Chapter 2

More about functions

Functions are an excellent means to manage the complexity of a program since they make it possible to subdivide a program into smaller subproblems that can be solved separately. This not only allows for shorter development times, but it also promotes reusability because a function is written once but used many times. In Python, function can also be *objects*, i.e., values that can manipulated just like any other value.

However, functions can be more helpful. For example, a function may rely on other functions while simultaneously relying on itself. These functions, known as *recursive functions*, are extremely useful for solving complicated problems.

On the other hand, this chapter discusses modules, which can be merely defined (at first sight) as a set of inter-dependent functions that constitute a cohesive unit. Using modules enhances code readability and facilitates code distribution. Python is especially well-known for its vast module ecosystems, which allow for almost anything, from basic operations to artificial intelligence and data analysis, while also going through networking, security, document management, image processing, and so on. This chapter presents some of the most frequently used basic Python modules.

1. Recursive functions

1.1. Definition

Let's reconsider the factorial function, which, given an integer n, computes the factorial function defined mathematically by $n! = n \times (n-1) \times ... \times 2 \times 1$:

```py
def fact(n:int) -> int :
    res=1
    for i in range(2,n+1):
        res=res*i
    return res
```

The fact function solves the problem by iterating explicitly over the numbers from 2 up to n. We call it an iterative function. In mathematics, the factorial function can also be defined by $n! = n \times (n-1)!$, which means that the factorial of n needs to multiply n by the factorial of $(n-1)$ which, in turn, needs to multiply $(n-1)$ by the factorial of $(n-2)$, etc. This is known as a recursion relationship, and it is a very useful way to define complex computations on integers or whatever else has a dependency between a larger and smaller version of the same problem.

Python allows such definitions, the previous code would just be rewritten as (be careful because this code is wrong as we will see later):

```py
def fact(n:int) -> int :
    return n*fact(n-1)
```

What is wonderful about this code is (besides its shortness) that it is just the mathematical definition of the factorial (the bigger problem is how to compute the factorial of n and the simpler one is the factorial of $(n-1)$).

Definition 2.1

A *recursive function* is a function that calls itself. A *recursion* is any computation that uses a recursive function.

1.2. Rules

Let's agree that a function that the call itself can be a bunch of fun. But let's run this code and see what happens when executing `fact(5)` : we get an error message "RecursionError: maximum recursion depth exceeded". So what actually happened? Let's first define what we mean by the *recursion depth*. In fact, the main program has a depth of 2. Now, whenever a function `f` of depth i calls a function g, so the depth of g is set to $i+1$. The initial call of `fact(5)` (depth=3) results in the following chain of calls: → `fact(4)` (depth = 4) → `fact(3)` (depth: 5) → `fact(2)` (depth: 6) → `fact(1)` (depth = 7). → `fact(0)` (depth = 8). → `fact(-1)` (depth: 9),…

In reality, the sequence of calls does not terminate (infinite loop), but instead, we get an error from Python since this one puts a restriction on the recursion depth (by default 1000) in order to prevent memory overflow. Furthermore, even if Python did not have such a restriction, the operating systems will eventually interrupt the program since there would be no memory to hold the recursive calls.

To use recursion, there are two rules:

- A recursion relationship that allows calling the same function with different parameters, given the following hypothesis: let's consider that f_i is the call with the recursion depth i; hence, the problem solved by f_{i+1} should be simpler than the one solved by f_i.
- The recursion should have a base case (there may be several), which is the simplest version of the problem that can be solved without recursion. The first rule should always lead to the base case.

In our example, the base case is $0! = 1$ (and $1! = 1$). We may rewrite it as:

```py
def fact(n:int) -> int :
    return n*fact(n-1) if n>1 else 1
```

The following table defines how calls work and what are the results of each call:

Call	Next call	Result
fact(5)	fact(4)	120
fact(4)	fact(3)	24
fact(3)	fact(2)	6
fact(2)	fact(1)	2
fact(1)	fact(0)	1

1.3. How to?

Now that our first recursive function is working, we can ask the following questions each time that we want to use recursion in order to solve a given problem:

1 – When can we use recursion?

2 – Is it useful?

3 – What kind of precautions should we take?
4 – How to operate?

We will answer each question by trying to give examples.

1- When can we use recursion?

Recursion can be used wherever loops can be used. Recursion can theoretically be used to rewrite any iterative program as long as the previously mentioned guidelines are followed. The problem needs to be recursively defined in order to use recursion.

Let's take an example. The iterative function that computes the maximum value of a list is the following:

```py
def max_value(arr:list[int]) -> int:
    m=arr[0]
    # this is not efficient since the comparison with arr[0] is useless
    for v in arr:
        if v>m:
            v=m
```

A recursive definition of the maximum function is as follows: the maximum value in a list is the maximum between the first element and the maximum value of the remaining elements. The base case is when there is just one value, which means that this value is the maximum:

```py
def rec_max_value(arr:list[int],start:int=0) -> int:
    if start==len(arr)-1:
        return arr[start] #the last element
    else:
        m=rec_max_value(arr,start+1)
        return m if m>=arr[start] else arr[start]
```

Remark: actually, Python allows for far more compact code. In the last program, we made an assignment to `m` to get the value of `rec_max_value` applied to the table starting from `start+1` . After that, we made a comparison. Assignment and comparison can be combined into one statement thanks to the *Walrus operator* (:=). We can just rewrite the body of `rec_max_value` as follows despite this not being recommended for a beginner to write such a code, but it is provided to illustrate how recursion can lead to very short programs):

```py
def rec_max_value(arr:list[int],start:int=0) -> int:
    return arr[start] if start==len(arr)-1 else m if (m:=rec_max_value(arr,start+1)) else arr[start]
```

.

2- Is it useful?

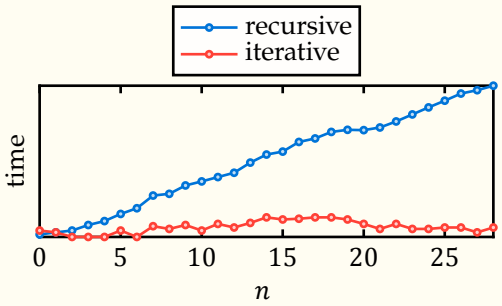
That depends on the problem at hand. Even recursion may produce code that may appear to be nice; yet, it is not necessarily efficient. Assume we have to choose between an iterative and a recursive version of a program, and the iterative version is not too difficult to construct. In general, we choose the iterative version. The recursive version can be quite inefficient in some situations. Consider a recursive version to compute the Fibonacci series (defined by: $u_0 = u_1 = 1$ and $u_{n+2} = u_n + u_{n-1}$). A straightforward implementation would be:

```py
def rec_fibo(n:int) -> int:
    if n<2:return 1
    else:return rec_fibo(n-1)+rec_fibo(n-2)
```

The iterative version is given by:

```py
def it_fibo(n:int) -> int:
    a,b=1,1
    for i in range(2,n):
        a,b=a+b,a
    return a
```

Execution times for these two codes are very different; let's compare all of that in the following chart (note that the y-scale is logarithmic for the sake of comparison):



It is obvious to see that the recursive implementation is far worse than the iterative one. We can build more effective recursive version for this program, but this is beyond the scope of this course.

Next, consider the problem of determining if a string is a palindrome, which means that it may be read from left to right as well as right to left. The following is the iterative version:

```py
def it_palindrome(s:str) -> bool:
    begin=0
    end=len(s)-1
    while begin<end:
        if s[begin]!=s[end]:break
        else:
            begin+=1
            end-=1
    return begin>=end
```

whereas the recursive version is given by (note that the inner function is recursive but the outer function is just used to call the inner function with the right arguments):

```py
def rec_palindrome(s:str) -> bool:
def __palindrome(begin:int,end:int) -> bool:
    if begin>=end:return True
    elif s[begin]!=s[end]:return False
    else:return __palindrome(begin+1,end-1)

return __palindrome(0,len(s)-1)
```

3- What kind of precautions should we take?

The first precaution is to ensure define the base case and to ensure that it can be reached. Otherwise, the code will raise a maximum recursion depth exception. The second precaution is that there won't be repeated recursive calls with the same parameters (just in the Fibonacci example), since this will drastically increases execution time.

4- How to operate?

Recursion has the major benefit of allowing the code to be easily inferred from the problem's decomposition. Generally speaking, we should first specify the base case and its solution. The next step is to specify how to

break down the problem into smaller ones and then use the solutions to the smaller problems for building the larger ones.

Recursion may be used to handle more complex problems where iterative programs would be difficult to develop (for example, *backtracking*), but this is not covered in this course.

2. Functions as values (first class values)

Assume we have an array `arr` that should be transformed by adding 1 to all its elements. This can be done with:

```py
def increment_all(arr):
    for i in range(len(arr)):
        arr[i]=arr[i]+1
```

Now, assume that we want to make another function that computes the square for each element:

```py
def square_all(arr):
    for i in range(len(arr)):
        arr[i]=arr[i]**2
```

Here, we can see that, except from the element update, the instructions for `increment_all` and `square_all` are the same. This update is dependent on the element's current value. Python enables a function to provide as an input to another function so that it can be applied to the elements, eliminating the need to write several functions with little variations. The following syntax performs this:

```py
def apply(arr,f):
    for i in range(len(arr)):
        arr[i]=f(arr[i])
```

The function `apply` is used in this case to execute the argument `f` on the current element and store the result in the same element. To increment all the elements, we should write:

```py
def incr(x:int):
    return x+1

def square(x:int):
    return x**2

apply(arr,incr)
```

The function `incr` is introduced as an ordinary argument to the apply. It's also worth noting that the function `incr` is quite simple; it just takes one parameter and returns it plus 1. In reality, the name of this function is not important; we are only interested in the fact that it accepts one input, `x` , and returns the value `x+1` . Python offers a specific syntax for creating *anonymous functions* that merely specify the input and output. We call them λ-expressions. The above example will be rewritten as:

```py
apply(arr,lambda x:x+1)
apply(arr,lambda x:x**2)
```

The expression `lambda x:x+1` means a function that takes an argument named `x` (or whatever we want) then returns `x+1` .

Remark: The pattern of applying a function to a list is so common that Python has a built-in function called `map` to do this. Therefore, the above code is equal to `list(map(lambda x:x+1,arr))` . Notice that `list` is required to generate a list from the *object* `map` .

Lambda expressions and passing functions as arguments are widely used in Python. For example, consider the function `sorted` . This function makes a sorted copy of an array. For example, `sorted([1,3,0,2])` returns the list `[0,1,2,3]` . Using the extra parameter `reverse=True` , it is possible to get the reverse order. What if we wish to have a more specific order of elements? Assume these values are the indices of another array (for example, `arr1=[4,8,0,5]` . `sorted([1,3,0,2],lambda x:arr1[x])` returns the array `[2,0,3,1]` .

3. Modules

Modules are another way to organize and reuse code. A module is a Python file that includes definitions and statements, like definition variables, functions, etc. Generally speaking, a module is *unit* that has a coherent collection of definitions to solve a specific problem. For example, we can have a module in a school management system for managing students, another for managing human resources, a third one for managing finances, and so on.

Let's consider a module that manages bank accounts as an example. An account is described by its number, the name of its owner, and the amount of money it holds (we will represent an account with an array). A list of accounts and the counter of accounts will be defined, followed by functions to search for accounts by name, deposit money, debit money, transfer money, and show the list of accounts. The information in figure 2.1 will be written into a file called `accounts.py`.

This Python code does not define a menu because it just manages bank accounts. To use this file, we will write another Python file ('menu.py') that will reference it. The first file will be imported using the `import accounts` syntax. All of the definitions in the `accounts.py` file can be used thanks to this line. However, the prefix `accounts.` (with the dot) should come before any function name or variable in `accounts.py`. We will have the following code:

```py
import accounts
if __name__=="__main__":
    print("C-Create an accountant")
    print("G-Make a deposit")
    print("D-Debit an account")
    print("T-Transfer money")
    print("L-List of accounts")
    ch=input("Please give your choice: ")
    if ch=="C":
        na)e=input("What is the name of the customer? ")
        accounts.create_account(name)
    ...
```

There is just one problem here: we will see the message "This code has been made by the Dev Team", why? In reality, Python will run whatever it finds when a module is imported, even the line `print(...)` . How can we get rid of it? We won't! But we can just make a restriction on it, i.e., the statement should not be executed when the module `accounts.py` is imported. This can be done thanks to the variable `__name__` which is equal to `__main__` for any main program, i.e., a program that is directly executed by the Python interpreter. For any imported module, this variable will be equal to the name of the module (`accounts` in this case). Hence, it is sufficient to test the value of this variable to prevent some instructions from executing when importation happens:

```py
if __name__=="__main__":
    print("This code has been made by the Dev Team")
```

```python
# The 'database' that contains the information about the accounts
accounts=[[] for _ in range(10000)]
nb_accounts=[]

def create_account(name:str,initial_amount:float):
    if nb_accounts==len(accounts):
        raise ValueError("No more customers are possible")
    accounts[nb_accounts]=[name,initial_amount]
    nb_accounts+=1

def search(name:str) -> int :
    for nb,acc in enumerate(accounts):
        if acc[0]==name:
            return nb
        raise ValueError(f"Customer {name} does not exist")

def deposit(nb:int,amount:float) :
    if nb<nb_accounts:
        accounts[nb][1]+=amount
    else:
        raise ValueError(f"Account no {nb} does not exist")

def debit(nb:int,amount:float) :
    if nb<nb_accounts:
        if accounts[nb][1]>=amount:
            accounts[nb][1]+=amount
        else:
            raise ValueError(f"There is not enough money on this account")
    else:
        raise ValueError(f"Account no {nb} does not exist")

def transfer(nb_from:int,nb_to:int,amount:float) :
    if nb_from<nb_accounts and nb_to<nb_accounts:
        if accounts[nb_from][1]>=amount:
            accounts[nb_from][1]-=amount
            accounts[nb_to][1]+=amount
        else:
            raise ValueError(f"There is not enough money on the first account")
    else:
        raise ValueError(f"At least one of these accounts does not exist")

print("This code has been made by the Dev Team")
```

Figure 2.1 - Code of `accounts.py`

If we do not want to use the prefix, especially when it is long, two options are available:

1. Creating an alias for the module using `import accounts as acc` . The new prefix, `acc.` , should be used to call the module's function. We can use any name for an alias, although some names are often used for some modules. For instance, `np` is often used as an alias for the module `numpy` , `plt` for the module `matplotlib.pyplot` , `sb` for the module `seaborn` , etc. Aliases can also be used for functions using the `pyline` syntax ("from accounts import create_account as ca").
2. Importing all module functions using the syntax `from accounts import *` . We should take caution while using this option since importing two modules with same named functions will cause a conflict (that is not easy to discover).

3.1. Overview of some of well-known basic Python modules

One of Python's attractive features is its ecosystems, which refer to the number of modules that can be imported by any user. These modules range from the most fundamental to those covering artificial intelligence, databases, editing documents, networks, and the web. Below is a summary of some of the most commonly used fundamental Python

modules. In reality, there are two types of modules: those belonging to the standard library[1] (installed with Python) and third-party modules written by different developers and installed through 'pip' (in this case, we can use the variable `__version__` to know the version of a module).

The `math` module

The `math` module is a standard module that includes a collection of mathematical functions for working with integers and real numbers (for complex numbers, we should use `cmath` . It is one of Python's fundamental modules. Its functions are implemented in C, making them faster than a pure Python implementation. Here are a few examples of functions.

Function	Description
`gcd(*ints)`	Computes the GCD of a list of integers.
`factorial(n)`	Computes the factorial of an integer.
`sqrt(x)` and `cbrt(x)`	Cmpute, respectively, the square and the cubic root.
`ln(x)` , `exp(x)` , `sin(x)` ...	Standard math functions.
`pi` , `e` , ...	Mathematical constants.
`prod(iterable,start)`	Computes the product of the elements in an iterable with a start value.

For instance the factorial function can be rewritten as (though not recommended since the function already exists):

```py
def fact(n):
    return math.prod(range(2,n+1),start=1)
```

The `random` module

The `random` module includes a number of helper pseudo-random functions for a variety of distributions. The module supports both discrete and continuous distributions. The standard documentation, however, advises against using this module for cryptographic purposes. Here are a few functions from this module:

Function	Description
`randint(a,b)`	Generates a uniform random integer value in the interval $[a, b]$.
`random()`	Generates a uniform random continuous value in the interval $[0, 1]$.
`gauss(mu,sigma)`	Generates a normal random continuous value $\rightsquigarrow N(\mu, \sigma)$.
`choice(iterable)`	Picks a random element from an iterable.
`shuffle(sequence)`	Randomly rearranges the element of a syntax.

Let's write a function that takes many non-empty sequences, then returns a random element from a random sequence:

```py
def general_choice(*seq):
    return random.choice(random.choice(seq))
```

[1]The complete documentation can be found on https://docs.python.org/3/library/index.html

The `datetime` module

The `datetime` module includes *classes* for handling times and dates. By classes, we mean particular data structures in which data is associated with functions. This module implements time and date arithmetic. For a date `dt` , we can access the following attributes: `dt.year` , `dt.month` and `dt.day` . For a time `t` , we can access the following attributes: `t.hour` , `minute` , `second` and `microsecond` . We can use the following *methods*:

Function	Description
`t-s`	Computes `timedelta` measuring how much time between the two dates.
`t+s`	Computes the date after a `timedelta` object.
`weekday`	Computes the day of the week.

Consider a program that tells the user on the day of the week he was born.

py
```
s=input("What is you birth date (MM/DD/YYYY)? ")
month,day,year=map(int,s.split("/"))
date=datetime.date(year,month,day)
days=["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
print(f"You were born on {days[date.weekday()]}")
```

The `numpy` module

`numpy` (also imported as `np`) is one of the most popular non-standard Python packages. It provides a range of efficient vector-based computations. It is considered a basic package since it serves as a basis for many additional modules (particularly in data analysis and artificial intelligence). This module adds support for large multidimensional arrays and matrices in an efficient way. Note that, unlike Python lists, arrays in `numpy` only hold one type of data. One notable advantage of this module is the ability to apply mathematical functions effectively on vectors rather than single elements. Here are some of the basic functions of this module:

Function	Description
`np.arrange(a,b,step)`	Similar to `range` but works with floats.
`np.array(sequence)`	Creates an array (or a matrix) from a sequence.
`array.resize(n,p)`	Resizes a given array.
`array.sum()`	Computes the sum of the elements of an array.
`np.add(a,b)`	Performs the addition of two arrays.
`np.matmul(a,b)`	Computes the product of two matrices.
`np.linealg(a,b)`	Solves the linear system $ax = b$.

The following code creates a data series with two columns: the first one corresponds to numbers from 0 to 5 with a step of 0.05, the second column is the sinus of the first column:

py
```
x=np.arange(0,5,0.05)
y=np.sin(x)
data=np.column_stack((x,y)) #column_stack concatenates x and y to make a two-columns data series
```

The `matplotlib.pyplot` module

This module supports data visualization in a variety of formats (dots, histograms, pie charts, curves, and so on). It is a rather a basic module in data visualization (other modules are more robust like `seaborn`) but it is still very powerful to create graphics. Here are some of its functions:

Function	Description
`plt.plot(x,y)`	Creates a curve such that x are the abscissas and y are the ordinates.
`plt.scatter(x,y)`	Creates a dot graph such that x are the abscissas and y are the ordinates.
`plt.show()`	Displays the graphics.

To plot the previous sinus series, we will use:

```py
plt.plot(x,y,'.-',mec = 'r', mfc = 'r',markersize=2)
plt.show()
```

This produces the following figure: