TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

— Part 1 —

(2 CFU out of 6 total CFU)

Link moodle: https://moodle2.units.it/course/view.php?id=11703

Teams code: 0ftoqj8

Python

What's Python?

It's a **high-level** programming language closer to human thinking
than to details of the machine behaviour

It's an **interpreted** language you need a compiler/interpreter to translate
this kind of programming language into a machine code

Python

What's Python?

It's a **high-level** programming language — closer to human thinking than to details of the machine behaviour

It's an **interpreted** language — you need a compiler/interpreter to translate this kind of programming language into a machine code

Why Python?

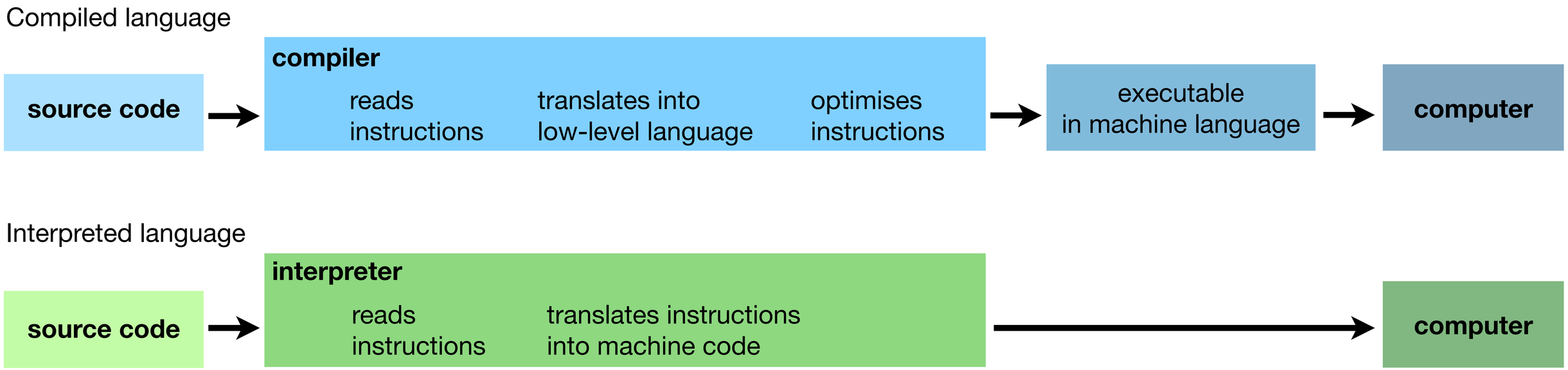
- Human-readable and close to human thinking
- Open source
- Developed by a community effort
- Contribution from users encouraged

Python

What's Python?

It's a **high-level** programming language — closer to human thinking than to details of the machine behaviour

It's an **interpreted** language — you need a compiler/interpreter to translate this kind of programming language into a machine code



Python

Language

Code

Natural language

Formal language

Structure

Syntax Set of rules which determines how a program is written and interpreted

Programming language

Python

Programming language

Quite strict

Instructions (statements) are interpreted (parsed).
To be understood they must be formally correct and only use the expected language constituents (token).

Formal language

Unique meaning independent on the context.

Syntax

Semantics Meaning of an instruction whose syntax is correct

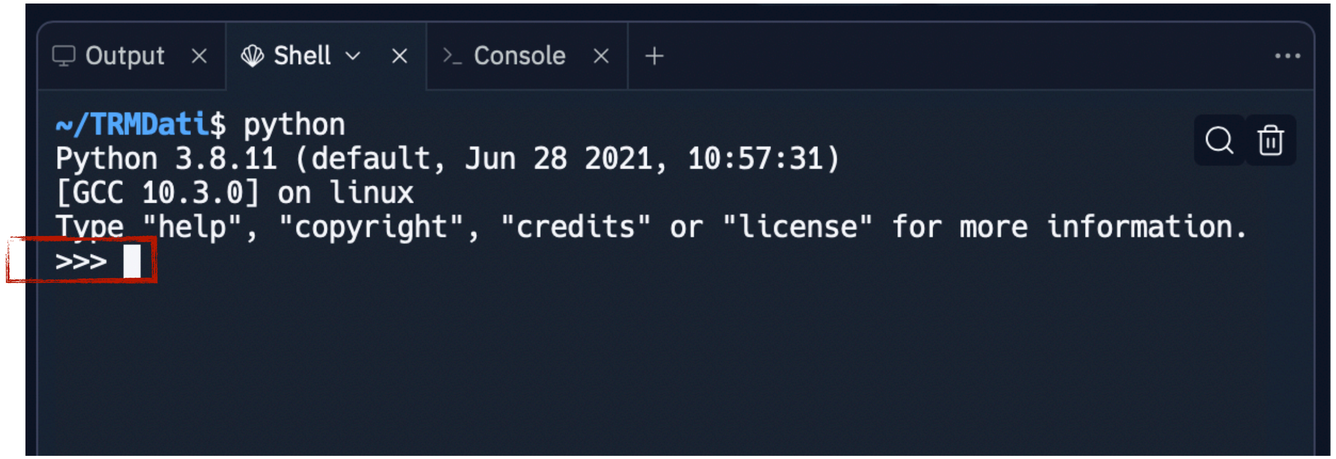
Python

Programming language

Can be used in two ways:

- interactively: the interpreter is given instructions directly, one by one
- withs scripts: the interpreter is provided with a set of instructions in a text file

Different versions, use Python > 3.7



On replit shell, type
python
to launch the interpreter

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

```
>>> 1 + 5&
  File "<stdin>", line 1
    1 + 5&
         ^
SyntaxError: invalid syntax
>>>
```

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

Python returns a runtime error if something goes wrong while executing an instruction

```
>>> 2 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

Python returns a runtime error if something goes wrong while executing an instruction

If semantic errors are there, Python does not return what you expect
(likely without issues during runtime)

Python scripts

Program/script: set of instructions in a given order that tells the interpreter how to compute or perform something

Types of instructions:

- Input
- Computation
- Condition check
- Iterate/repeat
- Output

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

```
>>> a = 1 + 2
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

```
>>> a = 1 + 2 \
... + 3 + 4 \
... + 5
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

Multi-line statements are implicitly assumed with parentheses.

```
>>> a = ( 1 + 2
... + 3 )
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

Multi-line statements are implicitly assumed with parentheses.

Multiple statements can stay on the same line, divided by the character ;

```
>>> a = 1 ; b = 2
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token **=**

Besides assignments, there are other statements, e.g., **import, while, if, for**

import allows you to import in your script instructions written in another file

```
>>> import this
```

Python comments

Comments: they describe in simple words what the source code is doing

Start with the hash character # and end with enter/new line

```
>>> # Add 2 to 1
>>> 1 + 2
3
>>>
```

Python interpreter neglects comments while executing
the set of instructions the script is made of

For multi-line comments, either start every line with # ,
or type the comment within triple quotes (“‘ comment ’” , “““ here ”””)

Python keywords

Keywords: ensemble of reserved words that cannot be used as variable names, function names, or any other identifiers instructions

Case sensitive: apart from False, None, True, all the others do not have capital letters

```
>>> # Can I assign a value to a keyword?
>>> False = 3
  File "<stdin>", line 1
    False = 3
    ^^^^^
SyntaxError: cannot assign to False
```

To check Python keywords:

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif'
, 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or'
, 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Values have different types and can be grouped into classes.

The built-in Python function *type* returns the type of a value.

```
>>> type(1)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type('hello')
<class 'str'>
>>>
```

integer number

float number

string of character

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Different types can do different things.

Python has the following built-in **data types**:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Python has the following built-in **data types**:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

covered in this course

Python variables

How to access the content of a string and slice it:

```
In [10]: string = 'Information'

In [11]: type(string)
Out[11]: str

In [12]: print(string[3])
o

In [13]: print(string[-1])
n

In [14]: print(string[0:4])
Info

In [15]: print(string[3:6])
orm

In [16]: print(string[2:-2])
formati
```

i-th element of a string

from the i-th to the j-th character of a string [i, j)

Lists behave similarly.

Python variables

Variables: nouns assigned to values stored in memory

Programs perform computations with variables to obtain results.

The token **=** links a value to a variable,
via an assignment statement.

It links the *lvalue* (variable on the left)
with its *rvalue* (value on the right)

```
>>> year = 2023
>>> month = 'October'
>>>
```

The Python interpreter evaluates variables
and returns their values:

```
>>> year, month
(2023, 'October')
>>> year
2023
```

The token = to assign is something different from == to verify value equality.

Python expressions

Expression: combination of values, variables, operators and calls to functions. The Python interpreter evaluates the written expression and returns the result.

Python operators

Operators: special tokens used to perform different operations

Python can perform operations only between variables / values of the same type.

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

If types are compatible (e.g., integers are a sub-set of floats), Python automatically cast (~convert) to the higher-level type (~upgrade).

Otherwise: Error!

Python operators

Operators: special tokens used to perform different operations

Python can perform operations only between variables / values of the same type.

```
In [1]: a = 1

In [2]: type(a)
Out[2]: int

In [3]: b = 1.0

In [4]: type(b)
Out[4]: float

In [5]: type(a) == type(b)
Out[5]: False

In [6]: a == b
Out[6]: True
```

Use the Python built-in function type( ) to check whether variables / values have the same type.

Do not just compare variable values!

Python operators

Operators: special tokens used to perform different operations

Python can perform operations only between variables / values of the same type.

```
In [7]: c = 4 + 3.5

In [8]: print(c);  type(c)
7.5
Out[8]: float
```

If types are compatible (e.g., integers are a sub-set of floats), Python automatically cast (~convert) to the higher-level type (~upgrade).

Python operators

Operators: special tokens used to perform different operations

Python can perform operations only between variables / values of the same type.

```
In [7]: c = 4 + 3.5

In [8]: print(c);  type(c)
7.5
Out[8]: float

In [9]: d = 4 + 'hello'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 d = 4 + 'hello'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

If types are compatible (e.g., integers are a sub-set of floats), Python automatically cast (~convert) to the higher-level type (~upgrade).

Otherwise: Error!

Python operators

Arithmetic operators: used with numeric values to perform common mathematical operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python operators

Comparison operators: used to compare two values

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python operators

Logical operators: used to combine conditional statements

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python operators

Membership operators: used to test if a sequence is presented in an object

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python operators

Membership operators: used to test if a sequence is presented in an object

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Extremely useful with lists:

```
In [12]: my_namelist = ['Marco', 'Mirko', 'Marika']

In [13]: if 'Mirko' in my_namelist:
    ...:     print("Yes!")
    ...:
Yes!
```

```
In [14]: my_list = [1,2,3,4]

In [15]: if 5 not in my_list:
    ...:     print("Not there")
    ...:
Not there
```

Python operators

When used with strings, the following operators assume a different meaning:

+ to concatenate strings

```
In [1]: 'Free' + 'Time'
Out[1]: 'FreeTime'
```

* to repeat strings (you can only use integer and strings)

```
In [2]: 4 * 'hi'
Out[2]: 'hihihihi'
```

When applied to lists, these operators behave in a similar manner.

Python operators

When used with strings, the following operators assume a different meaning:

- **+** to concatenate strings
- ***** to repeat strings (you can only use integer and strings)

When applied to lists, these operators behave in a similar manner.

```
In [18]: my_list = ['you', 'he', 'she', 'we', 'they']

In [19]: print(my_list[0])
you

In [20]: print(my_list[-2])
we

In [21]: print(my_list[-3:])
['she', 'we', 'they']

In [22]: my_list + my_list
Out[22]: ['you', 'he', 'she', 'we', 'they', 'you', 'he', 'she', 'we', 'they']

In [23]: 2 * my_list
Out[23]: ['you', 'he', 'she', 'we', 'they', 'you', 'he', 'she', 'we', 'they']

In [24]: 2 * my_list[0]
Out[24]: 'youyou'
```

Python operators

Exercises

Using the Python interpreter:

1. Type: hello + 3
   and make the result be 14.

2. Create the variables *value* and *percentage* to compute the 5% of 14350.

3. Assemble a string (e.g., ‘hello’) from a few other strings.

4. Assemble a sentence from a list of strings.

Python operators

Exercises

Using the Python interpreter:

1. Type: hello + 3
   and make the result be 14.

2. Create the variables *value* and *percentage* to compute the 5% of 14350.

```
In [1]: hello = 11

In [2]: print(hello + 3)
14

In [3]:

In [3]: percentage = 0.05

In [4]: value = 14350

In [5]: print(percentage * value)
717.5
```

Python operators

Exercises

Using the Python interpreter:

3. Assemble a string (e.g., ‘hello’) from a few other strings.

```
In [1]: string_1 = 'home'

In [2]: string_2 = 'hotel'

In [3]: string_3 = 'lounge'

In [4]: string_4 = string_1[0] + string_2[-2:] + string_3[0:2]

In [5]: print(string_4)
hello
```

Python operators

Exercises

Using the Python interpreter:

3. Assemble a string (e.g., ‘hello’) from a few other strings.

4. Assemble a sentence from a list of strings.

```
In [1]: string_1 = 'home'

In [2]: string_2 = 'hotel'

In [3]: string_3 = 'lounge'

In [4]: string_4 = string_1[0] + string_2[-2:] + string_3[0:2]

In [5]: print(string_4)
hello
```

```
In [6]: list_articles = ['A', 'an', 'the']

In [7]: list_animals = ['cat', 'dog', 'monkey']

In [8]: list_verbs = ['eats', 'sleeps', 'drinks', 'says']

In [9]: list_greetings = ['hi', 'hello', 'bye']

In [10]: list_other_words = ['and', 'for', 'to']

In [11]: list_fruit = ['apples', 'berries', 'bananas']

In [12]: list_final = list_articles[0]+' '+ list_animals[2]+' '+ list_verbs[0]+' '+ list_fruit[2]+' '+ list_other_words[0]+' '+ li
    ...: st_verbs[3]+' '+ 2*list_greetings[2]

In [13]: print(list_final)
A monkey eats bananas and says byebye
```

Python Types

Dictionaries

Python type used to store data values in key : value pairs.

A dictionary is a collection which is ordered, changeable and do not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values.

```
In [18]: my_dict_1 = {'brother':'Marco', 'sister':'Anna', 'dog':'Pluto'}   # dictionary of strings

In [19]: my_dict_2 = {'brother':1991, 'sister':1999, 'dog':2006}   # dictionary of numbers

In [20]: my_dict_2['brother'] = 1992   # to assign

In [21]: print(my_dict_2)
{'brother': 1992, 'sister': 1999, 'dog': 2006}

In [22]: print('My sister was born in : {}'.format(my_dict_2['sister']))
My sister was born in : 1999

In [23]: print('My sister is : {}'.format(my_dict_1['sister']))
My sister is : Anna
```

Python Types

Tuples

Along with lists and dictionaries (and sets), tuples make a built-in data type in Python used to store collections of data.

```
In [3]: my_tuple = (3, 3.0, 'three')

In [4]: type(my_tuple)
Out[4]: tuple

In [5]: print(len(my_tuple))
3
```

Python Types

Tuples and lists are both used to store collection of data

They are both heterogeneous data types (you can store any kind of data type in the same collection).

They are both ordered (the order in which you put the items are kept).

They are both sequential data types so you can iterate over their items.

Items of both tuples and lists can be accessed by an [index].

Main difference:

tuples cannot be changed

(tuples are immutable objects)

while lists can be modified.

```
In [11]: a_list = [1,2,'hello',4,5.3, True]

In [12]: a_tuple = (1,2,'hello',4,5.3, True)

In [13]: print(a_list)
[1, 2, 'hello', 4, 5.3, True]

In [14]: print(a_tuple)
(1, 2, 'hello', 4, 5.3, True)

In [15]: a_list[2] = 'bye'

In [16]: print(a_list)
[1, 2, 'bye', 4, 5.3, True]

In [17]: a_tuple[2] = 'bye'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [17], in <cell line: 1>()
----> 1 a_tuple[2] = 'bye'

TypeError: 'tuple' object does not support item assignment
```

The Python keyword None

None

The keyword *None* refers to a variable / value which exists, but it is not yet defined.

The keyword None has the following value: NoneType

```
In [8]: a = None

In [9]: type(a)
Out[9]: NoneType

In [10]: a is None
Out[10]: True
```

Assigning the None value to variable does not delete it:
space for the variable content is reserved
and filled with the value None

Use the keyword is to check whether a variable is None.

Python casting

Casting functions

To convert one Python type into another, there are casting functions.

Their name is that of the type which we want to convert the argument type into.

```
In [5]: float(3)
Out[5]: 3.0

In [6]: str(3.0)
Out[6]: '3.0'

In [7]: int('three')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 int('three')

ValueError: invalid literal for int() with base 10: 'three'
```

Python: conditions, blocks, indentations

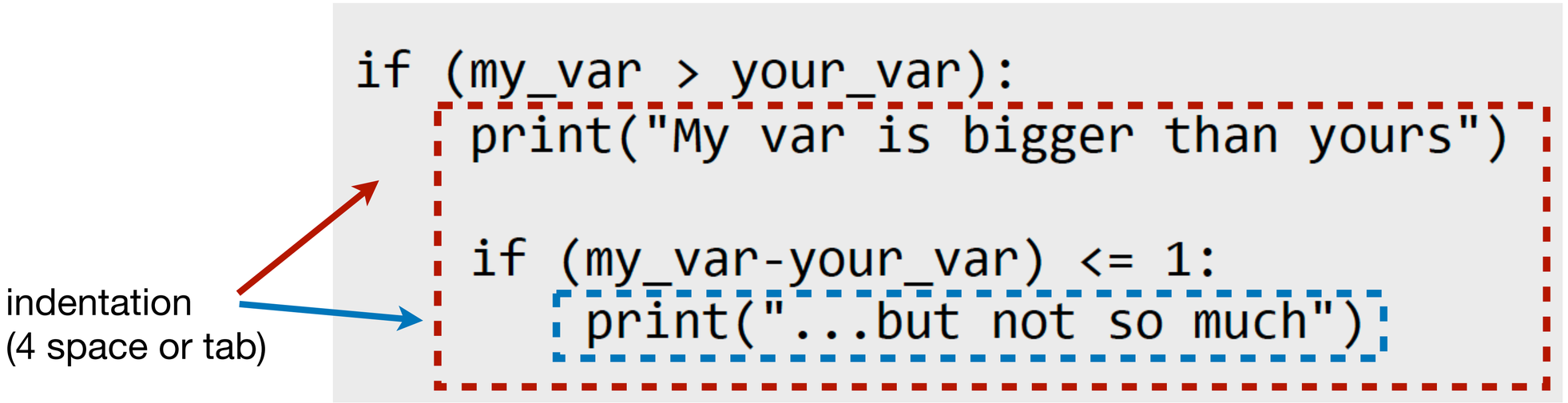
Conditional instructions

```
if (my_var > your_var):
    print("My var is bigger than yours")

    if (my_var-your_var) <= 1:
        print("...but not so much")
```

Python: conditions, blocks, indentations

Conditional instructions



Python: conditions, blocks, indentations

Additional conditions are added with **elif**

```
if (my_var > your_var):
    print("My var is bigger than yours")
    if (my_var-your_var) <= 1:
        print("...but not so much")
    elif (my_var-your_var) <= 5:
        print("...quite a bit")
    else:
        print("...a lot")
```

Python: loops

For / while loops

```
for item in my_list:
    print(item)
```

```
for i in range(10):
    print(i)
```

```
i=0
while i<10:
    print(i)
    i = i+1
```

Python: loops

For / while loops

```
for item in my_list:
    print(item)
```

```
for i in range(10):
    print(i)
```

```
i=0
while i<10:
    print(i)
    i = i+1
```

given a list of numbers

for each element in the list:
    if the element is smaller than 5:
        then, print it

Pseudo-code:
what to do

Actual code:
how to do

```
number_list = [13,12,34,4,51,8,27,18]

for item in number_list:
    if item < 5:
        print(item)
```

Python: loops

Online manuals, tutorials, official Python documentation help

```
for i in range(10):
    print(i)
```

Python range() Function

‹ Built-in Functions

Example

Get your own Python Server

Create a sequence of numbers from 0 to 5, and print each item in the sequence:

```
x = range(6)
for n in x:
  print(n)
```

Try it Yourself »

Definition and Usage

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Syntax

```
range(start, stop, step)
```

Parameter Values

Parameter	Description
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to stop (not included).
step	Optional. An integer number specifying the incrementation. Default is 1

More Examples

Example

Create a sequence of numbers from 3 to 5, and print each item in the sequence:

```
x = range(3, 6)
for n in x:
  print(n)
```

Python: loops

For / while loops

```
for item in my_list:
    print(item)
```

```
for i in range(10):
    print(i)
```

```
i=0
while i<10:
    print(i)
    i = i+1
```

```
for i, item in enumerate(my_list):
    print(i, item)
```

```
>>> my_list = ('orange', 'lemon', 'apple', 'strawberry')
>>> for i, item in enumerate(my_list):
...     print('position {}: item {}'.format(i, item))
...
position 0: item orange
position 1: item lemon
position 2: item apple
position 3: item strawberry
>>>
```

Python: modules

Python features several **modules**.

Each module contains instructions, constants and functions already available for the users.

Modules (like libraries) can be imported through the **import** statement.

Instructions of the module are available within the module **namespace** (i.e. the name of the module)

```
>>> a = sqrt(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>>
>>> import math
>>> b = math.sqrt(9)
>>> print(b)
3.0
```