Python and Bioinformatics

Pierre Parutto

November 18, 2016

Contents

Chapter 1

Code And Data Files

In computer science files represent the memory of a computer allowing you to store information. There exist numerous types of files: text, images, videos, even softwares and the operating system are contained themselves in files. A first classification leads to the distinction of two kinds of files that you will necessarily be confronted to: files containing code and files containing data (DNA sequences, some text, numbers, ...). In this chapter we will see how to handle both types: to import functions and variables from other Python code files and read and write data in files.

1.1 Code Files

For Python code files, we are going to further distinguish two types of files:

Definition 1
- **Modules** *are Python code files containing only function and variable definitions;*
- **Scripts** *are Python code files that are meant to be executed, they produce some results (displaying some value, generating a file, ...).*

Modules are meant to group together multiple function and variable definitions. Usually in modules there are no code outside of the definitions (as when you send me an homework). In the other hand, scripts usually do not define many functions but possess only code for performing a specific task (for example the test files). In the following we focus on modules.

1.1.1 Importing a Module

Definition 2 *In the context of modules, a name is a function or variable definition.*

In the following subsection we consider the following Python file named *my_functions.py*:

```
def my_seq(n, u0):
    if n == 0:
        return u0
    return u0 * n * my_seq(n-1, u0) + u0

my_n = 3
```

We will write the code in a file named *my_script.py* that **is in the same directory** as the file *my_functions.py*.

Definition 3 *Importing a module consists in fetching one or multiple name(s) into the program.*

There exist three different ways of importing the names defined in *my_functions.py* into the file *my_script.py*.

1.1.1.1 `import`

The first way uses the syntax:

```
import filename
```

where `filename` is the name of a Python module **without the ".py"** at the end.

The import statement fetches all the names from the module *filename* and prefix them with *filename*.

Warning

Using the `import` method, all names are prefixed by the filename: `filename.name`.

Example In the file *my_script.py*:

```
import my_functions

print(my_functions.my_seq(1, 5))
print(my_functions.my_n)
print(my_functions.my_seq(my_functions.my_n, 1))
```

Produces the output:

```
30
3
16
```

1.1.1.2 `from - import`

The second way uses the following syntax:

```
from filename import name1, name2, ..., namen
```

Where `filename` is the name of a Python module without the *.py* at the end and `name1, name2, ..., namen` is a list of names to import.

This command imports **only** the names `name1, ... namen` from the module *filename*. It **does not** prefix the names.

Example In the file *my_script.py*:

```
from my_functions import my_seq, my_n

print(my_seq(1, 5))
print(my_n)
print(my_seq(my_n, 1))
```

Produces the output:

```
30
3
16
```

1.1.1.3 `from - import *`

The third way uses the syntax:

```
from filename import *
```

Where `filename` is the name of a Python module without the *.py* at the end.

This command imports **all** the names from the module *filename*. It **does not** prefix the names.

Example In the file *my_script.py*:

```
from my_functions import *

print(my_seq(1, 5))
print(my_n)
print(my_seq(my_n, 1))
```

Produces the output:

```
30
3
16
```

1.1.2 Choosing The Best Import Method

You must be very careful to avoid name clashes when importing some names. A name clash appears when you import a name that already exists, you can have defined it yourself in the file or already imported it from another file, in your file.

Here is how I advise you to use the different methods:

- `import`: When you want to import multiple modules that can contain the same names. The prefix allows you to access specifically a name from a specific module.
- `from - import name1, ..., namen`: this is, by default, the method that you should use.
- `from - import *`: this is the laziest method as is allows to directly import all the names from a module without a prefix. Use this method if you have only 1 import or if you know what you do.

To illustrate these points consider the following example with the files *a.py*:

```
var = 1
```

and the file *b.py*

```
var = 5
```

We will import both files in the file *code.py* with the three methods:

1. With `import`:

   ```
   import a
   import b

   print(a.var)
   print(b.var)
   ```

   Produces the output:

   ```
   1
   5
   ```

   Here we can access the variables `var` from both modules.

2. With `from - import name1, ..., namen`:

   ```
   from a import var
   from b import var

   print(var)
   ```

Produces the output:

```
5
```

Here only the last value of `var`, from the file *b.py*, is kept. With this method it is **not possible** to access at the same time both the values of `var` from *a.py* and *b.py*.

3. With `from - import *`:

```
from b import *
from a import *

print(var)
```

Produces the output:

```
1
```

This time I imported `var` from *a.py* after the one from *b.py* hence it is the value from *a.py* that is kept. With this method also, it is **not possible** to access at the same time both the variables.

1.2 Manipulating Data Files

In Python there exists a specific type to represent files, the type `File`. To manipulate a file, first you need to open it in the code, then perform the treatments and finally close it. Closing a file tells your Operating System (Windows, Mac Os, ...) that it can release the ressources affected to the file.

1.2.1 Opening A File

To open a file use the function open:

```
open(filename: str, mode: str) -> File
```

Where `filename` is the name of the file you want to open and mode is the manipulation mode. This function returns a value of type `File`. The manipulation mode depends on the treatment you want to do on the files, the main modes are the following:

Name	Mode	Effect if the file does not exist	Effect if the file exists
Read	`"r"`	Error (file not found)	Nothing
Write	`"w"`	Creates it	Erase its content
Append	`"a"`	Creates it	Add new content at the end

There exist multiple other modes that basically perform a combination of these modes, you can find the full list on the Python documentation here.

Warning

By default Python searches the file in the same directory as the script.

1.2.1.1 Example

Consider the code file *mycode.py*:

```
f = open("toto.txt", "r")
```

It tries to open for reading (mode `"r"`) the file named `"toto.txt"` that is located in the same folder as the file *mycode.py*.

Remark

You can also open a file by giving its full location:

```
f = open("C:\lala\pooo\toto.txt", "r")
```

Opens the file named *toto.txt* located in the directory *C:\lala\pooo*.

1.2.2 Closing A File

Closing a file is really easy:

```
f.close()
```

where `f` is some variable of type `File`.

Warning

DO NOT FORGET TO CLOSE ALL THE FILES YOU HAVE OPENED AFTER YOU ARE DONE WITH THEM.

1.2.3 Reading A File

In the following we consider that the variable `f` represents a file opened in reading mode. Consider also the file *bohemian.txt*:

```
Is this the real life?
Is this just fantasy?
Caught in a landslide,
No escape from reality.
```

The easiest way to read files in Python is to read a complete line from it. In your computer a new line is represented by the character `"\n"`.

There exist three ways to read a file in Python.

1.2.3.1 `readline`

The `readline` function allows to read **1** line from a file:

```
varname.readline() -> str
```

where `varname` is a variable of type `File`.

Each time `readline` is called on a file, it returns the next unread line. When the last line of the file have been read, `readline` returns the empty string `""`.

Example In a Python interpreter, in the same directory as the file *bohemian.txt*:

```
>>> f = open("bohemian.txt", "r")
>>> f.readline()
"Is␣this␣the␣real␣life?\n"
>>> f.readline()
"Is␣this␣just␣fantasy?\n"
>>> f.readline()
"Caught␣in␣a␣landslide,\n"
>>> f.readline()
"No␣escape␣from␣reality.\n"
>>> f.readline()
""
>>> f.readline()
""
>>> f.close()
```

As you can see, each call to `readline` returns the next line of the file. When the file is finished, after the fourth call to `readline`, all calls to `readline` on this file will return the empty string `""`.

1.2.3.2 `readlines`

The function `readlines` directly returns the list of **all** the lines in the file:

```
varname.readlines() -> list
```

After a call to `readlines`, a file is considered to be finished, hence all the other calls to `readlines` will return the empty list `[]`.

Example In a Python interpreter, in the same directory as the file *bohemian.txt*:

```
>>> f = open("bohemian.txt", "r")
>>> f.readlines()
["Is␣this␣the␣real␣life?\n", "Is␣this␣just␣fantasy?\n" \
 "Caught␣in␣a␣landslide,\n", "No␣escape␣from␣reality.\n"]
>>> f.readlines()
[]
```

```
>>> f.close()
```

The first call to `readlines` returns the list of all lines while the second call returns the empty list.

1.2.3.3 `for`

Finally, it is also possible to read a file line by line directly using a `for` loop:

```
for varname in f:
    INSTRUCTION
    GROUP
```

Here `varname` will take the value of all successive lines of the file. At the first turn the first line, the second turn the second line, ..., until the end on the file.

Example In a Python interpreter, in the same directory as the file *bohemian.txt*:

```
>>> f = open("bohemian.txt", "r")
>>> for line in f:
        line
'Is␣this␣the␣real␣life?\n'
'Is␣this␣just␣fantasy?\n'
'Caught␣in␣a␣landslide,\n'
'No␣escape␣from␣reality.\n'
>>> f.close()
```

Remark

I advise you to use the `for` loop method by default.

1.2.4 Writing In a File

To write in a file, the syntax is the following:

```
varname.write(s: str) -> None
```

where `varname` is of type `File` opened in writing mode.

This writes the string `s` at the end of the file.

Warning

`write` does not put a line ending character `"\n"`. Hence if you want to finish the line in your file you have to add `"\n"` at the end of the string you write.

Example In the Python interpreter:

```
>>> f = open("coucou.yolo", "w")
>>> f.write("TinkyWinky")
>>> f.write("laalaa\n")
>>> f.write("Po" + "\n")
>>> f.close()
```

Creates the file *coucou.yolo* in the directory where your interpreter is set, it contains:

```
TinkyWinkylaalaa
Po
```

The two first calls to `write` only produce one line in the file as there are no `"\n"` character in the first call.

1.3 Useful Functions On Strings

I will present you two functions on strings that are useful when reading files.

1.3.1 Removing The New Line Character

Two simple ways to remove a `"\n"` character at the end of a string `s` are the following:

1. Using the slicing: `s[:len(s)-1]` (equivalent to `s[:-1]`).
2. Using the `rstrip` function: `s.rstrip("\n")`. The `rstrip` function has the following prototype: `varname.rstrip(c: str) -> str` where `varname` is a variable of type `str`. It removes all the successive characters `c` at the end of the string `varname`.

Example In the interpreter:

```
>>> "ABC\nDEF\n"[:-1]
"ABC\nDEF"
>>> "ABC\nDEF\n".rstrip("\n")
"ABC\nDEF"
>>> "ABC\nDEF\n\n".rstrip("\n")
"ABC\nDEF"
```

Note that `rstrip` only removes the characters at the end of the string, hence in examples 2 and 3 the `"\n"` character in the middle of the string remains.

1.3.2 Cutting A String Into Pieces

The function `split` allows to cut a string into a list of sub-strings:

`varname.split(c: str) -> list`

where `varname` is a variable of type `str`.

`split` splits the string `varname` at each positions where the character `c` appears. It produces the list of all the obtained sub-strings.

Example In the interpreter:

```
>>> "1,2,3".split(",")
["1", "2", "3"]
>>> "1,2,,3".split(",")
["1", "2", "", "3"]
```

Note that in the second case, as there are two following coma, it creates an empty string in the list.