

Introduction to Python Programming

Functions and sequence data types

Goals:

The goal of this lab assignment is to practice the notions of [functions](#) and to use and manipulate [sequence data types](#) !

If you have any question about the syntax or how to write a part of your code, please use the "memo", the Python documentation or any other resource.

For this course and its practical parts, you will use the PyCharm Edu Integrated Development Environment (IDE). Thus, your first work here is to create a new project named SIP_LAB2 in PyCharm Edu.

1 Functions

To know : Functions

In programming, a function is a block of **organized** and **reusable** code used to perform a specific action. Functions are a way to divide your code into useful blocks leading to more **readable, reusable, shareable** and less expensive code.

In Python, the rules to define a function are the following:

- Function blocks begin with the keyword **def** followed by the [function name](#) and parentheses **()** and **:**.
- Input [parameters](#) or [arguments](#) are placed within the parentheses.
- It is highly recommended that the first statement of a function is a documentation, the [docstring](#).
- The statement **return [expression]** exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as **return None**.

Examples are given below :

```
1 # to define a function
2 # Keyword "def", function name, list of arguments between parenthesis, colon
  ":", docstring and indented block for the content
3
4 def fib(n):
5     """
6     print the nth first numbers of Fibonacci
7     """
8     a,b = 0,1
9     for i in range(n):
10         print(a)
11         a,b = b,a+b
12
13 # Calling the fib function
14 >>> fib(5)
15 0
16 1
17 1
18 2
19 3
20
21
22 # A function can return a value (the function above returns None which is the
  variable for nothing)
23
24
25
```

```
26
27 def fact(n):
28     """
29     Return the factorial of n
30     :param n: an integer
31     :return: the value of n!
32     """
33     factorial = 1
34     for i in range(1,n+1):
35         factorial*=i
36     return factorial
37
38 >>> a = fact(10)
39 >>> print(a+2)
40 3628802
41 >>> print(fact.__doc__)
42     Return the factorial of n
43     :param n: an integer
44     :return: the value of n!
45
46 # We can ask a function to return multiple elements. In that case, they will be
   wrapped inside of a tuple
47
48 def euclidian_div(a,b):
49     """
50     Euclidian division
51     :param a: an integer
52     :param b: an integer
53     :return: the quotient and a remainder
54     """
55     return a//b,a%b
56
57 >>> euclidian_div(15,6)
58 (2,3)
59 >>> q,r = euclidian_div(23,9)
60 >>> q
61 2
62 >>> r
63 5
```

1.1 Writing your first functions !

Exercise 1 : Maximum

1. Write a function `max(a,b)` returning the maximum of `a` and `b`.
2. Using the previous function, write a function `max3(a,b,c)` that returns the maximum of three numbers.

1.2 Exploring Arguments and Parameters

To know : Functions arguments and parameters

When defining a function, we can specify a default value for the arguments, as shown below.

```
1 def div(a,b,euclidian=False):
2     if euclidian:
3         return a//b,a%b
4     else:
5         return a/b
6
7 >>> div(10,3)
8 3.3333333333333335
9 >>> div(10,3,True)
10 (3,1)
```

The same principle allows us to specify an argument by name in the function call.

```
1 >>>div(10,3,euclidian=True)
2 (3,1)
```

Exercise 2 : Familiar Functions

We consider the following function.

```
1 def print_arguments(a = 6, b):
2     print("Arguments: {0} and {1}".format(a, b))
```

1. The python interpreter will return an error ? Why ? Try to correct in such a way that the call of the function as `print_arguments(2)` will return Arguments: 2 and 6
2. For each of the following function calls, predict whether the call is valid or not. If it is valid, what will the output be? If it is invalid, what is the cause of the error?

```
1 print_arguments()
2 print_arguments(4, 1)
3 print_arguments(4, "four")
4 print_arguments(41)
5 print_arguments(a=4, 1)
6 print_arguments(4, a=1)
7 print_arguments(4, 1, 1)
8 print_arguments(a=4)
9 print_arguments(b=4, 1)
10 print_arguments(a=4, b="four")
11 print_arguments(b=1, a="four")
```

Exercise 3 : Euclidian Division

Rewrite the `euclidian_div` function, with two Boolean arguments `euclidian`, `remainder` allowing to ask for an Euclidian division with or without the remainder. The default behaviour should be Euclidian division without remainder.

How would you call this function to ask for the remainder without specifying the value of `euclidian` (because it's `True` by default)?

Exercise 4 : Bank Account

We consider the following code.

```
1 balance = 0
2
3 def deposit(amount):
4     global balance
5     balance = balance + amount
6
7 def withdraw(amount):
8     global balance
9     newbalance = balance - amount
10    if(newbalance > 0):
11        balance = newbalance
12    else:
13        amount=balance
14        balance=0
15    return amount
16
17 print(balance)
18 deposit(1000)
19 print(balance)
20 amount1=100
21 print(withdraw(amount1))
22 print(balance)
23 amount2=1000
24 print(withdraw(amount2))
25 deposit(1000)
26 amount = 2000
27 print(withdraw(amount))
28 print(amount)
```

1. What are the global variables, the parameters and the local variables ?
2. What will be displayed by the program ?

Exercise 5 : Variadic functions (From Stanford Python)

To know : Variadic functions

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- `*args` (Non Keyword Arguments)
- `**kwargs` (Keyword Arguments)

We consider the following function.

```
1 def variadic(*args, **kwargs):
2     print("Positional:", args)
3     print("Keyword:", kwargs)
```

For each of the following function calls, predict whether the call is valid or not. If it is valid, what will the output be? If it is invalid, what is the cause of the error?

```
1 variadic(1, 2, 3, 4)
2 variadic(1, 1, n=1)
3 variadic(n=1, 2, 3)
4 variadic()
5 variadic(cs="Computer Science", pd="Product Design")
```

```
6 variadic(cs="Computer Science", cs="CompSci", cs="CS")
7 variadic(5, 8, k=1, swap=2)
8 variadic(8, *[3, 4, 5], k=1, **{'a':5, 'b':'x'})
9 variadic(*[8, 3], *[4, 5], k=1, **{'a':5, 'b':'x'})
10 variadic(*[3, 4, 5], 8, *(4, 1), k=1, **{'a':5, 'b':'x'})
11 variadic({'a':5, 'b':'x'}, *{'a':5, 'b':'x'}, **{'a':5, 'b':'x'})
```

Exercise 6 : Average

Write a function `average` that accepts a variable number of integer positional arguments and computes the average. If no arguments are supplied, the function should return `None`.

Exercise 7 : Loops in a function

Using the keyword `continue` (that skips directly to the next iteration), create a function that for a given n , prints the numbers inferior to n that are multiple of 3 but not of 5.

Then, using the keyword `break` (that exits the loop), create a function that finds the first divider of a given number n .

1.3 Writing more complex functions

Exercise 8 : The Goldbach's conjecture

The Goldbach's conjecture is one of the oldest and best-known unsolved problems in number theory and all of mathematics. It states that *every even integer greater than 2 can be expressed as the sum of two primes*.

1. Write the function `is_prime(p)` that implements the following specification:

```
1 # Test if an integer p is a prime
2 #
3 # Input : an integer p that is greater than 1.
4 # Output : True if and only if p is prime, else false
5
```

2. Write the specification of the function `is_goldbach(n)` that returns `True` if and only if the conjecture is satisfied for n .
3. Write the code of the function `is_goldbach(n)`.
4. Write the code of the function `goldbach(n)` that implements the following specification :

```
1 # Verify if the Goldbach's conjecture is satisfied until a given rank
2 #
3 # Input : an integer n.
4 # Output : True if and only if every integer lower than n verifies the Goldbach's
           conjecture.
5
```

1.4 Recursive functions

To know : Recursive Functions

A recursive function is a function defined in terms of itself via self-referential expressions. All recursive functions share a common structure made up of two parts:

- Base case.

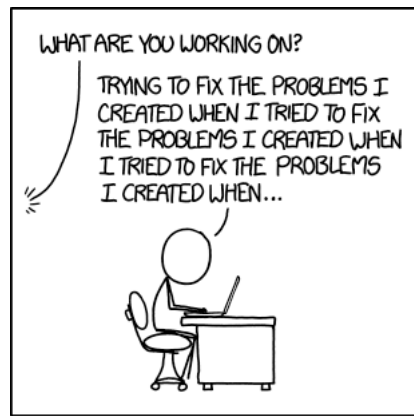


Figure 1: Source : <https://www.xkcd.com/1739/>

- Recursive case.

An example is given below with the recursive implementation of the factorial function.

```
1 def factorial_recursive(n):
2     # Base case: 1! = 1
3     if n == 1:
4         return 1
5
6     # Recursive case: n! = n * (n-1)!
7     else:
8         return n * factorial_recursive(n-1)
```

The visualization of this code execution is available here : <https://goo.gl/HeQjTV>

Remark:

Be careful that a recursive program uses much more memory than a regular iterative one. For example, if we implement fibonacci series in a recursive program, we have :

```
1 def fibonacci_recursive(n):
2     if n == 0 or n == 1:
3         return n
4     return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

This algorithm is not adapted to recursion: indeed, each call makes two more calls. So in the end we have 2^n function calls (for example this one takes around 10 seconds for $n = 31$ whereas the iterative one takes less than a second for $n = 50$). Each time you increase n by one, the time to execute is multiplied by two (around 60 days to compute the 50th term).

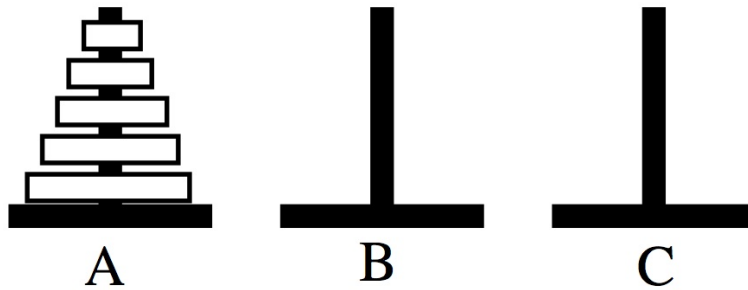
Recursion can be extremely powerful when used correctly. There are more complex algorithms using it (sorting algorithms like the Merge Sort for example) where the time complexity is reduced drastically thanks to recursion (for the Merge Sort we go from polynomial to pseudo-linear ($O(n \log(n))$) complexity).

Exercise 9 : Hanoi Towers

In this exercise, we will solve the Hanoi Tower Problem which is one of the example often used to explain recursion. The problem is the following.

We have a number n of disks on a pole, arranged like in the above picture. We want to move all the disks from pole A to pole C using pole B and following the subsequent rules:

- We can only move one disk at once
- We cannot put a disk on top of a smaller disk



- The disks should be arranged in C in the same way as they are in A.
1. Solve the problem for $n = 2$. How many steps ?
 2. Assume that we know how to solve it for $n - 1$. How to solve it for n disks. In how many steps ?
 3. Write a recursive program that solves the Hanoi Tower problem for each $n \geq 1$ and that displays all the steps on the standard output.

Exercise 10 : Digit sum

The objective of this exercise is to compute the sum of the digits of a number both iteratively and recursively.

1. Write a function `sum_digit_iter(n)` that takes a number n and returns the sum of the digits of the number in an iterative way.
2. Write a function `sum_digit_rec(n)` that takes a number n and returns the sum of the digits of the number in a recursive way.

`sum_digit_iter(2019)` and `sum_digit_rec(2019)` have to return 12 because $2+0+1+9=12$.

Exercise 11 : Digital root

The digital root is a value obtained by an iterative process of summing digits, on each iteration using the result from the previous iteration to compute a digit sum. The process continues until a single-digit number is reached. The number should be positive. For instance, for the number 1234, we have $1234 \rightarrow 1 + 2 + 3 + 4 = 10 \rightarrow 1 + 0 = 1$

1. Write a function `digital_root_iter(n)` that takes a number n and returns its digital root in an iterative way.
2. Write a function `digital_root_rec(n)` that takes a number n and returns its digital root in a recursive way.

Exercise 12 : Anagram

Write a function `is_anagrams(s1, s2)` that tests if a given string (`s1`) is an anagram of another string (`s2`). We will consider that two empty strings are anagrams. Try your code on `s1="Emperor Octavian"` and `s2="Captain over Rome"`.

2 Sequence data types

2.1 Lists

To know : Lists

- A list is a **mutable** sequence (ordered collection) of Python objects (type `list`).
- A list = different comma-separated values (items) between square brackets.

Some examples are given below.

```
1 >>> empty = [] # Empty list
2 >>> list2 = ['non', 'empty', 'list']
3 >>> len(list2) # Get the size of a list
4 3
5 >>> list2[1] # Access an element /\ Count starts at 0
6 'empty'
7 >>> list2[-1] # Negative index start at the end
8 'list'
9 >>> empty.append('new element') # Add a new element at the end
10 >>> empty + list2 # Concatenation operator
11 ['new element', 'non', 'empty', 'list']
12 >>> list2[1:3] # Slicing [start index :end index not included]
13 ['empty', 'list']
```

Exercise 13 : Occurrences in a list

We consider different implementations of the function `occ(e1,li)` that returns the first index of `e1` in the list `li` and `-1` if `e1` is not in `li`

```
1 def occ(e1,li):
2     for i in range(0, len(li), 1):
3         if e1==li[i]:
4             return i
5         else:
6             return -1
7     return -1
8
9 def occ(e1,li):
10    for i in range(0,len(li),1):
11        if e1==li[i]:
12            return i
13    return i
14
15 def occ(e1,li):
16    p=0
17    for i in range(0,len(li),1):
18        if e1==li[i]:
19            p=i
20        else:
21            p=-1
22    return p
23
24
25 def occ(e1,li):
26    p=0
27    for i in range(1,len(li),1):
28        if e1==li[i]:
29            p=i
30    return p
31
32 def occ(e1,li):
33    p=-1
```



```
34     i=0
35     while i < len(li):
36         if el==li[i]:
37             p=i
38             i=i+1
39     return p
40
41 def occ(el, li):
42     p=-1
43     i=0
44     while i < len(li) and p!=-1:
45         if el==li[i]:
46             p=i
47             i=i+1
48     return p
```

1. The different implementations are not correct. Why ? For each case, give an example of a list for which the function returns the wrong answer.
2. Correct the different implementations.

Exercise 14 : List reverse

1. Write the code of the function `reverse_iter(li)` that reverses the list `li` in an iterative way.
2. A list is a recursive datatype. Why ?
3. Write the code of a function `reverse_rec(li)` that reverses the list `li` in a recursive way.

Exercise 15 : Odd and Even

1. Write the code of the function `parity_sort(li)` that takes as argument a list `li` and returns a list such that all the even numbers are on the left and all the odd numbers on the right. For instance :

```
1 >>> lis=[5,8,4,2,3,1,10,9,7]
2 >>> print(parity_sort(lis))
3 [8,4,2,10,5,3,1,9,7].
4
```

2. Modify the function `parity_sort(li)` such that all the items of the same parity are ordered.

```
1 >>> lis=[5,8,4,2,3,1,10,9,7]
2 >>> print(parity_sort(lis))
3 [2,4,8,10,1,3,5,7,9].
4
```

Exercise 16 : List Manipulation

- What is the difference between the instructions `list2 = list1` and `list2 = list1.copy()` ? **Hint:** try changing `list1[0]` and look at `list2[0]`
- What is the difference between `list.copy()` and the `deepcopy` function from the `copy` module?
- What is the difference between `l.sort()` and `sorted(l)` ?

Exercise 17 : Sorting a list

Create a function that sorts a list of integers using the Insertion Sort algorithm. The algorithm loops over all elements of the list starting from the second one; the first element is considered as being at the right place. At each iteration, an element is inserted at the right place in the list, which might cause the other elements to be shifted. The algorithm stops when all the elements are sorted.

To know : List Comprehension

Another situation where the loop `for` is useful is to define iterable objects:

```
1 # This code generates a list containing the 100 first squared integers
2
3 my_list = [ k**2 for k in range(100) ]
4
5 # 2D-Array of size 10x10 containing zeros :
6
7 2Darray = [[ 0 for _ in range(10)] for _ in range(10)]
8
9 # Remark : When not needed in the loop, the iteration variable can be replaced
   with a "_" making it anonymous.
```

3 Tuples

To know : Tuples

Tuples are another way to keep track of data. Unlike lists, they are immutable, but have better performance for memory management.

- A tuple is an **immutable** sequence (ordered collection) of Python objects (type `tuple`).
- A tuple = different comma-separated values between (or no) parentheses.

```
1 >>> pair = (5,7)
2 >>> pair[0]
3 5
4 >>> pair[1]
5 7
```

The main difference with lists is that they can not be modified once they have been created. For example the following code will not work

```
1 >>> pair = (5,7)
2 >>> pair[0] = 3
```

Therefore each time we want to change a tuple, we need to create another one, like this:

```
1 >>> pair = (5,7)
2 >>> pair = (3, pair[1])
```

Exercise 18 : Second degree polynomials

The objective of this exercise is to write a Python program (a module) which solves real second degree polynomials through use of the quadratic equation. As a reminder, the quadratic equation gives two real roots x_1 and

x_2 of a polynomial in the form $f(x) = ax^2 + bx + c$ as $x_1 = \frac{b+\sqrt{b^2-4ac}}{2a}$ and $x_2 = \frac{b-\sqrt{b^2-4ac}}{2a}$ if the discriminant $\Delta = b^2 - 4ac > 0$. Else, if $\Delta = 0$ we have one real root as $x_1 = x_2 = \frac{-b}{2a}$ and at last, there is no real roots if $\Delta < 0$.

1. In a module named `quadratic`, write a function `quadratic_root (a,b,c)` that respects the following specification.

```
1  """
2      Solving of the quadratic equation f(x)=ax**2 + bx + c = 0
3      :param a: real number
4      :param b: real number
5      :param c: real number
6      :return: a tuple with the number of roots of the quadratic function ax**2 +
7              bx + c and their values
8  """
```

2. By using the trick if `__name__ == "__main__":`, complete the module `quadratic` such that the user is asking to enter some values for a , b and c when `quadratic` is used as a standalone program and the solve the equation $f(x) = ax^2 + bx + c = 0$ and prints the results on the standard output.
3. Write a new program that will import the module `quadratic` and the will use the `easygui` library to enter the values of the parameters a, b, c and to display the results of $f(x) = ax^2 + bx + c = 0$.
 - First, install the package `easygui` using PyCharmEdu.
 - Then, write a program that use three `enterbox` to ask and get the values of the parameters a, b, c and that displays the result through a `messagebox`. Default values would be 1,3,4.¹

¹Hint : the `help()` function is useful to understand the reused code.

4 I/O and File processing

To know : I/O

You can use the functions `print` and `input` to interact with the user.

```
1 >>> print("Hello World!")
2 Hello World!
3 >>> a = input("Type a Number:")
4 Type a number: 12 # The user types 12
5 >>> a
6 12
```

You can also interact with files with `myfile = open(file_name [, access_mode])`. The access mode is a string, `'r'` (or `'r+'`) for reading (or reading and writing), `'w'` (or `'w+'`) for writing (or reading and writing) by overwriting the file if it exists. To close the file, simply use `myfile.close()`. Using `myfile.write(string)` will by default write at the start of the file.

Exercise 19 : File Reading

Read from a file of your choice:

- the first 10 characters from the file, using `myfile.read(n)`
- the first 5 lines using `myfile.readline()`
- the last line using `myfile.readlines()`

5 Clean Code

5.1 Naming variables

Exercise 20 : Reading some code

Let's take a look at this code:

```
1 def f(v,g,e,x):
2     a = v*g
3     b = x+e
4     return [a]*b
```

What does it do ?

To know : Naming conventions

Two ways of naming your functions:

- **camelCase**: first nouns start with a lowercase and all over start with an uppercase
ex: `myVar`
- **snake_case**: everything is lowercase and separated by an underscore `'_'`
ex: `my_var`

With python the convention is to use **snake_case**.

Be sure to properly name your functions, variables, params so that people can understand what you are doing.

5.2 Comments

Exercise 21 : Quizz 1

You have one minute to tell me what this code does:

```
1 def newraph(n,t):
2     prevx = -1.0
3     x = 1.0
4     while (abs(x - prevx) > t):
5         prevx = x
6         x = x - (x*x - y) / (2*x)
7     print(r)
```

Exercise 22 : Quizz 2

You have one minute to tell me what this code does:

```
1
2 # Display the sum of x and y with a nice string
3 # Ex: 'The sum of 1 and 2 is 3.'
4 def sumProblemString(x, y):
5     sum = x + y
6     # We use string.format to nicely display the result
7     return 'The sum of {} and {} is {}'.format(x, y, sum)
8
9 # The main function
10 def main():
11     # Let's try with 2 and 3
12     print(sumProblemString(2, 3))
13
14     # Now with something bigger
15     print(sumProblemString(1234567890123, 535790269358))
16
17     # And now with 2 integers of your choice
18     # NOTE: Will send error if it is not an integer
19     # |--> Ex: 'abc'
20     a = int(input("Enter an integer: ")) # Enter the two integers
21     b = int(input("Enter another integer: ")) # required
22     print(sumProblemString(a, b))
```

- Try to input other parameters than a integer (like a list, a string). What happens ?
- [Advanced]: How to make sure the program handles correctly **a** and **b** when they are not integers

5.3 Docstring

Exercise 23 : Reusing programs

When would you use these functions? For what purpose? How to differentiate them?

```
1 def addInt(a,b):
2     return a + b
3
4 def concatenateList(a,b):
5     return a + b
6
```



```
7 def concatenateString(a,b):  
8     return a + b  
9  
10 def addTogether(a,b):  
11     if len(a) < len(b):  
12         a,b = b,a  
13     for i in range(len(b)):  
14         a[i] = addInt(a,b)  
15     return a
```

Exercise 24 : A first step through being a good programmer

Comment the last code bits using the Docstring Method (you can add extra comments if you want as well)