EVERYTHING IS AN OBJECT IN PYTHON

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems and programming for Automation M

3. Functions in Python

Notice

The course material includes material taken from Prof. Chesani and Prof. Martini courses (with their consent) which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Motivation

- Pieces of code that compute/perform a certain activity
- One of the first ways introduced in programming language to **re-use** a certain part of code
  - Allow to tackle the complexity
- Imagine you wrote a single program of 2000 lines, that performs a number of well-identifiable, understandable steps
  - A 2000 lines program is difficult to be understood as a whole…
  - … but if we split the code into parts corresponding to the steps, the comprehension becomes easier (divide and conquer!)
  - If each step can be enclosed in a well thought box, i.e. a function, then the structure of the program is simplified as well

Use

- They are similar to mathematical functions

- Input: a number of **parameters/arguments**
- Output: a value that is **returned**

- Calling a function:
  - Name of the function + round brackets
  - Within the brackets, the values the function will be applied to

- Example: the predefined function `abs(x)` takes in input a number as parameter `x`, and returns its absolute value.

```
y = abs(x)
```

Concept

- A function:
  - has a **name** (e.g. abs())
  - receives some **parameters**
  - is called by specifying values (**arguments**) for each parameter (there are exceptions)
  - once called, applies some **computation** over the arguments
    - two different calls of a function amounts to two different computations
  - finally, it **returns** a value

- Parameters: what the function **expects** according to the **definition**
  - `abs` expects one parameter, that is a number
- Arguments: what is **given** to the function when is **called**
  - `y = abs(x)` I am calling `abs` giving `x` as argument

Functions: result

- Functions **always** return a value (the result).
  - we can save such result in a variable
  - we can use such result directly as the argument of another function call

```
base = int('2'))
exp = int('10'))
# the outcome of int is saved in variables
print(pow(base, exp))
# the outcome of pow is used as argument for print
```

- If the function does not have any value as result, the `None` value is returned

Libraries

- Python provides a set of pre-defined functions
- Developers have defined their own functions, and packaged them into a set of libraries, and made them available
- It is possible to use these functions importing them using the syntax

```
import <library_name> # import
<library_name>.<name_of_function>(arguments) # use

import <library_name> as <alias> # import
<alias>.<name_of_function>(arguments) # use

from <library_name> import <name_of_function> # import
<name_of_function>(argument) # use
```

Defining Functions

- Remember that a function
  - has a **name**
  - receives some **parameters**
  - is called by specifying values (**arguments**)
  - applies some **computation**
  - (usually) **returns** a value

Defining Functions: Prototype

- Remember that a function
  - has a **name**
  - receives some **parameters**
  - is called by specifying values (**arguments**)
  - applies some **computation**
  - (usually) **returns** a value

- The name and the parameters are also referred as the **prototype** or **header**
- They are a sort of a "contract": if you want to use the services of that function:
  - you have to call it properly
  - you have to pass the right arguments

Defining Functions: Parameters

- Remember that a function
  - has a **name**
  - receives some **parameters**
  - is called by specifying values (**arguments**)
  - applies some **computation**
  - (usually) **returns** a value

- The prototype specify what the client should provide to the function for doing the computation: **formal parameters**
  - They are names

- When the function is called, the caller will provide specific values: **arguments** or **actual parameters**
  - They are values/expressions

Defining Functions: Computation

- Remember that a function
  - has a **name**
  - receives some **parameters**
  - is called by specifying values (**arguments**)
  - applies some **computation**
  - (usually) **returns** a value

- The computation is expressed through a block of instructions.
  - Usually, that block is called the **body** of the function.

- The `return <value>` instruction terminates the computation
  - If the block ends without return, the `None` value is returned

Defining Functions: Syntax

```
def <name> (<formal parameters>):
        <instructions block>
```

- **def** is a reserved name of the language
- <formal parameters>: comma separated names
- <instructions block>: sequence of commands all at the same indentation

```
def niceFunction(x):
    y = x*x
    for i in range(y):
        print('You are nice!')
    return y
```

Defining Functions: Example 1

```
def niceFunction(x):
    y = x*x
    for i in range(y):
            print('You are nice!')
    return y


a = niceFunction(3)
print(a)
```

We expect a number, not a list, dict, or string. But the function does not verify the data type.

Best Practices

- The header of a function does not provide any meaningful information on what the function compute: indeed, the contract is separated from its implementation!
  - How to know what a function does?
- Always name your functions with a meaningful name!!!
- It is a best practice to add a comment, in the form of a string, to explain in natural language what the function will compute
  - If you place your explanation immediately after the header, there are automatic tools that will generate the documentation for you

Defining Functions: Example 2

```
def niceFunction(x):
'''
Given a number, computes the power of 2 and prints a
nice message that amount of times

:param x (int): the number for which the power of two
                    will be computed
:return (int): the power of two of x
'''
    y = x*x
    for i in range(y):
            print('You are nice!')
    return y
```

comment

Semantics

- A `def` of function introduces into the internal state a **binding** between the name of the function and its body
  - No computation is done at the time of definition

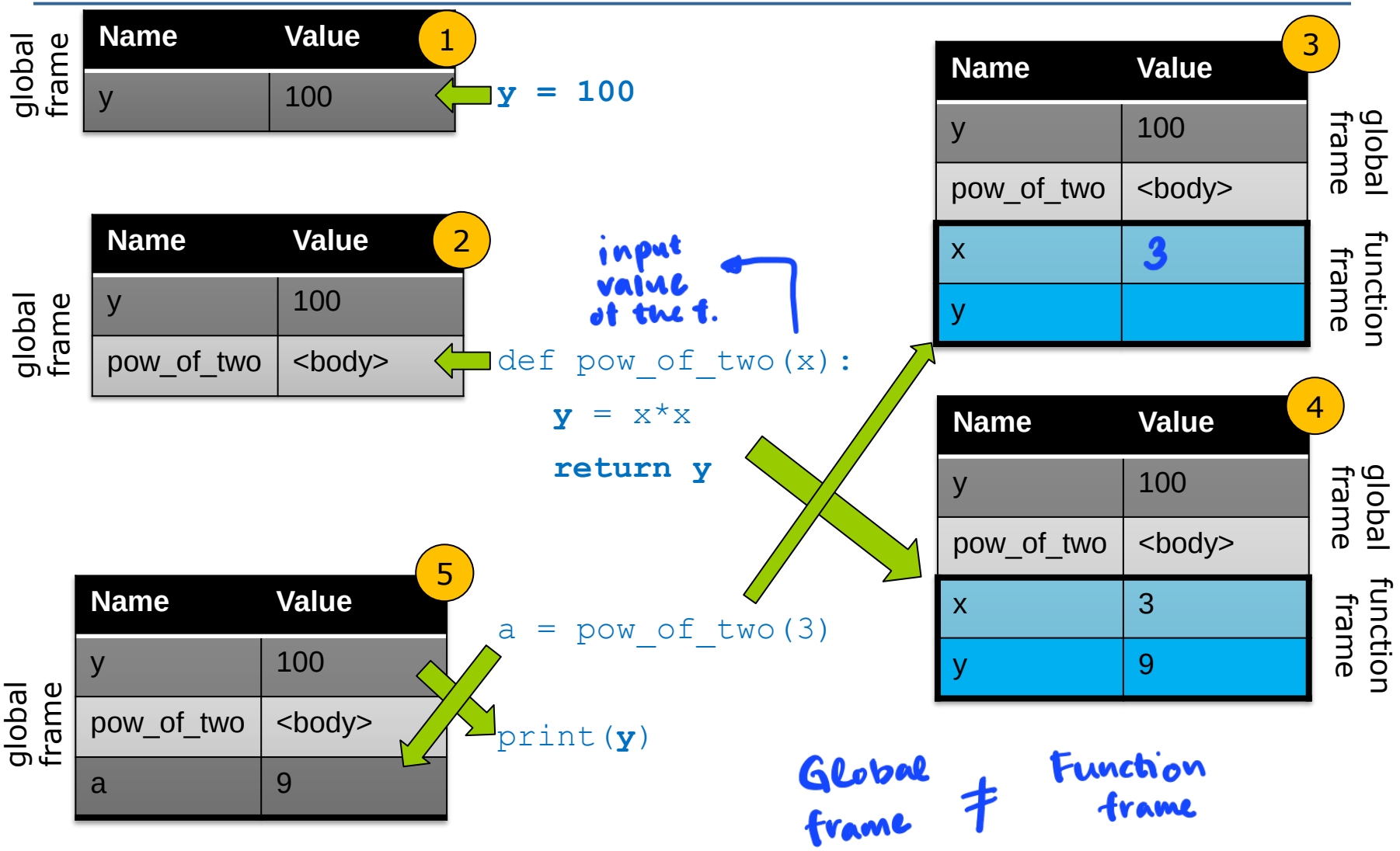
- What happens when, within a function, we use variables names that appear also out of the function?

```
y = 100
def pow_of_two(x):
    y = x*x
    return y

a = niceFunction(3) # a = 9
print(y) # What do we print here? 100? 9?
```

Call step by step

- When a function is called:
  1) its arguments are evaluated, if needed
  2) a frame, specific for that function call, is created
  3) inside the frame, links between the names of the formal parameters and the actual values are created
  4) if any variable appears in the left part of an assignment in any part of the body, the variable is considered local to the body: in the frame the name is prepared waiting to link it to a value
  5) the body is executed
  6) at the return, the execution is interrupted and the value is returned back to the caller
  7) the frame is removed from the internal state; bindings are deleted as well

Call step by step - Example



Visibility of Variables

- When the function is called, the machine looks at the block and for each variable decide if it will treat them as local or global
  - The variable appears in **any left** side of an assignment or is a formal parameter
    - The variable is considered as a **local** variable.
      - Any (global) variable is "covered" by the local one
      - You can not use the variable before assigning a value, unless it is a parameter
  - The variable **does not** appear in **any left** side
    - The variable's value is searched in the **global** frame

Visibility of Variables - Summary

- When a function is called a frame is created
- The frame will contain the links between the variables mentioned in the function (including the parameters), and their values
  - somehow, the global links are hidden by the local frame
- Variables local to a function will disappear at the end of the function!
  - Exception! You can use the keyword `global`: a variable local to a function will become global (and persistent)
    - It will keep existing even after the end of the function

Visibility of Variables - Exercise

- Try to guess the output

```
# 1
a = 5
def test():
    a = 7
    print(a)

print(a)
test()
```

print(a) → 5
test() → 7

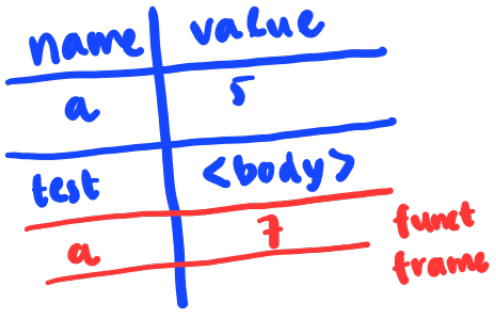
```
# 2
b = 5
def test():
    print(b)
    a = 7
    print(a)

test()
print(a)
```

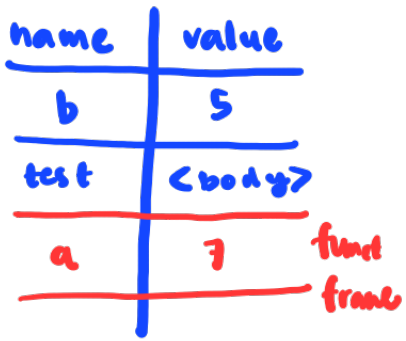
test() → 5
print(a) error
↳ doesn't exist

```
# 3
b = 5
def test():
    print(b)
    a = 7
    b = 3
    print(a)

test()
```

error

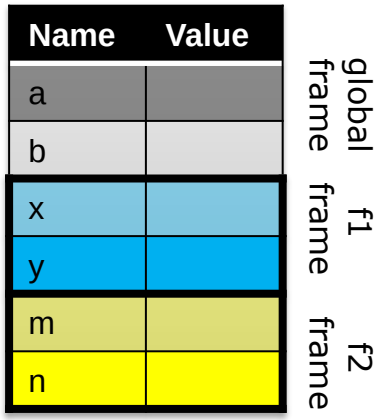
```
# 4
b = 5
def test(b):
    print(b)
    b = 3
    print(b)

test(11)
```





Call Stack

- What if a function will call another function?
  - a new frame the for the second call will be created
  - visibility rules will apply consequently (two different frames and a global frame)



- Suppose we call a function `f1`, that in turn will invoke function `f2`
  - Global frame; `f1` frame; `f2` frame

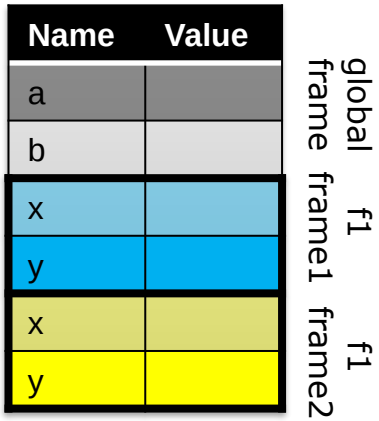
- The local frame for `f2` will be the last to be created, and the first one to be destroyed (LIFO (last in, first out))

- Frames are managed with a **LIFO** priority.
- Usually, such queue is referred as the **call stack**, or the **frame stack**.

Recursive Calls

- What if a function will call itself?

- A function that call itself is named **recursive function**
  - Each new call will generate a new frame, exactly in the same way as it was calling a different function
    - No confusion between variable names



Formal Parameters

- If the number of parameters is fixed and defined, formal parameters are defined by listing their names in the prototype of the function:

```
def f1(p1, p2, p3):
```

- If the number of parameters is variable is possible to define them
  - by specifying the last variable with a * preceding it:
    - that parameter will be linked to a tuple of values, and the number of formal parameters will be variable

```
def f2(a,b, *args):
```

def sum(a, b, *args)
↳ optional Params

  - by specifying the last variable with a ** preceding it:
    - that parameter will be linked to a dictionary

```
def f2(a,b, **args):
```

Formal Parameters - Example

```
def f1(a,b,c,):
    print(a)
    print(b)
    print(c)

def f2(a,b, *args):
    print(a)
    print(b)
    for i in args:
        print(i)
```

```
def f3(a, **kwargs):
    print(a)
    for key in kwargs.keys():
        print(key, '=', kwargs[key])
```

Default Values

- In the prototype it is possible to define **default** values for **optional** formal parameters:
  - the caller can omit to specify that value
  - the Python machine will assume that the formal parameter has actual value the default one
- Syntax: `def f1(a, b, c=5, d='pippo'):`

default val

- Notice that the formal parameters without defaults must always appear BEFORE parameters with defaults

- When calling a function with default values, it is possible to specify the optional arguments calling them by name:

```
x = f1(10, 9, d='pluto')
print(10, 20, sep="-") # prints "10-20"
```

List Comprehension

- Objective: create a new list
  - Starting from an existing iterable source
  - Possibly applying a selection on the values
  - Possibly applying a transformation

```
newlist = [expression for item in iterable if condition == True]
newlist = [transformation for item in source if selection]
```

- **iterable**: the source object from which the elements will be selected
  - **item**: the variable that will assume the values in iterable
- **condition**: a filter that will select for inclusion
- **expression**: the transformation to apply, it will be evaluated and its result will be added to the new list

List Comprehension – Example

- Example: given the list

```
fruitList = ['apple', 'mango', 'cherries', 'ananas', 'apricoat']
```

construct a new list that contains only strings not starting with 'a'.

- Possible approach: a function!

```
def fruitNotA(aList):
    result =[]
    for fruit in aList:
        if 'a' != fruit[0]:
            result.append(fruit)
    return result

newList = fruitNotA(fruitList)
```

- Use list comprehension:

```
newList = [x for x in fruitList if 'a'!=x[0] ]
```

Functions are Objects

- Remember: everything in Python is an object, even functions!
  - Functions can be assigned to variables
  - Functions can be passed as parameters
  - Functions can be returned as results

```
def apply_function(func, value):
    y = func(value)
    return y

n = apply_function(abs,-10)
print(n)
```


```
def return_function():
    return abs

func = return_function()
print(func(-10))
```