# Introduction to programming.

# Lecture 4

Handout freely adapted from material from Clément Guérin.

## Raise exception

The **raise** command should be called like this :

**raise** $NameError(string)$

It will immediately **stop** the code, write the line number where the **raise** command has been called and write $NameError : string$. It should be used to inform that the user is doing something that is not expected although strictly speaking it is not forbidden by Python. You can use the raise keyword to signal that the situation is exceptional to the normal flow.

Related to this is the $\textbf{Assert} statement

## Assert statement

Syntax

**if** condition :

**assert** $AsserionError, string$

If the condition is true, the program continue. If the condition is false, the program stops and throws an error.

The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement).

Although very similar the two commands usually do not serve the same purpose. The first one is meant to deal with the unexpected. The second one helps the programmer to have a better understanding of (and control on) his/her code. For instance, you can use to check that a variable has an expected value.

In [ ]:

```
while false:
    number = int(input("The table of which number do you want? "))
    if 9<number:
        raise ValueError("The number is greater than 9. Please do it again.")
    if 0<number<10:
        for i in range(1, 11):
            print(number,' x ',i,' = ',number*i,sep='')
```

In [146]:

```
#Example
def sendmoney(a):
    if 1000>a:
        print("the money has been sent")
        return a
    else:
        raise ValueError("Not enough money on your account")

sendmoney(500)
```

```
the money has been sent
```

Out[146]:

```
500
```

In [ ]:

```
a=12
#b=0
b=9
print('the division a/b is equal to',)
assert b!=0, 'the denominator is null'
print(a/b)
```

# Functions

## Syntax

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

We have the following components:

- Keyword **def** marks the start of function header.
- A function name my_function to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
- Arguments through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does. The command **help**(function_name) will return the text written there.
- One or more python statements describing the instructions. Statements are all indented.This block of text is called the **body** of the function
- An optional return statement to return a value from the function.

## Calling a function

To call a function we simply type the function name with appropriate argument value(s)--sometimes called actual parameter(s).

## Argument data types

You can pass to a function any data types: string, number, list, dictionary etc. You can also pass to it another function, even a method.

## Default argument value

An argument can get a default value. If a function is called with an empty argument, this default value is used

## Undefinite number of arguments

If you do not know how many arguments that will be passed into your function, add a * before the argument name in the function definition. This way the function will receive a tuple of arguments, and can access the items accordingly.

## Return

Returns a value to the caller of the function. Two points to remember:

- optional, but required if you want to reuse the output for something else
- multiple values can be returned

## Functions as parameters

It is possible to pass a function as argument to a function. This can give some extra freedom.

In [ ]:

```
# Example

def square(x):
    return x*x

square(2)

def add_number(x,y):
    """addition"""
    sum=(x+y)
    print(sum)
    return sum

add_number(1,2) # 1 and 2 are argument values

def find_max(a,b):
    """Find the max"""
   if(a > b):
      print(a,"is greater than",b)
   elif(b > a):
      print(b,"is greater than",a)

find_max(30, 45)  #Here we call the function and pass two values as arguments
find_max(45, 30)
```

In [ ]:

```
## argument data types

def my_function(str1,str2): #string
    print(str1)
    print(str2)

my_function("I'm string 1", "I'm string 2")

def my_function(L1,L2):  # lists
    print(L1)
    print(L2)

L1=['a','b','c']
L2=[1,2,3]
my_function(L1,L2)

def my_function(S1,S2):  # sets
    print(S1)
    print(S2)

S1={1,2,3}
S2={1,2,2,3}
my_function(S1,S2)

def my_function(dico1,dico2): # dico
    print(dico1)
    print(dico2)
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
             'Fanny':'fanny@trucmuch.lu',\
             'Robert':'robert@trucmuch.lu',\
       'Stephanie': (6812424239),\
       0:2}
dico2={0:7,'x':'x@trucmuch.lu'}

my_function(dico1,dico2)

def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

In [ ]:

```
## Default argument value

def my_function(country = "Norway"): # I am defining a default value for the argument
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

In [ ]:

```
# UNDEFINITE NUMBER OF ARGUMENTS

def adder(*num):
    sum = 0

    for n in num:  ## for loop over the tuple elements
        sum = sum + n

    print("Sum:",sum)

adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)



def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

In [ ]:

```
def first2items(list1):
  return list1[0], list1[1]

a, b = first2items(["Hello", "world", "hi", "universe"])
print(a + " " + b)
print(b + " " + a)
```

In [ ]:

```
# Passing a function as argument

def sum(val1,val2):
    return val1+val2

def prod(val1,val2):
    return val1*val2

def do_something(foo,val1,val2):
    return foo(val1,val2)

do_something(sum,3,4)
#do_something(prod,3,4)
```

# Mutable vs immutable arguments

In Python, immutable objects are those whose value cannot be changed after assignment or initialization. If you manage to change their value, it is not really a change, because you have in fact created a "second" object, with a new ID (a new memory address). Mutable objects are those whose value can be changed after their assignment or initialization. When the value of a mutable variable is changed its memory is not reallocated.

Numbers (integers, floats), strings and tuples are immutable. List, dict and set are mutable objects.

Depending on whether the given argument is mutable or not, you can modify it or not in the body of the function. Depending on the type of the arguments the output of the function changes.

In [ ]:

```
# Non-mutable arguments

def incrementation(x,k): ## x becomes y and
    x+=k
    # return  x
y=3
incrementation(y,10)# we try to change x into y
#y=incrementation(y,10) #To make it work we need to return a value and the assignment i
s there to store the value somewhere
y         # we get 3 because of this
```

In [ ]:

```
# Mutable argument
def incrementation(x,k):
    x+=k
L=[1,'truc',3]
incrementation(L,[2])#creates a variable x which is equal to L then x+=[2]
L
```

In [ ]:

```
#  Mutable argument--function that changes all  values of a dictionnary to a single str
ing argument.

def change(dico,c):
    for key in dico.keys(): ## to loop trhough all the keys
        dico[key]=c         ## to change the value of the key to c
dico={'k1' : 'truc', 'k2' : 'truc2'} ## example of a dictionary

change(dico,'othervalue') ## application of the function
```

# Nested functions

A nested function is a function which is defined inside another function. The enclosing function is called the outer or parent function, the nested function is called inner or child function. The inner function is "protected" from the outside world, and it cannot be called from outside the outer function.

# Recursive function

A recursive function is a function that calls itself (in its body).

A recursive function should not be confused with an iterative function. Recursion is the calling of a function by itself one or more times in its body. There is iteration when a loop repeatedly executes until the controlling condition becomes false.

Often an iterative function is more efficient.

Here are some examples.

In [6]:

```
# Example 1

def outer():                #outer or parent function
    x=3
    def inner():             #inner or child  function
        print(x)
    inner()                  # inner function called
#inner()                    # inner function cannt be called from the outside
a=outer()
print(a)
```

```
3
None
```

In [9]:

```
# Example 2


def outer():                #outer or parent function
    x=3
    def inner():             #inner or child  function
        y=4
        print(x+y)
    inner()
a=outer()
print(a)
```

```
7
None
```

In [150]:

```
def countdown(n):
    if n==0:                # base case
        print("completed")
    else:                   # recursive case
        print(n)
        countdown(n-1)      # recursive call

countdown(3)
```

```
3
2
1
completed
```

In [2]:

```
def factorial(n):
   # print("factorial has been called with n = ",n,sep='')
    if n==1:
        return 1
    else:
     #   print("intermediate result for ", n, " * factorial(" ,n-1, "): ",result,sep
='')
        return n*factorial(n-1)


factorial(4)
```

Out[2]:

```
24
```

In [ ]:

```
#The Fibonacci sequence is a sequence of numbers in which the nth number in
# the sequence is obtained by adding the two previous numbers in the sequence.

def Fibo(n):
    "F(n)=F(n-1)+F(n-2), F(1)=F(0)=1"
    if n<0:
        raise ValueError("Fibonacci terms begin at 0") # without this, Fibo(-1) would r
un forever.
    elif n==0:
        return 1 # First initial case
    elif n==1:
        return 1 # Second initial case
    else:
        return Fibo(n-1)+Fibo(n-2) # Here is the recursive call

def FFibo(n):
    """calculate the number of calls of Fibo(n)"""
    return 2 * Fibo(n) - 1  ## 1999 paper by John Robertson
```

In [ ]:

```
Fibo(500)
#FFibo(6)
```

In [ ]:

```
[Fibo(n) for n in range(18)] # to display the Fibonacci sequence up to 18-th term as a
 list
```

In [ ]:

```
#Fiboiterative
def Fiboiter(n):
    if n<0:
        raise ValueError("Fibonacci terms begin at 0") # without this, Fibo(-1) would r
un forever.
    elif n==0:
        return 1 # First initial case
    elif n==1:
        return 1 # Second initial case
    else:
        x=1 #0 th element
        y=1 # 1th element
        for i in range(1,n):
            # x i-1-th element and y to be the i-th element
            x,y=y,x+y
            # x i-th element and y to be the i+1-th element
        return y
```

In [ ]:

```
print(Fibo(10),"=",Fiboiter(10))
```

In [ ]:

```
Fiboiter(500)
#FFiboiter(6)
```

In [5]:

```
# Prints syracuse sequence for a given starting value k.
# Syracuse sequence. First term is k, then calculate the following terms using the foll
owing formula:
# i+1:=i/2 if i is even or 3xi+1 if i is odd. The sequence stops when 1 is returned
# Example for k=1. Syracuse sequence is 0, 4, 2, 1

## Syracuse sequence
def syr(k):
    """returns the Syracuse sequence itself as a list"""
    if k<=0:
        raise ValueError("Only for intergers")
    if k==1:
        return [1]
    else:
        if k%2==0:
            return [k] + syr(int(k/2))
        else:
            return [k] + syr(int(3*k+1))
def ssyr(k):
    """returns the position of 1 in the list"""
    return len(syr(k))-1
```

In [6]:

```
syr(7)
```

Out[6]:

```
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

In [7]:

```
ssyr(7)
```

Out[7]:

```
16
```

## Scope of a variable

In Python, a variable name can be thought of as a reference to a value. When we do the assignment a = 2, here 2 is an object stored in memory and a is the name we associate it with.

The scope is the part of the programme where the variable and its associated value can be accessed.

The rules are:

- The scope of every variable declared outside the functions is global. This means the variable is visible from anywhere in the programme.
- Unless stated otherwise, the scope of every variable declared within a function is local (to that function). Exceptions are when the keywords **global** and **nonlocal** are used. With **global** we overwrite the name binding initially made at the global level. With the **nonlocal** keyword we overwrite the name binding declared at the immediately higher level.
- To determine in which order Python should access a variable and its associated value, you can use the LEGB rule. LEGB rule stands for Local, Enclosed, Global, Built-in.

Example of built-in variable: underscore (*), used to ignore the values. If you don't want to use specific values while unpacking, just assign that value to underscore*().

In [12]:

```
## uderscore as a built-in variable

## ignoring a value
a, _, b = (1, 2, 3) # a = 1, b = 3
print(a, b)

## ignoring multiple values
## *(variable) used to assign multiple value to a variable as list while unpacking
## it's called "Extended Unpacking"
a, *_, b = (7, 6, 5, 4, 3, 2, 1)
print(a, b)
```

```
1 3
7 1
```

In [ ]:

```
# In the following examples, what will be printed?
#                            what is the value of x after the call?
#                            do we get error message?
x=2
def ExVar1():
    print(x)
def ExVar2():
    x=5
    #y=5
    print(x)
def ExVar3():
    print(x)
    x=5
    #y=5
def ExVar4():
    print(x)
    print('x=1')
```

In [ ]:

```
ExVar1()
ExVar2()  # global variables are protected--try with y=5 instead of x=5 in the function
ExVar3() #
ExVar4()
```

In [21]:

```
# GLOBAL Changing the value of a global variable from inside the function using global

x = 0 # global variable

def add():
    global x
    x = x + 2 # increment by 2
    print("Inside the function, x is", x)

add()
print("Outside the function x is", x) # Is is still 0?
```

```
Inside add(): 2
In main: 2
```

In [32]:

```
# NONLOCAL is restricted to the immediately higher level

x='grand-father'  # global variable
def f1():
    x='daddy'
    print("Inside f1, x is", x)
    def f2():
        #global x
        nonlocal x
        print("Inside f2, x is", x)
    f2()
f1()
```

```
Inside f1, x is daddy
Inside f2, x is grand-father
```

In [33]:

```
#Difference between global and non local keywords==overlap problem
def examplenothing():
    x='changedinexample'
    def insidenothing(): # inside... is a function only defined in the scope of exampl
e...
        x='changedinside' #x is not global within the scope  of insidenothing.
    insidenothing()
    print("the x in example... is ",x)

def exampleglobal():
    x='changedinexample'
    def insideglobal():
        global x        # x is understood as a global variable, here.
        x='changedinside'
    insideglobal()
    print("the x in example... is ",x)

def examplenonlocal():
    x='changedinexample'
    def insidenonlocal():
        nonlocal x      # x is understood as a local variable of examplenonlocal
        x='changedinside'
    insidenonlocal()
    print("the x in example... is ",x)
```

In [34]:

```
x='notchanged'
examplenothing() #x is not changed inside the first function
print("global x is ",x) # x is not changed globally
print(5*'-')
x='notchanged'
exampleglobal() #x is not changed inside the first function
print("global x is ",x) # x is  changed globally
print(5*'-')
x='notchanged'
examplenonlocal() #x is  changed inside the first function
print("global x is ",x) # x is not changed globally
print(5*'-')
```

```
the x in example... is  changedinexample
global x is  notchanged
-----
the x in example... is  changedinexample
global x is  changedinside
-----
the x in example... is  changedinside
global x is  notchanged
-----
```

One should not use **nonlocal** with recursively defined functions. It leads to a SyntaxError.

In [49]:

```
#Fibonacci sequence with a count of the number of recursive calls
t=0
def Fibo(n):
    "F(n)=F(n-1)+F(n-2), F(1)=F(0)=1"
    global t # Declare that t should be considered as global
    t+=1     # We do something on t
    if n<0:
        raise ValueError("Fibonacci terms begin at 0")
    elif n==0:
        return 1
    elif n==1:
        return 1
    else:
        print(t)
        return Fibo(n-1)+Fibo(n-2)

# Could you evaluate how fast is the number of calls growing?
#t=0
Fibo(4)
#t
```

```
1
2
3
7
```

Out[49]:

```
5
```

In [39]:

```
t=0
2*Fibo(5)-1
```

```
1
2
3
4
8
11
12
```

Out[39]:

```
15
```

In [42]:

```
t
```

Out[42]:

```
15
```

## Adding a "help" comment to your function

If you want to add a sentence that will be printed when you call $\mathbf{help(yourfunction)}$ you should simply add this sentence as a string to the first line of the body of the function. It does not do anything in the script.

In [51]:

```
def nicelydocumented():
    "Unfortunately, it does not do much. It just returns True all the time." # Help com
ment.
    "This line is not seen" #This line won't be shown.
    return True
```

In [53]:

```
nicelydocumented
```

Out[53]:

```
True
```

In [54]:

```
help(nicelydocumented)
```

```
Help on function nicelydocumented in module __main__:

nicelydocumented()
    Unfortunately, it does not do much. It just returns True all the time.
```

# Keywords argument(s)

When we call a function with some values, these values get assigned to the arguments according to their position. These are called called positional arguments.

Python allows functions to be called using keyword arguments. Such arguments are passed as "name=value" instead of just "value". When we call a function this way, the order (position) of the arguments does not matter.

In [8]:

```
def greet(name, msg='goodbye'):
    """
    This function greets to
    the person with the
    provided message.

    If message is not provided,
    it defaults to "Good
    morning!"
    """

    print("Hello",name + ', ' + msg)

greet("John")
greet("Kate",'how you doing?')
greet("Bruce","How do you do?")
```

```
Hello John, goodbye
Hello Kate, how you doing?
Hello Bruce, How do you do?
```

In [62]:

```
# 2 keyword arguments

greet(name = "Bruce",msg = "How do you do?")

#2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

# 1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

```
Hello Bruce, How do you do?
Hello Bruce, How do you do?
Hello Bruce, How do you do?
```

In [55]:

```
print('c','g',sep="*",end=" ** ") #'c' is a positional argument whereas
                                  #'endofthe[...]printline' is a keyword argument for
the keyword 'end'.
print('g','c')
```

```
c*g ** g c
```

In [64]:

```
def Displayingarguments1(a,b,c):#A simple example with only positional arguments
    "Displays the arguments, one on each line"
    print("positional argument a is ",a)
    print("positional argument b is ",b)
    print("positional argument c is ",c)
```

In [65]:

```
Displayingarguments1('yes',(1,2,3),'no') #As expected
```

```
positional argument a is  yes
positional argument b is  (1, 2, 3)
positional argument c is  no
```

In [66]:

```
Displayingarguments1(b='yes',a=(1,2,3),c='no')# You can call positional arguments as ke
yword arguments
```

```
positional argument a is  (1, 2, 3)
positional argument b is  yes
positional argument c is  no
```

In [69]:

```
#You need to give the good number of positional arguments
Displayingarguments1('yes',(1,2,3))
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-69-a7f5462f25de> in <module>
      1 #You need to give the good number of positional arguments
----> 2 Displayingarguments1('yes',(1,2,3))

TypeError: Displayingarguments1() missing 1 required positional argument:
 'c'
```

In [70]:

```
#You need to give the good number of positional arguments
Displayingarguments1('yes',(1,2,3),'no',3)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-70-d707e9ecd5c4> in <module>
      1 #You need to give the good number of positional arguments
----> 2 Displayingarguments1('yes',(1,2,3),'no',3)

TypeError: Displayingarguments1() takes 3 positional arguments but 4 were
 given
```

In [72]:

```
#You cannot give two values to b
Displayingarguments1('yes',(1,2,3),'no',b=(1,2))
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-72-1764298375a0> in <module>
      1 #You cannot give two values to b
----> 2 Displayingarguments1('yes',(1,2,3),'no',b=(1,2))

TypeError: Displayingarguments1() got multiple values for argument 'b'
```

In [76]:

```
#No positional argumentation is allowed after a keyword argumentation
Displayingarguments1('yes',b=(1,2,3),'no')
```

```
  File "<ipython-input-76-2a86d21a4f23>", line 2
    Displayingarguments1('yes',b=(1,2,3),'no')
                                         ^
SyntaxError: positional argument follows keyword argument
```

In [80]:

```
def Displayingarguments2(a,b,c,kw1='defautkw1',kw2='defautkw2',kw3='defautkw3'):#An exa
mple with both positional
                                                                                #and ke
yword arguments
    "Displays the arguments, one on each line"
    print("positional argument a is ",a)
    print("positional argument b is ",b)
    print("positional argument c is ",c)
    print("keyword argument kw1 is ",kw1)
    print("keyword argument kw2 is ",kw2)
    print("keyword argument kw3 is ",kw3)
```

In [81]:

```
Displayingarguments2('yes',(1,2,3),'no','turn',3,'nothing') #Each argument is called as
a positional argument
```

```
positional argument a is  yes
positional argument b is  (1, 2, 3)
positional argument c is  no
keyword argument kw1 is  turn
keyword argument kw2 is  3
keyword argument kw3 is  nothing
```

In [82]:

```
Displayingarguments2('yes',(1,2,3),'no') #First three arguments called are positional,
                                         #other arguments are empty, and get their defa
ut values
```

```
positional argument a is  yes
positional argument b is  (1, 2, 3)
positional argument c is  no
keyword argument kw1 is  defautkw1
keyword argument kw2 is  defautkw2
keyword argument kw3 is  defautkw3
```

In [83]:

```
Displayingarguments2('yes',(1,2,3),'no','turn', kw3='nothing') #3st three are called as
positional arguments
                                                               #kw1 is called as a posi
tional argument
                                                               #kw2 is given a defaut v
alue
                                                               #kw3 is called as keywor
d argument
```

```
positional argument a is  yes
positional argument b is  (1, 2, 3)
positional argument c is  no
keyword argument kw1 is  turn
keyword argument kw2 is  defautkw2
keyword argument kw3 is  nothing
```

In [84]:

```
#Do not give keyword arguments before positional ones
Displayingarguments2('yes',(1,2,3),'no', kw3='nothing','turn')
```

```
  File "<ipython-input-84-cd2f54d01c68>", line 2
    Displayingarguments2('yes',(1,2,3),'no', kw3='nothing','turn')
                                                                  ^
SyntaxError: positional argument follows keyword argument
```

In [117]:

```
def Displayingarguments3(a,b,c,kw):#An example with a mutable keyword argument.
    "Displays the arguments, one on each line"
    print("positional argument a is ",a)
    print("positional argument b is ",b)
    print("positional argument c is ",c)
    kw.append(1)
    print("keyword argument is ", kw)
```

In [118]:

```
Displayingarguments3(1,2,3,kw=[])# The defaut value of kw is changing!
```

```
positional argument a is  1
positional argument b is  2
positional argument c is  3
keyword argument is  [1]
```

# Yield

Yield is a keyword that is used like return, except the function will return a new object called **generator**. A generator is an iterable, viz. you can go through its elements with a for loop. The difference is that the generated values are not stored in memory, and are generated on the fly. Hence they can be used only once.

In [145]:

```
def createGenerator():
    mylist = range(3)
    for i in mylist:
        yield i*i

mygenerator = createGenerator() # create a generator
print(mygenerator) # mygenerator is an object!
for i in mygenerator:
    print(i)
```

```
<generator object createGenerator at 0x0000020E28ADD2A0>
0
1
4
```