Bocconi

COMPUTER SCIENCE

code 30424 a.y. 2020-2021

Lesson 12

Functions and exceptions

Università Bocconi
MILANO

ITEC - IT Education Center

Objectives of the lesson

- Learn how to create custom functions
- Handle errors

Functions in Python

- In Python there are many functions immediately available, called **built-in functions** such as print, input, str, int and float
- Many other functions are available in the Python **standard library** modules, such as **math** and **random**, or in additional third-party libraries
- In addition to these functions, it is also possible to create custom functions to perform tasks and operations based on our needs

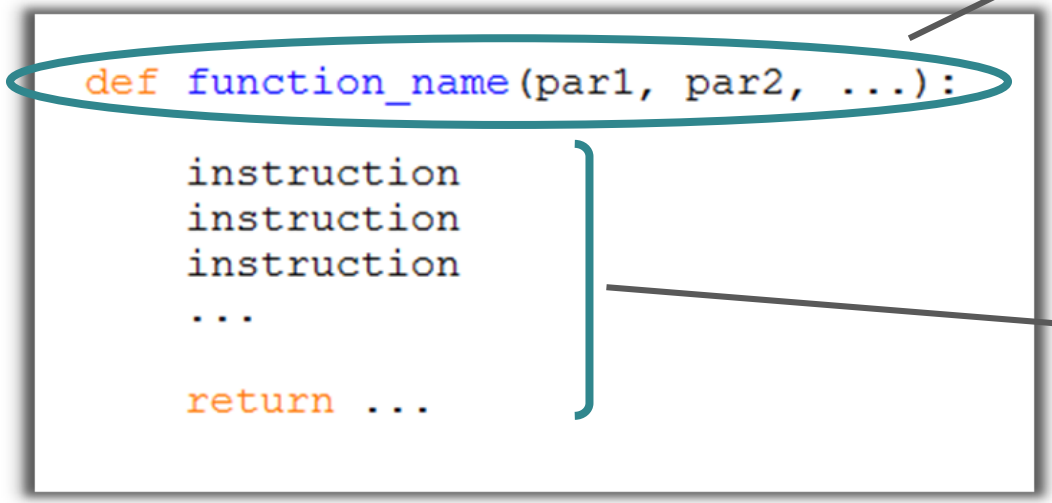
Functions

- A function is a group of instructions to which is given a name and that performs a specific task within a program
- Functions can perform tasks of any type: execute a calculation, perform an action, create an object, update values etc.
- In addition to performing calculations or actions, functions are important because they facilitate the creation of a **modular program**

Creating custom functions

A function is made up of two parts:

- header
- body



The header is made up of:

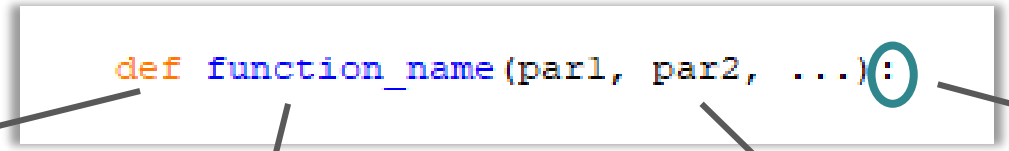
- the keyword def
- the name of the function
- the list of parameters between parentheses
- a colon

The body:

- contains all the instructions of the function
- can end with the return statement

The header of a function

Each element of the header has a specific task:



The keyword def is needed to start *defining* a function

The header always ends with the symbol « : »

The name uniquely identifies the function in the program.

To assign a name to a function we must follow some rules:

- the first character must be a letter or an underscore
- keywords or spaces cannot be used
- after the first letter, we can use letters (uppercase or lowercase), numbers or underscores
- Python distinguishes between uppercase and lowercase (case sensitive)

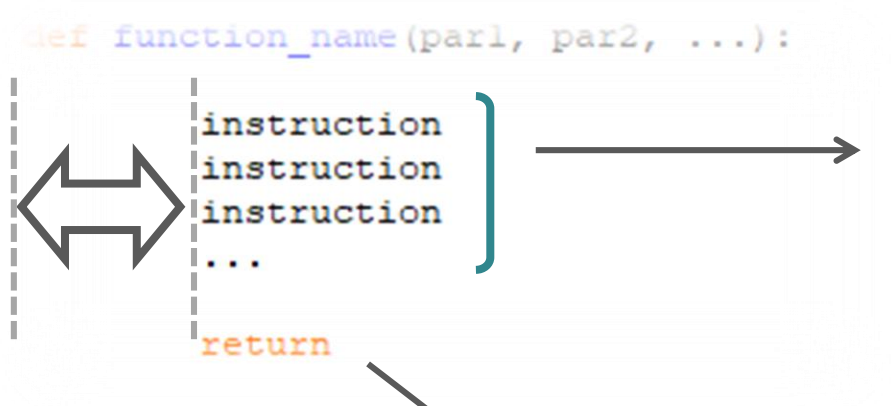
Parameters specify which arguments will be passed to the function:

- names are separated by commas
- parameters can be mandatory or optional

The body of a function

The body of a function contains a list of instructions (called block):

- all the instructions and the return statement must be indented
- it ends with the return statement when it is necessary that the function returns a value to the calling instruction. Without the return statement the result of the function is not stored



Each block can contain any type of instruction: calculations, conditional statements, call of other functions etc.

All the rows of the body must be indented

The return statement is necessary if the function must return a result to the calling instruction

Calling a function

- Defining a function allows Python to understand which task must be performed, but it does not execute it
- Each time we want to execute a function it is necessary to **call it** using the syntax:

```
function_name()
```

- If parameters have been specified in the definition, their value must be written in parentheses
- A function can be called in the Python shell, by a program or by another function

Examples of definition and call of a function

- **hello** function

```
def hello():
  print('Hello my friend, is everything ok?')
  print('Hope so!')
```

```
>>> hello()
Hello my friend, is everything ok?
Hope so!
```

- **pleased** function

```
def pleased():
    name = input("What's your name? ")
    print('Pleased to meet you, ' + name + '!')
```

```
>>> pleased()
What's your name? Martin
Pleased to meet you, Martin!
```

- **square** function

```
def square(x):
    print('The square of', x, 'is:', x*x)
```

```
>>> square(5)
The square of 5 is: 25
```

Function arguments

- Often we need to pass to a function some arguments, that are inputs it needs to perform its task. In these cases it is necessary to specify it through parameters when defining the function
- Parameters:
  - explain to Python that data will have to be passed to the function when it is called
  - allow to specify how the data should be used to perform the operations required by the function
- Parameters are specified in the parentheses after the name of the function, separated by commas

```
def division(num1, num2):
    print('The result of the division is:', num1/num2)
```

Mandatory and optional parameters

- Parameters can be mandatory or optional: when defining a function, it is always necessary to specify all the mandatory parameters first, then the optional ones
- For each optional parameter it must also be indicated the default value, which will be used if not specified in the function call
- Optional parameters are written in the function definition as:

parameter=default_value

- The general syntax for defining a function with parameters is the following:

```
def function(par1, par2, par3=value, par4=value):
```

Calling a function with parameters

- If mandatory parameters are specified when defining a function, a corresponding number of arguments should be provided when the function is called
- When a function with parameters is called, it is necessary to specify the arguments by position (in the same order in which the parameters were defined), or to use the assigned name (keyword arguments)
- In the call it is possible to use both arguments by position and keyword arguments, but in this case it is necessary to write first the arguments by position and then the keyword arguments
- It is possible to pass a variable as an argument to a function

Example of definition and call of functions with parameters

```
def calc(par1, par2, par3=10):
    total = (par1 + par2) / par3
    return total
```

```
>>> calc()
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    calc()
TypeError: calc() missing 2 required positional arguments: 'par1' and 'par2'
```

```
>>> calc(15,5)
2.0
```

```
>>> calc(20,80,5)
20.0
```

```
>>> calc(par2=50,par3=20,par1=30)
4.0
```

```
>>> calc(10, par2=5)
1.5
```

```
>>> calc(10, par3=3, par2=5)
5.0
```

```
>>> calc(par2=3, 5)
SyntaxError: positional argument follows keyword argument
```

Productive and void functions

- A **productive** function is a set of instructions that performs a specific task **returning**, when it ends, **a value to the instruction that called it**
- A function that performs a specific task, but that when it ends **does not return any value to the instruction that called it**, is a **void** (empty) function
- Productive functions always end with the return statement, which closes the body of the function. Any other instructions after the return statement are not taken into account
- The difference between void and productive functions therefore consists in:
  - the return statement (always part of productive functions but not of void ones)
  - the ability to return a value to the instruction that called them so that the value can be reused. If the value is not returned to the calling instruction it cannot be stored or reused

Productive and void functions compared

Productive function	Void function
`def square(x):` `    return x*x`	`def square(x):` `    print('The square of', x, 'is:', x*x)`

`>>> y = square(10)`
`>>> y`
`100`
	`>>> y = square(10)`
	`The square of 10 is: 100`
	`>>> print(y)`
	`None`

The function returned a value that was stored in the **y** variable and that can be used in other instructions	The function showed a text string on screen and returned no value. Variable **y** is empty. The function output cannot therefore be used in other instructions

Local and global variables

- A **global** variable can be reached by any instruction in a program
- A **local** variable can be reached only within its scope, that is within the part of the program in which it was defined, such as a function
- Within a function we can create local variables to facilitate calculations, temporarily store a value, make the code more readable

```
def calc(x, y):
    temp1 = (x + y) / y**2
    temp2 = x**2 - y**2
    return temp1 - temp2
```

```
w = 100

def calc(x, y):
    temp1 = (x + w) / y**2
    temp2 = (x**2 - y**2) * w
    return temp1 - temp2
```

The docstring

- When creating a function it is possible to insert a few lines of explanation or a comment by defining a docstring
- The docstring consists of a text string on one or more lines enclosed in triple quotes, inserted as the first statement immediately after the header, when defining a function
- The docstring content is visible when reading the code, when using the help function or in the *call tip*

```
def multiply(num1, num2, num3):
    ''' The function multiplies the three arguments and returns the result.\n \
        Arguments can be integer or decimal numbers'''
    return num1 * num2 * num3


>>> multiply(
          (num1, num2, num3)
          The function multiplies the three arguments and returns the result.
          Arguments can be integer or decimal numbers
```

Functions with loops and conditional statements

```
def sum_range(num):
  total = 0
  for n in range(1,num+1):
    total = total + n
  return total
```

```
def odd_even(num):
  count_odd = 0
  count_even = 0
  for n in range(1,num+1):
    if n % 2 == 0:
      count_even = count_even + 1
    else:
      count_odd = count_odd + 1
  print("Number of even numbers :",count_even)
  print("Number of odd numbers :",count_odd)
  return count_even
```

```
def discount(quant, price):
  if quant > 100:
    total = quant * price * (1 - 0.4)
  elif quant > 50:
    total = quant * price * (1 - 0.2)
  else:
    total = quant * price
  return total
```

Exceptions

- An exception is an event triggered by an error of various kinds
- If we expect an exception to occur, in Python we can write some code to handle it (*exception handling*)
- When an exception is not handled (*unhandled exception*) it causes an error and the exit from the block of instructions in which it occurs or from the program

Error types

Errors can be divided into three categories:

- Syntax errors
- Runtime errors
- Semantic errors

Syntax errors

- Syntax errors indicate that there is an error in how the code is written
- Python shows an error message (traceback message) in the shell and indicates the point in the code where the error occurs, but does not specify how to correct it
- The error message always starts with SyntaxError

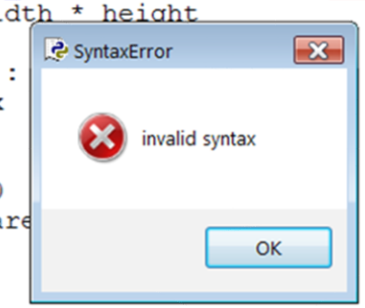
In the shell (after Enter)

```
>>> def rect_area(width, height)
SyntaxError: invalid syntax
>>>
```

```
>>> def square(x):
    y = x * x
    retun y

SyntaxError: invalid syntax
```

In the editor (after Run)

```
def rectangle_area(width, height)
    return width * height

def square(x):
    y = x * x
    retun y

toSquare = 10
result = square
print(result)
```



Runtime errors

- Runtime errors indicate that there is an error in the code, even if the syntax is correct
- Python shows an error message (in the shell only) indicating the part of the code that generates the error and specifying the cause

```
>>> print(result)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(result)
NameError: name 'result' is not defined
>>>
```

```
>>> print(1/0)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(1/0)
ZeroDivisionError: division by zero
>>>
```

Semantic errors

- Semantic errors occur when the program is executed without producing error messages, but the results are not the correct ones (they are inconsistent or not expected)
- They derive from a wrong code design (they are also called logic errors)
- They are tricky to find and require a step by step re-reading of the code, or the use of more sophisticated debugging tools **(Debugger)**

Exceptions handling

- Python provides some instructions for **handling errors**
- Handling errors means preventing them, instructing the program on what to do in case they should occur
- The most common instruction is try… except that lets you specify how errors should be handled
- It is possible to specify what the program should do for all types of errors or only for some of them

Example

Let's create a program that performs a simple division between two integer numbers entered by the user. The program must prevent from any kind of errors occurring while entering numbers:

- Text entered instead of numbers, or decimal numbers
- A zero entered in the denominator
- Any other error

```
try:
    a = int(input('Enter a number: '))
    b = int(input('Enter another number: '))
    print(a/b)

except ValueError:
    print('\nEnter integer numbers only!')
except ZeroDivisionError:
    print("\nOne of the two values is 0! \n \
    Please try again from the beginning with another number")
except:
    print('\nSomething's wrong: try again with other numbers!')
```

Debugging

- Debugging consists in searching for code errors and removing them
- Debugging takes place through a process of re-reading of the code, search for errors, modification of the code and re-execution of the program
- There are specific tools as the Debugger

Tools to learn with this lesson

- Creating custom functions
- Handling errors

Files of the lesson

In the zip file **30424 ENG - 24 - Lesson 12.zip** you will find these files:

- **30424 lesson 12 slides.pdf**: these slides
- .py files with examples of the lesson
- **Exercises 30424 lesson 12**: exercises on the content of the lesson

Book references

Learning Python:

Chapters 7, 10.4, 10.5, 10.6

Assignment:

Exercises of lesson 12