Introduction to programming with Python

generators

generators

Generators are a simple and powerful tool for creating iterators (any object that has implemented the __iter__ and __next__ methods). They can also be defined using regular functions that use the yield keyword whenever they want to return data.

```
>>> def gen():
...     for n in range(3):
...             yield n
...
>>> for n in gen():
...     print(n)
...
0
1
2
```

Generators, unlike regular functions, have the advantage of being able to pause and resume execution when the next() function is called. Additionally, their use is advantageous in terms of memory usage, as they generate one object at a time.

```
>>> def gen():
...     for n in range(5):
...             yield n
...
>>> iter_gen = iter(gen())
>>> next(iter_gen)
0
>>> next(iter_gen)
1
>>> next(iter_gen)
2
>>> next(iter_gen)
3
>>> next(iter_gen)
4
>>> next(iter_gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

generators - generator expression -

A quick and concise way to create a generator is by using a generator expression. The syntax is derived from Python's list comprehension (see subsequent slides).

The generator created through a generator expression has the same characteristics as one defined with a function but is more concise, although less versatile.

```
>>> generator = (n for n in range(5))
>>> iterator = iter(generator)
>>> next(iterator)
0
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
4
>>> next(iterator)
5
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

generators - generator expression -

It can be more efficient to use generator expressions, especially in combination with built-in functions typical of iterable objects (max(), min(), sum(), etc.)

```
>>> sum([x for x in range(5)]) # list comprehension
10
>>> sum(x for x in range(5)) # generator comprehension
10
```

The return value of the two functions is the same (sum() → sum all elements of an iterable object: 0+1+2+3+4=10), but in the first case, a list object will be created, the sum() function will be used and then the object deleted; while with the generator expression, a variable will be created to which the next element will be added in each iteration.

The getsizeof() function returns the size of an object in bytes. Below, you can see how a generator occupies much less memory than its equivalent list.

```
>>> from sys import getsizeof
>>> getsizeof([x for x in range(1000)])
9016
>>> getsizeof(x for x in range(1000))
112
```

syntax comprehension

python - syntax comprehension -

List comprehension provides a concise way to create any type of data structure. Common applications include creating new iterable objects where each element is the result of some operation applied to each member of another sequence or creating a subsequence of those elements that satisfy a specific condition.

```
>>> squares = []
>>> for n in range(11):
...     squares.append(n ** 2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

Traditional version:
- empty list;
- for loop;
- append every element to the power of 2.

Syntax comprehension:
- element of the sequence (optional: expression → n ** 2);
- for loop.

```
>>> squares = [n ** 2  for n in range(11)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

python - syntax comprehension -

Syntax comprehension can be used to generate every type of data structure and generators:

List

```
>>> l = [n for n in range(5)]
>>> l
[0, 1, 2, 3, 4]
```

Tuple

```
>>> t = tuple(n for n in range(5))
>>> t
(0, 1, 2, 3, 4)
```

IMPORTANT It is mandatory to use the tuple() constructor because simply using curly brackets create a generator.

Dict

```
>>> d = dict((k,v) for k,v in zip(range(5), 'abcde'))
>>> d
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

Set

```
>>> s = {n for n in range(5)}
>>> s
{0, 1, 2, 3, 4}
```

Generator

```
>>> g = (n for n in range(5))
>>> g
<generator object <genexpr> at 0x7f2481f7fb30>
```

python - syntax comprehension -

It's also possible to establish conditions (with conditional statements) within the structure of list comprehension.

Condition x % 2 != 0 to create a list with squared of odd numbers

```
>>> l = [x ** 2 for x in range(11) if x % 2 != 0]
>>> l
[1, 9, 25, 49, 81 ]
```

```
>>> l = [(n, 'even') if n % 2 == 0 else (n, 'odd') for n in range(5)]
>>> for el in l:
...         print(el)
...
(0, 'even')
(1, 'odd')
(2, 'even')
(3, 'odd')
(4, 'even')
```

- each element is inside a tuple;
- multiple conditions;

strings, byte, unicode

python - strings, byte, unicode -

Unicode is a system designed to represent every symbol/character of every language. It represents any letter, character, or ideogram as a 4-byte number (with UTF-32), thus allowing the representation of all existing languages. Each symbol is represented by a single number, and each number represents a single symbol.

```
>>> '\u0041' # hexadecimal representation
'A'
>>> chr(65) # decimal representation
'A'
```

I.E.: 'A' is always the Unicode value U+0041

In Python 3, strings are sequences of UNICODE characters, allowing you to handle any symbol or character regardless of the language used.

python - strings, byte, unicode -

However, it's immediately evident that using 4 bytes per character is very memory-intensive when, in reality, strings typically use only a few characters (usually belonging to the same language). In fact, characters beyond the first 65535 (the first 2 bytes) are rarely used.

In reality, most text contains MANY ASCII characters. Even a web page (written, for example, in Chinese) still contains spaces, HTML tags, CSS, and JavaScript. This is why UTF-8 encoding was created, a variable-length Unicode encoding optimized for representing ASCII characters.

UTF-8 is the default encoding used by Python 3 to handle files, both source code and general file management.

python - strings, byte, unicode -

Python implements two fundamental functions for encoding and decoding strings and decoding bytes:

encode() → It returns an encoded version of the string as a byte object.

decode() → It returns a decoded string from a byte object.

```
>>> '蛇'.encode('utf-8')
b'\xe8\x9b\x87'
```

```
>>> b'\xe8\x9b\x87'.decode()
'蛇'
```

class

python - class -

As already specified, everything in Python is an object. Classes provide a means to group data (class attributes) and functionality (class methods) together. Creating a new class generates a new object type, allowing you to create new instances of that object. Class instances can also have methods (defined by their class) to modify their state.

Convention: Class name are in CamelCase (whili method in snake_case)

```
>>> class Test:
...
...         a = 'variable inside a class: attribute'
...         def method(self):
...                 print('function inside a class: method')
...
```

#1 Instance of Test() class

#2 Accessing a class attribute

#3 Accessing p class method

```
>>> t = Test() # 1
>>>
>>> t.a # 2
'variable inside a class: attribute'
>>>
>>> t.p() # 3
function inside a class: method
>>>
```

python - classi -

Python provides special attributes and methods related to classes (these are recognizable because they are preceded and followed by double underscores, often called "dunder"). The instantiation operation creates an empty object. With the special method `__init__`, you can create objects with custom instances to define a specific initial state.

```python
class Human:

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def say_hi(self):
        print(f"Hello there, my name is {self.nome} {self.cognome}")
```

self → The first parameter of every class method is a conventional word that is a reference to the current instance of the class. This parameter is typically named self, but it doesn't need to be specified when calling a method, as Python automatically handles it.

Two instance of the same class with different arguments.

```
[5]: francesco = Human('Francesco', 'Faenza')
     francesco.say_hi()

     Hello there, my name is Francesco Faenza

[6]: francesco = Human('Giovanni', 'Rossi')
     francesco.say_hi()

     Hello there, my name is Giovanni Rossi
```

python - class -

In Python, classes support inheritance. The class that inherits is called the Subclass, and the one that is inherited from is called the Superclass. This allows you to create new classes based on existing classes, inheriting attributes and behaviors from the superclass to the subclass.

Inheritance:
Taking the previously created "Human" class, let's define a new class "Worker" that inherits from "Human"

#1 In order to make “Worker” inherit from “Human” the __init__ method, we have to call it explicitly into the inheriting class.
#2 Defining a new method for Worker class only.

```python
class Worker(Human):

        def __init__(self, name, surname, job_description):
            Human.__init__(self, name, surname) #1
            self.job_description = job_description

        def state_job_description(self): #2
            print(f"My job description is: {self.job_description}")
```

python - class -

#1 Instance of Worker class

#2 Worker inherited __init__ method from human, hence attributes ...

#3 … and methods

#4 Attribute and methods of Worker

```
[22]: cosimo = Worker('Cosimo', 'Giannini', 'construction worker') #1
      cosimo

[22]: Object of type Human <Giannini Cosimo>

[23]: cosimo.name #2

[23]: 'Cosimo'

[24]: cosimo.surname #3

[24]: 'Giannini'

[25]: cosimo.job_description #4

[25]: 'construction worker'

[26]: cosimo.say_hi() #3
      Hello there, my name is Cosimo Giannini

[27]: cosimo.state_job_description() #4
      My job description is: construction worker
```

python - class -

Python supports multiple-inheritance

We define two classes that will be used as parent classes for a sub-class

```
class Mammal:
    def mammal_info(self):
        print("Mammals can give direct birth.")

class WingedAnimal:
    def winged_animal_info(self):
        print("Winged animals can flap.")
```

python - class -

Now we define a class “Bat” which inherit from “Mammals” and “Winged animal”

#1 *Bat* inherit from *Mammal* and *WingedAnimal*

#2 Instance of Bat class

#3 Method of Mammal class

#4 Method of WingedAnimal class

```
[33]: class Bat(Mammal, WingedAnimal): #1
          pass

[37]: bruce = Bat() #2

[38]: bruce.mammal_info() #3
      Mammals can give direct birth.

[39]: bruce.winged_animal_info() #4
      Winged animals can flap.
```