01. [25p] Intro, datatypes and more

Up until now, you should have had some interaction with the *C language* (or even, with *C++*). Normally, you would write your code in a source file, compile it with **gcc** and most likely get a few dozen errors. Eventually, you would end up with a binary executable file (i.e.: an ELF [https://refspecs.linuxfoundation.org/elf/elf.pdf]).

Another language that you've recently started learning is *Bash*. The main difference between the two is that *Bash* is actually an interpreted language, meaning that you have a program (also called **bash**; confusing, I know) that continuously reads your input and has pre-determined routines that it executes for known command patterns.

Python can be seen as a combination of the two. On the one hand, it's an interpreted language, just like *Bash*. Try to run **python3** in your terminal and see for yourself. Notice how the prompt changed? The similarities with *Bash* though, pretty much end there. *Python* is closer to *C/C++* in that it's better structured, implementing concepts from Object-Oriented Programming (OOP) and supporting multiple programming paradigms.

Why should you learn *Python*:

- It's easier to learn than *C/C++*. While these are low-level languages, preferred when you need precise control over certain aspects such as the memory, *Python* is more flexible (e.g.: dynamic typing vs static typing) and lets you prototype ideas faster.
- It's widely used in the industry for writing “glue code”. Compiled languages are, of course, significantly faster than interpreted languages. If you want high performance cryptographic libraries or game engines, you use a compiled language. “So why learn *Python* at all?”, you may think. By providing wrappers for libraries written in other languages, *Python* is meant to help you build a system from multiple pre-existing components with the minimum amount of effort.
- It gives you more control in some cases. In *C++* for example, classes (i.e.: *C* structures but smarter) typically use access specifiers such as **public** or **private** to control what mechanisms you can interact with, as a developer. Sometimes (very rarely) you may actually know what you're doing and you might want to fiddle around with the implementation of a container [https://en.cppreference.com/w/cpp/container], let's say. Unless you are willing to modify and recompile that specific library, you'd be forced to employ some pretty dubious hacks to make the change. *Python* on the other hand, doesn't have access specifiers. By convention, anything within a class that you are not supposed to interact with will be prefixed with a '_' (e.g.: `_do_not_touch_me()`). But if you're keen on it, by all means… do as you wish.

Native data types

In *Python* you don't have to declare variables. You just… use them. By assigning a value to a variable, you create it. If a variable with the same name already exists, you overwrite it. Just because you don't specify a type during declaration (as in *C*), it doesn't mean that the variable doesn't have one. The type of the variable is inferred from the type of the value that is assigned. Naturally, performing operations on variables of different types might create problems. Adding an integer and a float is permitted because both are numeric types. Adding a string and boolean however… At least errors such as this, or dividing by zero [https://www.youtube.com/watch?v=asDlYjJqzWE] don't crash our shell.

```
$ python3
Python 3.9.7 (default, Aug 31 2021, 13:28:12)
[GCC 11.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> a = 'ana are mere'
>>> b = 55

>>> type(a)
<class 'str'>
```

```
>>> type(b)
<class 'int'>
>>> c = True
>>> type(c)
<class 'bool'>

>>> a = 3.1415
>>> type(a)
<class 'float'>

>>> 'ana are mere' + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "bool") to str
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Let's take a look at each of the more common types in *Python*:

Numbers

As you probably noticed in the snippet above, *Python* makes a distinction between different numeric types, such as *float* or *integer*. It, however, is not limited to only these two types. *Complex* numbers are also natively supported:

```
>>> a = 3
>>> b = 5
>>> c = 1 - 8j

>>> isinstance(c, complex)
True
>>> c.real
1.0
>>> c.imag
-8.0

>>> a *= 1j
>>> a
3j
>>> c * a - b
(19+3j)
```

Notice how we accessed the real and imaginary parts of the complex number `c`? Being able to do this should immediately remind of structures in *C*, or classes in *C++*. We should investigate this further…

```
>>> help(c)
>>> # use the arrows or PageUp / PageDown to navigate the help window
>>> # press 'q' to exit back to the shell prompt
>>> # yes, these are all comments
>>> # no, you don't have to copy them :p
```

The **help()** function is an interpreter built-in that accesses the documentation of a specific type, module, etc. The output might be a bit too… complete. But if you scroll all the way down, you will find what we're interested in:

```
 |  conjugate(...)
 |      complex.conjugate() -> complex
 |
 |      Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  real
 |      the real part of a complex number
```

So the *complex* type has two internal variables that hold the real and imaginary parts of the number. But what's this? It also has a method (i.e.: function) defined. Try to invoke **conjugate()** for `C`. Does it work? Does it change the **imag** and **real** internal variables?

Well, that was an interesting detour. But let's get back to our numbers. As you might imagine, all the basic arithmetic operators work in *Python* pretty much like they would in *C*; even the modulus operator. There are, however, two extra ones that might be useful: *exponentiation* and *floor division*:

```
>>> 5 ** 3
125

>>> # regular division can produce a float even if the operands are integers
>>> 9 / 4
2.25
>>> # floor division acts like regular division on integers in C
>>> 9 // 4
2
```

“What about bitwise operators?” Of course they're supported ;)
Here is a reminder [https://mathstats.uncg.edu/sites/pauli/112/HTML/secbinary.html] if you forgot how the binary representation of numbers works.

```
>>> x = 8
>>> bin(x)
'0b1000'

>>> bin(x << 1)
'0b10000'
>>> bin(x >> 2)
'0b10'
>>> bin(x | 0b0010)
'0b1010'
>>> bin(x ^ 0b1010)
'0b10'
>>> bin(x & 0b0111)
'0b0'
```

Strings

Strings in *Python* are more similar to the std::string [https://en.cppreference.com/w/cpp/string/basic_string] in *C++* than with those in *C*. Just a heads up, but this will be a running theme while you learn *Python*. Knowing about OOP [https://en.wikipedia.org/wiki/Object-oriented_programming#:~:text=Object%2Doriented%20programming%20(OOP),(often%20known%20as%20methods).] and *C++* is not necessary to understand the basic concepts in *Python* but it does help somewhat. Anyway, a string declaration works pretty much as you'd expect. Note that you can use both single and double quotes in their declaration. It doesn't matter which you prefer.

While the choice of single / double quotes is irrelevant in *Python*, the same cannot be said for **Bash**. Single quotes [https://www.gnu.org/software/bash/manual/html_node/Single-Quotes.html] inhibit the expansion of character sequences (e.g.: referencing a variable, like `${MY_VAR}`). Double quotes [https://www.gnu.org/software/bash/manual/html_node/Double-Quotes.html] do not. Make sure you don't mix up these concepts.

```
>>> # here, we use the + operator to concatenate four strings into one
>>> # we use \ to break the current line and continue our command on the next
>>> # do not copy the ... at the start; that's the multiline prompt
>>> # bonus point if you recognize the quote ;)
>>> s = "'I was there,' he would say afterwards, until afterwards "  + \
...     "became a time quite devoid of laughter. 'I was there, the " + \
...     "day Horus slew the Emperor.' It was a delicious conceit, "  + \
...     "and his comrades would chuckle at the sheer treason of it."

>>> # the length of the string can be obtained using len()
>>> len(s)
230

>>> # we can access a single character from the string, but it's still a string
>>> s[3]
'w'
>>> type(s[3])
<class 'str'>

>>> # we can also extract a substring
>>> # if we specify [x:y], it starts from the letter indexed x, but stops at y-1
>>> # if we avoid specifying x, it will consider it to be 0 implicitly
>>> # if we avoid specifying y, it will consider it to be len(s) implicitly
>>> s[0:38]
"'I was there,' he would say afterwards"
>>> s[:38]
"'I was there,' he would say afterwards"
>>> s[210:]
'sheer treason of it.'

>>> # here, things get funky
>>> # you can use a *negative* index to count *backwards* from the end
>>> s[-20:]
'sheer treason of it.'
```

As you can see, we access individual characters and substrings just like how you'd expect to access elements in an array. The fact is, everything we've just seen here applies identically to generic arrays. We'll see that, and more, in the next section. For now, look in `help(str)`; identify a function to convert all characters to lowercase, and another to find a substring in a string. Apply both of them to `s` in one command in order to find the string *'emperor'* (all lower case).

One final thing about strings: formatting. There are multiple ways of doing this, but this is most similar to the formatting in **printf()** and honestly, it's the one I prefer :p

```
>>> '%-10s are %.2f mere' % ('Ana', 15)
'Ana        are 15.00 mere'
```

Lists

When you think “list”, you might be reminded of the constant-sized arrays in *C*. But lists in *Python* are closer in design to the std::vector [https://en.cppreference.com/w/cpp/container/vector] container from *C++*, in that they can be arbitrarily resized at runtime. One difference, however, is that the elements contained in a list do not have to be of the same type (although it would be preferable if they were).

```
# notice how we don't need \ to split the command
# the interpreter expects that we may want to extend the declaration on multiple lines
>>> months = [ 'January', 'February', 'March',
...            'April',   'May',      'June',
...            'July',    'August',   'September',
...            'October', 'November', 'December' ]

>>> # most of what we tested in the Strings section also applies here
>>> len(months)
12
>>> months[0]
'January'
>>> months[-1]
'December'

# here, we extract all months starting with the one indexed 3rd, up until the 3rd from the end
# obtaining a subset of a list is called slicing
>>> months[3:-3]
['April', 'May', 'June', 'July', 'August', 'September']

# here, we extract every second month starting with January
# the z in [x:y:z] is the iteration step
# in this case, x and y didn't need to be explicitly stated
>>> months[0:12:2]
['January', 'March', 'May', 'July', 'September', 'November']
>>> months[::2]
['January', 'March', 'May', 'July', 'September', 'November']

# here, we extract months ranging from the one indexed 6th to the one indexed 0th (excluding it), in reverse order
>>> months[6:0:-1]
['July', 'June', 'May', 'April', 'March', 'February']
```

Next, let's look at a neat little trick called **list comprehension**. We haven't looked at control structures (i.e.: **for**, **if**, etc.) yet, but the following use cases are more or less self contained. Note that the **range()** function does not generate a list, by itself. What it does, it creates a number generator that can be interrogated to continuously get the next number in the range. The reason for this is not to overload the memory when working with larger values.

```
>>> # list() consumes all values that range() can offer
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> # we create an array for every element (we call it "it") in the range
```

```
>>> # in stead of list(), the for structure consumes the input itself
>>> [ it for it in range(10) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> # initialize the array with the cubes of the elements
>>> [ it ** 3 for it in range(10) ]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

>>> # do the same, but only for even elements in range
>>> # odd elements are in stead replaced with None
>>> # None is a special type that represents a NULL object
>>> [ it ** 3 if it % 2 == 0 else None for it in range(10) ]
[0, None, 8, None, 64, None, 216, None, 512, None]

>>> # include the cubes only of the odd elements
>>> # even elements are completely ignored
>>> [ it ** 3 for it in range(10) if it % 2 == 1 ]
[1, 27, 125, 343, 729]
```

Let's take a closer look at the final two examples. In the first, we used the *Python* equivalent of the *C* ternary operator: `it ** 3 if it % 2 == 0 else None`. This would roughly translate to `(it % 2 == 0) ? pow(it, 3) : None`. Notice that in this example, we either have to have `it ** 3` or `None`. In other words, we can't drop the `else`. Otherwise, we would get an invalid syntax error. For a similar outcome however, we have the second example. Here, the use of `if it % 2 == 1` at the end is specific to this type of array initialization and will most likely generate an error in any other context.

Since we know how to initialize an array, and access elements of an array, all that's left is manipulating an array.

```
>>> # first, create a simple list
>>> v = list(range(0, 20, 2))
>>> v
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> # delete the element with the index 4
>>> # del is a Python built-in keyword and can delete more than just list elements
>>> del(v[4])
>>> v
[0, 2, 4, 6, 10, 12, 14, 16, 18]

>>> # delete a whole range of values
>>> del(v[4:8])
>>> v
[0, 2, 4, 6, 18]

# delete the first occurrence of the element 4
>>> v.remove(4)
>>> v
[0, 2, 6, 18]

>>> # append a new element
>>> v.append(99)
>>> v
[0, 2, 6, 18, 99]
```

```
>>> # insert a new element on the first position
>>> v.insert(0, -99)
>>> v
[-99, 0, 2, 6, 18, 99]

>>> # extend the list with a few more numbers from a range()
>>> v.extend(range(-5, 5, 4))
>>> v
[-99, 0, 2, 6, 18, 99, -5, -1, 3]

>>> # concatenate v with another list, then overwrite v
>>> v += [ -10, 0, 10 ]
>>> v
[-99, 0, 2, 6, 18, 99, -5, -1, 3, -10, 0, 10]

>>> # sort all elements using the default criterion (i.e.: ascending order)
>>> v.sort()
>>> v
[-99, -10, -5, -1, 0, 0, 2, 3, 6, 10, 18, 99]
```

Tuples

While lists are variable-sized and any element can be changed, tuples are immutable. Meaning that they can't be changed in any way once they are created. Similarly to arrays, though, slicing still works. Because slicing creates a new tuple (or array) from an existing object, you are not really *changing* anything. We already used a tuple when doing string formatting earlier. They are defined exactly like a list, but using parentheses in stead of brackets. So why should we use tuples if we already have lists? For once, tuples are faster to iterate over than lists. Moreover, sometimes you might want your data to be read-only. For example, when you use the same tuple object as a dictionary key and expect it never to change.

```
>>> # this is how you declare a tuple
>>> t = ( 'pi', 3.1415, True )
>>> t
('pi', 3.1415, True)

>>> # splicing still works, and we get another tuple
>>> t[:2]
('pi', 3.1415)

>>> # a list can be generated from the content of a tuple
>>> list(t)
['pi', 3.1415, True]
>>> list(t) + [ 'e', .5772, True ]
['pi', 3.1415, True, 'e', 0.5772, True]

>>> # similarly, a tuple can be generated from a list
>>> tuple(list(t) + [ 'e', .5772, True ])
('pi', 3.1415, True, 'e', 0.5772, True)

>>> # tuples can be used as a shortcut to assign multiple values at once
>>> # many functions in Python return multiple values (how would you do this in C?)
>>> # by convention, you use _ to signify that you want to ignore a certain value
>>> t = ('c', 4, False)
>>> (x, y, _) = t
>>> x
```

```
'c'
>>> y
4
```

Dictionaries

These are unordered sets of key-value pairs. Each key must have a corresponding value. Also, each key is unique. Dictionaries work pretty much like an array, only that you don't *have* to use numeric indexes, but objects of your choice:

```
>>> # this is how you declare a dictionary (can also be empty)
>>> ip_addrs = { 'ocw.cs.pub.ro' : '141.85.227.65',
...              'google.com'    : '142.250.180.206',
...              'filelist.io'   : '104.21.16.66' }

>>> # blindly accessing elements is usually a bad idea
>>> ip_addrs['ocw.cs.pub.ro']
'141.85.227.65'
>>> ip_addrs['curs.upb.ro']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'curs.upb.ro'

>>> # better to first check if the key exists in the dictionary
>>> 'curs.upb.ro' in ip_addrs
False
>>> ip_addrs['curs.upb.ro'] = '141.85.241.61'
>>> ip_addrs['curs.upb.ro']
'141.85.241.61'

>>> # dictionaries also have specific methods, viewable with help()
>>> # these are a few self-explanatory examples
>>> ip_addrs.keys()
['google.com', 'filelist.io', 'curs.upb.ro', 'ocw.cs.pub.ro']
>>> ip_addrs.values()
['142.250.180.206', '104.21.16.66', '141.85.241.61', '141.85.227.65']
>>> ip_addrs.items()
[('google.com', '142.250.180.206'), ('filelist.io', '104.21.16.66'), ('curs.upb.ro', '141.85.241.61'), ('ocw.cs.pub.ro', '141.85.227.65')]
```