02. [25p] Scripting, control structures and functions

Scripting

In essence, *Python* is still a scripting language. Writing a *Python* script is very similar to writing a *Bash* script. After all, both use a specific interpreter. So let's take a look at how we might start:

my_first_script.py

```
#!/usr/bin/python3

import sys      # gives access to cli arguments

def main():
    print('Hello world!\nReceived %d arguments are: %s' % \
        (len(sys.argv), str(sys.argv)))

if __name__ == '__main__':
    main()
```

Quick reminder: you need to give the script executable permissions if you want to run it.

```
$ chmod +x my_first_script.py
$ ./my_first_script.py these are some arguments
```

First of all, you can download this file directly by clicking on the name at the top of the snippet. Next, let's take it step by step and understand what is going on:

- On the first line, we have a shebang (i.e.: `#!`); this specifies the path to the interpreter that the operating system should use to run this script. You can try to identify the location of **python3** in your system using `$ whereis python3`, but it should be the same.
- The `import sys` is new, but you should at least intuit what it does. Functionally, it works pretty much like a *C* `#include`. Practically, there are differences:
  - an `#include` directive takes the code in the specified header file and just copy-pastes it before proceeding with the compilation.
  - a *C* header file does not (usually) contain the actual code. Only definitions that your program should use without having access to the implementation. Once the compilation is finished, more often than not the actual code that is executed by invoking the functions in the included header resides in a shared library (e.g.: */usr/lib/libcrypt.so*). Some of these libraries are linked by default (e.g.: */usr/lib/libc.so*). For others, you may need to specify a linker flag (e.g.: `-lcrypt`).
  - in *Python*, when you import a module, the interpreter basically executes all the code in the module. After that, it returns to the same `import` line but now, all functions have been defined and variables have been declared. The only difference from *C* is that you must access them through the module name (e.g.: `sys.argv`).
- `def main():` declares the start of a function (shocker). Note that this function did not have to be named *“main”*. There is no such convention in *Python*, but this name makes it familiar and immediately obvious what its purpose should be. Also, the script execution entry point is not this function!
- looking at the `print()` function, there are a few things to discuss:

        - *Python* does not have curly brackets like *C/C++*. In stead, it uses indentation as a way to differentiate blocks of code. For example, if we want to specify the body of the **main()** function, all instructions in said body must be indented by exactly one *tab*. If one of these instructions is an **if** statement, its body must be indented by two *tabs*. Note that these tabs can either be the *'\t'* character, or four spaces. However, if you combine the two in the same source file, the interpreter will throw an error!
        - remember that when using the **python** shell earlier, we could just punch in a variable name or an expression and the result would be shown? This does not work inside a script. Simply writing `sys.argv` has the same effect as in *C*. The expression is evaluated. The interpreter determines that it is a variable. It returns the content of the variable but there's nothing more to the statement. The obtained content is lost and the interpreter moves on… This is why we must use the **print()** function.
    - Here, we meet our first real **if** statement: `if __name__ == '__main__':`. If we ignore function declarations, this is the first command that is actually executed. In fact, we could skip this **if** and just leave **main()** in stead. Nothing would be visibly different. The key word here is *visibly*. You see, the variable __**name**__ is a *Python* built-in that is set to the name of the current module. If our script is imported in another script (or a **python** shell), the __**name**__ variable would hold the name of the script (i.e.: *my_first_script*). However, if we run this script directly from **bash**, it's value would be set to *"__main__"*. In other words, without this **if**, whenever we would import this script as a module just to have access to the **main()** function, the invocation of main would be executed implicitly.

This may have been a lot to unpack. Take a moment and if you have any questions, ask away. If not, let's move on!

Control Structures

Conditional statements

Up until this point we've already seen the **if** statement used a few times. It should hold no mysteries :p

At the moment, there is no equivalent to the **switch case** from *C/C++*. A similar feature called **match** will be introduced in *Python 3.10*. Functionally, the two will be the same. The main differences would be the lack of a **default** clause keyword, and that of a need to use **break** with every **case** statement. For now, we won't bother with this.

```
>>> # get the current time
>>> import time
>>> lt = time.localtime()
>>> lt
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=7, tm_hour=18, tm_min=23, tm_sec=58, tm_wday=6, tm_yday=311, tm_isdst=0)

>>> # let's print different messages based on the time interval
>>> hour = lt.tm_hour
>>> hour
18

>>> # notice how we use elif
>>> if hour < 7:
...     print('night')
... elif hour < 12:
...     print('morning')
... elif hour == 12:
...     print('noon')
... elif hour < 17:
...     print('afternoon')
... else:
```

```
...     print('evening')
...
evening

>>> # in Python we use "not", "or", "and" in stead of "!", "||", "&&"
>>> import random
>>> r = random.randint(0, 100)

>>> if (r % 2 == 0 and r < 50) or not (r % 2 == 1 and r > 50):
...     print("was there a point?")
...
was there a point?

>>> # again, remember to perform checks before accessing dictionary items
>>> d = { }
>>> if r not in d:
...     d[r] = 0
...
>>> d[r] += 1
>>> d[r]
1
>>> d[r + 1] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

While statement

Again, the **while** in *Python* is not that different than the **while** in C/C++.

```
>>> r = random.randint(0, 100)
>>> guess = -1

>>> # continue and break work just like in C/C++
>>> while guess != r:
...     guess = int(input('take a guess: '))
...     if guess % 12 == 0:
...         continue
...     elif guess % 3 == 0:
...         break
...
>>> print('good job, I guess...')
```

For statement

This is probably different from what you may be used to from other programming languages. The **for** in *Python* does not iterate until a certain condition is satisfied. Nor does it let you specify the iteration step. As a result, it's not immediately replaceable with a **while** loop. In stead, **for** iterates over a sequence of items.

Remember using **for** to iterate over the items generated by the **range()** function? In that case, the iteration step should be established via **range()**, not the built-in constructs of the **for** statement.

Modifying elements of an iterable object (like a list or dictionary) with **for** may be prone to errors when the modified element is the iterator itself. Sometimes it's better to work with copies or create new objects entirely.

```
>>> # iteration step is given by range()
>>> for it in range(0, 10, 2):
...     print(it)

>>> # this is also a good method of iterating over available dictionary keys
>>> # NOTE: using the ip_addrs dictionary from earlier; not redeclaring it here...
>>> for key in ip_addrs:
...     print(key)
...
ocw.cs.pub.ro
google.com
filelist.io
```

This may be a bit random, but **for** and **while** loops can have **else** statements. The **else** block is executed if the **for** reaches the end of the iterable object or if the **while** statement's condition becomes *False*. If the loop is otherwise broken, the **else** block will not be executed.

```
>>> for key in ip_addrs:
...     if key == 'lwn.net':
...         print('good for you!')
...         break
... else:
...     print(':(')
...
:(
```

Functions

Defining functions is done using the **def** keyword, followed by its name and a parenthesized list of parameters, with a **:** at the end, for good measure. The body of the function starts on the next line.

Each function has its own local context. You can still access global variables from functions for read-only operations. However, if you want to modify a global variable, you need to mark it as such with the keyword **global**:

```
>>> x = 10
>>> def increment_x(value):
...     global x
...     x += value
...
>>> increment_x(5)
>>> x
15
```

Functions, of course, can have arguments. *Python* uses a system named *“Call by Object Reference”*. If you pass immutable objects (e.g.: numbers, strings, tuples) as arguments and overwrite them in the invoked function, the originals in the calling function will remain untouched and you will be working with a copy. This is known as *“Call by value”*. If your argument is mutable (e.g.: lists, dictionaries), the changes that you make will be reflected in the caller's environment. This is known as *“Call by reference”*.

```
>>> def modify_string(s):
...     s = 'a new string'
...     print(s)
...
>>> def modify_list(l):
...     l.append(99)
...     print(l)
...
>>> original_s = 'original string'
>>> original_l = [1, 2, 3]

>>> # call by value
>>> modify_string(original_s)
a new string
>>> print(original_s)
original string

>>> # call by reference
>>> modify_list(original_l)
[1, 2, 3, 99]
>>> print(original_l)
[1, 2, 3, 99]
```

This can get confusing… I know. On the upside, *Python* is very flexible in how you define your arguments. For example, this is how you can assign default values:

```
# defining a function with default argument values
>>> def prompt_user(prompt, expected='student', retries=3):
...     while retries != 0:
...         user = input(prompt)
...         if user == expected:
...             return True
...         retries -= 1
...     return False

>>> # only specifying the required argument
>>> prompt_user('Who are you? ')
Who are you? user
Who are you? student
True

>>> # overwriting default argument values
>>> prompt_user('*What* are you? ', 'I aM sPeCiAl', 0)
False

>>> # using keyword arguments to overwrite only specific arguments
>>> prompt_user('Who are you? ', retries=0)
False
```

A word of caution. The default initialization of arguments happens only once! If we use immutable objects (like numbers or strings) we won't have any surprises. But when using mutable objects (like lists), the default value can change over subsequent calls:

```
>>> # I mean... you _could_ still want this...
>>> def append_to_default(l=[ 0 ]):
...     l.append(l[-1] + 1)
...     print(l)
...

>>> append_to_default()
[0, 1]
>>> append_to_default()
[0, 1, 2]
>>> append_to_default()
[0, 1, 2, 3]
```