Programming For Data Science

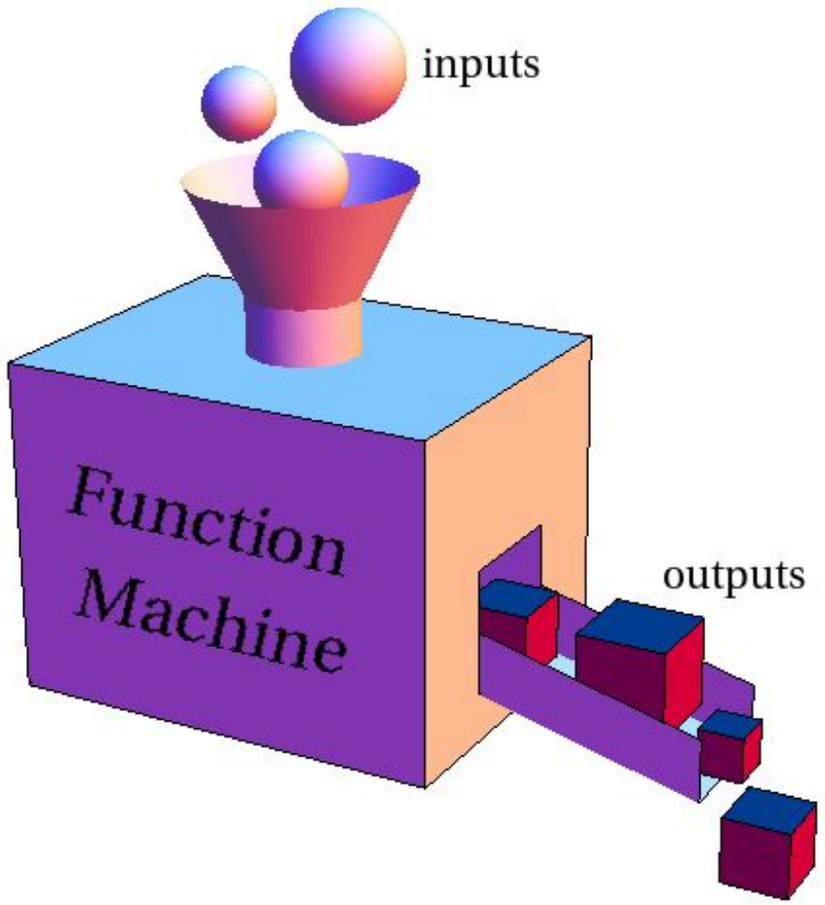
Part Four:

Make it Function!

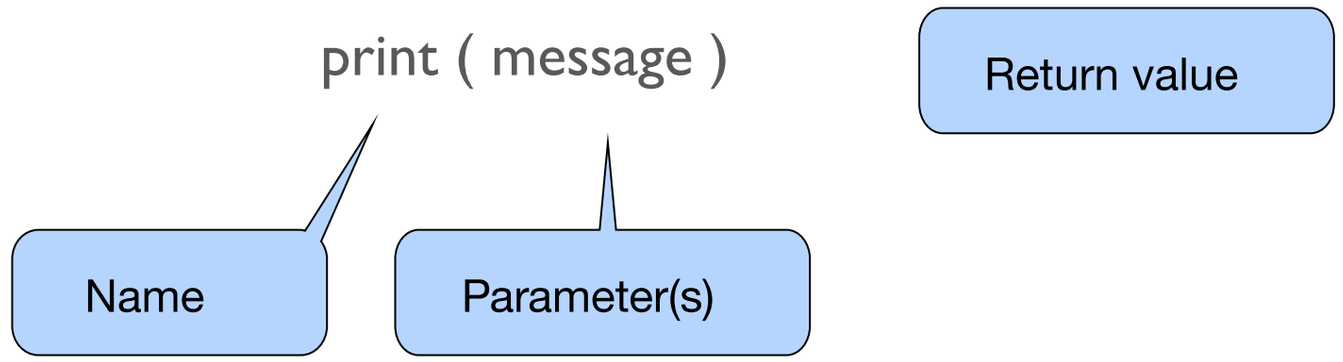
Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Simple functions



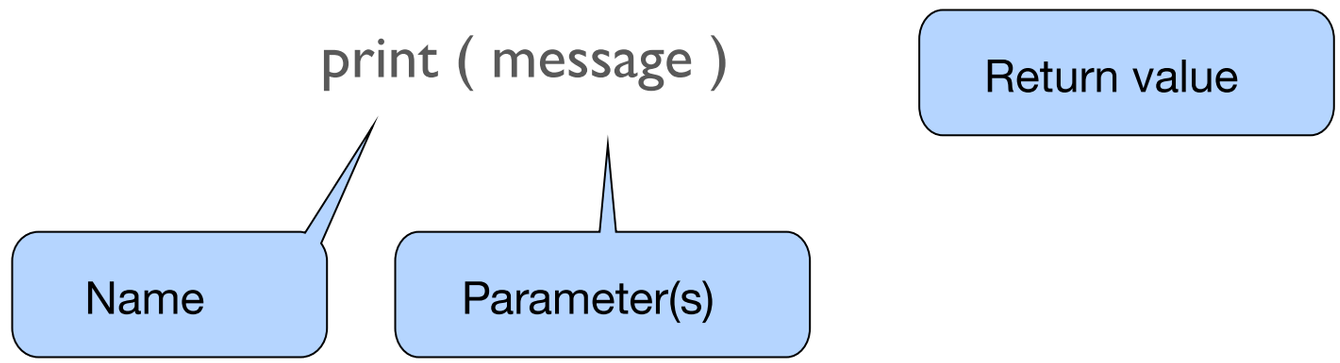
Simple functions



Similar to variable

- letters, digits, underscore, and cannot start with a digit
- typically, lower case

Simple functions



Parameters

Are not mandatory

Are the values supplied to the function to work with

int() has one parameter

print() has any number

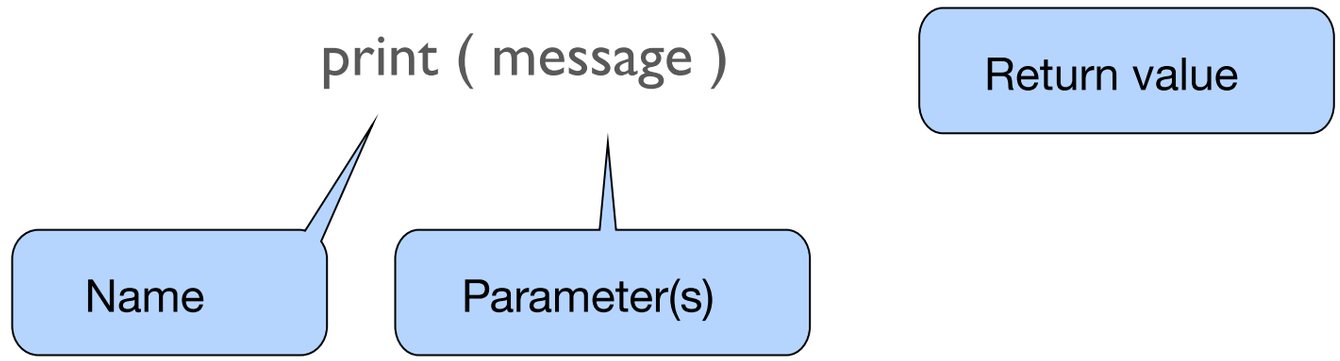
The order of parameters matters!

print(pow(base,exponent))

Also, you will get an **error** if you try to use a function with parameters it cannot work with

int(“pippo”)

Simple functions



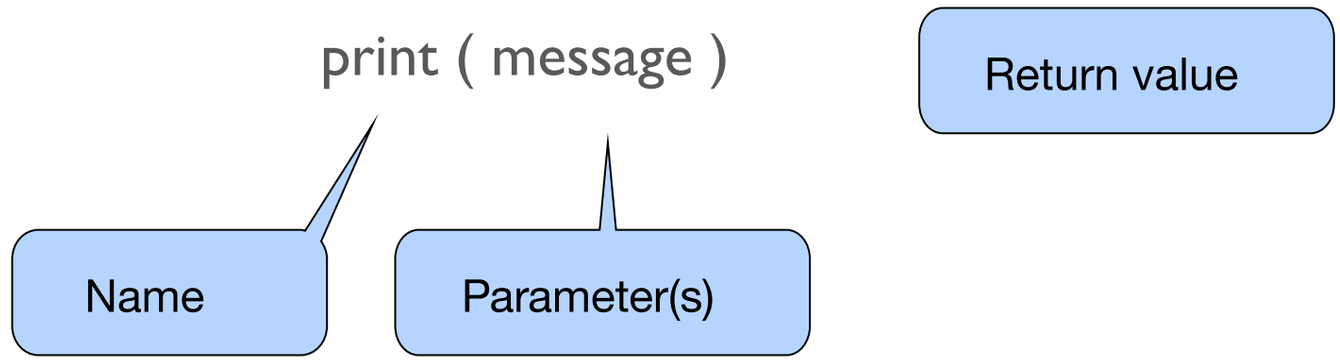
Parameters

print (message) is an example of **invocation** of a function

Somewhere, somebody **defined** what print() actually does

“Display all parameters, and then newline”

Simple functions

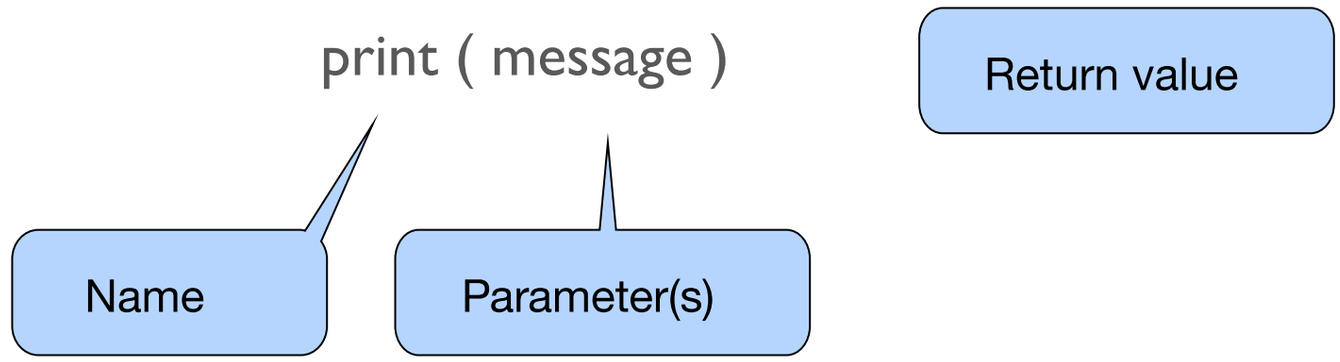


Parameters

In most programming languages, parameters can be passed either by reference or by value (or both)

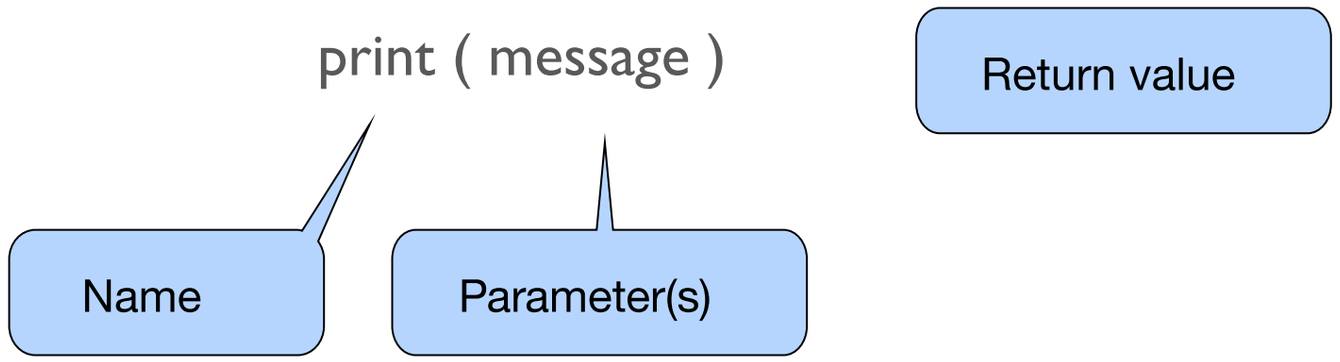
- **By reference**: the function has access to the variable that has been used during the invocation (hence, it might change it)
- **By value**: the function has access to a copy of the variable used during the invocation (hence, it cannot change it)

Simple functions



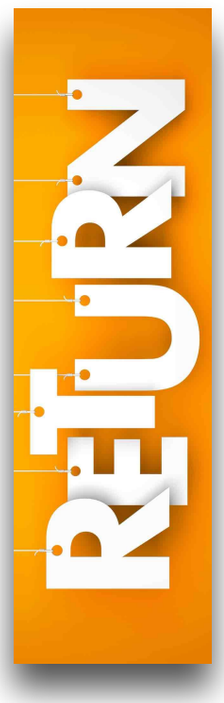
Parameters

Simple functions



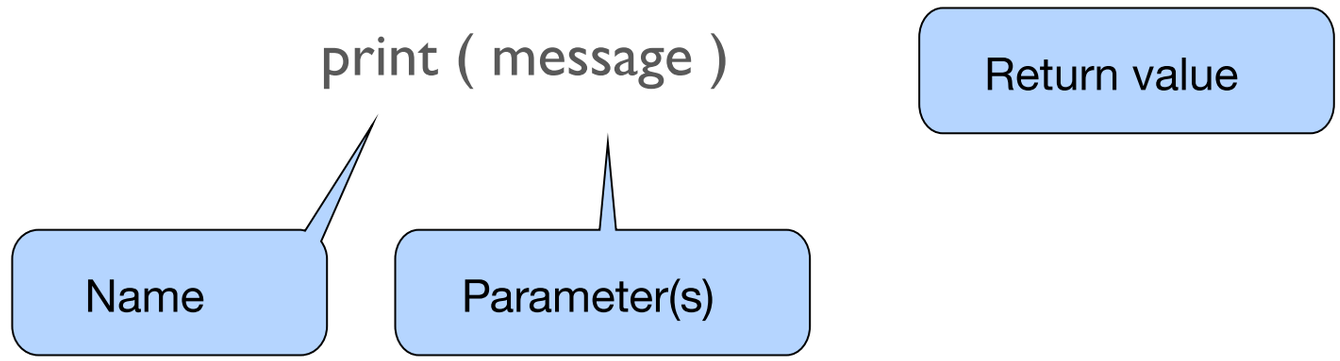
Not mandatory

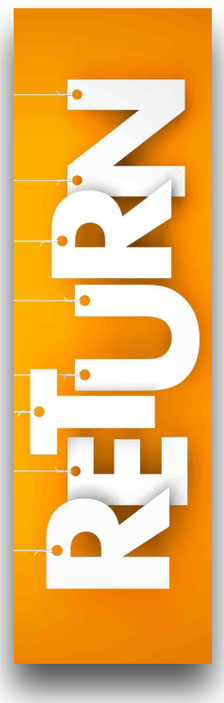
If a value is returned, it can be used in your code




```
x=2.1
y= '3'
z=int(x)
print(z)
print( int(y) )
```

Simple functions

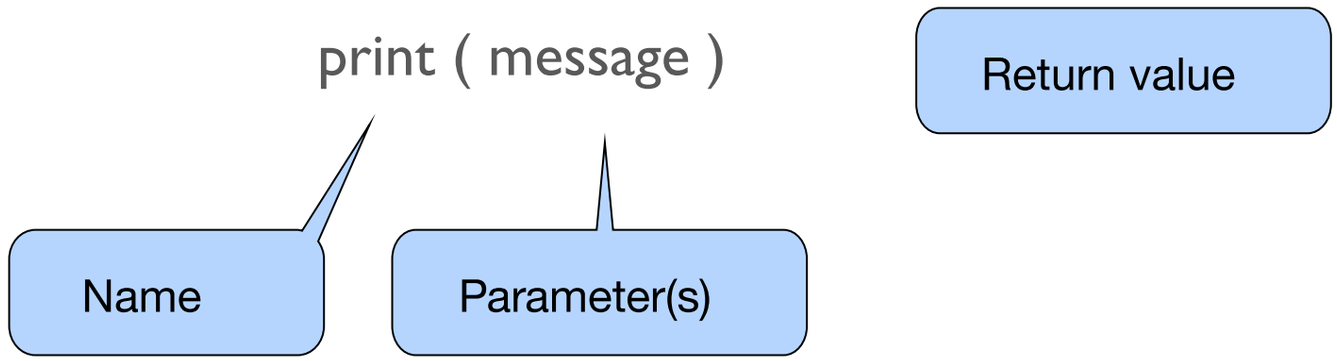


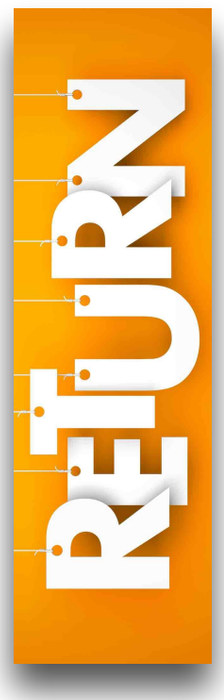



```
print( print( “Hello world” ) )
```

What does happen here?

Simple functions

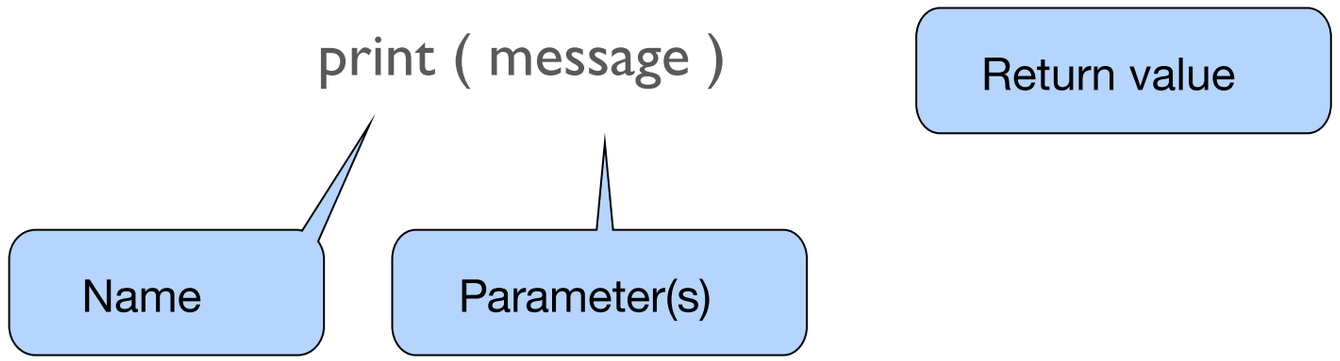


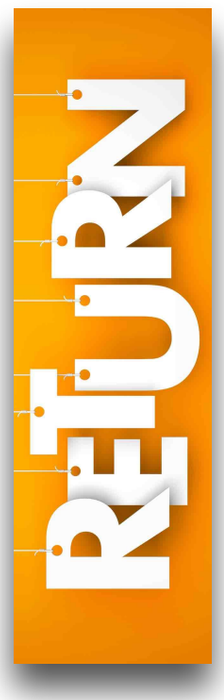


print(print(“Hello world”))

The invocation to the most extern print() evaluates its parameter

Simple functions

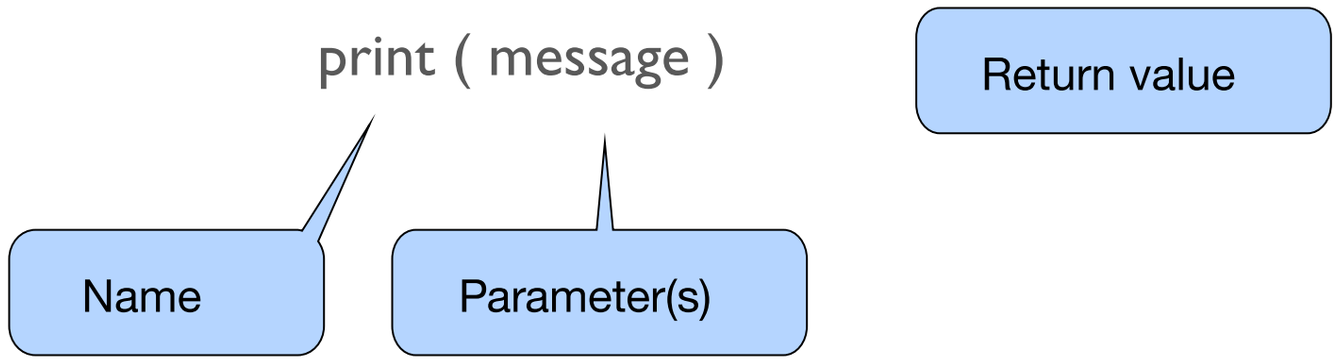


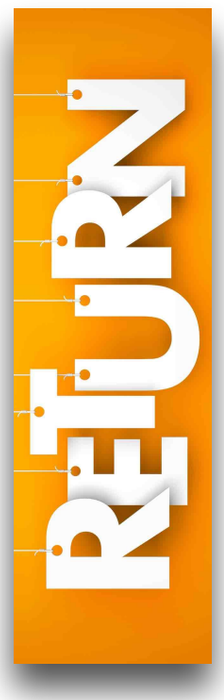


print(print(“Hello world”))

Its parameter is another invocation of the print() function

Simple functions





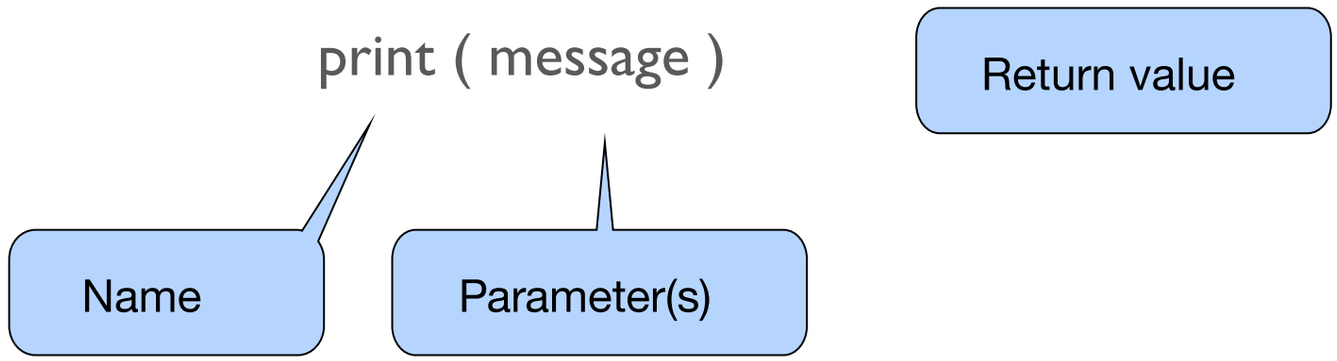
print(print(“Hello world”))

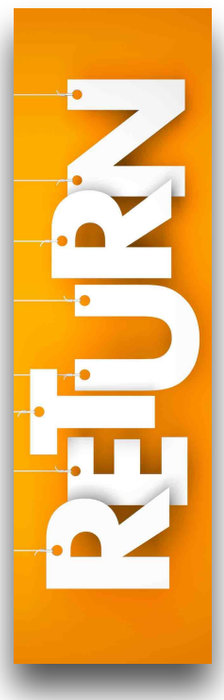
The second print() evaluates its parameter, that is a string, and displays it

Output:

Hello world

Simple functions





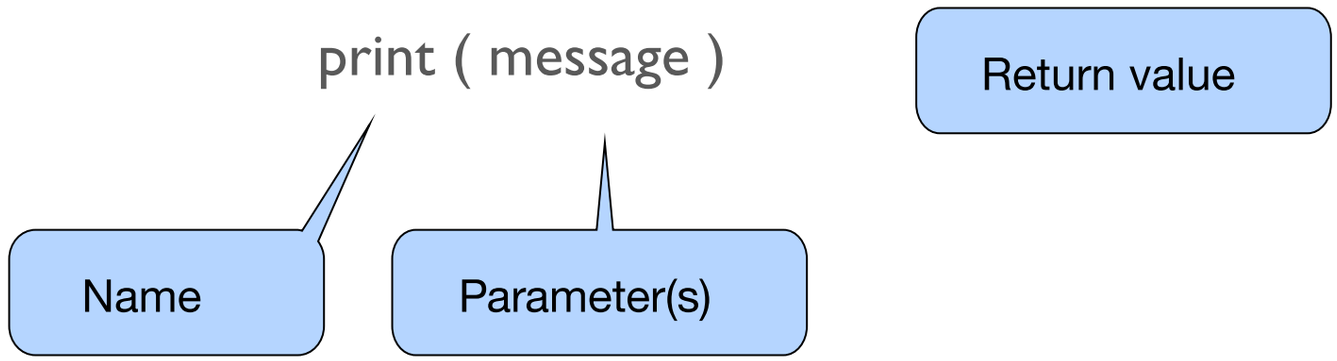
print(print(“Hello world”))

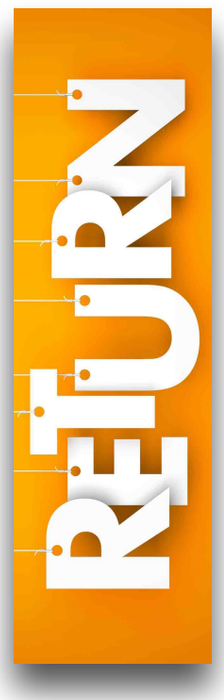
The second print() finishes its work, and returns, but is has **no return value**

Output:

Hello world

Simple functions





print(print(“Hello world”))

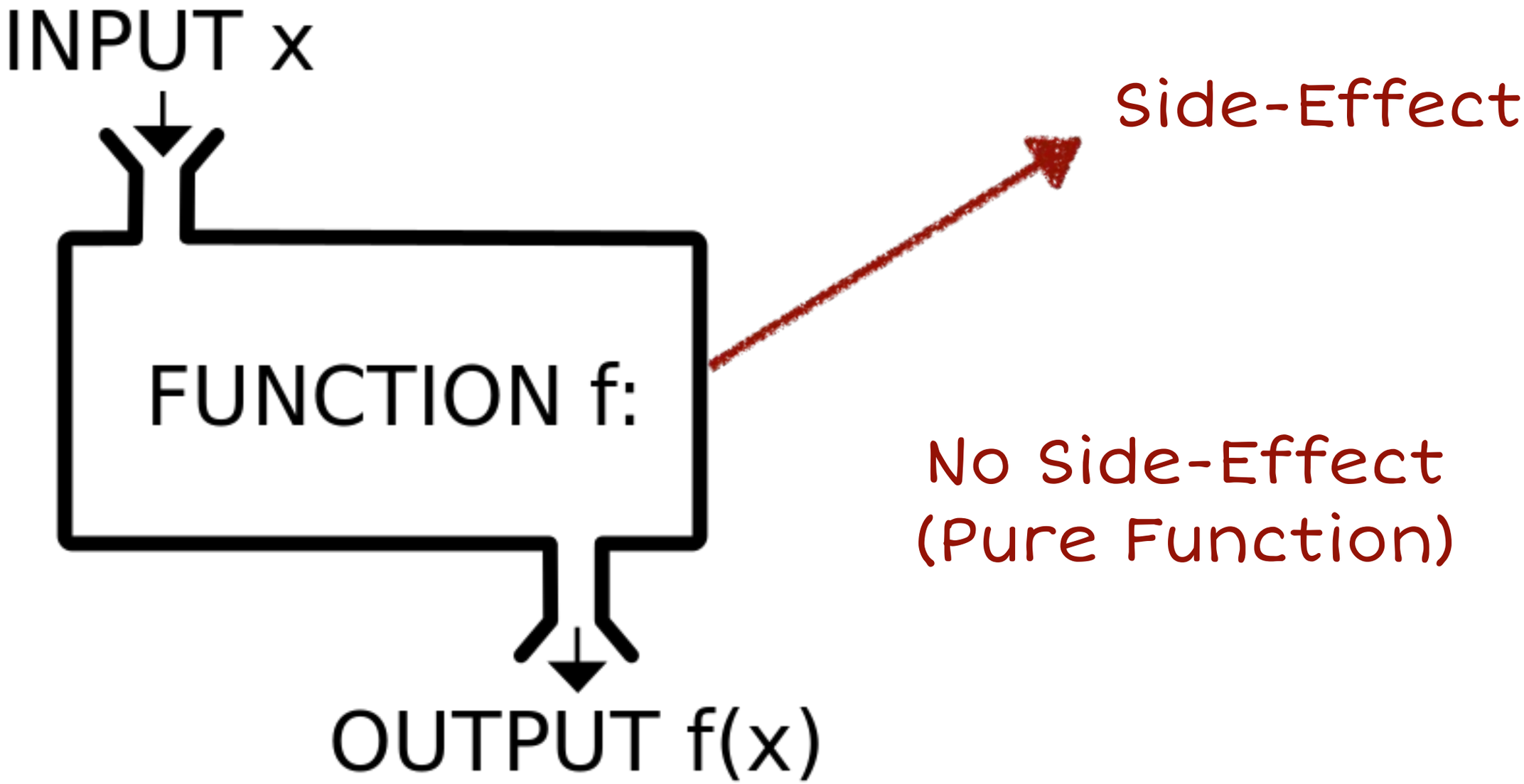
Hence the first print() has **nothing real** to display, and displays None.

Special value, meaning no value!

Output:

Hello world
None.

Function is a **black box**



Basic functions

- Casting
  - float(n)
  - int(n)
  - str(“n”)
- Calculations
  - abs(n)
  - max(m, n), min(m, n)
  - pow(x, y)
  - round(n), round(n, m)

round n to a whole number

round n to the number of decimals specified by m

round (1.58, 1) —> 1.6

Basic functions

- Casting
  - float(n)
  - int(n)
  - str("n")
- Calculations
  - abs(n)
  - max(m, n), min(m, n)
  - pow(x, y)
  - round(n), round(n, m)
    - print (7/11) *with only 3 decimals????*

Basic functions

- Casting
  - float(n)
  - int(n)
  - str("n")
- Calculations
  - abs(n)
  - max(m, n), min(m, n)
  - pow(x, y)
  - round(n), round(n, m)
    - print (7/11) — print (round(7/11, 3))

Basic functions

- len(n)
  - *returns the length of the parameter*
  - for now, n is a string
  - len(“pippo”) —> 5
  - len(“”) —> ????
  - len(‘can\’t’) —> ????
- input()
  - *asks the user for data*

Read input values

input() shows the input string (aka *prompt*) to the user

```
print (“Hi”)
name = input(“Your name? “)
print (“Hi”, name)
```

```
Hi
Your name? Ned Stark
Hi Ned Stark
```

Read input values

input() always returns a **string**

```
text = input(“enter a number: “)
print (“Square is”, text * text)
```

Read input values

input() always returns a **string**

```
text = input(“enter a number: “)
print (“Square is”, float(text) * float(text))
```

Print values

- print(n)
  - displays the parameters and **goes to the next line**
- Two special parameters: sep and end
  - sep indicates what should be printed between each of the parameters, and by default *is a space*
  - end indicates what print() should put after all the parameters have been displayed, and *by default is a newline*
  - print("A", "B", sep="x") —> AxB
  - print("A", "B", end="!") —> A B!

Format values

- format()
  - To create *formatted* strings
    - print(7/11) [*I want to display only 3 decimals*]
      - ???? (you know already how to do it!)

Format values

- format()
  - To create *formatted* strings
    - print(7/11) [*I want to display only 3 decimals*]
      - print(round(7/11, 3))

Format values

- format()
  - To create *formatted* strings
    - print(7/11) [*I want to display only 3 decimals*]
      - print(round(7/11, 3))
    - If there are more requirements in the formatting, it might be more difficult
      - ….also reserve 10 positions for it, and left align the result

Format values

- format()
  - Is a function that **works on strings**
  - It is called like this: string.format()
  - It **returns a new string** (the formatted version of the input string)
  - It can take any number of parameters, whose values are placed in the resulting string in places indicated by {}

Format values

- format()
  - Is a function that **works on strings**
  - It is called like this: string.format()
  - It **returns a new string** (the formatted version of the input string)
  - It can take any number of parameters, whose values are placed in the resulting string in places indicated by {}

```
print("Three numbers: {}, {}, and {}".format("one", "two", "three"))
```

```
Three numbers: one, two, three
```

Format values

- format()
  - Is a function that **works on strings**
  - It is called like this: string.format()
  - It **returns a new string** (the formatted version of the input string)
  - It can take any number of parameters, whose values are placed in the resulting string in places indicated by {}

```
print("Three numbers: {}, {}, and {}".format(1, 2, "three"))
```

```
Three numbers: 1, 2, three
```

Format values

- format()
  - Is a function that **works on strings**
  - It is called like this: string.format()
  - It **returns a new string** (the formatted version of the input string)
  - It can take any number of parameters, whose values are placed in the resulting string in places indicated by {}

```
print("Three numbers: {0}, {1}, and {2}".format(1, 2, 3))
```

```
Three numbers: 1, 2, 3
```

Format values

- format()
  - Is a function that **works on strings**
  - It is called like this: string.format()
  - It **returns a new string** (the formatted version of the input string)
  - It can take any number of parameters, whose values are placed in the resulting string in places indicated by {}

```
print("Three numbers: {2}, {1}, and {0}".format(1, 2, 3))
```

```
Three numbers: 3, 2, 1
```

Format values

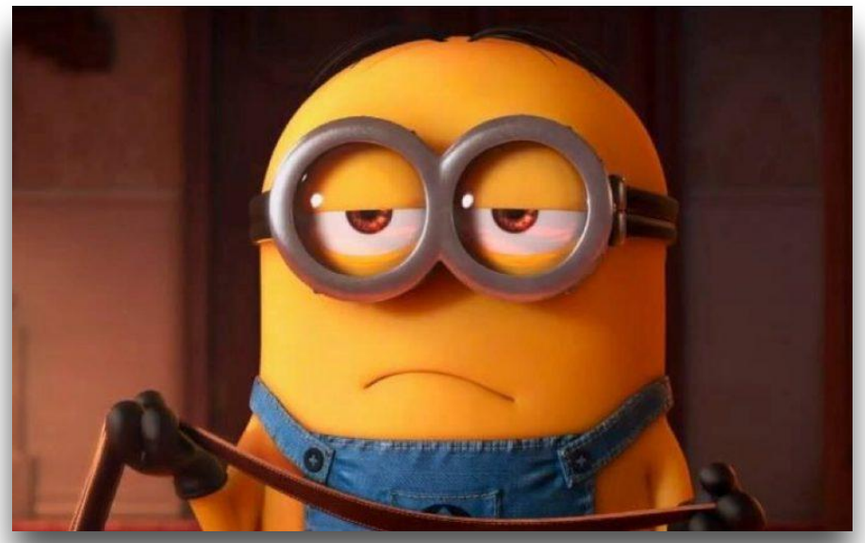
- format()
  - It can deal with any type of parameters, as long as there is a *suitable string representation*

```
print("Three numbers: {2}, {1}, and {0}".format(1, 2.0, "three"))
```

```
Three numbers: three, 2.0, 1
```

Format values

- format()
  - You can add **:** after the number in the {} (if any) for extra formatting instructions
    - Refer to the book for all details on precision, alignment, and decimals



Modules

- Contain useful functions
  - import <modulename>
  - from <modulename> import <function1>, <function2>, ….

```
import math
print(math.sqrt(4))
```

```
from math import sqrt
print(sqrt(4))
```

```
from math import sqrt as square
print(square(4))
```

Rename

Modules: math

The math module

- exp()
- log()
- log10()
- sqrt()

Modules: random

The random module: generating *pseudo-random* numbers

- **random()** —> random float in [0,1)
- **randint(x, y)** —> random integer in [x,y]
- **seed()** —> initializes the random generator
  - with or without parameter
    - seed(x), with x integer, will always generate the same sequence

(Suggested) Exercises

1. Write a program that takes as input the length of the sides of a rectangle and computes its perimeter and area
2. Write some code that asks the user for two numbers, then shows the result when you add them, and when you multiply them
3. Write ten times the string “I am writing ten times this string!”
4. All exercises at the end of Chapter 5 of the book

Programming For Data Science

Part Five:

What If...

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

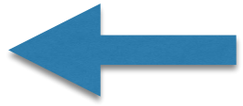
Conditions

A conditional statement consists of a **test** and one or more **actions**

- **Test:** it is a *boolean expression*
  - Evaluates to either **True** or **False**
    - The (only) Boolean values, predefined in Python

Boolean values

- True and False are of type **bool**
- In Python, *every value can be interpreted as a boolean value*
- **False:**
  - False
  - None



Remember this????

Boolean values

- True and False are of type **bool**
- In Python, *every value can be interpreted as a boolean value*
- **False:**
  - False
  - None
  - Every numerical value that is zero
  - “”
  - Every empty mapping (dictionaries)
  - Any function that returns one of the above

Boolean values

- True and False are of type **bool**
- In Python, *every value can be interpreted as a boolean value*
- **False:**
  - False
  - None
  - every numerical value that is zero
  - “”
  - Every empty mapping (dictionaries)
  - Any function that returns one of the above

And what is True?

Comparing booleans

Operator	Description
>	greater than
<	less than
==	equal to
<=	less or equal to
>=	greater or equal to
!=	lnot equal to

what is the outcome of 3 < I3?

Comparing booleans

Operator	Description
>	greater than
<	less than
==	equal to
<=	less or equal to
>=	greater or equal to
!=	lnot equal to

what is the outcome of “3” < “13”?

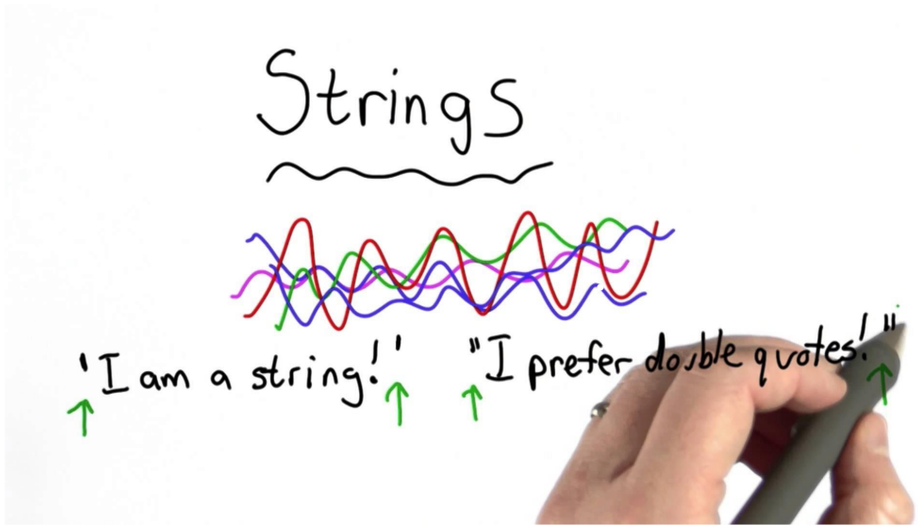
Comparing booleans

Operator	Description
>	greater than
<	less than
==	equal to
<=	less or equal to
>=	greater or equal to
!=	lnot equal to

what is the outcome of 3 < “13”?

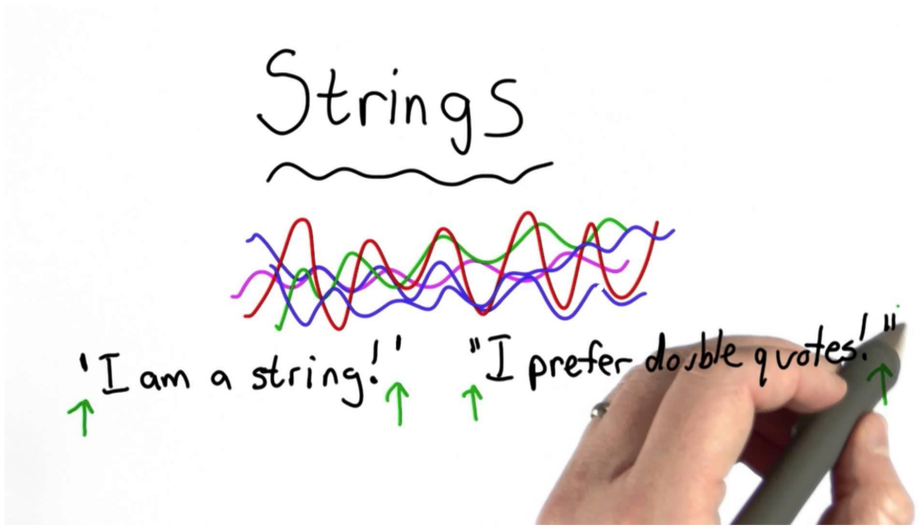
Membership

The **in** operator tests if the value to the left side of the operator is found in a "collection" to the right side of the operator



Membership

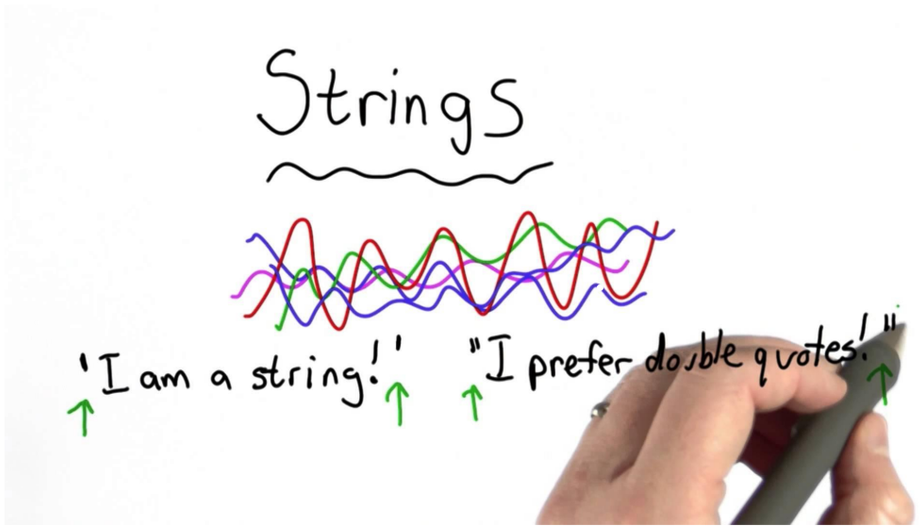
- The **in** operator tests if the value to the left side of the operator is found in a “collection” to the right side of the operator



- Strings are *collections* of characters
  - With the **in** operator you can check whether a *char is part of a string*
  - print(“y” in “Python”)

Membership

- The **not in** operator tests if the value to the left side of the operator is **not found** in a “collection” to the right side of the operator
  - print(“z” not in “Python”)



Logical operators

NOT	
A	**NOT A**
T	?
F	?

Logical operators

NOT	
A	**NOT A**
T	F
F	T

Logical operators

AND		
A	B	A AND B
T	T	?
T	F	?
F	T	?
F	F	?

NOT	
A	NOT A
T	F
F	T

Logical operators

AND		
A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

NOT	
A	NOT A
T	F
F	T

Logical operators

AND

A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

NOT

A	NOT A
T	F
F	T

OR

A	B	A OR B
T	T	?
T	F	?
F	T	?
F	F	?

Logical operators

AND		
A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

NOT	
A	NOT A
T	F
F	T

OR		
A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

Boolean expressions
evaluated
from left to right

```
x=1; y=0
print((x==0)or(y==0)or(x/y ==1))
```

What if….

A conditional statement consists of a **test** and one or more **actions**

```
if condition:
        # action


if x==5:
        print(“x is 5”)
```

If condition is TRUE, action is executed

Code Block

Note the indentation **(1 tab to the right)** of the action: it is necessary!

What if….

A conditional statement consists of a **test** and one or more **actions**

```
if condition:
    # action
    # action2

if x==5:
    print(“x is 5”)
```

Code Block

The code block is valid until the level of indentation is the same

Indentation: few rules

- Do not mix tabs and spaces
- 4 spaces is usually the standard
- Editors generally indent for you
- *It is not important the absolute indentation, but the relative one within the block*

```
x=4
if x==5:
        print("1")
      print("2")
```

Indentation: few rules

- Do not mix tabs and spaces
- 4 spaces is usually the standard
- Editors generally indent for you
- *It is not important the absolute indentation, but the relative one within the block*

```
x=4
if x==5:
    print("1")
    print("2")
```

Alternative decisions

You might want to distinguish between TRUE and FALSE condition

```
if condition:
    # action1
else:
    # action2
```

If condition is TRUE, action1 is executed, else action2

```
n = input("n: ")
if n % 2 == 0:
    print (“yes”)
else:
    print(“no”)
```

is n even?

Again on indentation

- *It is not important the absolute indentation, but the relative one within the block*

```
x=4
if x==5:
        print("1")
        print(“2")
else:
  print(“3”)
```

Multi-branch

You can choose one among several options

```
if condition1:
    # block1
elif condition2:
    # block2
else:
    # block3
```

If condition1 is TRUE, block1 is executed, else if condition2 is TRUE, block2 is executed. Else, block2 is executed

Several elif, only one else at the end

```
if x < y:
    print (“x is smaller”)
elif x > y:
    print (“y is smaller”)
else:
    print (“x and y are equal”)
```

Conditions can be complex!

Conditions can be combined using logical operators

condition1 and condition2 — TRUE iif both TRUE

condition1 or condition2 — TRUE iif at least one TRUE

not (condition1 and condition2) — TRUE iif at least one FALSE

not (condition1 or condition2) — TRUE iif both TRUE

not (condition1) and condition2 — TRUE iif c1 F and c2 T

```
x = 10; y = 5; z = 1
print (x < y and y < z)
print (not (x < y) and (z < y or z < x))
```

Early exits

- Occasionally it happens that you want to exit a program early when a certain condition arises

```
from sys import exit

….
if y == 0:
    exit
else:
   print (x/y)
```

Early exits

- Occasionally it happens that you want to exit a program early when a certain condition arises

```
from sys import exit


….

if y == 0:
    exit
else:
   print (x/y)
```

- When exit() is executed, the program terminates
  - Python raises a SystemExit exception
    - It depends on the editor….this is ok!

Exercises

- All exercises in Chapter 6 of the reference book

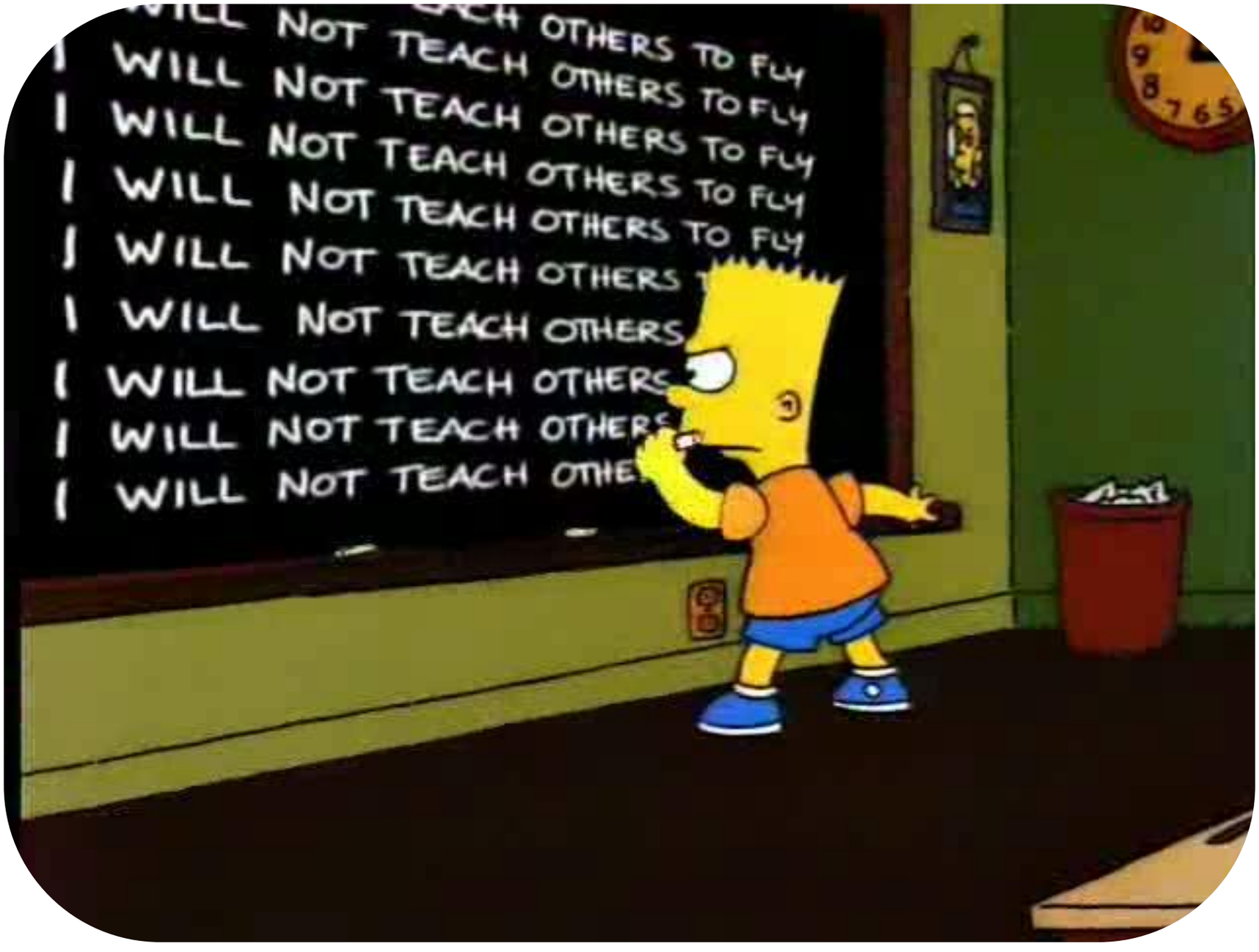
Programming For Data Science

Part Six:

Let’s Iterate

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it


I WILL NOT TEACH OTHERS TO FLY
I WILL NOT TEACH OTHERS TO FLY
I WILL NOT TEACH OTHERS TO FLY
I WILL NOT TEACH OTHERS TO FLY
I WILL NOT TEACH OTHERS
I WILL NOT TEACH OTHERS
I WILL NOT TEACH OTHERS
I WILL NOT TEACH OTHE

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
i = 0
while i < 100:
    print("I will not teach others to fly")
    i+=1
```

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
m = input("how many times? ")
i = 0
while i < int(m):
    print("I will not teach others to fly")
    i += 1
print("Done!")
```

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
m = input("??")
i = 0
while i < int(m):
    print i*i
    i += 1
```

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
password = “SheldonCooper”
input = input(“????”)
while input != password:
    input = input(“??? ”)
print(“wow”)
```

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
        Watch the indentation!
```

Repeat block until condition is TRUE

```
from pcinput import getInteger


total = 0; count = 0

while count < 5

    total += getInteger("Give a number")

    count += 1
```

Note the counters' values (programming style)

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
number = 1
total = 0
while (number*number)%100 != 0
    total += number
print( "Total is",  total)
```

What happens here?

Loops: while

Why repeating yourself when the computer can do that for you?

```
while condition:
    # block to repeat
```

Repeat block until condition is TRUE

Watch the indentation!

```
number = 1
total = 0
while (number*number)%100 != 0
    total += number
print( "Total is",  total)
```

Always check for endless loops!

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
for letter in “Python”:
    print(‘letter:', letter)
print(“Done”)
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
string = “Python”
for letter in string:
    print(‘letter:', letter)
print(“Done”)
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
string = “Python”
for pippo in string:
    print(‘letter:', letter)
    if (letter == ’t’):
        string = “Java”
print(“Done”)
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

range(3) —> 0, 1, 2

```
for x in range(3):
   print (x)

for x in range(1, 11, 2):
   print (x)
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
for fruit in (‘banana’, ‘apple’, ‘mango’):
   print 'Current fruit :', fruit
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
fruits = ['banana', 'apple',  ‘mango']

for fruit in fruits:
   print 'Current fruit :', fruit
```

Loops: for

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]
```

Controlling loops

```
i = 0
while i < 5:
    print(i)
    i += 1
else:
    print( "The loop ends, i is now", i)
print( "Done" )
```

Controlling loops

```
for fruit in ( "apple", "orange", "strawberry" ):
    print(fruit)
else:
    print( "The loop ends, fruit is now", fruit)
print( "Done" )
```

Controlling loops

- break
  - When Python encounters the break statement, it will no longer process the remainder of the code block for the loop

```
for grade in ( 8, 7.5, 9, 6, 6, 6, 5.5, 7, 5, 8, 7, 7.5 ):
    if grade < 5.5:
        print( "The student fails!" )
        break
    else:
        print( "The student passes!" )
```

Controlling loops

- break
  - It is usable only within loops
  - If executed, the else statement is not

Check this out!

```
i =1
while i <= 1000000:
    num1 = int("1"+str(i))
    num2 = int(str(i)+"1")
    if num2 == 3 * num1:
        print(num2 , "is three times",  num1)
        break
    i += 1
else:
    print( "No answer found" )
```

428571 is three times 142857

Controlling loops

- continue
  - When the continue statement is encountered in the code block of a loop, the current cycle ends immediately and the code loops back to the start of the loop

```
for letter in 'Python':
    if letter == ‘h':
        continue
    print (‘Current Letter :', letter)
```

P

y

t

o

n

Controlling loops

- continue
  - When the continue statement is encountered in the code block of a loop, the current cycle ends immediately and the code loops back to the start of the loop

```
var = 10
while var > 0:
    var = var -1
    if var == 5:
        continue
    print ('Current variable value :', var)
print ("Good bye!")
```

```
9
8
7
6
4
3
2
1
0
```

Controlling loops

- continue
  - When the continue statement is encountered in the code block of a loop, the current cycle ends immediately and the code loops back to the start of the loop

```
var = 10
while var > 0:
    if var == 5:
        continue
    var = var -1
    print ('Current variable value :', var)
print ("Good bye!")
```

What happens here?

Nested loops

Alternative way for implementing loops

```
for variable in collection:
    # block that can use variable
```

Iterate on all elements in the collection

```
for x in range(1, 11):
    for y in range(1, 11):
        print (x, y, x*y)
```

What is displayed?

Loop-and-a-half

- It is a particular way of designing a loop
- It is useful when you have several conditions that control a while loop
  - Used when it is not known how many loops there will be

```
while (cond1) and (cond2) and (cond3):
    do something
    if (cond):
        do something else
```

loop-and-a-half

```
while true:
    if (!cond1):
        break
    if (!cond2):
        break
    if (!cond3):
        break
    do something
    if (cond):
        do something else
```

Loop-and-a-half

- It is a particular way of designing a loop
- It is useful when you have several conditions that control a while loop
  - Used when it is not known how many loops there will be

```
while true:                                          loop-and-a-half
    do something
    if (sentinel condition):
        break  #(or continue)
    do something else
```

Loop-and-a-half

- It is a particular way of designing a loop
- It is useful when you have several conditions that control a while loop
  - Used when it is not known how many loops there will be

```
while true:
    x = getInteger()
    if (x < 0):
       break
    if (x == 0):
        continue
      y = getInteger:
    if (y == 0):
        continue
    print (x*y)
```

(Suggested) Exercises (1)

- Given two numbers A and B (from input), compute the the result of A/B and A%B not using the operators / and %
- Given numbers A, B and C (from input), compute the number of days in A years, B months, and C weeks
- Given a number X (from input), compute the number of years, months and weeks in X days
- Given a sequence of n integers (n from input), display the sum and the mean of the even numbers in the sequence
- Given a sequence n integers (n from input), display the sum and the mean of the numbers in the sequence that are even, multiple of 3 and larger than the previous number

(Suggested) Exercises (2)

All exercises in Chapter 7 of the reference book

- In particular:
  - 7.3
  - 7.5
  - 7.6
  - 7.8
  - 7.10
  - 7.11
  - 7.13