Il modulo os

Il modulo **os** ci permette di interagire con il sistema operativo in modo portabile, tramite una serie di funzioni. Per esempio, posso voler conoscere il path alla working directory corrente:

```
os.getcwd()
```

O il nome del sistema operativo:

```
os.name
```

Il login corrente (con che utente sono loggato sulla macchina):

```
os.getlogin()
```


Il modulo os

Possiamo anche eseguire un comando utilizzando la funzione system:

```
import os
os.system('ls -l')

os.system(command)
```

Esegue command (una stringa) in una subshell. L'output di command viene stampato su stdout. Il valore di ritorno che è system dependent. Nei sistemi Unix, il valore di ritorno è l'exit status del processo. Si guardi il manuale del comando.

Il modulo os

Se voglio utilizzare l'output prodotto dal comando, posso aprire uno stream apposito utilizzando la funzione `os.popen(cmd)`, che apre una pipe *to* o *from* il comando cmd:

```
stream = os.popen(cmd, mode='r', buffering=- 1)
```

Il to o il from dipende da come è settato il parametro mode, di default impostato come 'r'. Il valore di ritorno è un file object.

```
import os
stream = os.popen('ls -l')
output = stream.read() # ma anche readlines()
print(output)
```


subprocess

Posso avere una flessibilità maggiore utilizzando la funzione `subprocess.Popen()`, che non aspetta la terminazione del comando.

```
import subprocess

process = subprocess.Popen(['echo', 'Questa è una
stringa'], stdout=subprocess.PIPE,stderr=subprocess.PIPE)
stdout, stderr = process.communicate()
```

possiamo leggere l'output con i metodi read() e readline() su stdout

.

subprocess

Un’altra funzione analoga, meno flessibile di Popen() è run().

```
import subprocess
cmd = ‘ls -l’
process = subprocess.run(cmd.split(),
stdout=subprocess.PIPE,stderr=subprocess.PIPE)
process.returncode # codice di ritorno del processo
process.stdout # stdout associato al processo
```

Si consulti il manuale in linea, o la documentazione online:
https://docs.python.org/3/library/subprocess.html

async subprocess

Un’interessante opportunità è quella di mandare in esecuzione dei processi in modalità asincrona tramite il modulo asyncio di Python.

Si guardi la documentazione al seguente link:
https://docs.python.org/3/library/asyncio-subprocess.html

https://docs.python.org/3/library/asyncio.html

async subprocess

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Eseguire task in maniera asincrona è un'opportunità interessante in diversi contesti.

Gli oggetti in Python

Python può essere utilizzato sia come linguaggio di scripting che come linguaggio orientato agli oggetti.

La *programmazione orientata agli oggetti* (OOP, Object Oriented Programming) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi (da Wikipedia).

Una classe è un'entità che permette di definire il comportamento (un insieme di metodi) e lo stato (attributi) di un oggetto.

Un oggetto è un'istanza di una classe. Tutti le istanze di una classe hanno la stessa struttura e lo stesso comportamento.

Gli oggetti in Python

Un concetto molto importante della programmazione a oggetti è l'ereditarietà, che permette di creare nuove classi che estendono il comportamento di una classe di partenza.

Risulta quindi possibile aggiungere e/o modificare metodi di una classe di partenza in base alle esigenze.

Come definire una classe?

```
# Definisce una nuova classe MyClass
class MyClass:
    pass
# Genera una nuova istanza di MyClass
myInstance = MyClass()
```


Gli oggetti in Python

```
class MyClass(object):
    pass
```

object è la classe base del linguaggio Python, da cui tutte le altre classi devono ereditare. **Dalla versione 3 di python non è più necessario esplicitare object.** Per chi volesse più dettagli, consiglio di leggere l'interessante discussione su stackoverflow.

Nell'esempio sopra *pass* indica solamente che la classe non contiene un'implementazione.

Gli oggetti in Python

```
class MyClass(base-classes):
   istruzioni
```

base-classes è una lista di una o più classi dalla quale MyClass estenderà il comportamento. Python supporta quindi l'ereditarietà multipla a differenza di altri linguaggi come Java in cui ho alcune limitazioni.

Nel corpo della classe andranno specificati attributi, che possono essere metodi (di diverso tipo) e data object.

Gli oggetti in Python

```
class MyClass(base-classes):
   istruzioni
```

base-classes è una lista di una o più classi dalla quale MyClass estenderà il comportamento. Python supporta quindi l'ereditarietà multipla a differenza di altri linguaggi come Java in cui ho alcune limitazioni.

Nel corpo della classe andranno specificati attributi, che possono essere metodi (di diverso tipo) e data object.

Gli oggetti in Python

```
class MyClass(base-classes):
   istruzioni
```

Possiamo consultare alcune proprietà di una classe andando a vedere alcuni attributi base.

print(MyClass.__name__, Myclass.__bases__)

Stamperà sullo stdout il nome della classe e la classe o la lista di classi basi.

Gli oggetti in Python

- Elementi che appartengono ad un dato oggetto, accessibility tramite la classica notazione punto (**nome_oggetto.nome_metodo()**).
- Gli attributi sono *elementi dinamici*, la **creazione/rimozione/modifica** degli attributi può avvenire in qualunque momento della vita dell'oggetto (classe o istanza).
- In python istanze della stessa classe possono avere **attributi differenti**.
- Possono essere:
  - Di istanza
  - Di classe

Gli oggetti in Python

- Anche le classi sono oggetti su cui e' possibile aggiungere attributi
- In questo caso gli attributi sono detti attributi di classe.
- Il metodo piu' semplice e' quello di dichiarare all'interno della classe un attributo e assegnargli un valore:

```
class MyClass:
      a = 20
print(MyClass.a) # stampa 20
MyClass.a = 30
print(MyClass.a) # stampa 30
```


Gli oggetti in Python

- **Attributi di istanza** sono attributi associati con un'istanza specifica di un oggetto.
- Questi possono essere creati al momento dell'inizializzazione in init o in modo dinamico

```
class MyClass:
      def_init_(self):
            self.x = 10 # Aggiungi x come attributo

ist = MyClass()
ist.y = 20 # Aggiungi y
del ist.x # Rimuovi x
ist.y = 300 # Modifica y
```

Gli oggetti in Python

- Le varie istanze della classe contengono anche gli attribute di classe.
- Questi vengono automaticamente aggiunti come attributi di istanza e prendono come valore il valore che l'attributo di classe ha in quel momento.

```
class MyClass:
      a = 0

print MyClass.a # stampa 0
ist = MyClass()
print ist.a # stampa 0
MyClass.a = 10
print MyClass.a # stampa 10
print ist.x # stampa 0
ist2 = MyClass()
print ist2.x # stampa 10
```


Gli oggetti in Python

- Gli attributi di classe possono essere aggiunti e rimossi dinamicamente. Ad esempio:

```
class MyClass:
      a = 0

print MyClass.a # stampa 0
ist = MyClass()
MyClass.z = 10 # attributo z aggiunto
ist2 = MyClass()
ist2.z = 20
print MyClass.z # stampa 10
print ist2.z # stampa 20
print ist.z # AttributeError: 'MyClass' object has no attribute 'z'
del ist2.z
del MyClass.z
```

Gli oggetti in Python

- Gli attributi di classe possono essere recuperati tramite:
  - La funzione built-in ***dir().***
  - Tramite l'attributo di classe ***__dict__.***

```
class ExampleClass:
      a = 20

print(ExampleClass.__dict__)

print(dir(MyClass))
```

- dict può essere usato anche per recuperare gli attributi di un'istanza.

Gli oggetti in Python

Visibilità degli attributi

- Normalmente tutti gli attribute sono considerati ***pubblici.***
- Esiste la possibilità di rendere alcuni attribute ***privati.***
- Gli attributi che **iniziano con doppio underscore (es** `__score)` sono trattati come fossero privati dall'interprete.

Questi rappresentano funzioni e operatori particolari.

Esempi di attributi speciali sono:

- __class__ contiene un riferimento alla classe a cui appartiene l'oggetto;
- __name__ contiene il nome della classe;

Gli oggetti in Python

Metodi di istanza (instance method):

metodi definiti dentro il corpo di una classe invocabili solo sulle istanza della stessa
Il primo argomento, solitamente per convenzione self, e' il riferimento all'istanza su cui il metodo e' invocato

```
class MyClass:
    def method(self):
        print('metodo di istanza')

ist = MyClass()
ist.method()
```


Gli oggetti in Python

Metodi di classe (classmethod):

- sono funzioni definite dentro il corpo di una classe applicabili alla classe stessa;
- il primo argomento, è *cls*, ossia un riferimento alla classe stessa.

Il metodo più semplice per definire un metodo di classe in Python è utilizzare i ***decorators***, un costrutto sintattico.

```
class MyClass:
    @classmethod
    def cmethod(cls):
        print('metodo di classe')

MyClass.cmethod()
```

Gli oggetti in Python

Metodi statici (Static methods):

- si comportano come una funzione globale dentro il namespace della classe (simili a quelli di Java e C++).
- possono essere chiamati sia sulla classe stessa che su un'istanza della classe. Per creare un metodo statico si applica il decorator @staticmethod .

```
class MyClass:
      @staticmethod
      def smethod():
            print('metodo
            statico')

MyClass.smethod()
```


Gli oggetti in Python

Il costruttore di un oggetto Python viene specificato nella funzione **init**, che **crea e *inizializza*** un'istanza di una classe. Il metodo init viene utilizzato anche per aggiungere attributi a un'istanza.

```
class Square:
    def __init__ (self, side=1):
        self.side = side
    def area(self):
        return self.side * self.side

sq = Square()
print(sq.area())
```

attributo di istanza

Gli oggetti in Python

`init()` come tutte le altre funzioni può avere un numero arbitrario di parametri.

Questi parametri verranno utilizzati per l'inizializzazione:

```
class ExampleClass:
      def __init__(self, x, y):
          self.x = x
          self.y = y
```


Gli oggetti in Python

La funzione **init** non deve includere delle istruzioni di tipo return (deve ritornare None), altrimenti viene lanciata un eccezione:

```
class MyClass:
      def_init_(self, x, y):
          print('init called')
          return 1
```

```
>> TypeError:_init_() should return None, not 'int'
```

Gli oggetti in Python

Il metodo **__del__** viene chiamato quando una data istanza di un oggetto viene distrutta (utilizzando l'operatore **del**); il metodo e' l'equivalente di un distruttore nei linguaggi C++. Per come il linguaggio python gestisce il garbage collector, questo metodo viene eseguito solo dopo che tutte le referenze ad un oggetto sono state rimosse:

```
class MyClass:
    def _init_(self):
           print('init called')
    def __del__(self):
           print('del called')
ist = MyClass()
del ist
```


Le classi Data

Il linguaggio Python permette di definire anche classi le cui istanze sono un insieme di elementi, come una classe che definisce un oggetto rappresentante un piano cartesiano.

```
import dataclasses
@dataclasses.dataclass
class Point:
     x: float
     y: float

pt = Point(2,3.5)
```

Come gli altri tipi di classi, anche la dataclasses supportano la definizione di metodi e proprietà.

Le classi Data

Il linguaggio Python permette di definire anche classi le cui istanze sono un insieme di elementi, come una classe che definisce un oggetto rappresentante un piano cartesiano.

```
Import dataclasses
@dataclasses.dataclass
class Point:
      x: float
      y: float

pt = Point(2,3.5)
```

Qui non specifichiamo il metodo init…

Vediamo anche che x e y saranno di tipo float.

Tuttavia questa è solo un'indicazione (Type annotation) e non è vincolante.

Come gli altri tipi di classi, anche la dataclasses supportano la definizione di metodi e proprietà.

Le classi Enum

Le classi di tipo Enum sono quelle classi che servono a definire un catalogo o a enumerare i possibili valori per una property particolare, ad esempio un colore. Una classe Enum definisce un gruppo di tal valori con dei nomi simbolici utilizzabili come delle costanti globali. I valori di una classe enum sono chiamati **membri**, e per convenzione sono definiti in maiuscolo.

```
from enum import enum
class Color(Enum):
      RED = 1
      GREEN = 2
      BLUE = 3
```

```
from enum import enum, auto
class Color(Enum):
      RED = auto()
      GREEN = auto()
      BLUE = auto()
```

`auto()` permette di assegnare automaticamente un intero progressivo ai valori (partendo da 1).

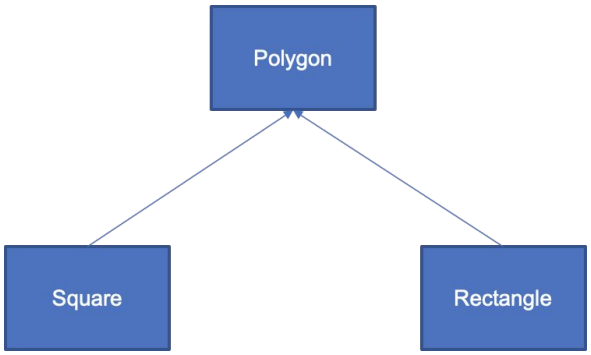
Ereditarietà

L'ereditarietà è il processo mediante il quale una classe **estende** un'altra classe assumendo gli attributi e i metodi di questa.

Le classi che derivano da un altra classe sono chiamate classi **figlio** (base) e le classi da cui derivano le classi figlio sono chiamate classi **padre**.

È importante notare che le classi figlio hanno la precedenza o estendono la funzionalità (ad es. Attributi e comportamenti) delle classi padri.

In altre parole, le classi secondarie ereditano i comportamenti delle classi super, ma possono anche specificare comportamenti diversi da seguire **override**.



Esempio pratico di ereditarietà - (1)

```
class Polygon:
    def __init__(self):
        self.name = "Polygon"

class Square(Polygon):
    def __init__(self, side):
        super().__init__()
        self.side = side
    def area(self):
        return self.side * self.side
```

```
class Rectangle(Polygon):
    def __init__(self, base, height):
        super().__init__()
        self.name = "Rectangle"
        self.base = base
        self.height = height
    def area(self):
        return self.base * self.height
```

Il metodo super() consente di delegare la chiamata di metodi a una superclasse o classe.

```
class super(type, object_or_type=None)
```

Qui andiamo a chiamare il metodo init() della superclasse per definire/ereditare anche gli attributi di istanza definiti in Polygon.

Esempio pratico di ereditarietà - (2)

```
def main():
    p = Polygon()
    s = Square(3)
    r = Rectangle(5,4)
    print(p.name, p)
    print(s.name, s.area())
    print(r.name, r.area())

if __name__ == '__main__':
    main()
```

Output:

Polygon <__main__.Polygon object at 0x7f8ba386efa0>

Polygon 9

Rectangle 20

Square stampa Polygon
Square eridità l'attributo di istanza name deinito in Polygon

Esercizio Classi 1

```
triangolo_1 = Triangolo(3,6)
triangolo_1.calcola_area()
triangolo_1.modifica_base(4)
triangolo_1.calcola_area()
triangolo_1.modifica_altezza(7)
triangolo_1.calcola_area()
```

Si scriva una classe Python chiamata **Triangolo** avente come attributi:

- base (float)
- altezza (float)

Entrambi gli attributi possono essere inizializzati alla creazione dell'istanza.

Come metodi:

- modifica_base → modifica la base del triangolo
- modifica_altezza → modifica l'altezza del triangolo
- calcola_area → calcola e stampa l'area del triangolo

Una volta implementata la classe, creare un'istanza che testi tutti e tre i metodi del triangolo.

Esercizio Classi 2 (1/2)

Creare una classe chiamata **Casa**. Tale classe avrà come attributi:

- arredamento (list) → inizialmente vuota
- numero_stanze (int) → inizializzabile alla creazione dell'istanza

Come metodi:

- arreda_casa → chiede all'utente di inserire in maniera indefinita un oggetto qualsiasi come arredamento della casa. L'inserimento dell'arredamento terminerà una volta che l'utente ha digitato "fine". Dopo aver digitato fine va mostrato l'arredamento aggiornato.
- elimina_oggetto → verrà chiesto all'utente di togliere un solo oggetto dalla lista dell'arredamento, e successivamente verrà mostrata la lista aggiornata. Nel caso in cui la lista è vuota o presenta un minimo di due oggetti non sarà possibile togliere oggetti dalla lista e verrà mostrato a video l'avvertimento e verranno mostrata la lista degli arredamenti. Nel caso in cui l'oggetto inserito non è presente nella lista verrà mostrato a video "oggetto non presente nell'arredamento della casa" e verrà mostrata la lista dell'arredamento.

Esercizio Classi 2 (2/2)

```
Inserisci un nuovo oggetto alla casa:sedia
Inserisci un nuovo oggetto alla casa:tavolo
Inserisci un nuovo oggetto alla casa:fine
Nuovo arredamento: ['sedia', 'tavolo']
Inserisci un nuovo oggetto alla casa:comodino
Inserisci un nuovo oggetto alla casa:poltrona
Inserisci un nuovo oggetto alla casa:fine
Nuovo arredamento: ['sedia', 'tavolo', 'comodino', 'poltrona']
Inserisci l'oggetto da eliminare nell'arredamento:tavolo
Arredamento aggiornato: ['sedia', 'comodino', 'poltrona']
Inserisci l'oggetto da eliminare nell'arredamento:comodino
Arredamento aggiornato: ['sedia', 'poltrona']
Inserisci l'oggetto da eliminare nell'arredamento:poltrona
Non è possibile eliminare un oggetto dalla lista
Arredamento: ['sedia', 'poltrona']
```

```
casa = Casa(5)
casa.arreda_casa() # --> aggiungere 2 oggetti
casa.arreda_casa() # --> aggiungere 3 oggetti
casa.elimina_oggetto() # --> rimuovere un oggetto
casa.elimina_oggetto() # --> rimuovere un oggetto
casa.elimina_oggetto() # --> rimuovere un oggetto --> darà avvertimento
```

Esercizio Classi 3

Creare una classe che rappresenti un libro e che avrà i seguenti attributi:

- titolo (stringa)
- nome autore (stringa)
- cognome autore (stringa)
- numero di pagine (intero)

Gli attributi sono inizializzabili alla creazione dell'istanza.
E avrà i seguenti metodi:

- Info() : mostra a video le informazioni relative al libro
- modifica_titolo(): modifica il titolo dell'istanza e mostra a video il nuovo valore
- modifica_autore(): modifica nome e cognome dell'autore dell'istanza e mostra a video il nuovo valore nell'ordine nome cognome
- modifica_numero_pagine(): modifica il numero di pagine dell'istanza e mostra a video il nuovo valore

Una volta terminata, inizializzare un oggetto e provare i metodi nel seguente ordine: info, modifica_titolo, info, modifica_autore, modifica_numero_pagine, info.

Esercizio Classi 3

Definire una sottoclasse, chiamata Rivista, della classe Libro dell'esercizio 8 e che abbia i seguenti attributi:

- anno (intero)
- colore (stringa)

che abbia i seguenti metodi:

- info(): restituisce una stringa con le informazioni relative alla rivista considerando anche gli attributi della classe padre.
- modifica_autore(): ridefinire modifica l'autore dell'istanza e mostra a video il nuovo valore nell'ordine cognome - nome e tutto in maiuscolo

Rappresentazioni e gestione delle informazioni

Per rappresentare e scambiare un'informazione possiamo affidarci ad un formato per la rappresentazioni di dati.

Affidarsi a un formato significa dare una rappresentazione chiara e standardizzata di un'informazione e del suo valore.

Un formato largamente utilizzato negli ultimi anni è sicuramente **JSON** (JavaScript Object Notation).

JSON viene utilizzato sia nel mondo Web che in quello dell'Internet of Things (IoT) per rappresentare i dati scambiati tra server e client, dispositivi e applicazioni, etc…

JSON

- JSON è un formato di interscambio dati leggero basato sul testo
- JSON è un formato testo, scritto utilizzando la sintassi degli oggetti Javascript
- JSON è autodescrittivo e semplice da leggere
- JSON essendo in formato testuale è indipendente dal linguaggio di programmazione usato.
- esistono diverse librerie per convertire oggetti in formato JSON e per leggere un oggetto JSON e avere una sua rappresentazione in Python
- La sintassi JSON è costituita da un insieme di coppie nome/valore, le coppie **nome**/**valore** sono separate da virgole `{"name": "Mario", "surname": "Rossi"}`.

JSON

- Gli oggetti sono definiti da parentesi graffe mentre le parentesi quadre definiscono un array. `{“name”: “Filippo”, “telephone”: [“+39XXXXX”, “+39YYYYY”]}`
- Il nome del dato richiede sempre i doppi apici.
- Un valore JSON deve rispettare dei tipi definiti.

AMMESSI	NON AMMESSI
stringa	funzione
numero	data
oggetto JSON	undefined
boolean	
null	

JSON

Alcuni esempi sono:

JSON strings `{ "name":"John" }`

JSON numbers `{ "age":30 }`

JSON objects `{"employee":{ "name":"John", "age":30, "city":"New York"}}`

JSON arrays `{ "employees":[ "John", "Anna", "Peter" ] }`

JSON booleans `{ "sale":true }`

JSON null `{"middlename":null }`

JSON in Python

Python ci mette a disposizione una libreria del linguaggio per lavorare direttamente con JSON. Il modulo JSON ci permette di lavorare sia a livello di stringhe sia a livello di file, con 4 funzioni principali. Le prime due permettono di lavorare a livello programmatico, senza appoggiarsi su file:

- **json.dumps(parametro)** prende in ingresso un oggetto Python e restituisce una stringa JSON
- **json.loads(parametro)** prende in ingresso una stringa JSON e restituisce un dizionario Python

JSON in Python

Esempi

```
import json

# encoding di un JSON
json.dumps({"name": "Filippo"})
>> '{"name": "Filippo"}'
json.loads('{"name": "Filippo"}')
>> {'name': 'Filippo'} # un dizionario python
```

JSON in Python

Le altre due funzioni (con un nome molto simile), ci permettono invece di lavorare con i file:

- **json.dump(data, file)** prende in ingresso un oggetto Python **data** e lo scrive su un **file** precedentemente aperto;
- **json.load(file)** prende in ingresso un oggetto di tipo **file** e restituisce un oggetto JSON;

```
{
    "dispositivo": {
        "power": "on",
        "parametri": {
            "p1": "0.0235849594",
            "p2": "0.4877774151",
            "p3": "2.0000022549"
        }
    }
}
```

```
import json
data = json.load(open("dati.json"))
data["dispositivo"]
data["dispositivo"]["power"]
data["dispositivo"]["parametri"]
```


JSON in Python

```
data = {'mappe': [{'id': '01', 'nome': 'test'},
                  {'id': '02', 'nome': 'prova'}],
        'dispositivo': {'power': 'on',
                        'parametri': {'p1': '0.0235849594',
                                      'p2': '0.4877774151',
                                      'p3': '2.0000022549'}}}
```

```
import json
data = { ... }
with open("output.json", "w") as outfile:
    json.dump(data, outfile)
```

Installare un nuovo pacchetto

La community python ci rende disponibile una mole di librerie con le quali è possibile svolgere una serie di compiti utilizzando un pacchetto di sorgenti creato da altri sviluppatori.

Come posso installare una libreria?

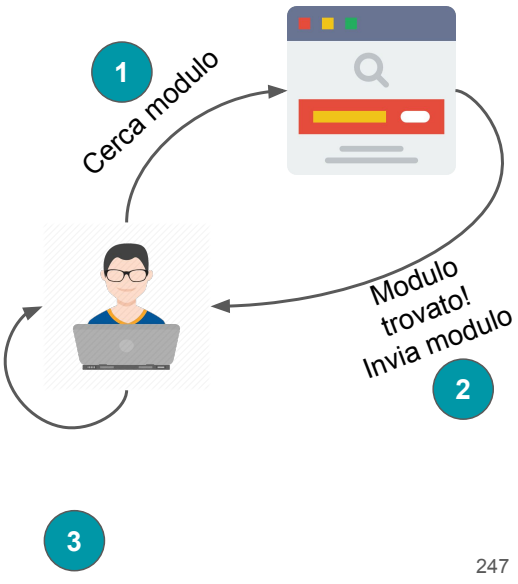
Dove posso trovare la libreria che mi serve?

Installo il modulo

Pip: il package manager per Python

Vi è la possibilità di installare moduli aggiuntivi che non fanno parte della libreria standard. Gli elementi in gioco sono due:

- **Python Package Index (PyPI):** è un repository che contiene decine di migliaia di moduli scritti in Python. Chiunque può scaricare moduli esistenti o condividere nuovi package su PyPI (link).
- **Pip:** è un tool che ci permette di cercare, scaricare ed installare moduli Python che si trovano sul Python Package Index. Ci consente inoltre di gestire i moduli già scaricati, permettendo di aggiornarli o rimuoverli.



Installare il tool pip - (1) Verifica della versione

Ogni versione di Python deve avere il suo pip! Se ad esempio abbiamo installato pip su python3.6 questo non funzionerà con Python2 o Python3.7.

Verifichiamo se per la versione Python3.6 installata è già presente il tool pip:

```
python3 -m pip -V
# oppure
pip3 -V

# Se non installato
# python3 -m pip -V
# /usr/bin/python3: No module named pip
```


Installare il tool pip - (2) Installare pip

```
# Scarichiamo il file necessario get-pip.py
# da eseguire per installare pip
wget https://bootstrap.pypa.io/get-pip.py

# Eseguiamo il file con la versione
# dell'interprete di cui vogliamo
# installare pip
python3.6 get-pip.py --user
```

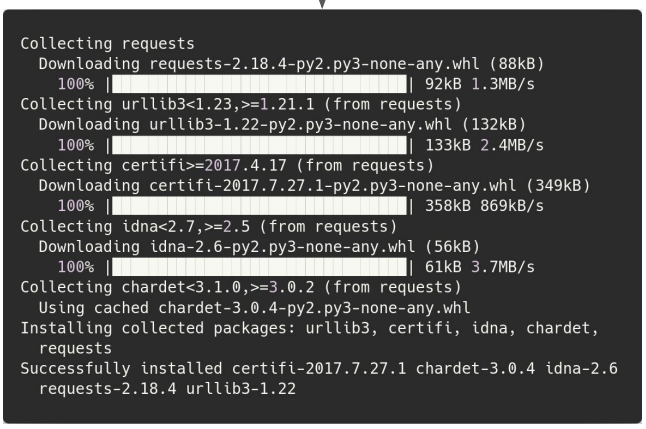
Una volta installato il tool verificare la corretta installazione eseguendo lo script della slide precedente.

Installare un modulo con pip

Per installare un package, basta eseguire:

```
python3 -m pip install -U <modulo>

# es:
# python3 -m pip install -U requests
```



Come possiamo vedere, pip non solo scarica e installa automaticamente il modulo, ma si assicura che tutte le dipendenze richieste dal modulo siano soddisfatte. Ovvero si occupa di scaricare altri moduli da cui dipende la libreria che vogliamo installare.

Altri comandi importanti

Per visualizzare lista di moduli installati, possiamo utilizzare il comando:

```
python3 -m pip list
```

Per cancellare un modulo e le sue dipendenze precedentemente installate:

```
python3 -m pip uninstall <modulo>
# es:
# python3 -m pip uninstall requests
```

Esercizio - 1

Seguire i seguenti passi:

1. Crea un modulo chiamato **Triangolo.py** contenente la classe **Triangolo**. Il suo inizializzatore **init** dovrà prendere gli argomenti angolo1, angolo2, angolo3. Assicurarsi che tali attributi vengano settati correttamente all'interno del corpo dell'inizializzatore
2. Creare una variabile **numero_di_lati** inizializzata a 3
3. Creare un metodo **controllo_angoli**, che si occuperà di stampare il valore degli angoli del triangolo e di ritornare vero o falso se la somma degli angoli del triangolo è uguale a 180 o no
4. Importa il modulo precedentemente creato in un nuovo file
5. Crea due oggetti triangolo assegnando al primo tre angoli la cui somma è 180 e al secondo tre angoli la cui somma è diversa da 180
6. Eseguire il metodo controllo_angoli per entrambi gli oggetti

Esercizio - 2

Data la classe precedentemente scritta, realizzare una funzione che prenda come parametro una lista di oggetti **triangolo** e scriva un file json che riporti esattamente il seguente formato di esempio sulla base del numero di triangoli presenti nella lista.


```
{
  "Numero esercizio": "2",
  "numero triangoli": "3",
  "triangoli": [
    {
        "numero triangolo": "1",
        "angolo 1": "50",
        "angolo 2": "80",
        "angolo 3": "50",
        "somma angoli": "180",
        "controllo angoli": "True",
    },
    {
        "numero triangolo": "2",
        "angolo 1": "22",
        "angolo 2": "30",
        "angolo 3": "120",
        "somma angoli": "172",
        "controllo angoli": "False",
    },
  ],
  "somma angolo 1": "72",
  "somma angolo 2": "110",
  "somma angolo 3": "170",
  "somma totale angoli": "252"
}
```

Esercizio - 3

Seguire i seguenti passaggi:

1. Creare un modulo Mensa.py, il quale conterrà un classe **Mensa**. La classe avrà un attributo locale chiamato **programma_settimana** e sarà di tipo dizionario.
2. Creare un metodo di classe chiamato **aggiungi_pasto_del_giorno()** il quale una volta invocato dovrà chiedere chiedere all'utente cosa verrà servito quel giorno come **primo**, **secondo** e **frutta**.
3. Una volta terminato tale elenco **[“primo piatto”, “frutta del giorno”]**, questo verrà inserito nel dizionario **programma_settimana**, secondo tale schema:

```
{"Lunedì": ["penne al sugo", "mela"], ...
```

4. Nel file **main.py** richiamare il modulo Mensa scritto in precedenza e implementare il codice per popolare il programma della settimana di tale mensa e che stampi a video il programma settimanale della mensa inserito.

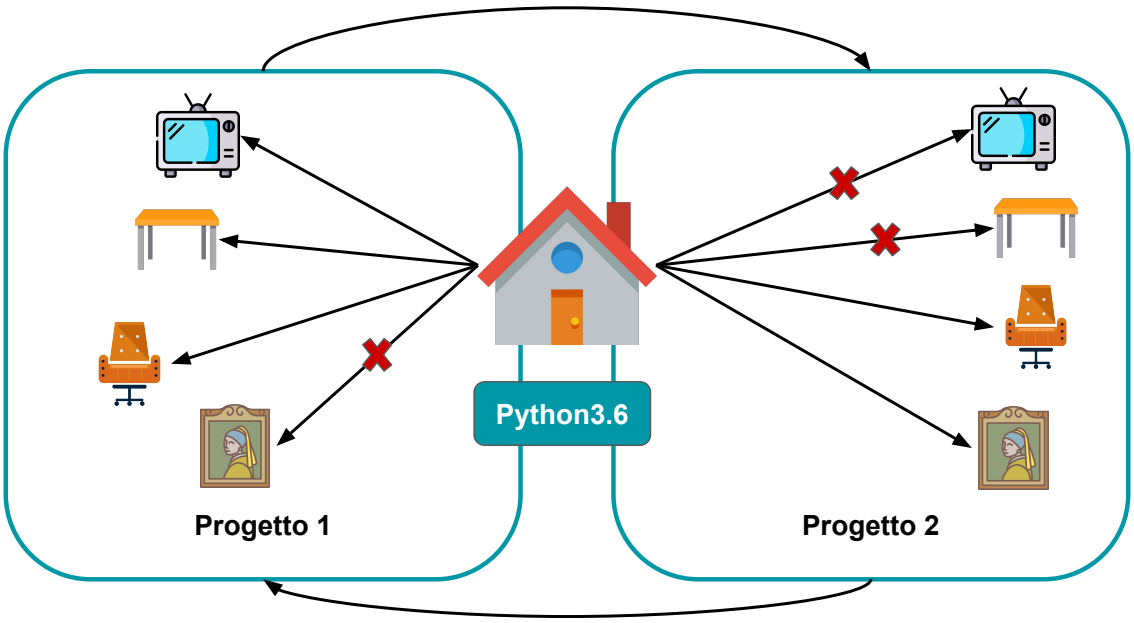
Esercizio - 4

Data la classe precedentemente scritta, realizzare una funzione che prenda come parametro l'oggetto **mensa** e scriva un file json che riporti esattamente il seguente formato di esempio sulla base del numero di triangoli presenti nella lista.

```
{
  "Numero esercizio": "4",
  "Nome esercizio": "Menù della mensa",
  "Menù": [
    {
        "Giorno": "Lunedì",
        "Piatti": [
          {
            "Primo": "Carbonara",
          },
          {
            "Frutta": "Mela",
          }
        ],
    },
    {
        "Giorno": "Martedì",
        "Piatti": [
          {
            "Primo": "Cappellacci",
          },
          {
            "Frutta": "Arancia",
          }
        ],
    },
  ],
}
```

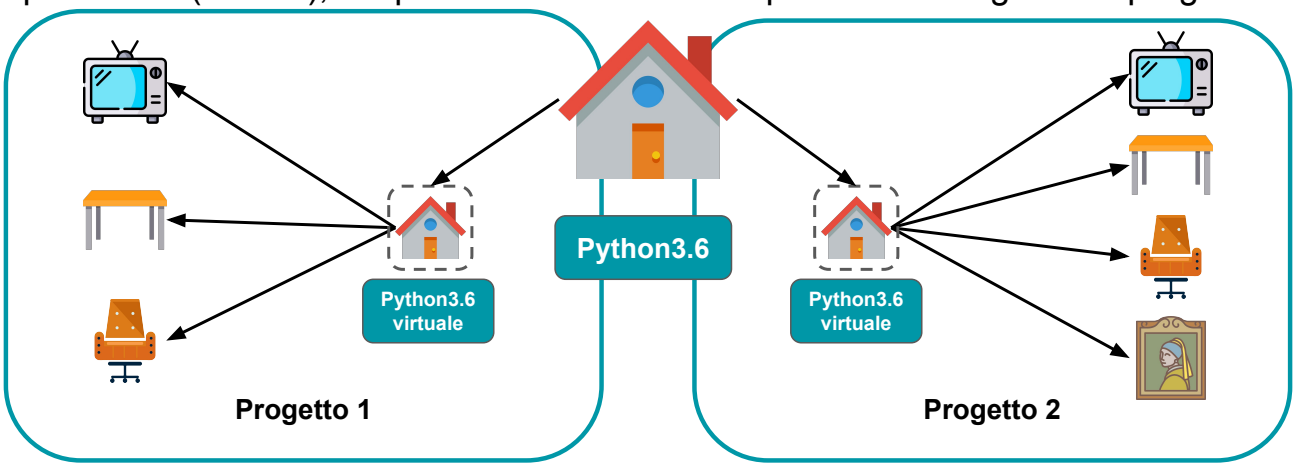
Ambiente virtuale Python

Mai installare moduli sull'interprete di sistema!



Cos’è un ambiente virtuale?

Lo scopo principale degli ambienti virtuali Python è quello di creare un ambiente isolato per i progetti Python. Ciò significa che ogni progetto può avere le proprie dipendenze (moduli), indipendentemente dalle dipendenze di ogni altro progetto.



Il modulo virtualenv in Python

Per poter creare un ambiente virtuale è possibile utilizzare diversi modi. Tra questi abbiamo quello di utilizzare un modulo **virtualenv** installato tramite pip. Una volta creato l’ambiente virtuale, verrà creata di conseguenza una cartella (nome_ambiente) che conterrà il nostro ambiente virtuale.

```
# installiamo il modulo virtualenv con pip
python3.6 -m pip install virtualenv --user

# creiamo una cartella qualsiasi
mkdir test-virtual

# spostiamoci nella cartella creata
cd test-virtual

# creariamo l'ambiente virtuale
virtualenv -p python3.6 <nome_ambiente>
```

```
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.6
│   ├── pip
│   ├── pip3
│   ├── pip3.6
│   ├── python -> python3.6
│   ├── python3 -> python3.6
│   └── python3.6 -> /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
├── include
├── lib
│   └── python3.6
│       └── site-packages
└── pyvenv.cfg
```

Attivare l’ambiente virtuale

Per entrare all’interno dell’ambiente è necessario attivare uno specifico file all’interno della cartella che rappresenta l’ambiente virtuale stesso, quindi digitare il seguente comando:

```
source <nome_ambiente>/bin/activate
```

```
(envname) $
```

Una volta eseguito si può notare che la riga del terminale è stata cambiata, mostrando una parentesi con il nome dell’ambiente attivato.

Per disattivare l’ambiente:

```
deactivate
```