fun 2.9**: Let's write a challenging script from scratch. You will build a script that will use Praat's SpeechSynthesizer to synthesize an English CVC word (e.g., “hat”, “had”, “got”, “pan”), segments and annotates that word automatically into a TextGrid, obtains mean F1, F2 and F3 values from the vowel, prints the results and a phonetic transcription for the whole word into Praat's Info window, and saves the same lines into a “.txt” file outside your script.
    - The script will have to ask the user for the word that the user wants to synthesize. In order to do that, you'll need to use a form.
        - Test that the word entered by the user is three characters long and that the middle segment is a vowel. If the string doesn't meet the required characteristics, make the script crash gracefully and send feedback to the user as to why the script crashed.
    - You'll need to take a look at `New > Sound > Create SpeechSynthesizer`. Use “English_rp” and voice “m1”. The synthesizer will provide you with the TextGrid and the phonetic transcription.
    - You'll have to navigate between objects in your script by resorting to their unique ID number, and to do that you'll have to store that ID somehow into variables.
    - Remember to extract the formant values from a Formant object (let's use the Burg method) and not using Praat's editor or manual analyses.
    - You'll have to use at least 1 procedure in your script. The procedure must be stored outside your script and it has to be included into your main script via `include`.