Python language

Module 1

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Useful materials

Textbooks.

- **Think Python, 3rd edition** – Green Tea Press
  (English version: AllenDowney/ThinkPythonItalian: LaTeX source for the Italian Translation of Think Python. (github.com))
- "**Fluent Python**," L. Ramalho, ed. O'Reilly, 2015.

Prerequisites

01 **Python installed**

It is advisable to create a virtual environment.

02 **IDE**

Visual Studio Code, PyCharm.

Alternatively

03 **Google Account.**

Which allows the use of Google Colab.

OnlineGDB

Values and Data Types

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Values and types

Value

A *fundamental unit of data*, such as a number (2, 2.5) or a string, that a program processes.

Type

A *category* of values.

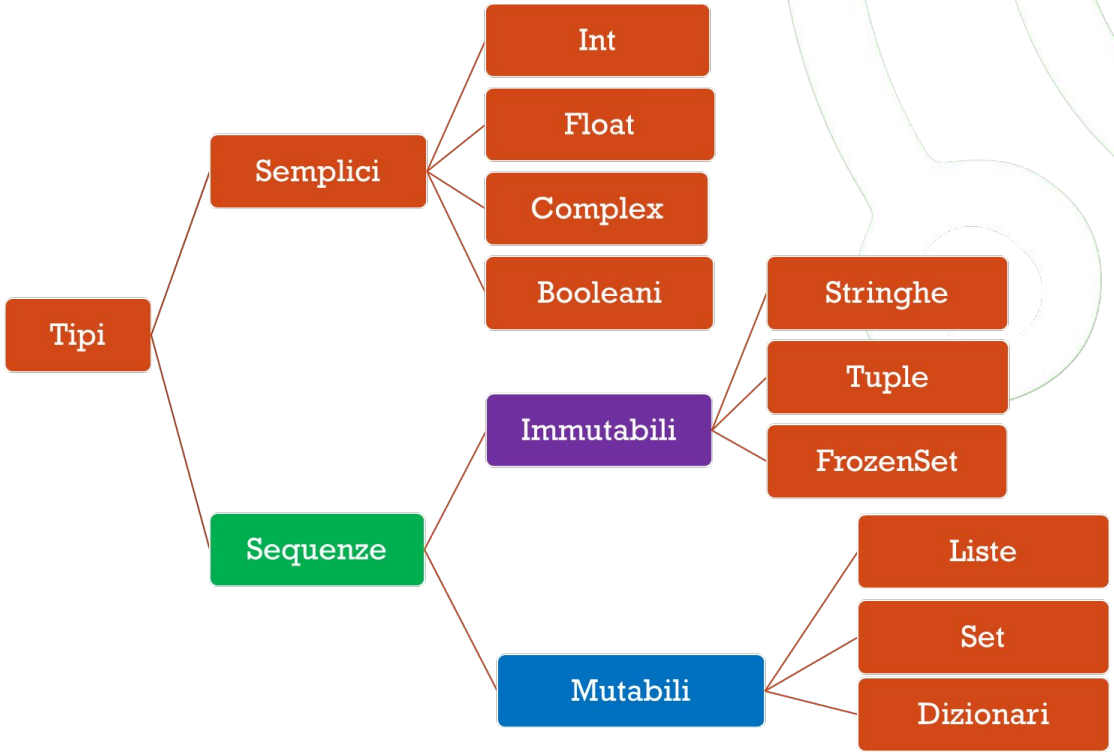
Types

Type

A *category* of values.

```
2 42.0 'Hello World!' True
```

Types


Tipi
Semplici
Int
Float
Complex
Booleani
Sequenze
Immutabili
Stringhe
Tuple
FrozenSet
Mutabili
Liste
Set
Dizionari

Variables and Assignment

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Variables and Assignment

A **variable** is a **container** that

- *holds* a piece of data;
- it has a **name,** which is used to refer to that data in the program;
- the contents of the variable can change at any time *by assigning it* a new value. First, however, the variable must be ***declared--how?***

An **assignment statement** is used to *create* a *new variable*, specifying its **name**, and *assign it* a **value**, or to *change* the *value* of an already *declared* variable.

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
>>> x = n = 10j
```

Variable names.

The Python interpreter has ***reserved* keywords**. These are used to recognize the structure of the program, so they ***cannot* be used** as **variable names**.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Types of Variables

- To find out *what type* a given *value belongs to*:

type()

```
>>> var = 10
>>> type(var)
<class 'int'>
>>> var = 10.0
>>> type(var)
<class 'float'>
```

- To know whether a variable is of a particular type:

isinstance(var, type)

```
>>> var = 5.9
>>> isinstance(var, int)
False
```

Operators

Arithmetic Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Integer division (//)
- Modulus (%)
- Power elevation (**)

Comparison Operators

- Equal to (==)
- Different from (!=)
- Less than (<)
- Less than or equal to (<=)
- Greater than(>)
- Greater than or equal to (>=)

Operatori Booleani

- `and`: Ritorna `True` se entrambi gli operandi sono veri, altrimenti `False`
- `or`: Ritorna `True` se almeno uno degli operandi è vero, altrimenti `False`
- `not`: Ritorna `False` se l'operando è vero, `True` se l'operando è falso

Operatori Binari

- *shift a sinistra* di n posizioni (x<<n)
- *shift a destra* di n posizioni (x>>n)
- *and* tra bit (x & y)
- *or* tra bit (x | y)
- *or esclusivo* tra bit (x ^ y)
- Inversione di bit (~x)

Expressions and instructions

An **expression** is a *combination* of *values*, *variables*, and *operators*.
An instruction is a portion of code that the Python interpreter can execute that has some effect

```
>>> n = 17
>>> print(n)
```

When an expression contains multiple operators, the sequence in which the calculation is performed depends on the order of the operations. The acronym **PEMDAS** is a useful way to remember the precedence rules:

- Parentheses
- Power elevation
- Multiplication and Division
- With equal precedence, evaluation takes place from left to right.

Control structures

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Control Structures

Control structures (or constructs) allow the sequence of execution of program instructions to be altered when certain conditions occur, going to **divert** (selection) or **repeat** (iteration) the flow. Control constructs can be:

- **Selective**: they choose blocks of code to execute based on a condition.
- **Iterative**: they repeat blocks of code until a condition is true or for a set number of times.

Selective constructs

- `if`
- `if-else`
- `match`

Selective constructs

- **`if`**
- `if-else`
- `match`

Conditional execution

Condition

```
if x > 0:
    print('x is positive')
```

Selective constructs

- **`if`**
- `if-else`
- `match`

Conditional execution

```
if x < 0:
    pass
```

If true, nothing happens!

Selective constructs

- **`if`**
- `if-else`
- `match`

Conditions in series

```
if x < y:
  print('x is less than y')
elif x > y:
  print('x is greater than y')
else:
  print('x and y are equal')
```

Conditions can also be nested

Selective constructs

- `if`
- **`if-else`**
- `match`

Alternative execution

```
if x % 2 == 0:
  print('x is even')
else:
  print('x is odd')
```

Selective constructs

- `if`
- `if-else`
- **`match`**

from version `3.10` onwards

```
match day:
    case "Monday":
        print("Start the week!")
    case "Friday":
        print("Almost weekend!")
    case "Saturday" | "Sunday":
        print("It's weekend!")
    case _:
        print("Normal day.")
```

Iterative constructs

- **for**
- while

```
for i in range(4):
  print('Hello!')
```

Iterative constructs

- `for`
- **`while`**

```
while n > 0:
  print(n)
```

It reads: as long as (`while`) the condition (`n>0`) is verified, do…

Iterative constructs

- for
- **while**

```
while True:
    row = input('> ')
    if row == 'end':
        break
    print(row)
```

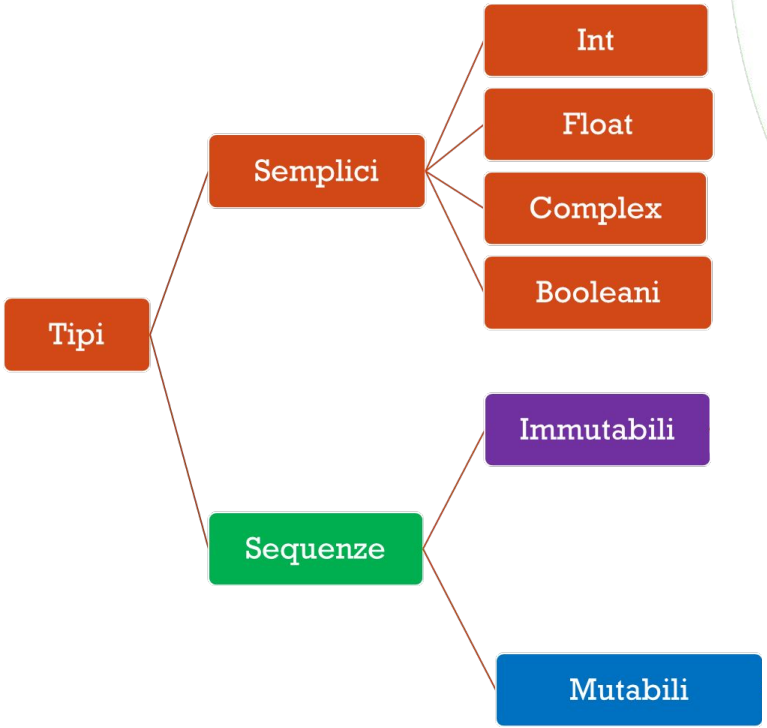
ITINERIS

Values and Data Types pt.2

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Types



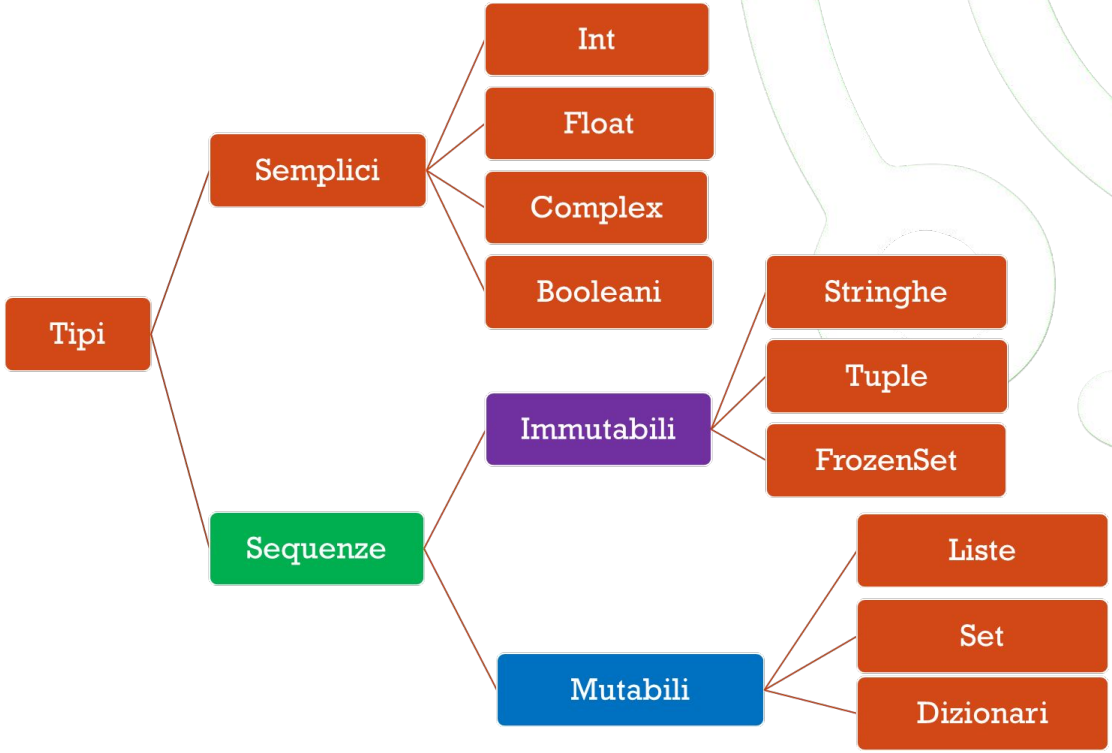
Immutable

They cannot be changed after they are created. If we tried to change an element, Python would generate an error.

Mutables

Can be modified after creation: we can change, add or remove elements.

Types


Tipi
Semplici
Int
Float
Complex
Booleani
Sequenze
Immutabili
Stringhe
Tuple
FrozenSet
Mutabili
Liste
Set
Dizionari

Immutable sequences

Strings

```
var = 'text'
var = 'text'
```

- sequence of *characters*.
- Individual characters can be accessed using the square bracket operators:

```
>>> fruit = "apple"
>>> letter = fruit[1]

>>> i = 1
>>> letter = fruit[i]
>>> letter = fruit[i+1]
>>> letter = fruit[1.5] # what's going on?
```

TypeError: string indices must be integers

Immutable sequences

Strings - conversions between types

- We can convert a number to a string with the **`str()`** function:

```
>>> num = 12345
>>> s = str(num)
>>> type(s)
<class 'str'>
```

- We can convert a string into the integer or float number it represents using the **`int()`** and **`float()`** functions, respectively:

```
>>> s_as_int = int('20')
>>> s_as_float = float('2.1')
```

Operations on strings

Operator	Returns	Description
`len(string)`	`int`	Returns the length of the string
`string + string`	`str`	Concatenates the strings
`string * int`	`str`	Replicate string
`string in string2`	`bool`	Checks whether one string is contained in another
`string[int:int]`	`str`	Extracts a sub-string

Slicing

A *segment* or *portion of a string* is called a **slice**. The operation of selecting a portion of a string is called **slicing:**

We start at the n-th character (**inclusive**) `[n : m]` We arrive at the m-th character (**excluded**)

```
>>> s = 'Monty Python'
>>> s[:5]
'Monty'

>>> s[6:12]
'Python'
```

String traversal

While

```
index = 0
string = 'Meditel'

while index < len(string):
  letter = string[index]
  print(string)
  index = index + 1
```

For

```
string = 'Meditel'

For letter in string:
   print(letter)
```

Methods on strings

Method	Returns	Description
`string.upper()`	`str`	Returns the string in upper case
`string.startswith(string)`	`bool`	Checks whether the string starts with another
`string.endswith(string)`	`bool`	Checks if the string ends for another
`string.find(string)`	`int`	Returns the position of a sub-string
`string.find(string,int)`	`int`	Returns the position of a sub-string, starting at one position.
`string.count(string)`	`int`	Counts the number of repetitions of a sub-string
`string.replace(str,str)`	`str`	Replaces sub-strings
`string.strip(string2)`	`str`	Removes strings at the sides

Immutable sequences

Tuples

```
t = (1, 2, 3, 5, 8)
```

- Sequence of values, separated by commas. Elements can be of different types: `t = ('abc', 1, False)`
- We can create a tuple as follows:

```
t = tuple()
```

- We can create a tuple with a single element as follows:

```
>>> t1 = ('a',) # without comma is a string!
>>> type(t1)
<class 'tuple'>.
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence.

Immutable sequences

Tuple

```
fruits = ("apple," "banana," "cherry")
```

packing

```
(green, yellow, red) = fruits
```

unpacking

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits
(green, *tropic, red) = fruits
```

Immutable sequences

Tuples

```
t = (1, 2, 3, 5, 8)
```

- Square brackets are used to refer to an element of the tuple:

```
t[0] # 1
```

- Like strings, you can use the slicing operator to select successive elements:

```
t[1:4] # (2, 3, 5)
```

- Tuples are immutable; you cannot change their elements. But we can replace one tuple with another:

```
t = (0,) + t[1:] # t = (0, 2, 3, 5, 8)
t = (10,) + t[:3] # t = (10, 1, 2, 3)
```

Crossing a tuple

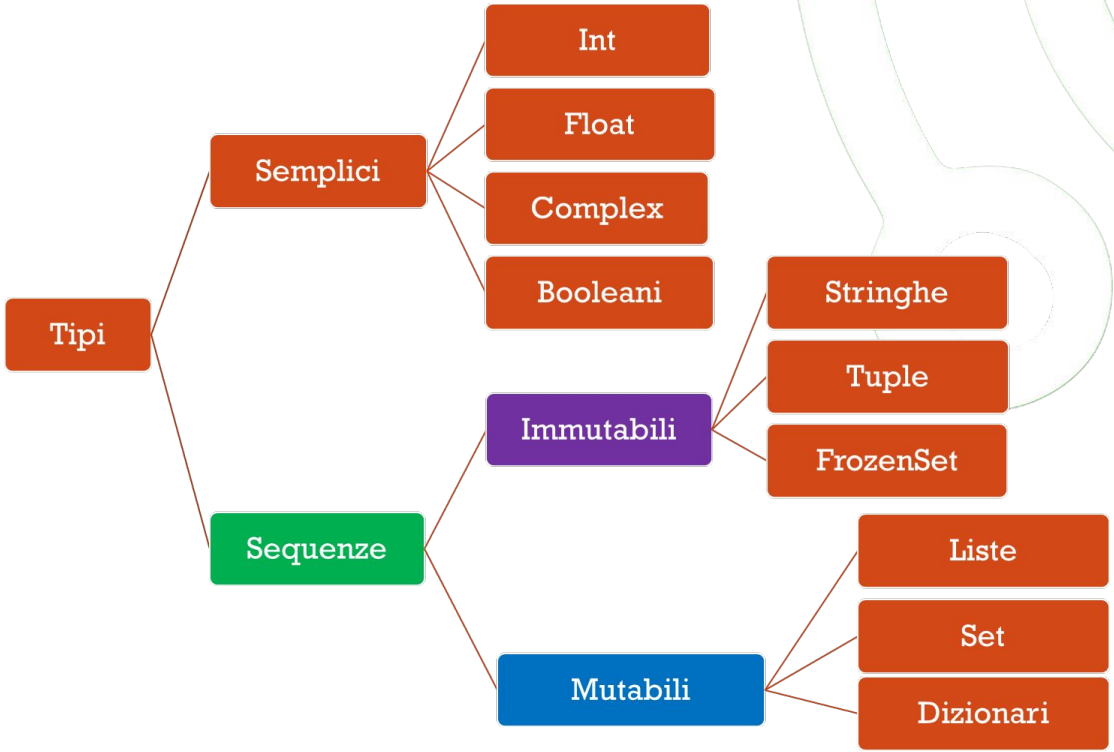
While

```
t = ("apple," "banana," "cherry")
i = 0
while i < len(t):
  print(t[i])
  i = i + 1
```

For

```
t = ("apple," "banana," "cherry")
for i in range(len(t)):
  print(t[i])
```

Types


Tipi
Semplici
Int
Float
Complex
Booleani
Sequenze
Immutabili
Stringhe
Tuple
FrozenSet
Mutabili
Liste
Set
Dizionari

Mutable sequences

Lists

```
list = [1, 2, 3]
```

- **Ordered** *sequence* of *elements*. We can **add**, **remove** or **modify** elements.
- Elements **can be *duplicated***, also elements need not all be of the same type: `list = ['spam', 2.0, 5, [10, 20]]`
  - A list within another list is called a **nested list**.
- We can create a list with its constructor:

```
list = list((1, 2, 3))
```

- **To modify** a list element: `list[0] = 3`.

Mutable sequences

Lists

```
list = [1, 2, 3]
```

- To access the elements of a list we use square brackets enclosing **the element index (integer!):**

```
elem = list[0]
```

  - If we tried to read or modify an element that does not exist, we would get an `IndexError` error message.
  - With an **index of negative value**, we count backward from the end of the list.

```
elem = list[-1] # 3
elem2 = list[-2] # 2
```

Operations on lists

Operator	Returns	Description
`range(int,[int])`	`range`	Lets you construct a list of integers
`len(list)`	`int`	Returns the length of the list
`list1 + list2`	`list`	Concatenates two lists
`list * list`	`list`	Replicate list
`object in list`	`bool`	Checks whether object is contained in an lsita
`list[index]=object`	`list`	Replaces the item in index position of the list
`list[int:int]`	`list`	Returns the sublist
`of the list[index]`	`None`	Deletes the element in index position from the list
`of the list`	`None`	Deletes the entire list

Crossing a list

```
list = [1, 2, 3]
for elem in list:
    print(elem)
```

```
list = [1, 2, 3]
for i in range(len(list)):
    list[i] = list[i] * 2
```

List Comprehension

List comprehension: a *method* or *construct* that can be used to define and create a list from a string or other existing list. We can also filter and transform data.

```
list_filtered = [element
                 for element in original_list
                 if condition(element)]


transform_list = [transform(element)
                  for element in original_list]
```

Loop with List Comprehension

```
list = [1, 2, 3]

for elem in list:
    print(elem)
```

```
list = [1, 2, 3]

[print(elem) for elem in list]
```

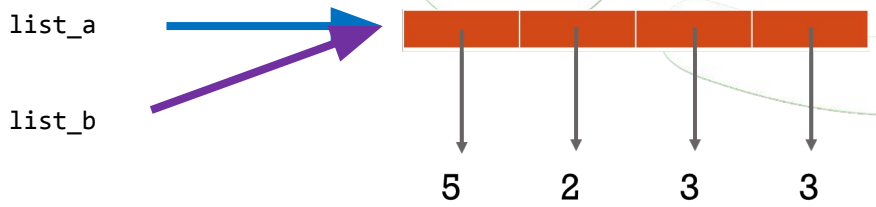
Methods on lists

Method	Returns	Description
`list.append(object)`	`None`	Adds an element to the end of the list
`list1.extend(obj)`	`None`	Extends the list with another iterable object
`List.clear()`	`None`	Removes all elements from the list
`list.insert(int,object)`	`None`	Inserts an element at position `int`
`list.remove(object)`	`None`	Removes the first instance of an object
`list.pop(index)`	`object`	Removes and returns the element at `index` position
`list.reverse()`	`None`	Reverses the list
`list.sort()`	`None`	Sorts the list
`list.count(object)`	`int`	Counts the number of repetitions of an object

Copy of a list

```
list_a = [5, 2, 3, 3]
list_b = list_a
```

The two lists point to the same object. If I modify `list_b` I will also modify `list_a`. What we have created is called an ***alias***.



To make a copy

```
list_c = list_a[:]
```

Mutable sequences

Set

```
s = {6, 2, 7, 3}
```

- collection of *unique*, *unordered* elements.
- The elements may appear in a different order each time we use the sequence, so indexes or keys cannot be used to refer to them.

- To create a set we can use the constructor:

```
s = set((1, 2, 3))
```

- You can have elements of different types:

```
s = set((1, True, "apple"))
```

Join Sets

- The **`union()`** and **`update()`** methods join all elements of both sets.
  - The same result as using the **`union()`** method can be obtained with the **|** (ONLY BETWEEN SETS) operator.
  - The **`update()`** method updates the original set, it does not generate a new one.

- The **`intersection()`** method keeps only duplicate elements (i.e., common to all sets), returning a new set with the common elements.
  - The same result is obtained with the **&** (ONLY BETWEEN SETS) operator.
  - The **`intersection_update()`** method keeps only duplicate elements, in the starting set.

Join Sets

- The **`difference()`** method returns a new set with elements from the first set that are not present in the other sets.
  - The same result is obtained with the **`-`** (ONLY BETWEEN SETS) operator.
  - The **`difference_update()`** method updates the starting set with all the elements that are not present in the other sets.

- The **`symmetric_difference()`** method keeps all elements except duplicates (i.e., those not common to both sets).
  - The same result is obtained with the **`^`** operator (ONLY BETWEEN SETS).
  - The symmetric_difference_update**`()`** method updates the starting set with all set elements except duplicates.

Methods on sets

Method	Returns	Description
`s.add(object)`	`None`	Adds an element to the set
`s.update(s2)`	`None`	Adds the elements of a set/iterable into a set
`s.remove(object)`	`None`	Removes an object from the set (raises exception)
`s.discard(object)`	`None`	Removes an object from the set (does not raise exception)
`s.pop()`	`obj`	Removes and returns the removed object
`s.clear()`	`None`	Empties the set
`of the` `s`	`None`	Totally clears the set

Mutable sequences

Dictionaries

```
dictionary = {'a': 1, 'b': 2}
```

- A collection of ordered, unduplicated elements of **key: value pairs.**
  - When we say dictionaries *are ordered*, it means that the elements have a *defined order,* and that order will not change.
  - **The keys** must be **immutable.**
  - Each **key** is **associated with** a **unique value.**
  - The `dict` function creates a new dictionary:

```
dictionary = dict(name = "Mara," age = 26)
```

- To add elements to the dictionary:

```
dictionary['key'] = 1
```

Mutable sequences

Dictionaries

```
dictionary = {"name": "Luke", "age": 30}
```

- To access the elements of a dictionary, we must refer to the name of the key, inside the square brackets:

```
name = dictionary["name"]
```

```
name = dictionary.get("name")
```

- To get all the keys in the dictionary, in the form of a list:

```
list_keys = dictionary.keys()
```

- To get all values present in the dictionary, in list form:

```
list_values = dictionary.values()
```

- To get all present in the dictionary, in the form of tuples in a list:

```
items = dictionary.items()
```

Dictionary traversal

```
for x in my_dict: #iterate on keys
  print(x)

for x in my_dict: #it on the keys and print the values
  print(my_dict[x])

for x in my_dict.values(): #iterate on the values
  print(x)

for x in my_dict.keys(): #iterate on keys
  print(x)

for x, y in my_dict.items(): #iterate on the key-value pair
  print(x, y)
```

Nested dictionaries

```
myfamily = {
  "child1" : {
    "name" : "emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "linus",
    "year" : 2011
  }
}
```

```
print(myfamily["child2"]["name"])
```

```
for x, obj in myfamily.items():
  print(x)

  for y in obj:
    print(y + ':', obj[y])
```

Functions

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Built-in functions

The functions available in a programming language are called **predefined**, or built-in. The set of such functions is called a ***library***, and all of them constitute

The Python Standard Library

What it is.

- ❑ It is the collection of modules and packages included with Python. It provides ready-to-use functionality to address common problems in programming (e.g., `math` and `random`).

Includes:

- ❑ Built-in modules (in C) → access to system functionality (e.g., file I/O).
- ❑ Modules in Python → standardized solutions, cross-platform portability.

Components:

- ❑ **Windows** usually includes the entire library and additional components.
- ❑ For **Unix-like** devices Python is in the form of a collection of packages, so you may need OS tools to be able to add additional optional components.

Built-in functions

In order to call a function from libraries such as `math` and `random`, you need to use the combination

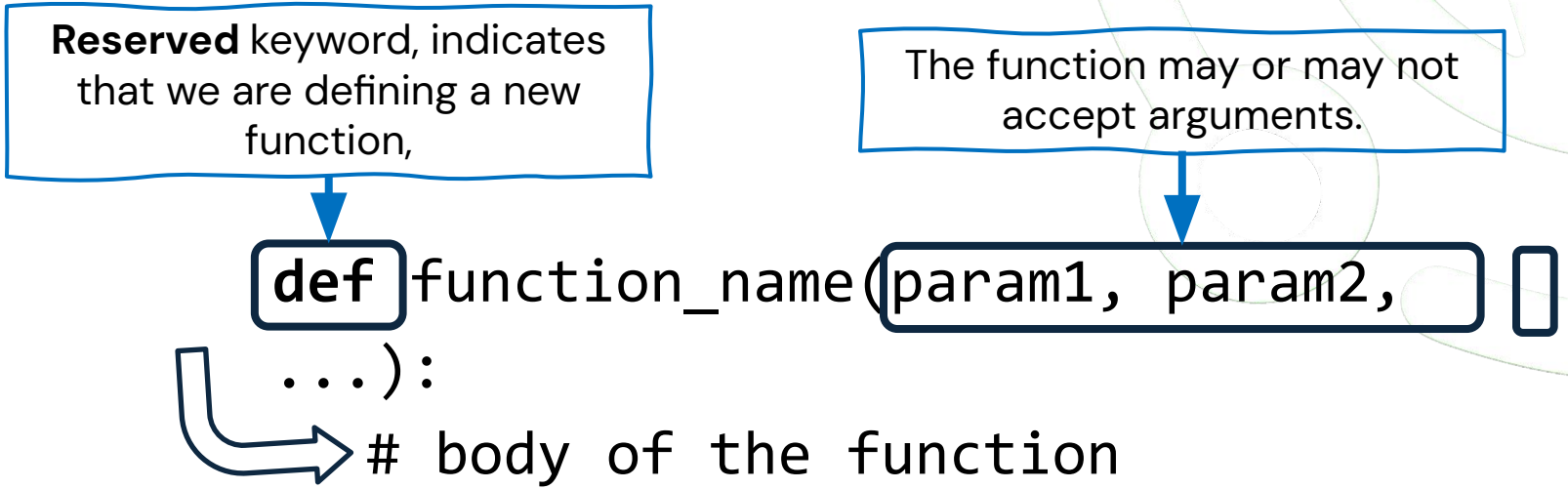
`from - import`

Syntax:

```
from library_name import function_name
```

- `library_name` is the symbolic name of a library;
- `function_name` can be:
  - The name of a specific function of that library (this will allow only that function to be used);
  - The symbol * indicating all functions of that library;
- If the `from import` combination is not used correctly, the function call will produce an error.
- To download a library: `pip install library_name`

Declaration of a function



Passing parameters

```
def function_name(param1, param2,
...):
    # body of the function
```

- Parameters are passed in ***object reference*** mode: a copy of the variable is passed.
- **Please note**: a python variable does not contain objects, but object references → `a = b` does not create a copy of object `b`, but the two variables refer to the same object.
- A function **can modify** the object that the variable passed as a parameter refers to ***only if the corresponding object is modifiable***.

Arbitrary Keyword Arguments

```
def function_name(param1, **kwargs):
    # function body
```

- If we don't know how many keyword arguments will be passed to your function, we add two ** asterisks before the parameter name in the function definition.
- This way, the function will receive a dictionary of arguments and can access the elements accordingly.

Default Parameter Value

```
def function_name(param1 = default):
    # function body
```

Positional-Only Arguments

```
def function_name(param1, /):
    # function body
```

- We are specifying that a function accepts only positional arguments, or only keyword arguments.
- Without the use of , / in the function definition, we are allowed to use keyword arguments even if the function expects positional arguments. If we add , /, we will get an error if we try to pass an argument as a keyword.

Keyword-Only Arguments

```
def function_name(*, param1):
    # function body
```

- To specify that a function accepts only keyword arguments, we add * before the arguments:
  - Without *, you are allowed to use positional arguments even if the function expects keyword arguments.
  - With *, you get an error if you try to pass a positional argument.

Return value

```
def compute_sum(x, y):
    sum = x + y
    return sum

def compute_sum(x, y):
    return x + y
```

`sum` is a **variable local** to the function.

Return value

```
def compute_sum_sub(x, y):
    sum = x + y
    sub = x - y
    return sum, sub
```

The two values are returned as a **tuple**.

```
def compute_sum_sub(x, y):
    return x + y, x - y
```

Lambda functions

A lambda function is a small anonymous function. It can accept any number of arguments, but can contain only one expression.
Syntax:

```
lambda arguments : expression
```

- The power of lambda functions is best seen when **they are used as anonymous functions inside another function**:

```
def multiplier(n):
    return lambda x: x * n

# Create a function that multiplies by 3
per_three = multiplier(3)

print(per_tre(5)) # Output: 15
```

Functions as "first-class entities"

In a language, there are often *entities* on which *restrictions* are placed *on what the program can do*.

- In C, arrays are "second-class citizens" compared to integers.

Typically, a "first-class" entity can:

- be **created dynamically during execution** (e.g., as the result of an operation);
- be **assigned to a variable** or **element of a data structure;**
- **be passed as a parameter** to a **subprogram;**
- be **returned** as the result of a **subprogram.**

Functions as "first-class entities"

- In Python 3, functions are objects (in the OOP sense), and they have the same "rights" as all other objects in the program.
- **We can create a function at runtime**. The definition of a function is for all intents and purposes an instruction, which takes effect only at the time it is executed.

```
num=int(input("Enter a number:"))
if num==42:
    def test(x):
        print("The answer is 42")
else:
    def test(x):
        print("x=",x)
test(10)
```

Assignment to data variables/structures

```
def f1(x):
    return x*2

def f2(x):
    return x**2

lst=[f1, f2]

print(lst[0](5))
print(lst[1](5))
```

Pass as parameter

```
from math import sin, cos

def derivative(f, x):
    dx = 1e-7
    return (f(x+dx) - f(x))/dx

print(derivative(sin, 0))
print(derivative(cos, 0))
```

High-order functions

- A function that takes a function as a parameter (or returns a function) is called a high-order function.
- High-order functions make it easier to write reusable code by allowing "customization" of a function.

Pattern Strategy: I customize the behavior of an object by passing it another object that defines part of the behavior of the first one.

High-order-functions: I customize the behavior of a function (but also of an object) by passing it a function that defines part of the behavior of the former.

High-order functions: an example

```
lst=["apple", "grape", "raspberry", "banana"]

print(sorted(lst))

print(sorted(lst, key=len))

def remove_first_letter(s):
    return s[1:]

print(sorted(lst, key=take_off_first_letter))
```

High-order functions to manipulate sequences

```
lst=["apple," "grape," "raspberry," "banana"]

def reverse(s):
    return s[::-1]

newlst = list(map(reverse, lst))
print(newlst)
```

High-order functions to manipulate sequences

```
lst=["apple," "grape," "raspberry," "banana"]

print(sorted(lst))

print(sorted(lst, key=len))

def remove_first_letter(s):
    return s[1:]

print(sorted(lst, key=take_off_first_letter))
```

Exercise 1

Write a function

```
check_temperature(temperatures, threshold)
```

that receives as input a list of temperatures and a threshold. The function must return True if at least one value exceeds the threshold, otherwise `False`.

Exercise 2

Write a function

```
simulate_time_cycle(n_operations,
                    mean_time_s)
```

that calculates the total production time (in minutes) for a given number of operations.

Python Arrays

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Arrays

An **array** is a variable used to hold multiple values at once.

- Python does not support arrays, but simulates their structure using lists:

```
cars = ["Ford", "Volvo", "BMW"]
```

- For more advanced arrays (e.g., numerical computation), the `NumPy` library is used:

```
pip install numpy
import numpy as np

arr = np.array([1,2,3,4])
print(type(arr)) # <class 'numpy.ndarray'>
```

Size of a NumPy array

- **0-D arrays (scalars)**: are the elements of an array. Each value that makes up an array is a 0-D array.
  - `arr = np.array(42)`
- **1-D arrays (one-dimensional array):** array consisting of 0-D elements.
  - `arr = np.array([42, 3, 55])`
- **2-D arrays (**array): array consisting of 1-D elements.
  - `arr = np.array([[42, 3, 55], [1, 3, 4]])`
- **3-D** arrays: array consisting of 2-D elements.
  - `arr = np.array([[[42, 3, 55], [1, 3, 4]],[1, 2, 3], [5, 6, 7]])`

To check the size: **`arr.ndim`**

Shape of a NumPy array

NumPy arrays have an attribute called `shape` that returns a tuple with each index containing the corresponding number of elements:

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

returns `(2, 4)`, which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4 elements.

To create an array of a given size:

```
arr = np.array([1, 2, 3, 4], ndmin=5)
```

Reshape a NumPy array

NumPy arrays have an attribute called `shape` that returns a tuple with each index containing the corresponding number of elements. To change the shape and go from one dimension to another, the reshape operation can be used:

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
newarr =[[ 1 2 3]
 [ 4 5 6]
 [ 7 8 9]
[10 11 12]]
```

The reshaping operation can be performed as long as the elements to be *reshaped* are the same in both forms.

```
newarr = arr.reshape(2, 2, -1)
```

Flattening of a NumPy array

The flattening operation is the conversion operation to go from a multidimensional array to a one-dimensional array:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)
```

`newarr` will be `[1 2 3 4 5 6]`.

Accessing a NumPy array

- **1-D arrays**.
  - `arr = np.array([42, 3, 55])`
  - `elem = arr[1]`
- **2-D arrays.**
  - `arr = np.array([[42, 3, 55], [1, 3, 4]])`
  - `elem = arr[0, 1] # second element - first row`
- **3-D arrays.**
  - `arr = np.array([[[42, 3, 55], [1, 3, 4]],[1, 2, 3], [5, 6, 7]])`
  - `elem = arr[0, 1, 2] # third element - second array - first row`
- Again, negative indexes can be used. The selection will start from the end of the array.

Slicing 1-D arrays

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

second_elem = arr[2]

from_third_elem = arr[3:]

first_three_elem = arr[:3]
```

We can use the `step` value to determine the step of slicing:

```
elems = arr[1:5:2]
Returns each element, from index 1 to index 5, skipping one.
[2 4]
```

Slicing 2-D arrays

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

- From the second element, cut elements from index 1 to index 4 (not included):

```
elems = arr[1, 1:4] # [7 8 9]
```

- From both arrays, returns the element at position 2:

```
elem = arr[0:2, 2]
```

- From both elements, returns the slice from index 1 to index 4 (not included):

```
elem = arr[0:2, 1:4]
```

Data Type.

Verification of the data type:

```
arr.dtype
```

Creating an array with a defined data type:

```
arr = np.array([1, 2, 3, 4], dtype='S')
```

```
i - integer
b - boolean
u - unsigned integer
f - float
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type ( void )
```

Data type

```
arr = np.array(['a', '2', '3'], dtype='i')
```

- The interpreter raises a ValueError, since a string like 'a' cannot be converted to an integer. The best way to change the data type of an existing array is to create a copy of the array with the `astype()` method.

- The `astype()` function creates a copy of the array and allows you to specify the data type as a parameter.

```
arr = np.array([1, 0, 3])
new_arr = arr.astype(bool)
```

Iteration with ndtier()

The nditer() function is a utility function that can be used from very simple to very advanced iterations.

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]])
```

Iteration over all elements

```
for x in np.nditer(arr):
  print(x)
```

Iteration on elements with step size

```
for x in np.nditer(arr[:, ::2]):
  print(x)
```

Iteration with ndenumerate()

```
arr = np.array([1, 2, 3])
```

```
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Join of NumpPy arrays

```
                              arr1 = np.array([1, 2, 3])
                             arr2 = np.array([8, 9, 10])
>>>arr = np.concatenate((arr1, arr2))
[1 2 3 8 9 10]
                          arr1 = np.array([[1, 2], [3, 4]])
                          arr2 = np.array([[5, 6], [7, 8]])
>>>arr = np.concatenate((arr1, arr2), axis=1)
[[1 2 5 6]
 [3 4 7 8]]
>>>arr = np.stack((arr1, arr2), axis=1)
[[[1 2]
  [5 6]]
 [[3 4]
  [7 8]]]
```

Split of NumpPy arrays

```
arr1 = np.array([1, 2, 3, 4, 5, 6])
```

```
>>>arr = np.array_split(arr1, 3)
[array([1, 2]), array([3, 4]), array([5, 6])]

>>>arr = np.array_split(arr1, 4)
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

- The `split()` method is available, but it does not adjust the elements when there are fewer elements in the source array for the split, as in the previous example. In that case `array_split()` works correctly but `split()` fails.

- We can specify the axis along which to perform the split:

```
newarr = np.array_split(arr, 3, axis=1)
```

Search

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
```

- `x = np.where(arr == 4)`
  - The value 4 is present at indexes 3, 5, 6.
- `x = np.searchsorted(arr, 7)`
  - Locates the index where the element should be inserted. The search starts from the left, in case we want to start from the right:
    - `x = np.searchsorted(arr, 7, side = 'right')`
  - The search can also involve multiple values. The return value is an array containing the indexes in which the elements could be inserted.

ITINERIS

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani