# Systèmes d'Information & Programmation (SIP)


Python Memo

**CentraleSupélec**

2019-2020

# Contents

# Elements of Python Syntax

**Note:** this document is not meant to be exhaustive but covers most of the main notions seen in the TD's.

## Variable declaration and assignment

In Python, both the declaration (i.e. giving a name to a variable) and the assignment (i.e. giving a value to a variable) of variables are done in only one instruction using the = symbol.

```
>>> name_of_the_variable = "value"
# the value of the variable name_of_the_variable is the string
   "value".
```

## Types

Every variable has a type. However, because Python is a weakly typed language (meaning that variables are not bound to a specific type), the type of the variable is not explicitly specified but implicitly found by Python as the variable is declared. The most common types in Python are integers, floating-point numbers, complex numbers, booleans, strings, lists, tuples, dictionaries and sets. The last five types are also data structures.

### Integers

Integer is the type denoting natural numbers, i.e. $\mathbb{N}$. Integers support all usual mathematical operations `+`, `*`, `-`. Note that the division of integers will return a floating number.

- `//` returns the quotient of the Euclidean division.
- `\%` returns the remainder of the Euclidean division.

```
# 7 = 3 * 2 + 1
>>> 7 // 2
3
>>> 7 % 2
1
```

### Floats

Floats denote floating(-point) numbers that are an approximation of real numbers, i.e. $\mathbb{R}$. Floats support all usual mathematical operations but note that, because they are only approximations, some results can be incorrect from a mathematical point of view, due to loss of precision. Indeed, some decimal numbers cannot be represented exactly in binary, resulting in small roundoff errors.

```
>>>1.2-1.0
0.19999999999999996
```

### Complex numbers

Complex denotes a complex number, i.e. $\mathbb{C}$. The symbol `j` denotes the complex number $i$.

- `z = complex(a,b)` declares a complex variable of real part $a$ and imaginary part $b$.
- `z.real` returns the real part of $z$.
- `z.imag` returns the imaginary part of $z$.
- `z.conjugate()` returns $\overline{z}$, the complex conjugate of $z$.

```
>>> z = complex(3,2)
>>> z.real, z.imag
(3.0, 2.0)
>>> z.conjugate()
(3-2j)
```

### Booleans

Booleans can take only two values `True` and `False` and denote the truth values of logic.

- **and** returns the logical conjunction of two booleans.
- **or** returns the logical disjunction of two booleans.
- **not** returns the logical negation of a boolean.
- `^` returns the exclusive disjunction (XOR) of two booleans.

```
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> True ^ False
True
```

Comparison operators allow us to define predicates that take a boolean value. The most usual ones are:

- `>` for greater than

- `<` for less than
- `>=` for greater than or equal to
- `<=` for less than or equal to
- `==` for equal to
- `!=` for not equal to

Be careful not to mix the equality test `==` and the assignment operator `=`.

### Typecasting

Casting an object is the operation of changing its type. Casting is done by calling the desired type on the object, i.e. `int`, `float`, `bool`, etc.

```
>>>string="10010"
>>>float_number=float(string)
>>>integer_number=int(string)
>>>print(float_number)
10010.0
>>>print(integer_number)
10010
```

### Range

Range denotes a sequence of integers from a start value (inclusive) to a stop value (exclusive). If the start value is omitted, then it takes the default value 0. If a step value is specified, it changes the increment (or decrement for a negative step value).

- `range(i, j)` returns $i,\ i+1,\ i+2, \ldots,\ j-1$.
- `range(j)` returns $0,\ 1,\ 2, \ldots,\ j-1$.
- `range(i, j, k)` returns integers in $[\![i, j-1]\!]$ written as $i+nk$, $n \in \mathbb{N}$.

## Conditions

Conditional expressions allow us to execute certain instructions in a given case. The key word **if** is used to run an instruction if a condition is satisfied, i.e. if the boolean in the condition statement is equal to `True`. The key word **else** is used to run other instructions if the condition is not satisfied. Finally, the key word **elif** as a contraction of 'else' and 'if' can be used to avoid having too many indexation levels.

```
if (boolean):
    ...
elif (boolean):
    ...
else:
    ...
```

## Loops

Loops allow us to repeat an instruction as many times as necessary. There are two types loops : the 'while' loop and the 'for' loop.
The 'while' loop is repeated as long as the boolean is equal to `True`. Beware of infinite loops if the statement never becomes false.

```
while (boolean):
    ...
```

The 'for' loop iterates over the elements of a sequence and is built as follow:

```
for element in sequence:
```

Note that `element` is a variable created by the loop that does not need instantiation and will successively take each value in `sequence`. The most common way to use a 'for' loop is to repeat an operation a given number of times. In this case, the sequence is usually a range object:

```
for i in range(n):
    ...
for i in range(start, end, step):
    ...
```

## Functions

Writing code often means reusing some block of instructions. A function allows us to regroup several instructions in a block that can be called using the function's name.

### Creating functions

In order to create a function, one must use the key word **def** for define and follow the syntax below:

```
def name_of_the_function(arg1, arg2, ...):
    ...
```

```
3     return
```

### General functions

- `type` returns the type of the object given in argument.
- **print** prints a value to a stream or to the standard output (∼ the console).
- `input` reads and returns a string for the standard input (∼ the console). When using the `input` function, it is a good practice to check that the input value matches the requirements to be used later on (e.g. type requirement, value requirement, etc.).

## Data Structures

### Strings

A string allows us to store a sequence of letters (also called characters). The sequence is written between simple quotes or double quotes, e.g `'this is a string'` and `"this is also a string"`. Here are the main operations for string objects:

| | |
|---|---|
| `s = ""` | s takes the value empty string |
| `len(s)` | returns the length of the string s |
| `s[i]` | returns the value of index i of the string s |
| `s[-1]` | returns the last value the string s |
| `s[i:j]` | returns the substring between indexes i and j-1 (slicing) |
| `c in s` | returns `True` if c is in the string s, `False` if not |
| `c not in s` | returns the negation of c in s |
| `s1 + s2` | returns the concatenation of $s_1$ and $s_2$ |
| `s1.find(s2)` | returns the index of the first letter of the string $s_2$ in the string $s_1$ if $s_1$ contains $s_2$ and -1 if not |
| `s.center(n)` | returns n length string in which s is centered |
| `s.replace(s1,s2)` | returns s where every occurrence of $s_1$ is replaced by $s_2$ |
| `s.upper()` | returns s where every character is converted to uppercase |
| `s.lower()` | returns s where every character is converted to lowercase |
| `s1 >= s2` | returns a boolean depending on the comparison of the 2 strings |
| `s.split(sep)` | splits the string into a list according to the given separator (defaults to any whitespace) |

### Lists

A list is the simplest way to do collections, it stores a sequence of objects of any type (integers, floating numbers, booleans or even lists). Lists are similar to strings but they can store more than just characters. Note that there is no conceptual limit to the size of a list but a physical one coming from the memory limit. Here are the main operations for lists:

| | |
|---|---|
| `t=[]` | t takes the value empty list |
| `t=[1, "Hello", 3.4]` | t is a list storing `1` , `"Hello"` and `3.4` |
| `len(t)` | returns the length of the list t |
| `t[i]` | returns the value of index i of the list t |
| `t[-1]` | returns the last value the list t |
| `t.append()` | adds the value given in argument at the end of the list t |
| `t[i:j]` | returns the sublist between indexes i and j-1 (slicing) |
| `del(t[i])` | removes the value of index i from the list t |
| `x in t` | returns `True` if x is in the list t, `False` if not |
| `x not in t` | returns the negation of x in t |
| `t1 + t2` | returns the concatenation of $t_1$ and $t_2$ |
| `t.remove(x)` | removes first occurence of the value x |
| `t.insert(i,x)` | adds the element x at index i in t |
| `t.reverse()` | reverses the elements of the list t |
| `t.sort()` | sorts the elements of the list t |
| `sorted(t)` | returns a sorted copy the list t |
| `t = [f(i)for i in range(n)]` | t becomes an n length list fulfilling: t[i] = f(i) |
| `filter(f, t)` | returns a list of elements of $t$ for which the function $f$ returns `True` . |
| `t.pop(i)` | removes from the list t and returns the element of index i |

### Tuples

Tuples can be seen as immutable lists as they can store any type of object but cannot be modified: it is not possible to change, add or remove an object from a tuple. Tuples are defined between parenthesis as opposed to the square brackets used for lists. Here are the main operations for tuples:

| | |
|---|---|
| `t = ()` | t takes the value empty tuple |
| `t = ('Gif', 'Rennes', 'Metz', 800)` | t is a tuple storing `'Gif'` , `'Rennes'` , `'Metz'` and `800` |
| `t = (x,)` | t takes the value of a single-element tuple |
| `len(t)` | returns the length of the tuple t |
| `t[i]` | returns the value of index i of the tuple t |
| `t[-1]` | returns the last value the tuple t |
| `t[i:j]` | returns the subtuple between indexes i and j-1 (slicing) |
| `x in t` | returns a boolean depending on the presence of the element x in the tuple t |
| `x not in t` | returns the negation of x in t |
| `t1 + t2` | returns the concatenation of $t_1$ and $t_2$ |

### Sets

A set is an unordered collection that can store each object only once. When adding an object to a set that is already in a set, nothing happens and the size of the set remains the same. Here are the main operations for sets:

| | |
|---|---|
| `s = ()` | s takes the value empty set |
| `s = {1, 2, 3}` | s is a set storing `1` , `2` , and `3` |
| `set(l)` | returns a set from list $l$. |
| `s1.intersection(s2)` | returns the intersection of sets $s1$ and $s2$. |
| `s1.union(s2)` | returns the union of sets $s1$ and $s2$. |
| `len(s)` | returns the number of items in set $s$ |

### Dictionaries

Dictionaries allow us to store objects like lists (they are a collection). However instead of storing objects according to an order, they map keys to values. For instance, a dictionary can be used to represent an address book where the values would be the addresses and the keys would be the name of the contacts. Here are the main operations for dictionaries:

| | |
|---|---|
| `d = {}` | d takes the value empty dictionary |
| `d[k] = v` | adds a new key $k$ to $d$ associated with the value $v$ or replaces the current associated value of key $k$ by the value $v$ |
| `d = {key1:value1, key2: value2}` | d takes the value of a dictionary containing two (key,value) pairs |
| `len(d)` | returns the length of the dictionary d |
| `d[k]` | returns the value associated to the key k |
| `del(d[k])` | removes the value of key k from the dictionary d |
| `k in d` | returns a boolean depending on the presence of the key k in the dictionary d |
| `k not in d` | returns the negation of k in d |
| `d.keys()` | returns the list containing all of the keys of the dictionary d |
| `d.values()` | returns the list containing all of the values of the dictionary d |
| `d.items()` | returns the list containing all of the tuples (key,value) of the dictionary d |
| `d.pop(k)` | removes k from the dictionary d and returns the element associated to key k |

## File processing

```
1 with open("poem.txt", "r") as poem:
```

```
    # You can iterate: for line in poem:
    string = poem.read(3) # reads 3 characters (whole file if the number is absent)
    line = poem.readline() # readline() gives you 1 line
    lines = poem.readlines() # List of all the lines
    # If you open with "w" to overwrite the file or "a" to append you can write
    poem.write("Hello world!")
    poem.writeline("A string with a line ending")
# File closed here
```

## Packages and Libraries

### Import

The key word **import** imports a library whereas the syntax **from** `...import` only imports a part of the library.

```
import numpy as np # import the numpy library
from math import pi, exp # import the pi variable and the exp function
from math import * # import the whole library
```

It is also possible to import a user-defined module. Let's assume an *aux.py* file containing a *gcd* function is in the same folder as a *main_file.py* file. Then the **import** key word allows us to import the *gcd* function from the auxiliary file to the main file:

```
### aux.py file

def gcd(a,b):
    """
    Calculate the Greatest Common Divisor of a and b, assuming b != 0.
    """
    while b > 1:
        a, b = b, a%b
    return a

### main_file.py file

from aux import gcd

print(gcd(125,75))

```

```
17 ### when running main_file
18 >>> 25
```

### The math module

From the Python documentation: "This module provides access to the mathematical functions defined by the C standard. [...] Except when explicitly noted otherwise, all return values are floats". Most mathematical functions and constants are available in this library such as:

| | |
|---|---|
| `math.sqrt(x)` | returns the square root of $x$ |
| `math.exp(x)` | returns the exponential of $x$ |
| `math.log(a [, b])` | returns the logarithm of $a$ in base $b$ (default is $e$) |
| `math.floor(x)` | returns the floor of $x$ as an integer, i.e the largest integer $n <= x$ |
| `math.factorial(n)` | returns the factorial of $n$ (which has to be an integer), i.e. $1 \times 2 \times \ldots \times n$ |
| `math.cos(x)` | returns the cosine of $x$ |
| `math.sin(x)` | returns the sine of $x$ |
| `math.pi)` | the mathematical constant $\pi = 3.141592\ldots$, to available precision |
| `math.e)` | the mathematical constant $e = 2.718281\ldots$, to available precision |
| `math.inf)` | a floating-point positive infinity, i.e. greater than any number |

### The random module

From the Python documentation: "This module implements pseudo-random number generators for various distributions". Here is a simple list of functions available from the library:

| | |
|---|---|
| `random.randint(a,b)` | returns a random integer between $a$ (inclusive) and $b$ (inclusive) |
| `random.random()` | returns a random float between 0 and 1 |
| `random.randrange(i,j [,k ])` | returns a random number from `range(i,j,k)` |
| `random.choice(seq)` | returns a random element from the non-empty sequence `seq` |
| `factorial(n)` | returns the factorial of $n$ (which has to be an integer), i.e. $1 \times 2 \times \ldots \times n$ |

# Clean Code

### Naming variables and functions

- No accents, nor spaces. No abbreviations. Use explicit names.
- Begin the name with a lowercase letter.
- Two naming conventions: `nameInCamelCase` [1] or `name_in_snake_case` [2].

### Code readability

- Skip lines to isolate the different parts of the code.
- Avoid too many indent levels: use auxiliary functions.
- Add comments using the # symbole

### Modularity

- Gather useful functions by theme in the same *.py* file.
- Import this file in the main script if you want to use the functions.
- More than one auxiliary function file can be used. You can divide your code as you see fit.

### Independence, coherence and concision

- It is better if a function from an auxiliary file does not call another function from another auxiliary file.
- Coherent programming: always use the same language (english, french, ...), the same naming convention, ...
- Look for concision. E.g.Boolean structures: `if boolean == True:` is the same as `if boolean:`)

### Don't Repeat Yourself

Do not do the same calculations over and over again: use loops and functions to automate the process.

[1] Camel case convention: each new word begins with a capital letter
[2] Snake case convention: the words are separated by the underscore symbol

```
print("Please")
print("do not")
print("do")
print("this")

list_to_print = ["Do","this","instead"]
for word in list_to_print:
    print(word)
```

# Tests and Exceptions

## Exceptions

Exceptions are errors that can occur when running Python code.

### How to handle exceptions

A **try** ... except block is used to catch an exception as shown in the following example :

```
# trying to get the inverse of a number
try:
    my_number = input('please enter an integer number\n')
    my_number = int(my_number)  # cast to int
    inserve_of_my_number= 1/my_number   # calculate the inverse
    print('the inverse of', my_number, 'is',
       inserve_of_my_number)

except ValueError :
    # catch the error raised if the input is not an integer
    print('input must be an integer')

except ZeroDivisionError :
    # catch the error raised if the input is 0
    print('zero is not inversible')
```

### Most Frequent Errors

- `IndexError`: you probably tried to call `t[n]` although the maximal index of $t$ is $n' < n$.
- `ZeroDivisionError`: you probably tried to divide a number by 0.
- `NameError`: you probably misspelled a variable or forgot to instantiate it.
- `TypeError`: you probably tried to apply a function to an object of the wrong type.

Please refer to `https://docs.python.org/3/library/exceptions.html` for a complete documentation on the built-in errors.

## Tests

This part aims to give you the tools for unit testing ; but we'll first see what it is and why it is important. Unit testing is one of the basic aspects of testing.

### But what is unit testing anyway ?

Unit tests are tests - small pieces of Python code in our case - that will assert that each unit - functions in our example - of our code is doing what it is supposed to do, and will keep on doing what it is supposed to even as we make new improvements to our code.

**Why is it important ?**

Imagine you're working on a project that has 3 functions in it at the beginning : they are working fine, and some other developers are using your functions. Everything is beautiful under the sun, but then you decide to write a fourth function, which is using function 2. However, your new function is not working that well, and you decide to change function 2 because it's not exactly what you need now.
Your fourth function works fine now, and everything is perfect again under the sun ... or is it ? Because some of the other developers that were using your second function are now protesting that its new version is the source of all their problems !
Unit tests can help with this kind of situation : for each functionality of your program, there will be a test that will assert that it still works as supposed after any modification.

**Is unit testing still relevant when working on a project alone ?**

Yes, because the other developers mentioned earlier could be you in two months for example !

**When to do unit testing ?**

Is it a good idea to test the code when everything is developed ? No, because the more complex your program is, the more difficult testing will be : it's best to do it as you are writing your code.

**NB about clean code :** it is better if each function can be tested independently.

## How to write unit tests in Python ?

There are multiple ways of doing unit testing in Python, but one of the easiest way is by using the framework `pytest`.
Writting Tests
To write unit tests using the *unittest* module, here is a simple protocol:

1. For each file of your project named $filename.py$ create a file named $filename\_test.py$ or $test\_filename.py$.
2. Import the *unittest* module and the main file you are willing to test.
3. Create a testing class.
4. For each function in the $filename.py$ file named $function$, write a test function in the $filename\_test.py$ file and call it $test\_function$.
5. Each testing function has to test the behavior of the function from the main file using assert methods such as $assertEqual$, $assertTrue$, etc.
6. Call `unittest.main()` in a `__name__ == "__main__"` conditional statement.

You can refer to the unittest documentation for a complete list of assert methods (look for the TestCase class in `https://docs.python.org/3/library/unittest.html`).
Here is a simple example taken from the hangman TD:

```
### hangman.py file
def update_turns(a, b, c):
    """
    Update the number of turns left.
    The user looses a turn if the guessed character is not in
       the secret word
    - a : guessed character
    - b : secret word
    - c : number of turns left
    """
    # this function is made 'difficult' to read on purpose
    return c if b in a else c-1

### hangman_test.py file
import unittest
import hangman as hg

class TestHangman(unittest.TestCase):
)

    def test_update_turns(self):
        self.assertEqual(hg.update_turns('p', 'python', 9),9)
        self.assertEqual(hg.update_turns('a', 'python', 9),8)
        self.assertEqual(hg.update_turns('', '', 9),9)
        self.assertEqual(hg.update_turns('', 'c', 9),9)
        self.assertEqual(hg.update_turns('a', '', 9),8)

if __name__ == "__main__":
    unittest.main()
```

**Running tests** To run the tests, simply run the *filename_test.py* file.

***Credit:*** *Special thanks to François Cointe (ECP 1980) for the cover page cartoon: "I am pitiful in Python".*