Scientific Programming

Lecture A04 – Functions

Andrea Passerini

Università degli Studi di Trento

2019/06/26

Acknowledgments: Alberto Montresor

Table of contents

How to define a function

Functions

Functions are named blocks of code. They take inputs and produce outputs.

Abstract syntax

```
def f(arg1, arg2, ...):
    # the code
    return <result-exp>
```

- The arguments (`arg1`, `arg2`, etc.) are variables that specify how many inputs the function takes.

Calling a function

```
the_result = f(value1, value2, ...)
```

Example

```
def plus(a, b):
    r = a + b
    return r

x = 5
y = 10

z = plus(x, y)
print(z)
```

Why functions?

Motivations

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Fruitful vs void functions

```
from math import sqrt

def hypotenuse(side1,side2):
    return sqrt(side1**2 + side2**2)

def printWarnings():
    print("I never said most of the things I said.")
    print("Yogi Berra")

x = hypotenuse(3,4)
y = printWarnings()
print(x,y)
```

```
I never said most of the things I said.
Yogi Berra
5.0 None
```

A first explanation about naming

The name of the variables passed to the function has nothing to do with the name of the arguments

- In the example, the values of the variables `x,y` are visible inside the function as `a,b`:
- When called,
    - `a` takes the value of `x`
    - `b` takes the value of `y`

```
def plus(a, b):
    r = a + b
    return r


x = 5
y = 10


z = plus(x, y)
print(z)
```

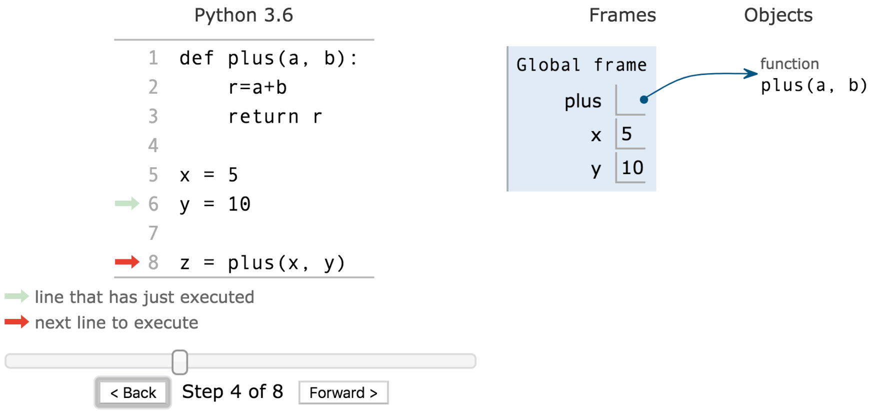
A first explanation about naming

The name of the variables used to store the result inside and outside the call has nothing to do with each other

- In the example, the result is stored
  - in variable `r` inside the function
  - in variable `z` by the caller.
- When the call is concluded,
  - `z` takes the value of `r`

```
def plus(a, b):
    r = a + b
    return r


x = 5
y = 10

z = plus(x, y)
print(z)
```

A first explanation about naming



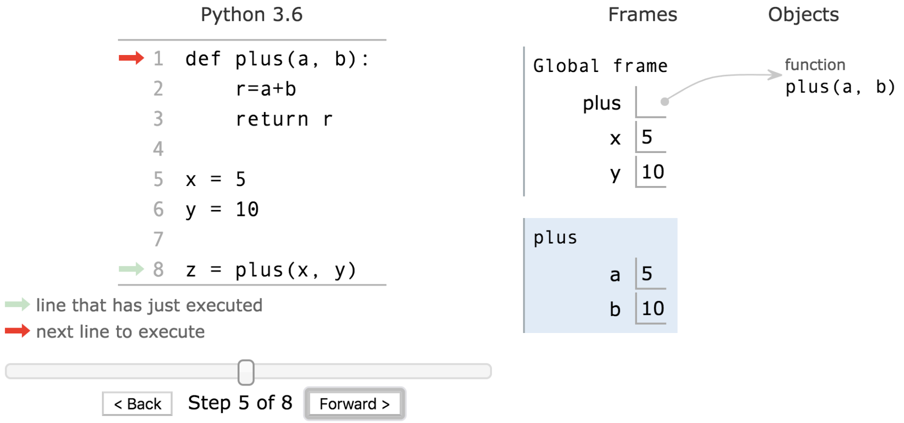
Python 3.6

```
def plus(a, b):
    r=a+b
    return r

x = 5
y = 10

z = plus(x, y)
```

➡ line that has just executed
➡ next line to execute

Step 4 of 8

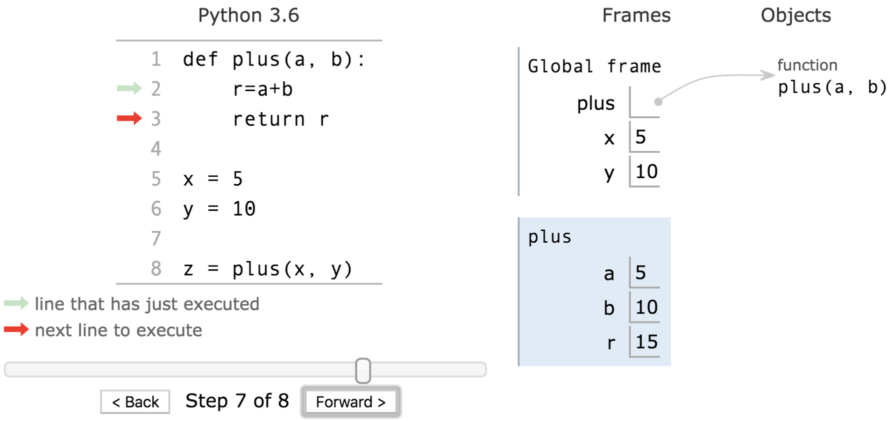
A first explanation about naming



Python 3.6

```
def plus(a, b):
    r=a+b
    return r

x = 5
y = 10

z = plus(x, y)
```

A first explanation about naming

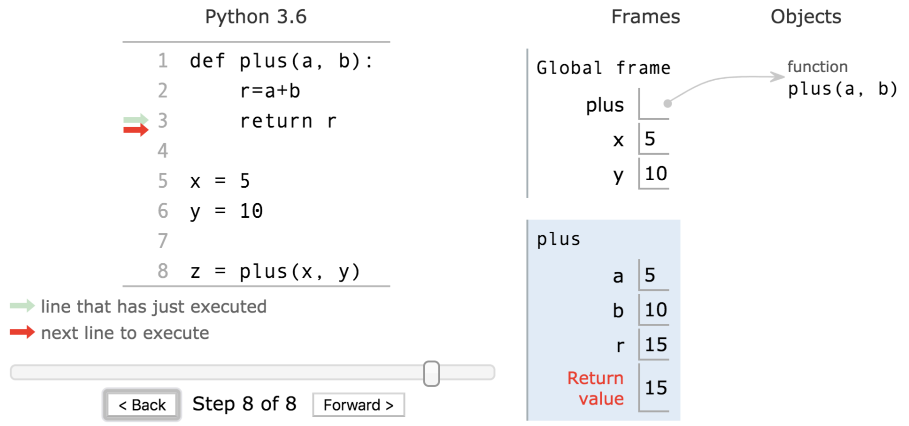


Python 3.6

```
def plus(a, b):
    r=a+b
    return r

x = 5
y = 10

z = plus(x, y)
```

➡ line that has just executed
➡ next line to execute

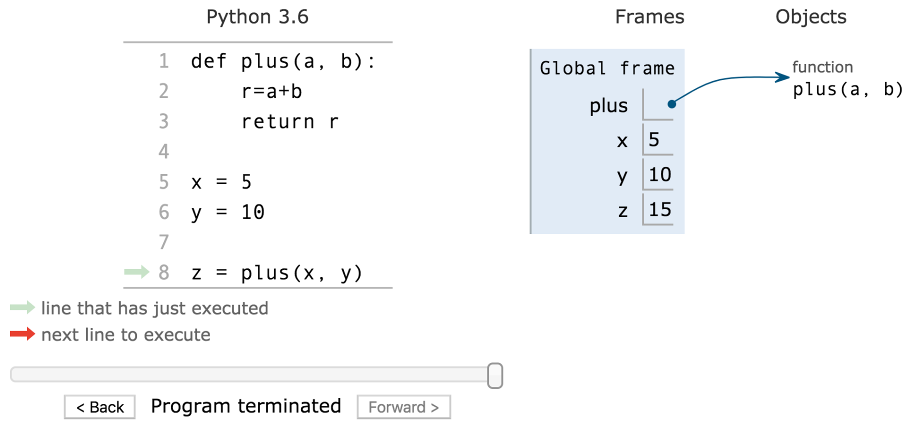
A first explanation about naming



Python 3.6

```python
def plus(a, b):
    r=a+b
    return r

x = 5
y = 10

z = plus(x, y)
```

A first explanation about naming



Python 3.6

```
def plus(a, b):
    r=a+b
    return r

x = 5
y = 10

z = plus(x, y)
```

Function definition

A function does nothing until it is called

```
print("beginning")

def f():
    print("I do stuff")

print("end")
```

```
beginning
end
```

Function definition

If called after its definition, the function is executed without problems

```
print("beginning")

def f():
    print("I do stuff")

f()
print("end")
```

```
beginning
I do stuff
end
```

Function definition

Functions must be defined before they are called

```
print("beginning")
f()

def f():
    print("I do stuff")

print("end")
```

```
beginning
Traceback (most recent call last):
  File "lecture.py", line 3, in <module>
    f()
NameError:  name 'f' is not defined
```

Function definition: some explanations

Unlike many languages, `def` is a statement that:

- creates a new object of type function by reading the code indented after `def`
- assign it to a variable called as the name of the function

The name of the function is like any other variable; can be copied in other variables, used as a function parameters, etc.

```
print("beginning")
def f():
    print("I do stuff")
print(type(f))
fun = f
fun()
print("end")
```

```
beginning
<class 'function'>
I do stuff
end
```

Exercise

Problem

Write a function that given in input a positive integer `n`, returns the factorial of `n`.

```
def fact(n):
    res = 1
    for k in range(1, n + 1):
        res = res * k
    return res

factorials = [fact(n) for n in range(1,11)]
print(factorials)
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Multiple results

A function may return multiple results

```
def multiresult():
    result_1 = "AA"
    result_2 = 0.12
    result_3 = "*"
    return result_1, result_2, result_3
```

Internally, Python interprets the return statement as returning a tuple. In practice, the above code is equivalent to:

```
def multiresult():
    return ("first result", 0.12, "something else")
```

Multiple results

When a "multi-result" function is called, the resulting tuple can be assigned to a variable; elements have to be extracted individually

```
def multiresult():
    return ("first result", 0.12, "something else")

result = multiresult()
res0 = result[0]
res1 = result[1]
res2 = result[2]
print(res0+res2, res1)
```

```
first resultsomething else 0.12
```

Multiple results

Otherwise, the “automatic unpacking” feature of Python can be used

```
def multiresult():
    return ("first result", 0.12, "something else")

res0, res1, res2 = multiresult()
print(res0+res2, res1)
```

```
first resultsomething else 0.12
```

Multiple results

Automatic unpacking only works with the same number of elements

```
def multiresult():
    return ("first result", 0.12, "something else")

res0, res1 = multiresult()

Traceback (most recent call last):
  File "prova.py", line 4, in <module>
    x,y = fun()
ValueError: too many values to unpack (expected 2)
```

Exercise

Problem

Write a function that takes two lists as input and returns their intersection, i.e. the objects that appear in both of them.

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

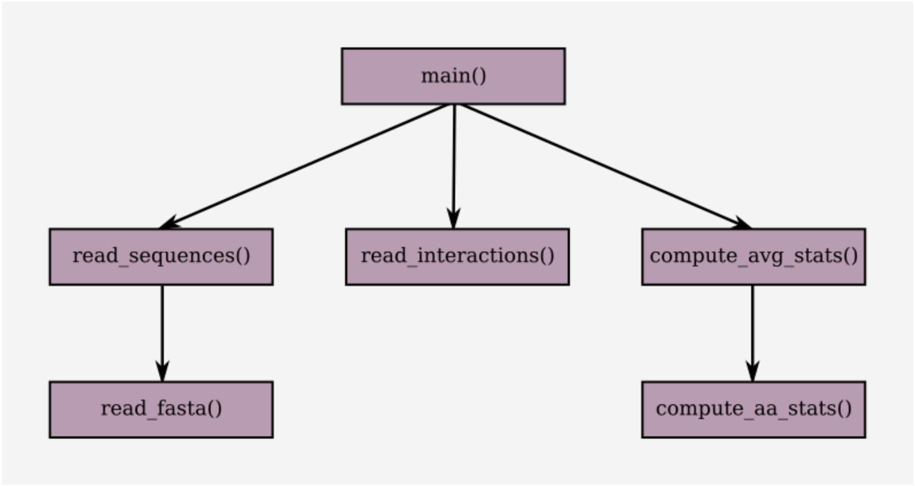
Polymorphism

Like all good functions in Python, `intersect()` is polymorphic. That is, it works on arbitrary types, as long as they support the expected interface - being iterable.

```
print(intersect([1,2,3], [2,3,4]))
print(intersect("ABC", "CBO"))
print(intersect((1,2,4), [3,4,1]))
```

```
[2, 3]
['B', 'C']
[1, 4]
```

Problem decomposition



Problem decomposition

```python
def read_fasta(path):
    """Takes a path to a FASTA file, returns a
    header->sequence dict."""
    return #"1A3A:A", "MANLFKLG..."

def read_sequences(paths):
    """Reads a bunch of FASTA files, returns a
    list of dicts."""
    header_to_seq = {}
    for path in paths:
        header, seq = read_fasta(path)
        header_to_seq[header] = seq
    return header_to_seq
```

Problem decomposition

```
def read_interactions(path):
    """Reads physical protein interactions from a
    file. Returns a list of pairs of strings."""
    return #[("1A3A:A", "5AA3:F"), ("5AA3:F", "5K9C:A")]

def compute_aa_stats(seq1, seq2):
    """Compute amino acid statistics, e.g.
    co-occurrence."""
    return #cooccurrence, #mutual_information
```

Problem decomposition

```python
def compute_avg_stats(sequences, interactions):
    """Takes a list of statistics (in some format) and
    computes the average statistics."""
    stats = []
    for prot1, prot2 in interactions:
        if prot1 in sequences and prot2 in sequences:
            seq1 = sequences[prot1]
            seq2 = sequences[prot2]
            stats.append(compute_aa_stats(seq1, seq2))
    return #Compute statistics
```

Problem decomposition

```python
def main():
    """The whole (fake) program."""

    # Read the sequence files
    paths = []
    ans = input("path to FASTA file: ")
    while len(ans) > 0:
        paths.append(ans)
        ans = input("path to FASTA file: ")
    sequences = read_sequences(paths)

    # Read the interaction file
    ans = input("path to interaction data: ")
    interactions = read_interactions(ans)

    # Print the average stats
    print("average stats =", compute_avg_stats(sequences, interactions))

main()
```

Namespaces and scopes

Namespace

A namespace (sometimes also called a context) is a naming system for making names unique to avoid ambiguity.

- Naming people: firstname surname [birthday][birthplace]
- Naming websites: subdomain.domain.top-level-domain

Scope

The scope of a name is the area of a program where this name can be unambiguously used, for example inside of a function.

Namespaces and scopes

To associate a name, with a particular namespace, Python uses the location of the assignment of such name

In other words, the place where you assign a name in your source determines the namespace it will live in, and hence its scope of visibility.

Local variables

By default, all names assigned inside a function are associated with that function's namespace (local namespace)

```
def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

- Names assigned inside a `def` can only be seen by the code within that `def`.
- `x` is called a local variable

```
Inside 88
Traceback (most recent call last):
  File "lecture.py", line 6, in <module>
    print("Outside", x)
NameError:  name 'x' is not defined
```

Global variables

Names defined outside functions are associated with the global namespace.

```
# Var defined before the
# function and the call
x = 88
def func():
    print("Inside", x)

func()
print("Outside", x)
```

```
Inside 88
Outside 88
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` is called a global variable

Global variables

Names defined outside functions are associated with the global namespace.

```
def func():
    print("Inside", x)

# Var defined before the call
x = 88
func()
print("Outside", x)
```

```
Inside 88
Outside 88
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` is called a global variable

Global variables

```
def func():
    print("Inside", x)

func()
# Var defined after the call
x = 88
print("Outside", x)
```

```
Inside 88
Traceback (most recent call last):
  File "lecture.py", line 2, in func
    print("Inside", x)
NameError:  name ’x’ is not defined
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` was not defined before the call

Local and global variables

If a variable exists in both the local and global namespace, the copies are distinct.

```
x = 99

def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

```
Inside 88
Outside 99
```

- Inside the function, the local namespace for `x` is used.
- Outside the function, the global namespace for `x` is used.

Local variables

Local and global variables may coexist in the same function.

```python
x = 99
y = 100

def func():
    x = 88
    print("Inside", x, y)

func()
print("Outside", x, y)
```

```
Inside 88 100
Outside 99 100
```

- Inside the function, the local namespace for `x` is used.
- Outside the function, the global namespace for `x` is used.
- The global namespace for `y` is used

Local variables

```
x = 99
y = 100

def func():
    x = 88
    print("Inside", x, y)
    y = y+1

func()
print("Outside", x, y)
```

- If a variable is assigned inside a function, it becomes a local variable
- At the time of assignment inside the function, the local variable y is undefined.

```
Traceback (most recent call last):
  File "lecture.py", line 6, in func
    y = y+1
UnboundLocalError:  local variable 'y' referenced before
assignment
```

Local variables

Functions may be defined inside other functions. In this case, variables assigned in enclosing functions are nonlocal to nested functions.

```
def func():
    def func2():
        x = 100
        print("Inside", x, y)

    y = 100
    func2()

x = 99 ; y = 99
func()
print("Outside", x, y)
```

```
Inside 100 100
Outside 99 99
```

- Inside `func()`, `x` is local and `y` is non local
- Inside `func2()`, `y` is local
- Outside the functions, `x` e `y` are global.

Global variables

The `global` statement tells Python that a function plans to change one or more global names

```
x = 99

def func():
    global x
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

```
Inside 88
Outside 88
```

- Both `x` labels refer to the same variable in the global namespace

Namespaces

If the prior section sounds confusing, it really boils down to three simple rules. Within a `def`:

- Name assignments create local names by default.
- Name references search at most four scopes: local, then enclosing functions (if any), then global, then built-in (LEGB rule).
- Names declared in `global` and `nonlocal` statements map assigned names to enclosing module and function scopes, respectively.

Note

Each call to a function creates a new local scope. Every time you call a function, you create a new local scope – that is, a namespace in which the names created inside that function will usually live.

Rules of thumb

The rules that we have seen and many other that we are not going to see enable the inexperienced programmer to make a huge mess. The following are rule of thumbs that you should follow to avoid caos

- For the moment, avoid nesting functions and the `nonlocal` statement
- Always, try to minimize global variables and side effects

Argument-passing basic

Basic rules

- Arguments are passed by automatically assigning objects to local variable names
- Assigning to argument names inside a function does not affect the caller.
- Changing a mutable object argument in a function may impact the caller.

Argument-passing basic

- Immutable arguments are effectively passed "by value."
  Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in place anyhow, the effect is much like making a copy.
- Mutable arguments are effectively passed "by pointer".
  Objects such as lists and dictionaries are also passed by object reference, and the object can be effectively modified.

Passing immutable arguments

```
def func(a):
    a = 99

b = 88
func(b)
print(b)
```

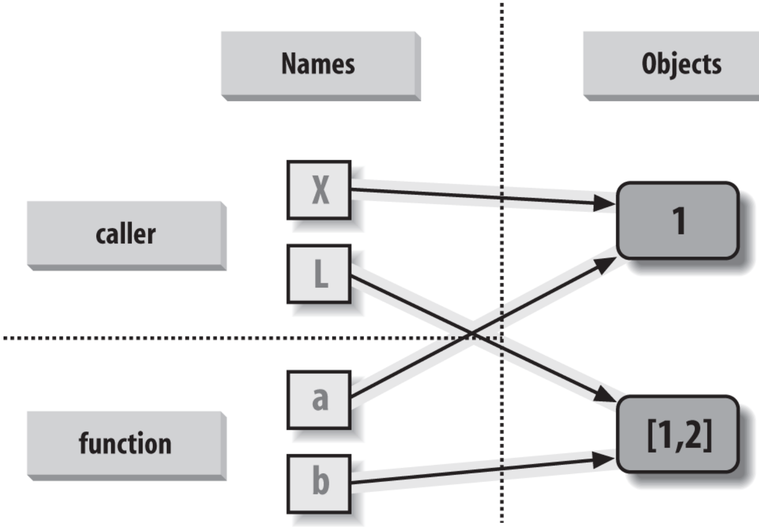
```
88
```

Passing mutable arguments

```
def func(a, b):
    a = 2
    b[0] = 99

X = 1
L = [1,2]
func(X,L)
print(X,L)
```

```
1 [99,2]
```

Passing mutable arguments



Avoid changes - use copies

Sometimes, the caller knows that the function is going to modify mutable objects, but she wants to avoid modifications. The mutable object must be copied before the call.

```
def func(a, b):
    a = 2
    b[0] = 99

X = 1
L = [1,2]
func(X,L[:])
print(X,L)                        1 [1,2]
```

Avoid changes - use copies

Sometimes, the functions needs to modify a mutable object (for example, to sort it), but this change should not be reported back to the caller. The mutable object must be copied in the function.

```
def func(a, b):
    a = 2
    b = b[:]
    b[0] = 99

X = 1
L = [1,2]
func(X,L)
print(X,L)                        1 [1,2]
```

Passing parameters by name

Knowing the name of the parameters, is it possible to pass the values by specifying `name=value`. This is useful in combination with defining defaults (see next slide).

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

f(1, 2, 3)                       1 2 3
f(c=1, b=2, a=3)                 3 2 1
f(1, c=3, b=2)                   1 2 3
print("C", end="++\n")           C++
```

Defining defaults

```
def f(a, b=2, c=3):
    print(a, b, c)
```

```
f(1)                        1 2 3
f(4,5)                      4 5 3
f(2,3,4)                    2 3 4
f(1, c=5)                   1 2 5
```

Exercise

Problem

Create a function `check_alphanumeric()` that takes a string and returns `True` if and only if the string is alphanumeric (contains only alphabetic or numeric characters).

```python
def check_alphanumeric(s):
    for c in s.upper():
        if c not in "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789":
            return False
    return True
```

Exercise

Problem

Write a function that given a string `s`, returns `True` if and only if `s` is palindromic.

```
def palindromic(s):
    L = list(s)
    L.reverse()
    return s == "".join(L)
```