



Introduzione e Configurazione di Visual Studio Code

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Cos'è Python?

- Linguaggio di programmazione ad oggetti;
- Linguaggio di alto livello (C++ / Java);
- Linguaggio interpretato;
- Linguaggio interattivo;
- Prototipazione veloce;
- Gestione automatica della memoria;
- Sintassi semplice;
- Tipizzazione dinamica;
- Portabilità;



On Platform

- Indipendente dalla piattaforma;
- Interprete scritto in C;
- Disponibile per tutte le piattaforme;
- Open Source
- Ultima release 3.11.5 (<https://docs.python.org/3/>);

Materiale Utile

- How to Think Like a Computer Scientist, Allen Downey Jeffrey Elkner Chris Meyers, Green Tea Press
<http://www.greenteapress.com/thinkpython/thinkCSpy.pdf>
- Pensare da informatico, Allen Downey Jeffrey Elkner Chris Meyers, Green Tea Press
https://www.google.it/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwjp0ey754vfAhWlp-osKHZCEAVgQFjAAegQICBAC&url=http%3A%2F%2Fwww.python.it%2Fdoc%2FHowtothink%2FHowToThink_ITA.pdf.gz&usg=AOvVaw0HZS7xER--MQ5yMI2a1KII
- A WhirlWind Tour of Python, Jake VanderPlas, O'REILLY
<https://s3-us-west-2.amazonaws.com/python-notes/a-whirlwind-tour-of-python-2.pdf>
- The Hitchhiker's Guide to Python!
<https://docs.python-guide.org>
- Fluent Python: Clear, Concise, and Effective Programming
https://github.com/piyusharma95/gyaan_ke_panne/blob/master/Fluent%20Python%20Clear%20Concise%20and%20Effective%20Programming.pdf

Interprete

- Python dispone di un interprete interattivo molto comodo e potente:
 - 🐍 Per avviarlo è sufficiente digitare *python* al prompt di una shell*
 - 🐍 Il prompt (`>>>`) che appare sullo schermo è pronto a interpretare ed eseguire le nostre istruzioni

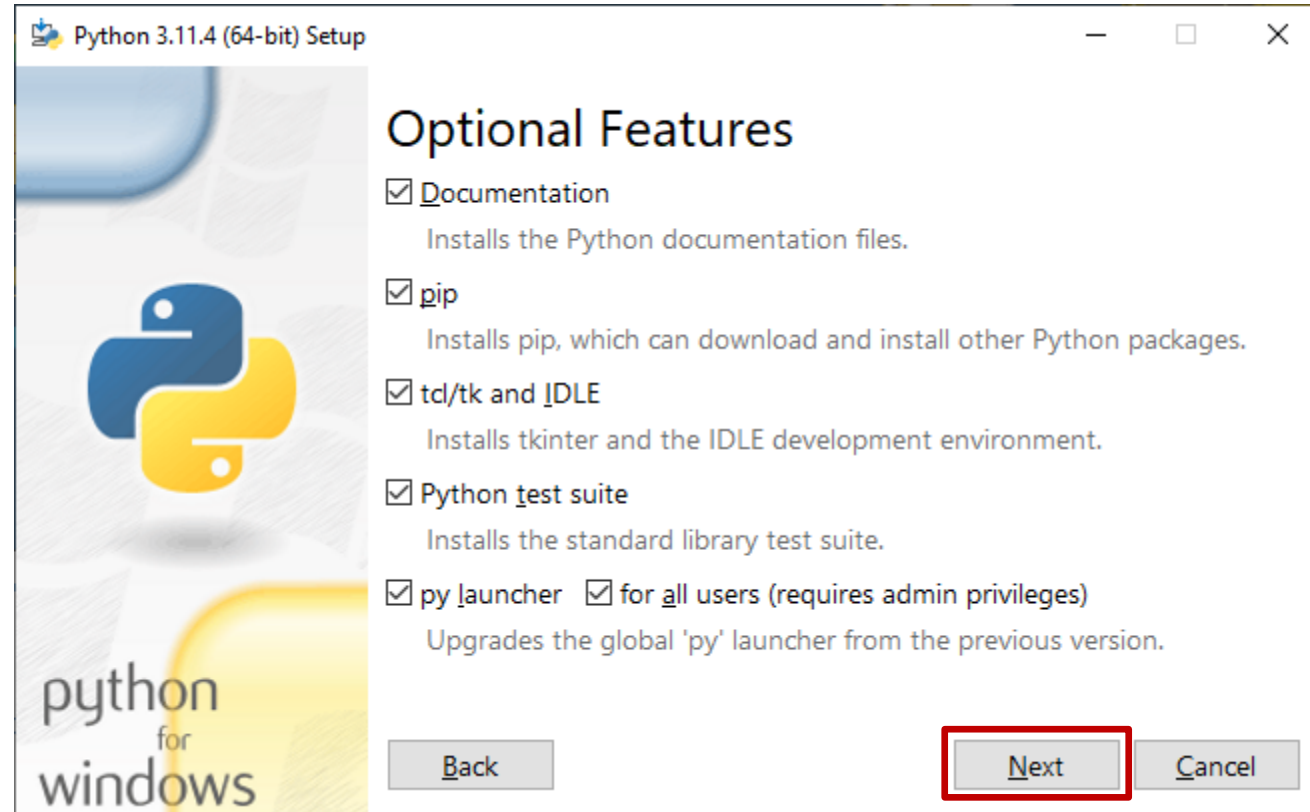
```
Python 3.7 (64-bit)
Python 3.7.6rc1 (tags/v3.7.6rc1:bd18254b91, Dec 11 2019, 20:31:07) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 3+5
8
>>> "Hello World!"
'Hello World!'
>>> _
```

* L'interprete deve essere installato e aggiunto alla variabile di sistema PATH.

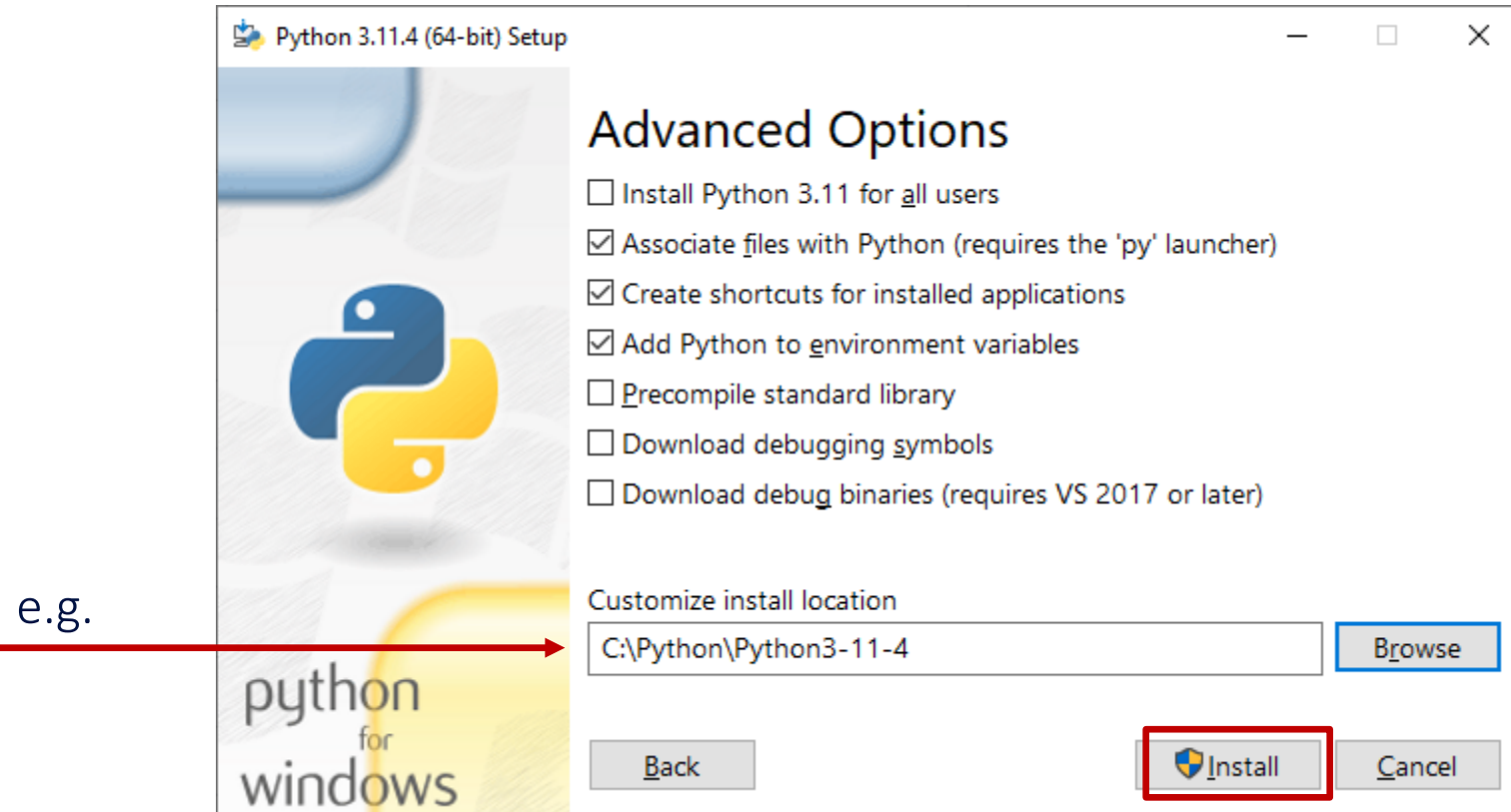
Interprete

- L'interprete è un file denominato:
 - 🐍 *python* su Unix
 - 🐍 *python.exe* su Windows
- Se invocato senza argomenti presenta un'interfaccia interattiva;
- Può essere seguito dal nome di file contenente comandi Python. In tal caso il file verrà interpretato ed eseguito;
- I file sorgente Python sono file di testo, generalmente con estensione “.py”;
- L'interprete Python può essere scaricato dal sito ufficiale (<https://www.python.org/downloads/>)

Interprete – Installazione (Windows)



Interprete – Installazione (Windows)



Interprete – Installazione (Windows)

- Completare l'installazione eventualmente disabilitando il limite sulla lunghezza dei path;
- Verificare l'installazione dell'interprete aprendo il prompt dei comandi e digitando python;
- Se l'interprete interattivo non si avvia, disconnettere l'utente per assicurarsi che il path di sistema venga aggiornato e riprovare.

Interprete – Installazione macOS

- L'installazione di sistema di Python su macOS non è supportata da VSCode. Si consiglia di utilizzare un sistema di pacchetti (e.g. *homebrew*) per installare una nuova versione dell'interprete:

```
brew install python3
```

Interprete – Installazione Ubuntu

- Se usate Ubuntu dovrete già avere una versione di python3 funzionante. Per vedere quale versione di Python 3 hai installato, apri un prompt dei comandi ed esegui:

```
python3 --version
```

- Se stai utilizzando Ubuntu 16.10, puoi facilmente installare Python 3.6 con i seguenti comandi:

```
sudo apt-get update  
sudo apt-get install python3.6
```

Interprete – Installazione Ubuntu

- Se stai utilizzando un'altra versione di Ubuntu (ad esempio l'ultima versione LTS) o desideri utilizzare un Python più attuale, ti consigliamo di utilizzare il PPA deadsnakes:

```
sudo apt-get install software-properties-common
```

```
sudo add-apt-repository ppa:deadsnakes/ppa
```

```
sudo apt-get update
```

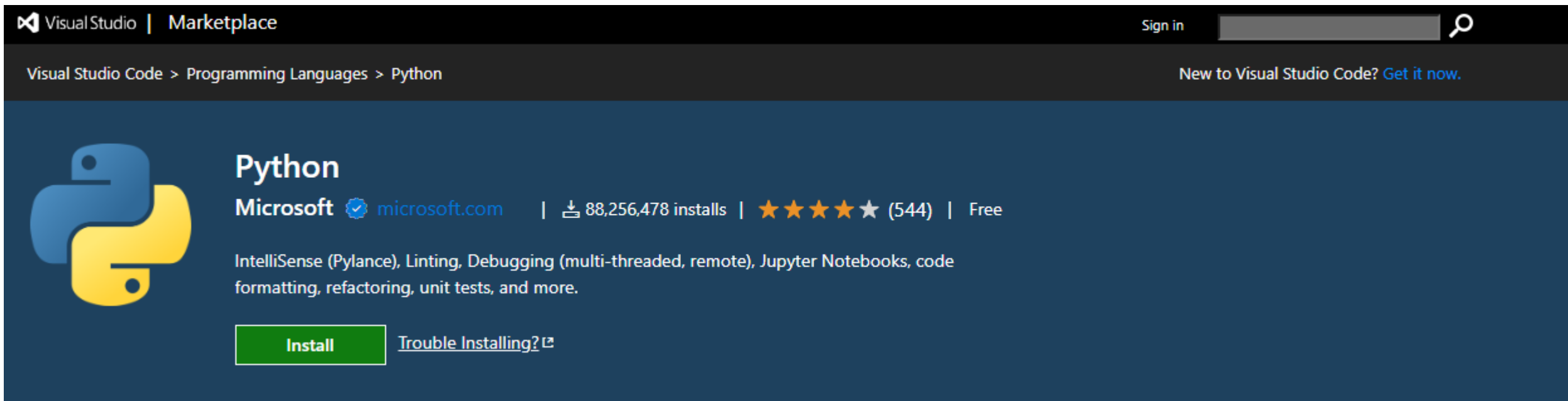
```
sudo apt-get install python11.4
```

- Se stai usando un'altra distribuzione Linux, è probabile che tu abbia Python 3 preinstallato. In caso contrario, usa il gestore di pacchetti della tua distribuzione. Ad esempio su Fedora:

```
sudo dnf install python3
```

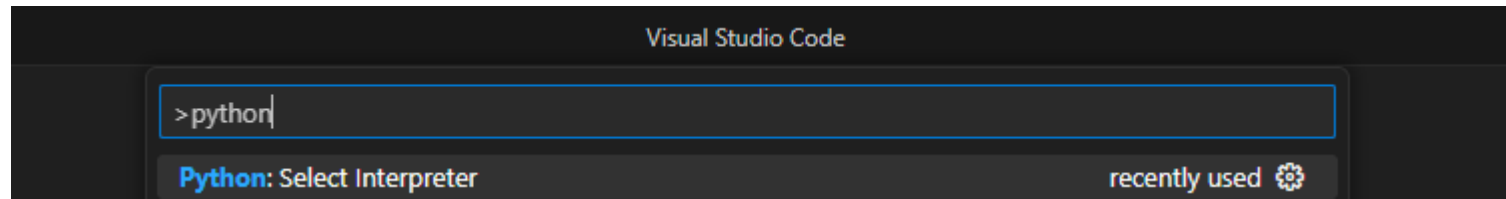
Visual Studio Code – Python Setup

- Se non lo hai già fatto, installa VS Code;
- Successivamente, è possibile installare l'estensione Python per VS Code, scaricabile da Visual Studio Marketplace. L'estensione si chiama **Python** ed è pubblicata da Microsoft.

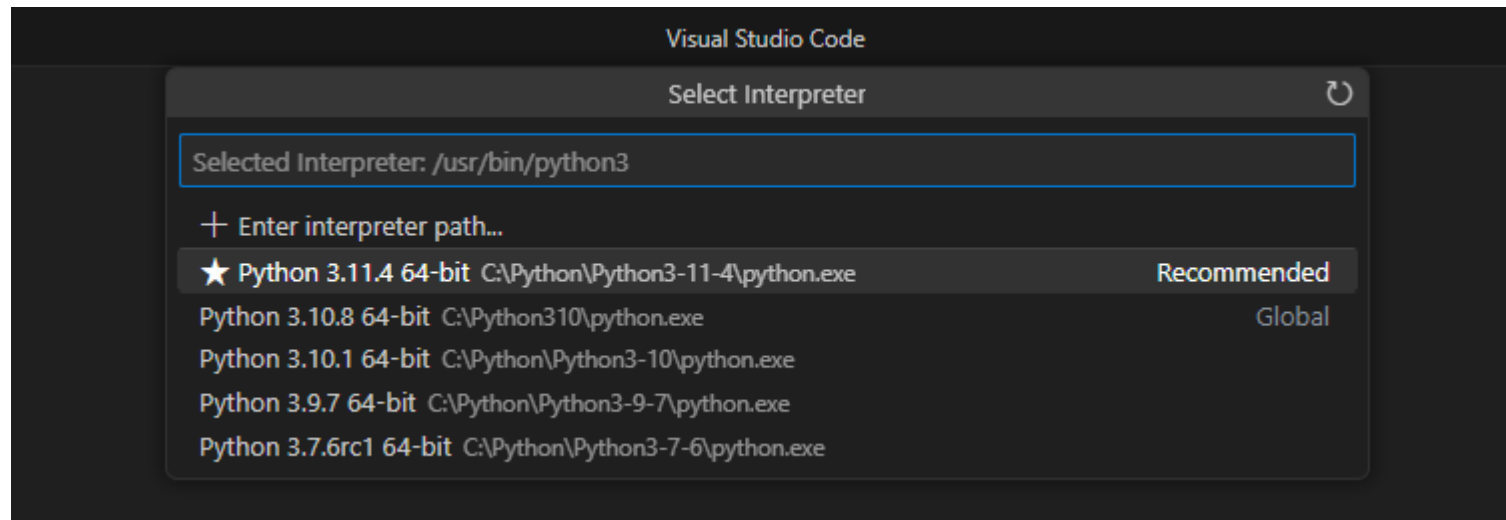


Selezionare l'Interprete Python

- Da VSCode, seleziona un interprete Python 3 aprendo il riquadro dei comandi (CTRL+MAIUSC+P o dal menu View/Command Palette) e digitando «Python: Select Interpreter»:

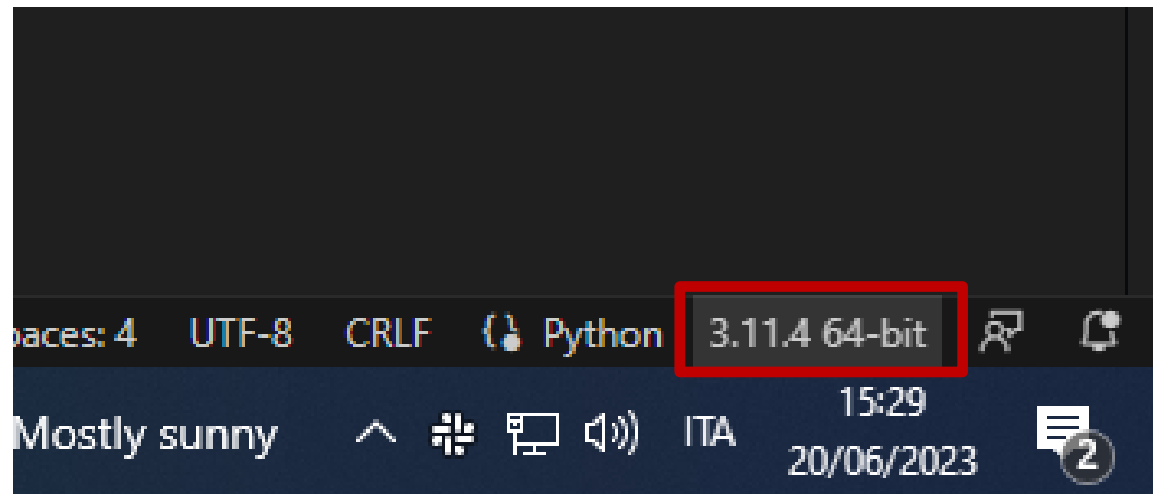


- Selezionare quindi l'interprete desiderato, se presente, o aggiungere il percorso dell'interprete desiderato:



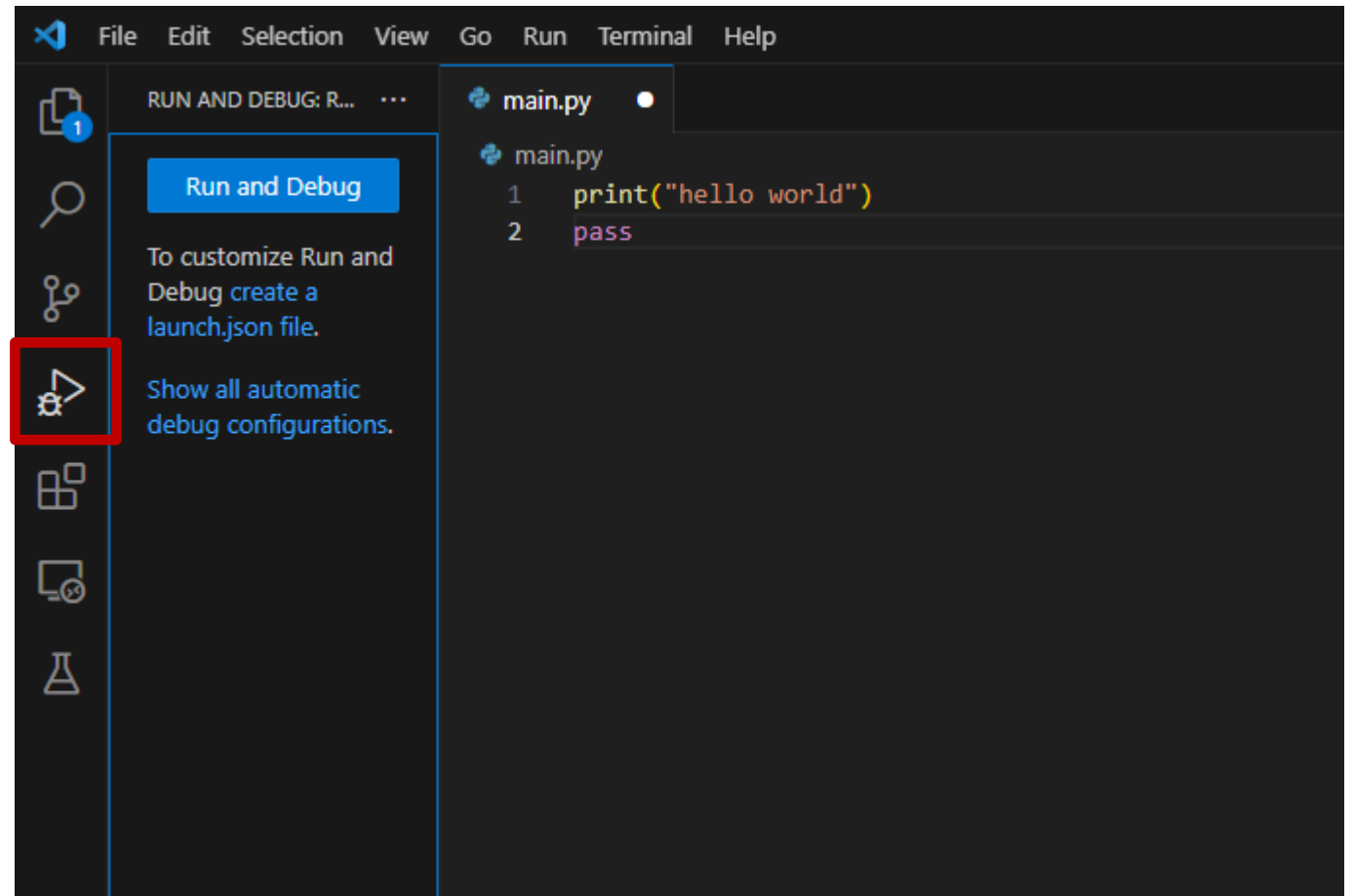
Selezionare l'Interprete Python

- Una volta configurato l'interprete, possiamo creare il primo progetto (cartella) ed aggiungere al suo interno il primo file python (*e.g. main.py*);
- Una volta aperto il progetto e selezionato un file python, la configurazione dell'interprete è disponibile anche nella barra in basso a sinistra di VSCode, da cui è possibile raggiungere rapidamente il menu analizzato nella slide precedente.



Debugger

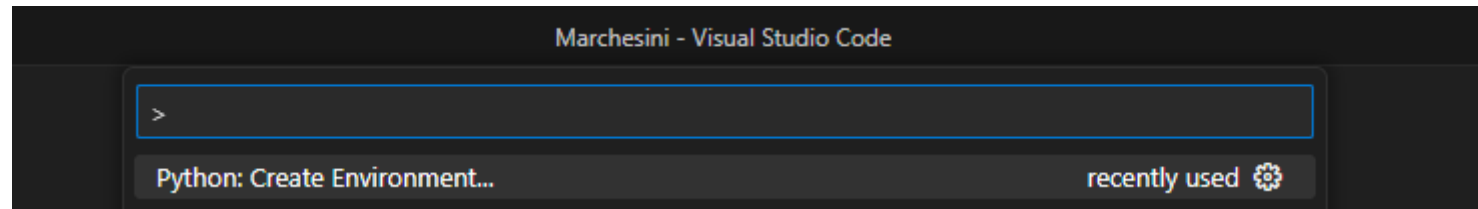
- A questo punto possiamo scrivere il primo programma di esempio e avviare il debugger (CTRL + SHIFT + D, o click sull'icona del debugger);
- È possibile creare una configurazione di debug personalizzata o usare quella di default «Python: File»;
- Per aggiungere un breakpoint si può fare click accanto all'istruzione su cui si desidera interrompere l'esecuzione.



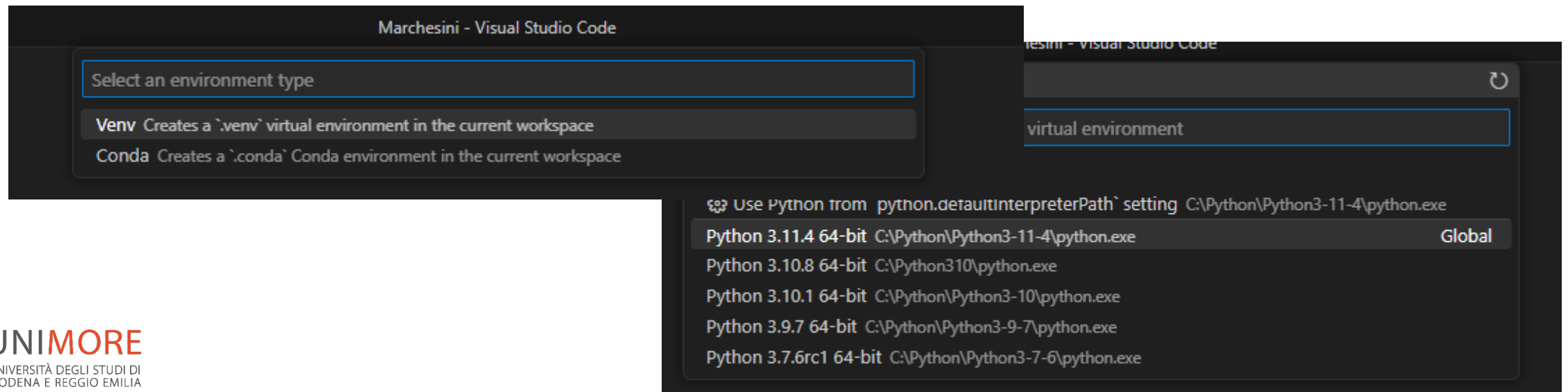
Ambienti Virtuali

- Una delle best practice quando si sviluppa codice Python consiste nell'usare *ambienti virtuali* specifico per ogni progetto;
- Una volta attivato l'ambiente, tutti i pacchetti installati sono isolati dagli altri ambienti, incluso l'ambiente dell'interprete globale, riducendo molte complicazioni che possono derivare da conflitti tra versioni dei pacchetti o pacchetti stessi;
- Per creare ambiente un *ambiente virtuale* ci sono due possibilità:
 - 🐍 *venv*: è uno strumento per la sola creazione e gestione di ambienti virtuali. Solitamente usato in combinazione con *pip* (gestore dei pacchetti python)
 - 🐍 *conda*: è allo stesso tempo un gestore di ambienti virtuali e pacchetti python.

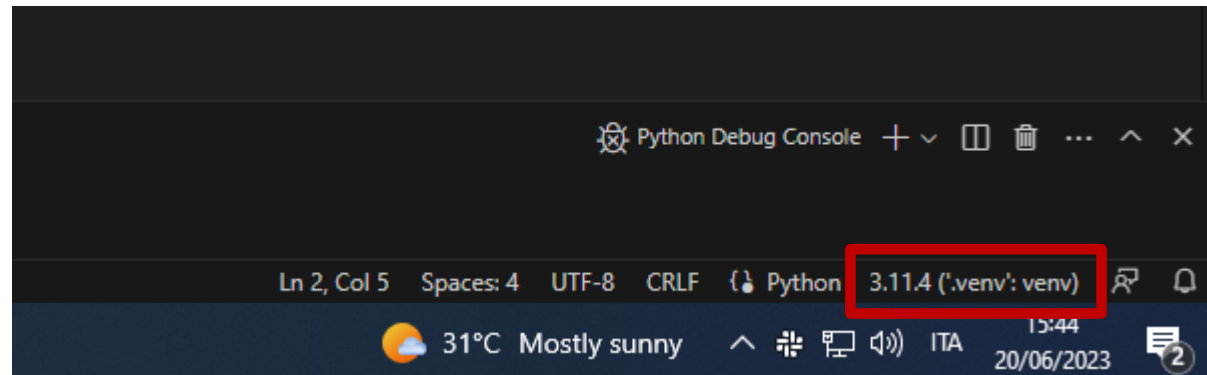
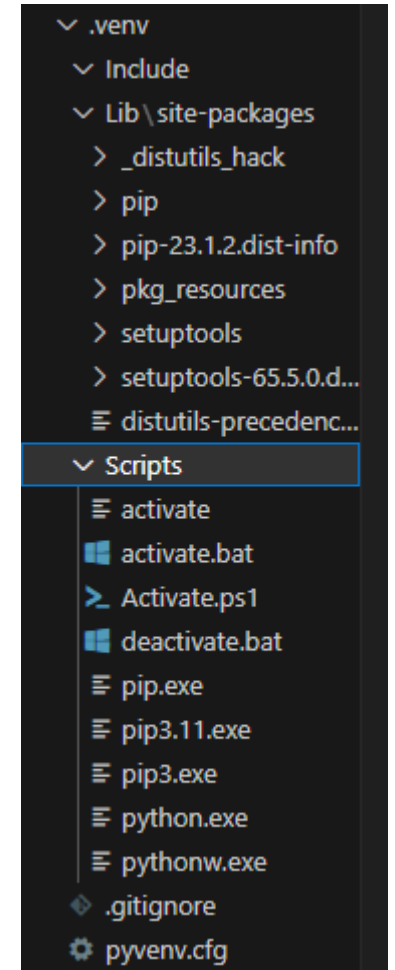
- È possibile creare un ambiente virtuale direttamente di VSCode aprendo il riquadro dei comandi (CTRL+MAIUSC+P o dal menu View/Command Palette) e digitando «Python: Create Environment»:



- Selezionare quindi “venv” e l’interprete di partenza da cui creare l’ambiente virtuale:



- La procedura dovrebbe creare una cartella `.venv` simile a quella riportata nella slide;
- Al suo interno trovate:
 - 🐍 La sottocartella `Lib\site-packages` contenente tutti i file relativi ai pacchetti installati;
 - 🐍 La sottocartella `Scripts` contenente la copia dell'interprete da cui è stato creato l'ambiente virtuale, il gestore dei pacchetti, e altri script come `activate.bat` per l'attivazione dell'ambiente virtuale;
- Se creato tramite VSCode, l'interprete virtuale viene automaticamente attivato:



- È possibile creare un *venv* da linea di comando (prompt dei comandi, o terminale di VSCode) usando il comando:

```
python -m venv /path/to/new/virtual/environment
```

- L'environment sarà creato a partire dalla versione «corrente» di python;
- Per attivare un ambiente virtuale da terminale è sufficiente eseguire lo script «activate.bat» contenuto all'interno della cartella *Script* del dell'ambiente virtuale;
- Il nome dell'ambiente virtuale attivo è visualizzato a sinistra della linea di comando. Se non specificato, verrà usato il python di default installato sul sistema:

```
(.venv) PS C:\Users\Federico\Desktop\Marchesini>
```

- Per uscire da un ambiente virtuale basta eseguire il comando *deactivate*.

Installazione Pacchetti

- Per installare un pacchetto con *pip* è possibile eseguire il comando:

```
pip install <nome-pacchetto>[==versione]
```

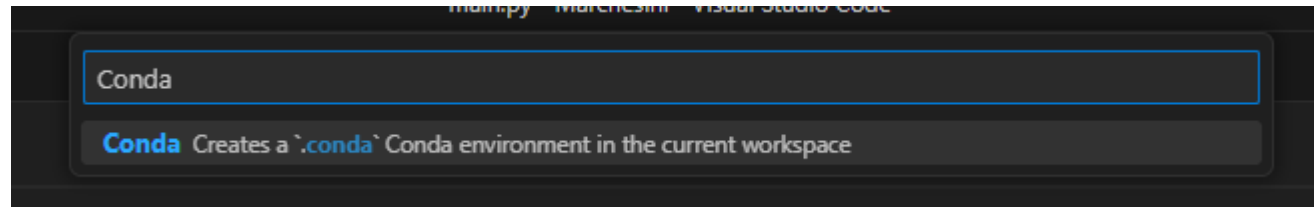
- Se volessimo ad esempio installare la libreria «OpenCV» potremmo lanciare il seguente comando, che installerà l'ultima versione disponibile su *pywheel*:

```
pip install opencv-python
```

- I pacchetti saranno installati all'interno dell'ambiente virtuale attivo o sul python di sistema nel caso in cui nessun ambiente sia attivo;
- Tramite il comando **pip list** è possibile elencare tutti i pacchetti attualmente installati e le relative versioni.

Conda

- Per prima cosa occorre installare *conda*. Ci sono due opzioni:
 - 🐍 Anaconda (<https://www.anaconda.com/download/>) - include *conda*, *python*, e numerosi altri pacchetti;
 - 🐍 Miniconda (<https://docs.conda.io/en/latest/miniconda.html>) - include solo *conda*, *python*, i pacchetti da cui questi dipendono e un piccolo numero di altri pacchetti utili, tra cui pip, zlib e pochi altri.
- Una volta installato è possibile creare un ambiente virtuale direttamente di VSCode aprendo il riquadro dei comandi (CTRL+MAIUSC+P o dal menu View/Command Palette) e digitando «Python: Create Environment»:



- Selezionare quindi “Conda” e la versione di Python da usare per creare l’ambiente virtuale.



Introduzione al Python per Programmatori C++

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Dal C++ al Python

- Probabilmente è più facile mostrare le caratteristiche salienti del Python con un programma di esempio;
- Vogliamo scrivere un semplice gioco di indovinelli, agli utenti viene chiesto di indovinare un numero intero segreto facendo al massimo 5 tentativi.

- Implementazione C++

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }

    return 0;
}
```

- Implementazione Python

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

Python vs C++

- Mettiamo fianco a fianco i codici C++ e Python per confrontarli;
- In primo luogo, la funzione principale non è necessaria in Python. Quando esegui uno script Python, verrà eseguito tutto ciò che non è definito in una funzione o in una classe.

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }
}
```

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

Python vs C++

- Inoltre, non è necessario dichiarare variabili in Python. In Python, le variabili sono solo etichette che puntano a istanze di oggetti in memoria e non sono associate staticamente a un tipo di dato specifico.

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }
}
```

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }
    return 0;
}
```

- I blocchi in Python non sono identificati dalle {}, ma da una combinazione di : e rientri;
- A differenza del C++, l'indentazione è obbligatoria in Python;

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }
    return 0;
}
```

- Puoi utilizzare spazi o tabulazioni per far rientrare il codice (ufficialmente quattro spazi):
- Tuttavia NON devi mescolare spazi e tabulazioni nel tuo codice. Ciò comporterà un errore di sintassi. Scegline uno e sii coerente.

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

- In Python, i commenti in linea sono identificati da un «#»

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    int num_of_guesses = 1;
    string user_guess, secret = "1234";

    // Read in user's guess
    cout << "Please enter password: ";
    cin >> user_guess;

    while (user_guess != secret && num_of_guesses < 5) {
        cout << "Incorrect.\n";
        cout << "Please enter password: ";
        cin >> user_guess;
        num_of_guesses++;
    }

    if (user_guess == secret) {
        cout << "Correct\n";
    } else {
        cout << "Incorrect. Game over.\n";
    }
    return 0;
}
```

```
num_of_guesses = 1
secret = "1234"

# Read in user's guess
user_guess = input("Please enter password: ")

while user_guess != secret and num_of_guesses < 5:
    print("Incorrect. ")
    user_guess = input("Please enter password: ")
    num_of_guesses += 1

if user_guess == secret:
    print("Correct")
else:
    print("Incorrect. Game over.")
```

Eseguire uno Script Python

- Rispetto a C++, Python è un linguaggio interpretato;
- Non occorre quindi compilare il codice e lanciare l'eseguibile risultante, ma è possibile eseguire lo script Python direttamente con l'interprete;
- In background, l'interprete convertirà automaticamente lo script in bytecode (un'istruzione simile al linguaggio assembly, ma che deve essere interpretata ed eseguita da una macchina virtuale o da un interprete);
- Supponendo che non vi siano errori di sintassi, l'interprete eseguirà quindi lo script. La compilazione avviene senza interruzioni in background, senza che tu te ne accorga;
- Per fare la prima prova puoi copiare lo script dell'indovinello e salvarlo nel file **guessing_game.py**;
- Puoi eseguire lo script dal prompt dei comandi con **python guessing_game.py** oppure eseguirlo tramite VSCode.

Eseguire Python Interattivamente

- Come già visto nelle prime slide, è possibile eseguire Python in maniera interattiva usando il terminale fornito da VSCode (Terminal/New Terminal o CTRL+SHIFT+ò) o lanciando il comando **python** dal prompt dei comandi;

```
(.venv) PS C:\Users\Federico\Desktop\Marchesini> python
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
>>> "Questa è una stringa!"
'Questa è una stringa!'
>>> x = 3 + 5 + 8
>>> i = 0
>>> while i < x:
...     print(i)
...     i += 1
...
0
1
2
```

Eseguire Python Interattivamente

```
3
4
5
6
7
8
9
10
11
12
13
14
15
>>>
```

- Se hai bisogno di uscire dal prompt di Python, digita `exit()` o `quit()`.

Tipi Built-in di Base

- In linea con la sua filosofia di mantenere le cose semplici, rispetto al C++ il Python offre una raccolta di tipi di dato di base molto più ridotta:
 - 🐍 Numeri: `int`, `float`, `complex`
 - 🐍 Booleani: `bool`
 - 🐍 Stringhe: `str`
- In realtà esiste anche il tipo `bytes`, ma probabilmente non lo userete molto spesso.

Tipi Built-in di Base: Numeri

- I numeri possono essere:
 - 🐍 **int** (numeri interi)
 - 🐍 **float** (numeri in virgola mobile) - questi sono in realtà equivalenti ai numeri in doppia precisione del C++ (**double**)
 - 🐍 **complex** (numeri complessi)
- Non esiste una distinzione tra **short**, **long**, **long long**, **unsigned**. Un intero è un **int** (la dimensione in memoria viene adattata automaticamente), così come un numero con la virgola è un **float**.
- Prova a digitare i seguenti comandi sul prompt interattivi e vedi cosa ottieni:

```
>>> type(42)
>>> type(3.412)
>>> type(3.2e-12)
>>> type(1+2j)
```

Tipi Built-in di Base: Booleani

- I valori booleani (**bool**) possono essere **True** o **False**. Si noti che questi sono in maiuscolo e non in minuscolo come in C++;
- Puoi provare ad eseguire:

```
>>> type(True)
>>> type(False)
```

Tipi Built-in di Base: Stringhe

- Le stringhe in Python (**str**) sono una sequenza di caratteri;
- In Python, le stringhe sono racchiuse tra 'apici', "virgolette", '''tripli apici''' o ""triple virgolette"";
- Puoi provare ad eseguire:

```
>>> 'Questa è una stringa.'  
>>> "Questa è una stringa."  
>>> '''Questa è una stringa.'''  
>>> """Questa è una stringa."""
```

- Tutti e quattro sono equivalenti. Quindi 'la mia stringa' è identica a "la mia stringa";
- Puoi anche scrivere stringhe multilinea con '''tripli apici''' o ""triple virgolette"". I *whitespace* vengono mantenuti:

```
>>> '''Never gonna give you up,  
... Never gonna let you down,  
... Never gonna run around and desert you.  
... '''  
'Never gonna give you up,\nNever gonna let you down,\nNever gonna run around and desert you.\n'  
>>>
```

A capo e ()

- Per mandare a capo uno statement posso usare “\” o “()” :

```
x = (3 + 4
    + 2)
y = 7 + 8 \
    + 2
```

- Gli spazi bianchi all'interno di una linea non hanno significato:

```
x=1+2
x = 1 + 2
x      =      1      +      2
```

- Le parentesi tonde “()” possono essere usate per raggruppare operazioni o effettuare chiamate a funzione:

```
2 * (3 + 4)

myfun()
```

Tutto è un Oggetto

- In Python tutto è un oggetto!
- Quindi `356`, `3.2`, `4+5j`, `"Gatto"`, `False` sono tutti oggetti;
- È possibile usare la funzione `isinstance()` per controllare se un oggetto (primo parametro) è un'istanza o sottoclasse di una determinata classe (secondo parametro). Si noti che in Python tutto deriva dalla classe `object`:

```
>>> isinstance(2020, object)
>>> isinstance(4.33, object)
>>> isinstance(5j, object)
>>> isinstance(True, object)
>>> isinstance("COVID-19", object)
```


Tutto è un Oggetto

- Pertanto, quelli che seguono sono i costruttori per il rispettivo tipo built-in; questi accettano parametri di input e creano un'istanza del tipo specificato:

```
>>> int()
>>> float()
>>> complex()
>>> bool()
>>> str()
>>> int(-9.789)
>>> float(9)
>>> float("nan")
>>> float("inf")
>>> float("-inf")
>>> complex(1,2)
>>> bool(1)
>>> bool("")
>>> str("Hello")
>>> str(10)
```

Parole Riservate

- Come in C++, in Python ci sono alcune parole chiave riservate;
- Per visualizzare quali sono, è possibile eseguire:

```
>>> import keyword  
>>> keyword.kwlist
```

- Si noti che i tipi di dato built-in (**int**, **str**, ecc.) NON sono parole chiave, ma classi. Quindi puoi tecnicamente usare questi come nomi di variabili (ma davvero non dovresti farlo!);
- Degna di nota è la parola chiave **None**. È sostanzialmente equivalente a **nullptr** in C++.
- Come ogni altra cosa, anche **None** è un oggetto. Più specificamente, è un'istanza della classe **NoneType**.

```
>>> type(None)  
<class 'NoneType'>
```

Funzione `print()`

- La funzione `print()` permette di visualizzare a video un qualsiasi oggetto Python per cui è definito il metodo `__str__()`:

```
x = 3 + 2
y = "ciao"

print(x)  # Visualizza 5 a video
print(y)  # Visualizza ciao a video
```

- Attenzione!** Nella versione 2.x di Python `print` era uno statement del linguaggio e non una funzione.



La Semantica delle Variabili in Python

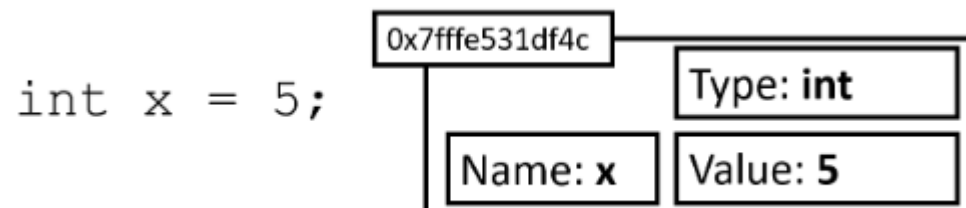
Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

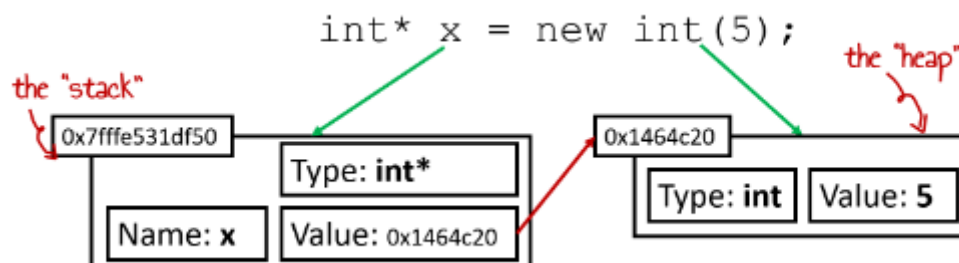
Costantino Grana <costantino.grana@unimore.it>

C++ vs Python

- In C++, una variabile è uno spazio riservato in memoria e ha un tipo, un nome e un valore ad essa assegnati:

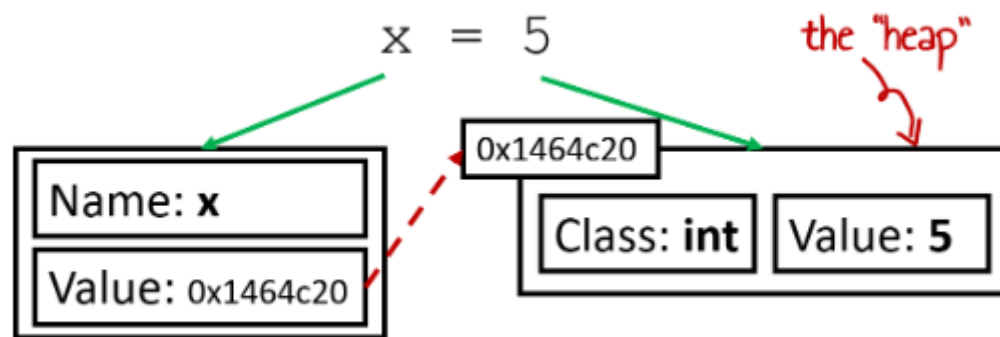


- Le variabili in Python sono più simili ai puntatori (ma senza le complicazioni dei puntatori C++) in quando puntano ad un oggetto nell'heap (allocazione dinamica della memoria);
- Ricordiamo che in C++ un puntatore è una variabile in cui il valore è l'indirizzo di ciò a cui punta. Una variabile puntatore è anche vincolata staticamente ad essere di un certo tipo.



C++ vs Python

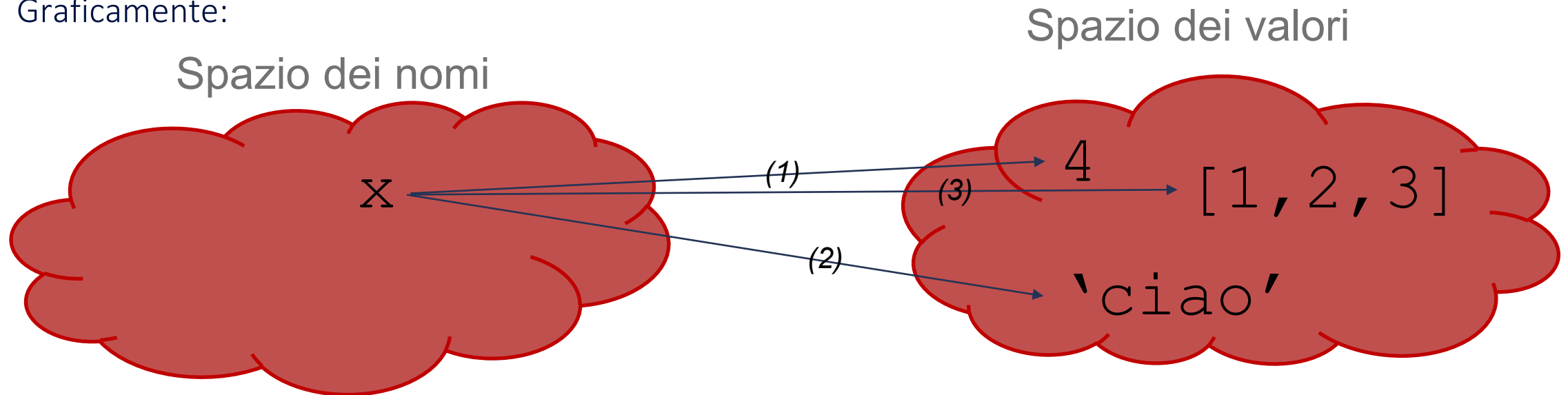
- In Python, una variabile è semplicemente un'etichetta (o un soprannome) che punta sempre a un oggetto nell'heap della memoria (ricorda, tutto è un oggetto in Python, incluso **None**);
- A differenza del C++, una variabile non ha nemmeno un tipo. Punta semplicemente a un oggetto (che ha un tipo!):



- Inoltre, a differenza del C++, non devi preoccuparti di liberare la memoria in Python. Le **delete** saranno fatte automaticamente in background, e tu potrai concentrarti sulle cose che contano davvero!

Spazio dei Nomi e dei Valori

- Quando si parla di semantica delle variabili possiamo distinguere due concetti:
 - 🐍 Spazio dei *Nomi*: contiene tutte le etichette esistenti nel *namespace* corrente che puntano ad oggetti allocati (dinamicamente);
 - 🐍 Spazio dei *Valori*: contiene gli oggetti allocati (dinamicamente).
- Graficamente:



```
(1) x = 4           # x è un intero
(2) x = 'ciao'      # ora x è una stringa
(3) x = [1, 2, 3]   # ora x è una lista
```

Nomi Differenti per lo Stesso Oggetto

- La tipizzazione dinamica usata dal Python è ciò che lo rende estremamente facile da leggere e veloce da scrivere;
- Attenzione però, se due “puntatori a variabile” puntano allo stesso oggetto, la modifica di uno cambierà anche l'altro:

```
x = [1, 2, 3]
y = x

print(x)  # Visualizza x, ovvero [1, 2, 3]
print(y)  # Visualizza y, ovvero [1, 2, 3]

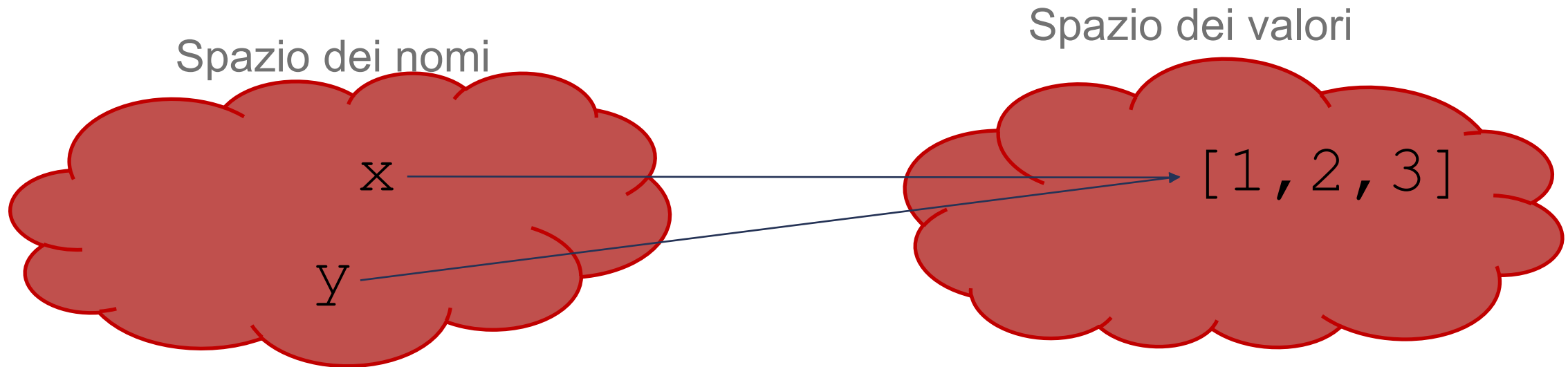
x.append(4)  # Aggiungo l'elemento 4 alla lista x

print(x)  # Visualizza x, ovvero [1, 2, 3, 4]
print(y)  # Visualizza y, ovvero [1, 2, 3, 4]
```


Nomi Differenti per lo Stesso Oggetto

- Infatti, questo è quello che accade in Python quando eseguiamo il codice:

```
x = [1, 2, 3]  
y = x
```



Oggetti Mutabili e Immutabili

- Questa rappresentazione potrebbe complicare le operazioni aritmetiche, quindi Python fa distinzione tra oggetti **mutabili** ed **immutabili**;
- *Numeri, stringhe e tutti gli oggetti semplici sono **immutabili**, ovvero se ne può cambiare il valore solamente cambiando l'oggetto a cui questi puntano!*

```
x = 10
y = 10
x = x + 5

print("x =", x)
print("y =", y)
```

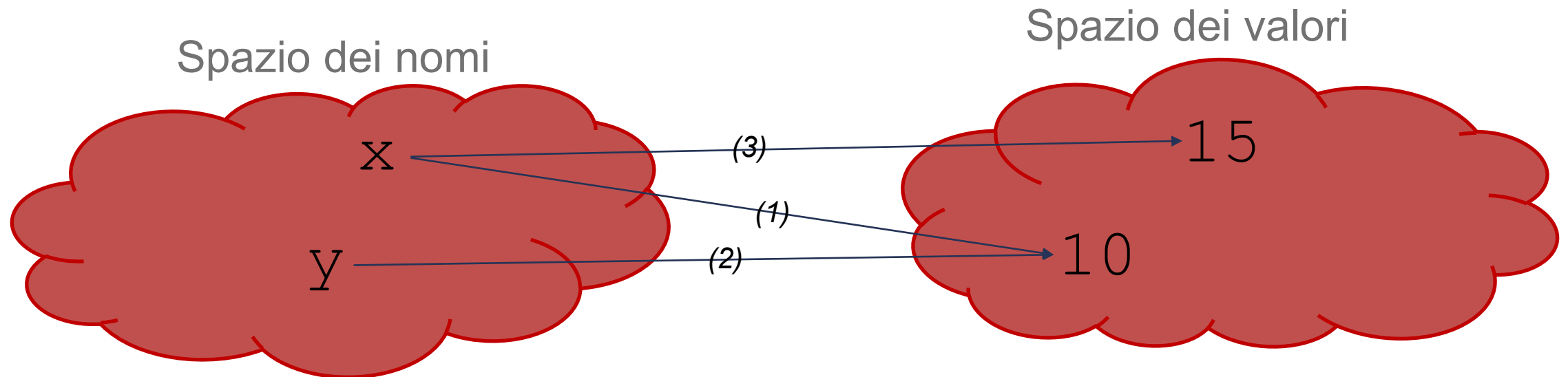
- Cosa ci aspettiamo venga visualizzato dalle due `print()`?

Oggetti Mutabili e Immutabili

- Risposta:

$x = 15$

$y = 10$



```
(1) x = 10
(2) y = 10
(3) x = x + 5
```

Oggetti Mutabili e Immutabili

- Gli oggetti come liste, tuple e dizionari sono invece **mutabili**, ovvero possono essere modificati;
- Come nelle slide precedenti, una eventuale modifica di un oggetto mutabile avrà quindi «ripercussioni» su tutte le etichette che puntavano all'oggetto;
- E bene ripetere che le etichette non hanno alcun tipo di informazione ad esse connessa, ma il Python NON è un linguaggio type-free;
- Tutte le informazioni, compreso il tipo, sono connesse agli oggetti a cui le etichette puntano;
- Per visualizzare il tipo associato ad un oggetto possiamo invocare la funzione **type()**, passandole un *literal* o un'etichetta che punta all'oggetto in analisi.

Oggetti in Python

- Come già accennato, tutto è un oggetto in Python!
- Naturalmente, è possibile definire nuove classi. In Python 3, tutte le classi ereditano dalla classe **object** di default;

```
class Person:  
    pass  
  
person = Person()
```

- Il codice sopra crea un'istanza di **Person** (chiamando il costruttore) e assegna l'istanza alla variabile **person**;
- **pass**: In Python, **pass** è un'istruzione nulla. L'interprete NON ignora l'istruzione **pass**, ma semplicemente non accade nulla quando l'istruzione viene eseguita. L'istruzione **pass** è utile, ad esempio, quando si demanda l'implementazione di una funzione o di una classe «al futuro».

Oggetti in Python

- Il codice della slide precedente è equivalente al seguente codice C++:

```
// C++ equivalent
class Person {};

int main() {
    Person* person = new Person();
    delete person;
    return 0;
}
```

- Si noti che Python crea sempre istanze di oggetti nell'area *heap* della memoria. In questo Python si differenzia dal C++ che con l'istruzione **Person person;** o **Person person = Person()** allocherebbe la classe staticamente;
- A costo di sembrare prolisso, ricorda che in Python una variabile non ha un tipo. È ciò a cui punta la variabile che ha un tipo! Questa è la chiave per capire come "pensare" alle variabili in Python rispetto a C++.

Operatori Matematici

- Operatori matematici come `+`, `-`, `*`, `/` e `%` funzionano come «previsto» in Python;
- Una nota particolare va fatta sull'operatore `/` che si comporta come previsto matematicamente (restituendo un **float**);
- Se si vuole eseguire la divisione intera occorre usare l'operatore `//`;
- Come in matematica, la precedenza degli operatori aritmetici può esser modificata con le parentesi tonde `()`:

Operator	Name	Description
<code>a + b</code>	Addition	Sum of a and b
<code>a - b</code>	Subtraction	Difference of a and b
<code>a * b</code>	Multiplication	Product of a and b
<code>a / b</code>	True division	Quotient of a and b
<code>a // b</code>	Floor division	Quotient of a and b, removing fractional parts
<code>a % b</code>	Modulus	Remainder after division of a by b
<code>a ** b</code>	Exponentiation	a raised to the power of b
<code>-a</code>	Negation	The negative of a
<code>+a</code>	Unary plus	a unchanged (rarely used)

Operatori Matematici

- Attenzione, il comportamento dell'operatore `/` è diverso in Python 2.x;
- Infatti, l'operatore esegue la divisione intera come il `//` in Python 3;
- Se per qualche motivo state lavorando su codice legacy e siete costretti ad usare Python 2.x, potete ovviare a questo problema importando la divisione del Python 3 dal modulo `__future__`:

```
from __future__ import division, print_function
```

- Lo stesso vale per la funzione `print()` che, come detto, in Python 2.x era uno *statement* del linguaggio.

Operatori di Confronto

- Gli operatori di confronto <, >, <=, >=, ==, != sono gli stessi del C++;
- È inoltre possibile concatenare le espressioni insieme! 💣

```
>>> x = 5
>>> 3 < x < 6
True
>>> 5 < x <= 8
False
```

Operation	Description
a == b	a equal to b
a != b	a not equal to b
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Operatori Logici

- Gli operatori logici `!`, `&&`, `||` del C++ si traducono nelle parole inglesi **not**, **and**, **or**

```
>>> not True
>>> True and False
>>> True or False
>>> x = 7
>>> y = 2
>>> x > 3 and y < 1
>>> not x >= 5 and x < 3      # make sure you understand the
>>> not (x >= 5 and x < 3)    # difference between these two
>>> not 5 + y > 3 * x or 4 == x + 3
```

Operatori di Identità e Appartenenza

- In Python esistono anche i seguenti operatori di «identità» e «appartenenza»:

Operator	Description
<code>a is b</code>	True if <code>a</code> and <code>b</code> are identical objects
<code>a is not b</code>	True if <code>a</code> and <code>b</code> are not identical objects
<code>a in b</code>	True if <code>a</code> is a member of <code>b</code>
<code>a not in b</code>	True if <code>a</code> is not a member of <code>b</code>

- La differenza fondamentale tra `is` e `==` è che il primo confronta gli indirizzi e ritorna **True** solo se sono uguali, il secondo, invece, confronta i valori;

Operatori di Identità e Appartenenza

- Puoi provare:

```
>>> a = 1000
>>> b = 1e3
>>> a == b
True
>>> a is b
False
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
```

- O anche:

```
>>> a = 1000
>>> b = 10**3
>>> a == b
True
>>> a is b
False
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
```

Operatori di Identità e Appartenenza

- Attenzione però:

```
>>> a = 10
>>> b = 10**1
>>> a == b
True
>>> a is b
True
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
```

- Questo accade perché gli oggetti rappresentanti numeri «piccoli» vengono mantenuti sempre in memoria;
- Quanto piccoli? Dipende dall'implementazione, ma non dovrebbe mai interessarci. Sulla nostra macchina è 256.
- È sulla vostra? E «in negativo»? Provate voi!

Comando di Assegnamento

- In Python, puoi assegnare lo stesso oggetto a più variabili contemporaneamente:

```
>>> x = y = 558
>>> print(x)
>>> print(y)
```

- Puoi anche eseguire più assegnamenti contemporaneamente:

```
>>> x, y, z = 5, 5, 8
>>> print(x)
>>> print(y)
>>> print(z)
```

- Ciò consente naturalmente di scambiare facilmente i valori delle variabili, senza richiedere (nel codice) una variabile intermedia:

```
>>> x = 5
>>> y = 8
>>> x, y = y, x
>>> print(x)
>>> print(y)
```

Comandi di Assegnamento «Aumentati»

- È possibile utilizzare gli operatori di assegnazione "scorciatoia" proprio come in C++: +=, -=, *=, /=, %=, //=, **=
- $x += y$ equivale a $x = x + y$.
- Purtroppo NON È POSSIBILE fare $x++$ e $--y$ in Python. Questo è legato ad una delle filosofie alla base di Python: *dovrebbe esserci preferibilmente un solo modo (ovvio) di fare qualcosa*;
- A differenza dell'assegnamento, i comandi di assegnamento aumentati non possono essere concatenati, ma non dovresti mai farlo nemmeno in C++.



Controllo del Flusso di Esecuzione

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

if – else – elif

- Oltre alle istruzioni **if** e **if-else**, Python offre anche il comando **if-elif-else**. **elif** è solo l'abbreviazione di "else if", e si allinea anche meglio con **else**!

```
if user_guess == 42:  
    print("Correct!")  
elif user_guess < 42:  
    print("Too low")  
else:  
    print("Too high")
```

while

- I cicli **while** funzionano più o meno come in C++, quindi basta un esempio (quello iniziale) per vedere la sintassi:

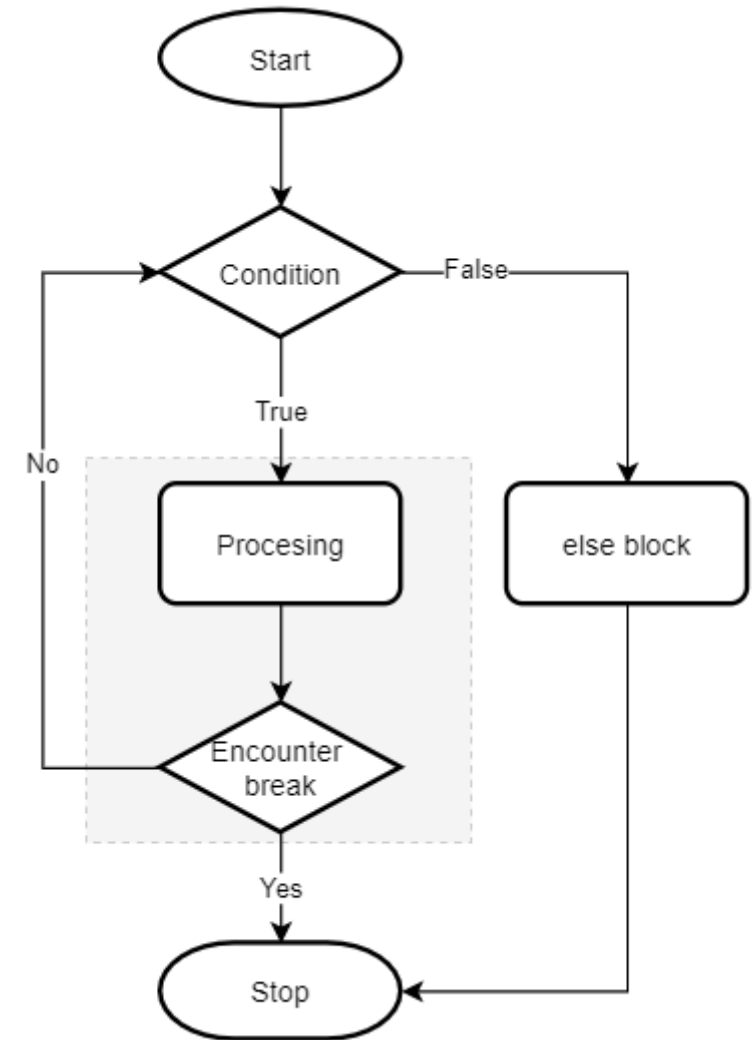
```
while user_guess != secret and num_of_guesses < 5:  
    print("Incorrect.")  
    user_guess = input("Please enter password: ")  
    num_of_guesses += 1
```

- **break** e **continue** possono essere usati come in C++;
- I cicli **do-while** non esistono in Python.

while -- else

```
while condizione:  
    istruzioni  
else:  
    istruzioni
```

- In questa sintassi, la condizione viene verificata all'inizio di ogni iterazione. Il blocco di codice all'interno dell'istruzione **while** verrà eseguito finché la condizione è **True**;
- Quando la condizione diventa **False** e il ciclo viene eseguito normalmente, verrà eseguita la clausola **else**. Tuttavia, se il ciclo viene terminato prematuramente da un'istruzione **break** o **return**, la clausola **else** non verrà eseguita affatto;
- Il diagramma di flusso a destra illustra la clausola **while...else**.



while – else (esempio)

- Un possibile esempio di uso della clausola **else** dopo il **while** è l'algoritmo di ricerca di un numero in una lista:
- Ancora non abbiamo introdotto le liste, ma la vostra fantasia vi permetterà di comprendere il codice che segue:

```
numbers = [1, 1, 3, 5, 8, 13]
# to_find = 21
to_find = 13
i = 0
while i < len(numbers):
    if numbers[i] == to_find:
        print("Trovato!")
        break
    i += 1
else:
    print("Non trovato!")
```

Esercizi

- Ora è il tuo turno!
- Per prendere valori in input da tastiera si può usare la funzione `input()`;
- La funzione accetta una stringa `s` come parametro e, dopo aver visualizzato `s` in output, legge da standard input (di default la tastiera) fino a quando non verrà letto il carattere a capo;
- La funzione ritorna una stringa con la linea letta dal file, senza l'a capo.

Esercizi

- Ora è il tuo turno!
- Per prendere valori in input da tastiera si può usare la funzione `input()`;
- La funzione accetta una stringa `s` come parametro e, dopo aver visualizzato `s` in output, legge da standard input (di default la tastiera) fino a quando non verrà letto il carattere a capo;
- La funzione ritorna una stringa con la linea letta dal file, senza l'a capo.

- Se serve, posso usare il costruttore di uno qualsiasi dei tipi built-in per convertire la stringa ad esempio in `int`, `float`, ecc.

for

- I cicli **for** in Python sono diversi dai tradizionali cicli `for` in C++ e sono in realtà più facili da capire;
- In effetti, i cicli **for** in Python sono simili ai cicli *range-based* in C++;
- In Python, i cicli **for** iterano su di un *iterable*, ad esempio su una sequenza come una lista (ma di questi dettagli parleremo più avanti);

```
for i in range(10):  
    print(i, end=" ")  
# Stamperà "0, 1, 2, 3, 4, 5, 6, 7, 8, 9,"  
print()
```

```
for i in range(5, 10):  
    print(i, end=" ")  
# Stamperà "5, 6, 7, 8, 9,"  
print()
```

```
for i in range(5, 10, 2):  
    print(i, end=" ")  
# Stamperà "5, 7, 9,"  
print()
```

for

- La funzione **range(start, stop, step)** costruisce un oggetto di tipo **range** (iterabile) rappresentante i valori da **start** (incluso) a **stop** (escluso) con passo **step**. I parametri **start** e **step** sono opzionali, se omessi, verranno considerati rispettivamente 0 e 1. L'oggetto **range** è *immutabile*;
- Anche il ciclo **for** ammette la clausola finale **else** con la stessa logica vista per il ciclo **while**;
- **end** è un parametro opzionale della funzione **print()**. Di default vale `\n`, ma può essere modificato. In sostanza rappresenta cosa deve essere stampato a video dopo aver completato la stampa del/degli oggetto/i specificati come primo parametro.

Esercizi

- Ora è il tuo turno!
- Rifai gli esercizi precedentemente assegnati usando questa volta il ciclo **for**.



Le Funzioni

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Funzioni

- Le funzioni in Python sono le stesse del C++, tranne per il fatto che non è necessaria la tipizzazione statica (né per il parametro né per il tipo restituito);

```
def compute_something(x, y, z):  
    a = x + 2*y  
    a = z*a + 3*x  
    return a  
  
result = compute_something(5, 6, 3)  
print(result)
```

- **def** è una parola chiave per indicare una definizione di funzione;
- Se non includi l'istruzione **return**, la funzione restituirà automaticamente **None**;
- Sebbene tu possa restituire solo un oggetto, quell'oggetto può essere una sequenza (ad esempio tupla), quindi tecnicamente puoi restituire più di un oggetto, ma questo lo vedremo più avanti.

Funzioni

- Come già specificato nell'esempio introduttivo, la funzione principale non esiste in Python. Quando esegui uno script Python, verrà eseguito tutto ciò che non è definito in una funzione o in una classe;
- Se nello script vengono definite funzioni occorre invocarle perché il codice al loro interno venga eseguito;
- A volte può essere utile specificare di che tipo dovrebbero essere i parametri e/o il valore di ritorno di una funzione. Questa procedura prende il nome di *type hints*:

```
def greeting(name: str) -> str:  
    return 'Hello ' + name
```

- Nell'esempio sopra stiamo dicendo che la funzione **`greeting`** si aspetta un argomento di tipo **`str`** e ritorna un oggetto di tipo **`str`**;
- Il *type hinting* non viene usato dall'interprete, ma può essere usato dell'IDE per fornire suggerimenti su quali sono i metodi a disposizione per quel determinato oggetto;
- Si può usare anche al di fuori delle funzioni.

Funzioni

- Come in C++, l'associazione argomento parametro è posizionale per i parametri obbligatori a meno di specificare il nome dei parametri;
- Python consente di fornire anche argomenti predefiniti, in modo che il povero utente non debba fornire esplicitamente un valore per ogni singolo parametro;

```
def my_function(a, b, c=2, d="great", e=5, f="john", g="python", h=-1, i=0, j=[]):  
    print(a, b, c, d, e, f, g, h, i, j)  
    return None
```

- Un utente che vuole invocare la funzione precedente deve fornire i primi due argomenti posizionali. Quindi può facoltativamente fornire eventuali argomenti aggiuntivi. Tutto ciò che non viene fornito utilizzerà gli argomenti predefiniti definiti dalla funzione;

```
>>> x = my_function("first", "second")  
first second 2 great 5 john python -1 0 []  
  
>>> x = my_function("first", "second", "arg for c", "arg for d")  
first second arg for c arg for d 5 john python -1 0 []
```

Funzioni

- Il chiamante può anche utilizzare le parole chiave per fornire i parametri facoltativi senza preoccuparsi del posizionamento del parametro:

```
x = my_function("first", "second", i=2, f="josiah")
```

- Si noti che il chiamante deve fornire gli argomenti posizionali (obbligatori) prima di fornire qualsiasi argomento con nome. È ovvio il perché!

```
x = my_function(d="no good", "first", "second") # Not allowed. Python will tell you so.
```

- Analogamente, non è possibile definire un parametro che sia un argomento posizionale dopo averne dichiarato uno con nome. Quindi quanto segue è illegale:

```
def my_function(a="first", b, c, d="default d"): # Python says no way!  
    return None
```

Funzioni

- Il passaggio dei parametri in Python avviene per assegnamento (*by assignment*);
- Essendo che l'assegnamento consiste nella creazione di un nome (etichetta nello spazio dei nomi) che punta ad un oggetto (lo stesso oggetto, nello spazio dei valori) questo potrebbe creare confusione;
- Tenendo però in considerazione l'esistenza di oggetti *mutabili* ed *immutabili* dovrebbe essere intuitivo il risultato del seguente codice:

```
def foo(a):  
    a = a * 2  
  
a = 6  
foo(a)  
print(a)
```

- O anche:

```
def foo(a):  
    a.append(4)  
  
a = [1, 2, 3]  
foo(a)  
print(a)
```

Funzioni

- Attenzione però, la cosa può diventare molto subdola:

```
def foo(a = [1, 2 ,3]):  
    a.append(4)  
    print(a)  
  
foo()  
foo()
```

- Che cosa verrà visualizzato?

Funzioni

- Attenzione però, la cosa può diventare molto subdola:

```
def foo(a = [1, 2 ,3]):  
    a.append(4)  
    print(a)  
  
foo()  
foo()
```

- Che cosa verrà visualizzato?

```
[1, 2, 3, 4]  
[1, 2, 3, 4, 4]
```

- Perché?

Funzioni

- Quando facciamo questo,

```
def foo(a = [1, 2, 3]):  
    a.append(4)  
    print(a)
```

- definiamo la funzione foo e diciamo che se non viene fornito il parametro a, questo punterà all'oggetto *mutabile* [1, 2, 3]. Alla prima chiamata la lista viene modificata e la seconda chiamata si trova come valore di default proprio lo stesso oggetto, che però questa volta ha un 4 in più alla fine.
- Quindi?
- Take home message: **mai utilizzare oggetti mutable come parametri di default se il parametro deve essere modificato!**

Funzioni – Numero Arbitrario di Argomenti

- Se non si conosce in anticipo quanti argomenti verranno forniti, Python consente di utilizzare un asterisco (*) per indicare che il parametro consente al chiamante di fornire un numero variabile di argomenti. Python "impacchetta" gli argomenti in una tupla e li assegnerà al parametro.
- Nota: l'asterisco non ha nulla a che fare con i puntatori! È un operatore di impacchettamento/de-impacchettamento a seconda del contesto in cui viene utilizzato.

```
def enroll(*courses):  
    print(type(courses))  
    for course in courses:  
        print('Joined course: ', course)  
  
enroll("Probabilistic Inference", "Introduction to Machine Learning",  
       "Computer Vision", "Graphics")
```

Funzioni – Numero Arbitrario di Argomenti con Nome

- Una funzione Python può anche accettare un numero arbitrario di *argomenti con nome*;
- Per indicarlo occorre utilizzare doppi asterischi (**);
- Questa volta, ti ritroverai con un dizionario contenente le coppie (parola chiave, valore) fornite dall'utente. È quindi possibile elaborare queste coppie secondo necessità. Utile ad esempio quando fornisci al chiamante molte opzioni personalizzabili!

```
def customise_page(**kwargs):  
    for key, value in kwargs.items():  
        if key == "background":  
            set_background(value)  
        elif key == "width":  
            set_width(value)  
        elif key == "avatar":  
            set_avatar(value)  
        else:  
            print("Unknown keyword {key}.")  
            return  
  
customise_page(background="red", width=500, avatar="selfie.jpg")
```

Esercizi

- Ora è il tuo turno!
- Rifai gli esercizi precedenti, impacchettando la logica in funzioni.



Tipi di Dato Strutturati

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Tipi di Dato Strutturati

- Il Python fornisce i seguenti tipi di dato strutturati:

Type Name	Example	Description
list	[1, 2, 3]	Ordered collection
tuple	(1, 2, 3)	Ordered collection (not mutable)
dict	{'a':1, 'b':2, 'c':3}	Unordered (key,value) mapping
set	{1, 2, 3}	Unordered collection of unique values



Liste (list)

Liste (**list**)

- Python NON offre array di dimensioni fisse come il C++;
- Tuttavia, offre qualcosa di probabilmente migliore: le liste. Una lista è un oggetto che può essere modificato e ridimensionato al bisogno;

```
>>> students = ["Abraham", "Bella", "Connor", "Da-ming", "Enya"]
```

- È possibile mescolare diversi tipi di oggetti nella stessa lista (se e quando questo dovrebbe essere fatto è una domanda diversa!);

```
>>> mixed_buffet = [1, "Two", 3.03, 4j]
```

- È possibile avere liste nidificate, cioè una lista all'interno di un'altra lista (all'interno di un'altra lista, ecc.);

```
>>> list_in_list = [1, 2, [3, 4, [5, 6, 7], 8], 9, [10, 11]]
```

- È anche possibile usare il costruttore **list()** per costruire un nuovo oggetto lista (ricorda: tutto è un oggetto 😊!)

```
>>> x = list("abc")    # converts the string "abc" into a list ["a", "b", "c"]
```

Liste (**list**) – Lunghezza & Indexing

- Per conoscere la lunghezza di una lista è possibile utilizzare la funzione `len()`:

```
>>> numbers = [2, 3, 5, 7, 11]
>>> print(len(numbers))
```

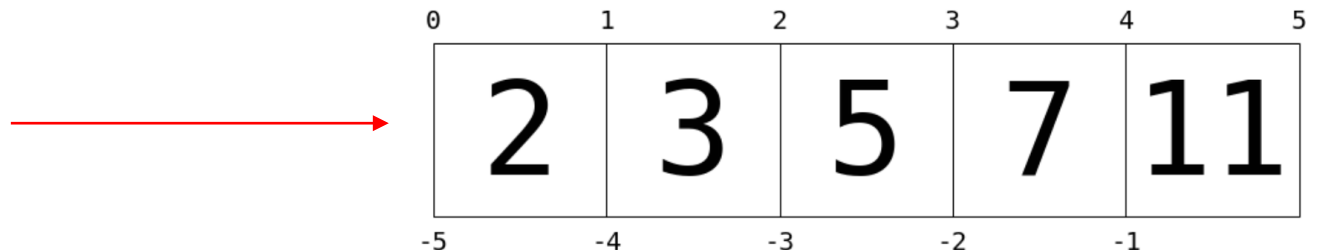
- È possibile accedere a un elemento di una lista proprio come faresti con un array in C++:

```
>>> numbers = [2, 3, 5, 7, 11]
>>> print(numbers[0])      # First element: 2
>>> print(numbers[1])      # Second element: 3
```

- Python ti consente anche di accedere agli elementi di una lista in ordine inverso, usando indici negativi:

```
>>> print(numbers[-1])     # First element from the end: 11
>>> print(numbers[-2])     # Second element from the end: 7
```

Schema di indicizzazione
per la lista [2, 3, 5, 7, 11]



Liste (**list**) – Lunghezza & Indexing

- Una lista dentro una lista? Nessun problema, anche in questo caso possiamo utilizzare l'indexing:

```
>>> x = [1, 2, [3, 4, 5], 6]
>>> print(x[2])           # [3, 4, 5]
>>> print(x[2][1])        # 4
```

Liste (**list**) – Slicing

- Lo *slicing* permette l'accesso ad elementi multipli della lista:

```
>>> numbers = [2, 3, 5, 7, 11]
>>> print(numbers[0:3])    # Stampa la lista [2, 3, 5]
>>> print(numbers[:3])     # Stampa la lista [2, 3, 5]
>>> print(numbers[:])      # Stampa la lista [2, 3, 5, 7, 11]
>>> print(numbers[0:3:2])  # Stampa la lista [2, 5]
```

- In generale, lo schema d'uso è il seguente:

`list_name[<start(included)>:<stop(excluded)>:<step>]`

- Dove **start** è l'indice dell'elemento da cui partire (**incluso**), **stop** è l'indice dell'elemento a cui fermarsi (**escluso**) e **step** è la granularità (il passo) con cui «estrarre» gli elementi;
- Tutti e 3 i parametri sono facoltativi. Il loro valore di default, se omessi, è rispettivamente **0**, **len(list_name)**, **1**.

Liste (**list**) – Indexing & Slicing

- Sia l'*indexing* che lo *slicing* possono anche essere usati per settare i valori delle liste:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> a[2:6:2] = [24, 24]    # modifica a in [1, 2, 24, 4, 24, 6, 7]
>>> a[-1] = 17            # modifica a in [1, 2, 24, 4, 24, 6, 17]
```

- Lo *slicing* può essere effettuato con valori negativi, con la stessa logica dell'*indexing*, ma occorre prestare attenzione, non è sempre scontato l'effetto;
- Nello *slicing*, a differenza dell'*indexing*, quando start/stop precedono/seguono indici effettivi di elementi della lista vengono automaticamente adattati e non generano eccezioni.

Liste (**list**) – Modifica

- Puoi aggiungere nuovi elementi alla fine della lista usando il suo metodo **append()** (la lista è un oggetto!);

```
>>> numbers = []           # this creates a new empty list
>>> numbers.append(6)
>>> print(numbers)
>>> numbers.append(2)
>>> print(numbers)
```

- Puoi anche aggiungere più elementi ad una lista con il metodo **extend()**;

```
>>> numbers = [1, 2, 3]
>>> numbers.extend([4, 5, 6])
>>> print(numbers)
```

Liste (**list**) – Modifica

- È possibile eliminare elementi da una lista utilizzando l'operatore **del**;

```
>>> numbers = [1, 2, 3]
>>> print(numbers)
>>> del numbers[1]
>>> print(numbers)
```

- La [documentazione ufficiale](#) fornisce molti altri metodi per l'oggetto lista, ad esempio:

 `.insert()`

 `.pop()`

 `.remove()`

 `.reverse()`

 `.index()`

 `.sort()`

Liste (**list**) – Operatori

- Gli operatori + e * sono stati ridefiniti per le liste;
- È possibile concatenare due liste "aggiungendo" una all'altra:

```
>>> print([1, 2] + [3, 4, 5])
```

- Puoi anche replicare gli elementi in una lista "moltiplicandola" per uno scalare:

```
>>> print([1, 2] * 3)
```

- L'operatore di appartenenza (**in**) è utile anche per controllare se un particolare oggetto è in un elenco. Questo restituisce un **bool** (**True** o **False**).

```
>>> print(3 in [1,2,3,4])
>>> print(3 not in [1,2,3,4])
>>> print("snake" in ["I", "am", "not", "afraid", "of", "Python"])
```


Liste (**list**) – Copia

- Hai già visto che:

```
>>> a = [1, 2, 3]
>>> b = a
```

- Non copia la lista. Le due etichette puntano allo stesso oggetto, quindi una modifica si ripercuoterebbe anche su b (e viceversa);
- Se volessimo copiare la lista potremmo utilizzare il metodo `copy()`:

```
>>> b = a.copy()
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3]
>>>
```

Liste (**list**) – Copia

- Attenzione però, la copia è *shallow*. Quindi gli oggetti all'interno della lista non vengono copiati, quindi:

```
>>> a = [1, [1, 2, 3], 3]
>>> b = a.copy()
>>> a[1].append(4)
>>> a
[1, [1, 2, 3, 4], 3]
>>> b
[1, [1, 2, 3, 4], 3]
>>>
```

- La *deepcopy* richiede l'uso del modulo copy:

```
>>> import copy
>>> a = [1, [1, 2, 3], 3]
>>> b = copy.deepcopy(a)
>>> a[1].append(4)
>>> a
[1, [1, 2, 3, 4], 3]
>>> b
[1, [1, 2, 3], 3]
>>>
```



Tuple (tuple)

Tuple (**tuple**)

- Python fornisce anche un altro tipo di sequenza chiamato **tuple**;
- Le **tuple** sono sostanzialmente delle liste NON modificabili;
- Quindi non puoi aggiungere/rimuovere elementi a/da una tupla dopo averla creata;
- Le tuple sono utili per rappresentare raggruppamenti ordinati di dimensioni fisse di elementi, come ad esempio coordinate 2D, 3D;
- Creare una nuova tupla è come creare una lista, tranne per il fatto che si usa (parentesi tonde) invece di [parentesi quadre]:

```
>>> my_vector = (1, 3, 4)
>>> print(my_vector)
```

- Puoi anche convertire una lista esistente (o qualsiasi tipo di sequenza Python) in un nuovo oggetto tupla:

```
>>> my_list = [1, 3, 4]
>>> my_vector = tuple(my_list)
>>> print(my_vector)
```

Tuple (tuple)

- Puoi usare le tuple proprio come faresti con le liste:

```
>>> x = (1, 2, 3, 4, 5)
>>> print(x[0])
>>> print(x[1:3])
>>> print(x[-1])
>>> print(x[:4:2])
>>> print(x[::-1])
```

- ... tranne per il fatto che non è possibile modificare gli elementi:

```
>>> x[1] = 6      # Python will complain!
>>> x.append(6)   # ... and complain again!
```

- **Una stranezza di Python!** Se la tua tupla ha un solo elemento, assicurati di aggiungere una virgola dopo il primo elemento per indicare che si tratta effettivamente di una tupla. Altrimenti, Python la tratterà come una singola variabile (la parentesi sarà solo una parentesi). *Brutto, lo so.*

```
>>> non_tuple = (1)
>>> print(non_tuple)
>>> print(type(non_tuple))
>>> singleton = (1, )
>>> print(singleton)
>>> print(type(singleton))
```

Tuple (**tuple**) - Funzioni

- Quando abbiamo introdotto le funzioni abbiamo detto: «sebbene tu possa restituire solo un oggetto, quell'oggetto può essere una sequenza (ad esempio una tupla), quindi tecnicamente puoi restituire più di un oggetto!»;
- Adesso potete comprenderne meglio il significato:

```
def do_more(x, y):  
    return (2*y, x+y)  
  
results = do_more(1, 2)  
print(type(results))  
  
x, y = do_more(1, 2)  
print(type(x))  
print(type(y))
```

- Nota che come le liste, le tuple possono contenere oggetti di tipo diverso, quindi le funzioni in Python possono veramente ritornare più «cose».

Esercizi

- Ora è il tuo turno!
- Trovi gli esercizi proposti nel file `esercizi_02.md`;



Una Piccola Parentesi: i Decoratori

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Funzioni – Uno Sguardo in Profondità

- Come già detto, le funzioni Python sono oggetti;
- Questo significa che le funzioni possono essere passate e utilizzate come argomenti, proprio come qualsiasi altro oggetto;
- Consideriamo le seguenti tre funzioni:

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we are the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```

- `say_hello()` e `be_awesome()` sono funzioni che si aspettano `str` come input;
- La funzione `greet_bob()`, invece, si aspetta una funzione come argomento.

Funzioni -- Uno Sguardo in Profondità

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we are the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```

- Possiamo, ad esempio, passarle la funzione `say_hello()` o `be_awesome()`:

```
>>> greet_bob(say_hello)  
'Hello Bob'  
  
>>> greet_bob(be_awesome)  
'Yo Bob, together we are the awesomest!'
```

- Si noti che `greet_bob(say_hello)` si riferisce a due funzioni, ma in modi diversi: `greet_bob()` e `say_hello`;
- La funzione `say_hello` è nominata senza parentesi. Ciò significa che viene passato solo un riferimento alla funzione; la funzione non viene eseguita;
- La funzione `greet_bob()`, invece, è seguita da parentesi tonde, quindi verrà invocata come di consueto;

Funzioni – *Inner Functions*

- È possibile definire funzioni all'interno di altre funzioni;
- Tali funzioni sono chiamate funzioni interne (*inner function*);
- Ecco un esempio di una funzione contenente due funzioni interne:

```
def parent():  
    print("Printing from the parent() function")  
  
    def first_child():  
        print("Printing from the first_child() function")  
  
    def second_child():  
        print("Printing from the second_child() function")  
  
    second_child()  
    first_child()
```

- Cosa succede quando chiami la funzione parent()? Pensaci per un minuto, prova a scrivere l'output su carta (o al PC), ma senza eseguire il codice;

Funzioni – *Inner Functions*

```
>>> parent()
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

- Si noti che l'ordine in cui sono definite le funzioni interne non ha importanza. Come con qualsiasi altra funzione, la stampa avviene solo quando le funzioni interne vengono eseguite;
- Inoltre, le funzioni interne non sono definite finché non viene chiamata la funzione padre;
- Esistono solo all'interno della funzione **parent()** come «variabili locali»;
- Se provi ad invocare **first_child()** dovresti ottenere un errore:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'first_child' is not defined
```

- Ogni volta che chiami **parent()**, vengono definite e in questo caso anche eseguite le funzioni interne **first_child()** e **second_child()**, ma a causa del loro *scope* locale, non sono disponibili al di fuori della funzione **parent()**, neanche dopo la sua invocazione;

Funzioni – Funzioni come Valore di Ritorno

- Python ti consente anche di avere funzioni come valori di ritorno. L'esempio seguente restituisce una delle funzioni interne dalla funzione `parent()`:

```
def parent(num):  
    def first_child():  
        return "Hi, I am Emma"  
  
    def second_child():  
        return "Call me Liam"  
  
    if num == 1:  
        return first_child  
    else:  
        return second_child
```

- Nota che in questo caso restituiamo `first_child` senza usare le parentesi. Ricorda che questo significa che stai restituendo un riferimento alla funzione `first_child`;
- Al contrario, `first_child()` si riferisce al risultato dell'esecuzione della funzione;

Funzioni – Funzioni come Valore di Ritorno

```
>>> first = parent(1)
>>> second = parent(2)

>>> first
<function parent.<locals>.first_child at 0x7f599f1e2e18>

>>> second
<function parent.<locals>.second_child at 0x7f599dad5268>
```

- L'output alquanto criptico significa semplicemente che la prima variabile fa riferimento alla funzione locale **first_child()** all'interno di **parent()**, mentre second punta a **second_child()**;
- Ora puoi utilizzare **first** e **second** come se fossero funzioni regolari, anche se non è possibile accedere direttamente alle funzioni a cui puntano:

```
>>> first()
'Hi, I am Emma'

>>> second()
'Call me Liam'
```

Decoratori

- Riesci a indovinare cosa succede quando chiami `say_whee()`? Provalo;

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = my_decorator(say_whee)
```

Decoratori

- Riesci a indovinare cosa succede quando chiami `say_whee()`?

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = my_decorator(say_whee)
```

```
>>> say_whee()  
Something is happening before the function is called.  
Whee!  
Something is happening after the function is called.
```

- La cosiddetta *decorazione* avviene nell'ultima riga di codice; il nome `say_whee` punta alla funzione interna `wrapper()` dopo aver eseguito quell'istruzione;

Decoratori

- In parole povere: i decoratori racchiudono una funzione, modificandone il comportamento;
- Prima di andare avanti, diamo un'occhiata a un secondo esempio;
- Poiché `wrapper()` è una normale funzione Python, il modo in cui un decoratore modifica una funzione può cambiare dinamicamente;
- Per non disturbare i tuoi vicini, il seguente esempio eseguirà il codice decorato solo durante il giorno:

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

Zucchero Sintattico

- Il modo in cui abbiamo decorato `say_whee()` è un po' goffo;
- Prima di tutto, si finisce per digitare il nome `say_whee` tre volte. Inoltre, la decorazione viene nascosta sotto la definizione della funzione;
- Python ti consente di usare i decorator in modo più semplice con il simbolo `@`;
- L'esempio seguente fa esattamente la stessa cosa, ma con una sintassi «migliore»:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

Zuccherò Sintattico

- Quindi, `@my_decorator` è solo un modo più semplice per dire `say_whee = my_decorator(say_whee);`
- È questo il modo di applicare un decoratore a una funzione!

Riutilizzare i Decoratori

- Ricorda che un decoratore è solo una normale funzione Python;
- Possiamo quindi spostare il decoratore nel suo modulo per utilizzarlo in molte altre funzioni;
- Crea un file chiamato **decorators.py** con il seguente contenuto:

```
def do_twice(func):  
    def wrapper_do_twice():  
        func()  
        func()  
    return wrapper_do_twice
```

- Ora puoi utilizzare questo nuovo decoratore in altri file eseguendone l'importazione:

```
from decorators import do_twice  
  
@do_twice  
def say_whee():  
    print("Whee!")
```

Decorare Funzioni con Argomenti

- Supponi di avere una funzione che accetta alcuni argomenti. Puoi ancora decorarla? Proviamo:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

- Sfortunatamente, l'esecuzione di questo codice genera un errore:

```
>>> greet("World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

- Il problema è che la funzione interna `wrapper_do_twice()` non accetta argomenti, ma le è stato passato `name="World"`;
- Potresti risolvere questo problema lasciando che `wrapper_do_twice()` accetti un argomento, ma poi non funzionerebbe per la funzione `say_whee()` che hai creato in precedenza;

Decorare Funzioni con Argomenti

- La soluzione è usare `*args` e `**kwargs` nella funzione wrapper interna. Questa accetterà quindi un numero arbitrario di argomenti posizionali e di argomenti con nome.
- Riscrivi `decorators.py` come segue:

```
def do_twice(func):  
    def wrapper_do_twice(*args, **kwargs):  
        func(*args, **kwargs)  
        func(*args, **kwargs)  
    return wrapper_do_twice
```

- La funzione interna `wrapper_do_twice()` ora accetta qualsiasi numero di argomenti e li passa alla funzione che decora. Ora entrambi i tuoi esempi `say_whee()` e `greet()` funzionano:

```
>>> say_whee()  
Whee!  
Whee!  
  
>>> greet("World")  
Hello World  
Hello World
```

Restituire Valori da Funzioni Decorate

- Cosa succede al valore di ritorno delle funzioni decorate?
- Sta al decoratore decidere. Diciamo che tu voglia decorare una semplice funzione:

```
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

- Se provi ad usarla:

```
>>> hi_adam = return_greeting("Adam")
Creating greeting
Creating greeting
>>> print(hi_adam)
None
```

Restituire Valori da Funzioni Decorate

- Spiacenti, il tuo decoratore ha «mangiato» il valore restituito dalla funzione;
- Poiché `do_twice_wrapper()` non restituisce esplicitamente un valore, la chiamata `return_greeting("Adam")` ha finito per restituire `None`;
- Per risolvere questo problema, devi assicurarti che la funzione **wrapper** restituisca il valore di ritorno della funzione decorata;
- Per farlo puoi cambiare il file `decorators.py`:

```
def do_twice(func):  
    def wrapper_do_twice(*args, **kwargs):  
        func(*args, **kwargs)  
        return func(*args, **kwargs)  
    return wrapper_do_twice
```

- In questo modo viene restituito il valore restituito dall'ultima esecuzione della funzione:

```
>>> return_greeting("Adam")  
Creating greeting  
Creating greeting  
'Hi Adam'
```


Introspezione

- Una grande comodità quando si lavora con Python, specialmente nella shell interattiva, è la sua potente capacità di introspezione;
- L'introspezione è la capacità di un oggetto di conoscere i propri attributi in fase di esecuzione. Ad esempio, una funzione conosce il proprio nome e la propria documentazione:

```
>>> print
<built-in function print>

>>> print.__name__
'print'

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    <full help message>
```

Introspezione

- L'introspezione funziona anche per le funzioni che definisci tu stesso:

```
>>> say_whee
<function do_twice.<locals>.wrapper_do_twice at 0x7f43700e52f0>

>>> say_whee.__name__
'wrapper_do_twice'

>>> help(say_whee)
Help on function wrapper_do_twice in module decorators:

wrapper_do_twice()
```

- Tuttavia, dopo essere stato decorato, `say_whee()` è diventato molto confuso sulla sua identità;
- Ora riporta di essere la funzione interna `wrapper_do_twice()` all'interno del decoratore `do_twice()`;
- Anche se tecnicamente vero, questa non è un'informazione molto utile;

Introspezione

- Per risolvere questo problema, i decoratori dovrebbero usare il decoratore `@functools.wraps`, che conserverà le informazioni sulla funzione originale;
- Aggiorna di nuovo `decorators.py`:

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

- Non è necessario modificare nulla della funzione `say_whee()` decorata:

```
>>> say_whee
<function say_whee at 0x7ff79a60f2f0>

>>> say_whee.__name__
'say_whee'

>>> help(say_whee)
Help on function say_whee in module whee:

say_whee()
```

Esercizi

- Ora è il tuo turno!
- Definisci un decoratore per misurare e visualizzare il tempo di esecuzione di una funzione e provalo sull'ultimo esercizio della scorsa volta (elimina duplicati);
- Per misurare il tempo di esecuzione di una funzione è possibile usare il modulo **time** di Python o ancora meglio **timeit**;
- Il secondo è meglio del primo per 3 motivi:
 - 🐍 Ripete i test molte volte per eliminare l'influenza di altre attività sulla tua macchina, come lo il *flushing* del disco o lo *scheduling* sistema operativo (**timeit.timeit**);
 - 🐍 Disabilita il *Generational Garbage Collector* per impedire che il processo falsi i tempi di esecuzione;
 - 🐍 Seleziona il timer più accurato per il tuo sistema operativo, **time.time** o **time.clock** in Python2 e **time.perf_counter()** in Python3;

Esercizi

- Ora è il tuo turno!

```
from timeit import default_timer as timer

start = timer()
# your code...
end = timer()
print(end - start) # time in seconds
```



Una (Seconda) Piccola Parentesi: Le Stringhe (str)

Stringhe

- Le stringhe Python (**str**) fanno parte dei tipi built-in di base, ma è questo il momento giusto per spiegarle meglio (dopo aver visto liste e tuple);
- Una stringa Python è anch'essa una sequenza di caratteri. Quindi puoi accedere a singoli caratteri, eseguire tagli (slicing) e iterare su stringhe proprio come faresti con liste/tuple:

```
>>> my_name = "Federico Bolelli"
>>> print(len(my_name))
>>>
>>> print(my_name[2])
>>> print(my_name[-1])
>>> print(my_name[0:6])
>>> print(my_name[-4:])
>>> print(my_name[0:10:2])
>>> print(my_name[::-1])
>>>
>>> for character in my_name:
...     print(character)
...
>>>
```

Stringhe

- Puoi anche usare gli operatori `+`, `*`, `in` e `not in` come in una lista:

```
>>> my_name = "Federico"
>>> print(my_name + my_name)
>>>
>>> my_name = "Federico"
>>> print(my_name * 7)
>>>
>>> my_name = "Federico"
>>> my_name += my_name
>>> print(my_name)
>>>
>>> print("r" in my_name)
>>> print("r" not in my_name)
```

- Nota che un oggetto `str` è immutabile, quindi come per le tuple non puoi usare l'`append()` su una stringa.

Stringhe

- Python fornisce molti metodi per facilitare la manipolazione delle stringhe. La [documentazione ufficiale](#) fornisce un elenco completo dei metodi di `str`;
- Eccone alcuni che potrebbero essere utili: `.split()`, `.join()`, `.strip()`, `.startswith()`, `.endswith()`, `.replace()`, `.upper()`, `.lower()`, `.capitalize()`, `.title()`.

Stringhe -- Formatting

- *f-strings (formatted string literals)* aiutano a rendere la formattazione delle stringhe più semplice e più leggibile;

```
>>> name = "Federico"
>>> count = 10
>>> possession = "cars"
>>>
>>> print(f"{name} has {count} {possession}")
```

- È inoltre possibile formattare i valori in virgola mobile;

```
>>> pi = 3.14159265359
>>> print(f"PI with three significant digits: {pi:.3}")
>>> print(f"PI with three decimal points: {pi:.3f}")
>>> print(f"PI with three significant digits, 9-character width is {pi:9.3}")
>>> print(f"PI with three significant digits, 9-character width padded with 0 is {pi:09.3}")
```

- E puoi giustificare le stringhe sinistra/centro/destra (dandogli una lunghezza fissa).

```
>>> title = "Title"
>>> print(f"| {title : <20} | {title : ^20} | {title : >20} |")
```

Stringhe -- Escaping

```
>>> 'the boy's mother'
      File "<stdin>", line 1
        'the boy's mother'
            ^
SyntaxError: invalid syntax
>>>
```

- Per risolvere questo problema possiamo usare il carattere di escape \ (come in C++):

```
>>> 'the boy\'s mother'
```

- È anche possibile racchiudere la tua stringa utilizzando un tipo diverso di *quoting* (consigliato per una migliore leggibilità):

```
>>> "the boy's mother"
>>> '''the boy's mother'''
>>> 'he said, "hello"'
```

Esercizi

- Ora è il tuo turno!



Gli Insiemi (set)

Set

- Un set non contiene elementi duplicati, inoltre l'ordine degli elementi contenuti nel set non dovrebbe essere importante;
- Per definire un set si usano le parentesi graffe {} in Python:

```
>>> vowels = {"a", "e", "i", "o", "u"}
>>> print(vowels)           # note the ordering
>>> vowels = {"i", "o", "a", "a", "e", "u", "e", "i", "a"}
>>> print(vowels)
```

- È possibile convertire una qualsiasi altra sequenza (ad esempio lista o tupla) in un set usando il costruttore `set()`. Ecco un'applicazione:

```
>>> words = ["let", "it", "go", ",", "let", "it", "go", ",", "can't",
...          "hold", "it", "back", "any", "more", "!"]
>>> vocabulary = set(words)
>>> print(vocabulary)
```

- Allo stesso modo, è possibile costruire una sequenza (ad esempio lista o tupla) a partire da un set:

```
>>> words = ["let", "it", "go", ",", "let", "it", "go", ",", "hold", "it", "back", "any", "more", "!"]
>>> vocabulary = set(words)
>>> words_nodup = list(vocabulary)
>>> words_nodup
['let', '!', 'any', 'back', 'go', 'hold', 'more', 'it', ',']
```

Set

- Si noti che la soluzione di eliminazione dei duplicati da una lista che fa uso dei set non preserva l'ordinamento;
- Per creare un insieme vuoto, devi sempre usare il costruttore `set()`. Non è possibile utilizzare `{}` perché identifica un dizionario vuoto (discuteremo dei `dict` subito dopo i set):

```
>>> empty_set = set()
>>> print(empty_set)
>>> print(type(empty_set))
>>> empty_dict = {}
>>> print(type(empty_dict))
```

- Non è possibile accedere direttamente ad un elemento in un set. Perché? (Suggerimento: pensa alla definizione di insieme);

```
>>> x = {1, 2, 3}
>>> print(x[1])           # NO!
```

- Dovrai convertire un set in un elenco (ad esempio lista) se hai bisogno di accedere a singoli elementi:

```
>>> x = {1, 2, 3}
>>> y = list(x)
>>> print(y[1])
```

Set – Aggiungere Elementi

- Per aggiungere un nuovo elemento al `set` puoi usare il metodo `add()`:

```
>> x = {1, 3, 2}
>>> x.add(4)
>>> print(x)
```

- Puoi anche aggiungere più elementi in una volta sola con il metodo `update()`:

```
>>> x = {8, 2, 5}
>>> x.update([3, 5, 2, 3])
>>> print(x)
```


Set – Rimuovere Elementi

- Puoi rimuovere un elemento da un set con il metodo **discard()**. Python non farà nulla se provi a rimuovere un elemento che non è nel set:

```
>>> x = {3, 9, 7}
>>> x.discard(3)
>>> print(x)
>>> x.discard(8)
>>> print(x)
```

- In alcuni casi è utile sapere se l'elemento che si vuole eliminare era o meno presente nel set. In questi casi puoi usare il metodo **remove()**. Se provi a rimuovere un elemento che non è nel set verrà generata un'eccezione:

```
>>> x = {3, 9, 7}
>>> x.remove(8)      # What is the error message?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 8
```

Set – Operazioni

- Le principali operazioni che è possibile eseguire sui set sono elencate di seguito:

Operation	Math Notation	Python Operator	Object Method
Union	$A \cup B$	<code>a_set b_set</code>	<code>a_set.union(b_set)</code>
Intersection	$A \cap B$	<code>a_set & b_set</code>	<code>a_set.intersection(b_set)</code>
Difference	$A \setminus B$	<code>a_set - b_set</code>	<code>a_set.difference(b_set)</code>
Symmetric difference	$A \Delta B$	<code>a_set ^ b_set</code>	<code>a_set.symmetric_difference(b_set)</code>
Membership	$A \in B$	<code>x in b_set</code>	
Non-membership	$A \notin B$	<code>x not in b_set</code>	
Proper Subset	$A \subset B$	<code>a_set < b_set</code>	
Subset	$A \subseteq B$	<code>a_set <= b_set</code>	<code>a_set.issubset(b_set)</code>
Proper Superset	$A \supset B$	<code>a_set > b_set</code>	
Superset	$A \supseteq B$	<code>a_set >= b_set</code>	<code>a_set.issuperset(b_set)</code>



I Dizionari (dict)

Dizionari

- Python implementa i dizionari nel tipo integrato **dict**. Potresti aver incontrato qualcosa di simile nella libreria standard C++: le mappe. Queste strutture prendono anche il nome di array associativi, tabelle hash, ecc a seconda del linguaggio;
- Nel caso in cui non si abbia familiarità, un dizionario funge da tabella di ricerca;
- È composto da coppie (chiave, valore). Si utilizza una chiave univoca per recuperare il valore corrispondente;
- Un caso d'uso di esempio per un dizionario potrebbe essere il recupero del nome di uno studente in base all'ID;
- Ecco un esempio di come definiresti un **dict** in Python:

```
>>> student_dict = {"00-01-30": "Ali", "00-02-11": "Simon",  
...                 "00-05-67": "Francesca", "00-09-88": "Cho"  
...                 }  
>>> print(student_dict)
```

Dizionari

- O in alternativa:

```
>>> student_dict = {}                                # or student_dict = dict()
>>> student_dict["00-01-30"] = "Ali"
>>> student_dict["00-02-11"] = "Simon"
>>> student_dict["00-05-67"] = "Francesca"
>>> student_dict["00-09-88"] = "Cho"
>>> print(student_dict)
```

- Cosa succede se successivamente si esegue `student_dict["00-01-30"] = "Josiah"`? Python lo consentirà? Cosa succederà ad **"Ali"**? Provalo per confermare!

Dizionari - Recupero

- Come ci si aspetterebbe, è possibile recuperare il valore di un dizionario data una chiave:

```
>>> best_student = student_dict["00-02-11"]  
>>> print(best_student)
```

- Ma cosa succede se la chiave non è presente?

```
>>> student_dict["00-11-22"]    # What error message do you get?
```

Dizionari - Recupero

- Dovresti sempre verificare se la chiave esiste prima di provare a utilizzarla accedere ad un valore;

```
>>> if "00-11-22" in student_dict:  
...     active_student = student_dict["00-11-22"]
```

- In alternativa, puoi utilizzare il metodo `get()` di `dict` per recuperare un valore. Il metodo assegnerà opportunamente un valore predefinito se la chiave non esiste (default **None**);

```
>>> real_student = student_dict.get("00-09-88", "John Doe")  
>>> print(real_student)  
>>> imaginary_student = student_dict.get("11-22-33", "John Doe")  
>>> print(imaginary_student)
```

Dizionari – Altri Metodi Utili

- Se occorre conoscere la lista completa delle chiavi puoi usare il metodo `.keys()`:

```
>>> id = student_dict.keys()
>>> print(ID)
>>> print(type(id))
>>> ids[0] # Can you do it?
```

- Se hai bisogno di accedere agli elementi dell'istanza `dict_keys`, puoi inserirli in un elenco. In effetti, puoi semplicemente inserire il dizionario direttamente in un elenco per ottenere un risultato analogo;

```
>>> id_list = list(ids)
>>> print(id_list[0])
>>>
>>> id_list = list(student_dict)
>>> print(id_list[0])
```

- Allo stesso modo, il metodo `.values()` fornisce l'elenco dei valori;
- Esiste anche il metodo `.items()` che fornisce un elenco di coppie (chiave, valore).

Raggruppamento di Dati con dict e tuple

- È possibile utilizzare un dizionario Python come se fosse una **struct** «flessibile» del C++. Ad esempio, puoi avere un dizionario che rappresenta un singolo studente, con le proprietà dello stesso rappresentate come chiavi:

```
>>> student = {"name": "Josiah",  
...            "id": "00-02-11",  
...            "degree": "MSc Computing"  
...            }  
>>> print(student["name"])  
>>> print(student["id"])  
>>> print(student["degree"])
```

Raggruppamento di Dati con dict e tuple

- Tecnicamente puoi avere qualsiasi oggetto come valore. È molto frequente avere strutture nidificate:

```
>>> student = {"name": "Josiah",
...             "id": "00-02-11",
...             "degree": "MSc Computing",
...             "courses": [
...                 {
...                     "title": "Introduction to Machine Learning",
...                     "code": "60012"
...                 },
...                 {
...                     "title": "Computer Vision",
...                     "code": "60006"
...                 }
...             ]
...         }
>>> print(student["courses"])
>>> print(student["courses"][0])
>>> print(student["courses"][0]["title"])
>>> print(student["courses"][0]["code"])
```

Raggruppamento di Dati con dict e tuple

- Come precedentemente accennato, anche le **tuple** possono essere usate per raggruppare dati:

```
>>> course1 = ("60012", "Introduction to Machine Learning")
>>> course2 = ("60006", "Computer Vision")
>>>
>>> student_record = ("Josiah", "00-02-11", "MSc Computing", [course1, course2])
>>>
>>> print(student_record[1])
>>> print(student_record[3])
>>> print(student_record[3][0])
```

Esercizi

- Ora è il tuo turno!



Programmazione Orientata agli Oggetti in Python

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Funzioni Built-in

- Arrivati a questo punto dovresti aver ormai capito che tutto è un oggetto in Python 😅;
- Ecco un promemoria su come definire una classe e creare un'istanza di una classe:

```
class Person:
    pass

person = Person()
print(type(person))
```

- Mentre i concetti OOP generali sono gli stessi del C++, ci sono molti dettagli specifici in Python; alcuni di questi richiedono un modo di pensare diverso da quello a cui sei stato esposto con in C++;
- In primo luogo, se tutto ciò di cui hai veramente bisogno è una **struct**, allora forse dovresti considerare di usare un **dict** invece di una classe!
- Nella slide che segue è riportata una definizione di classe di esempio in Python e come utilizzarla. Copia e incolla il codice in un file di testo, salvalo come person.py ed esegilo con python3 person.py per controllare l'output.

Funzioni Built-in

```
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def greet(self, message):
        print(f"{self.firstname} {self.lastname} says {message}!")

    def get_fullname(self):
        return f"{self.firstname} {self.lastname}"

    def __str__(self):
        return f"A person named {self.get_fullname()} aged {self.age}"

person = Person("Federico", "Bolelli", 20) # What do you mean I don't look 20? 🤔
person.greet("I love Beer 🍺🍺!")
print(person.age)
print(person.get_fullname())
print(person)
print(type(person))
print(isinstance(person, Person))
print(isinstance(person, object))
```

Costrutture

- Diamo prima un'occhiata al metodo `__init__()`. Il doppio underscore su entrambi i lati è una convenzione Python che indica che questo è un metodo *speciale* che fa *cose speciali*;
- Sono chiamati metodi *speciali* o *dunder* (dunder == double underscore);
- Ne vedremo altri più avanti;

```
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

person = Person("Federico", "Bolelli", 20) # Invoking the constructor
```

- `__init__()` funge da costruttore. Quando crei un'istanza di classe (ultima riga), Python creerà un nuovo oggetto in memoria heap ed eseguirà il metodo `__init__()`;
- Il parametro `self` è simile al puntatore `this` del C++;

Costrutture

- Quando crei una nuova istanza con **Person()**, quello che accade in background è che un nuovo oggetto viene creato e assegnato a **self**;
- Per rendere il concetto più concreto, quello che segue è (molto probabilmente) ciò che accade in background;

```
# new object of type Person created in heap, and assigned to the variable self
self = object.__new__(Person)

# The __init__() method for class Person is invoked
Person.__init__(self, "Josiah", "Wang", 20)

return self
```

- Quindi **self** è un riferimento alla nuova istanza di **Person** che Python ha appena allocato nella memoria heap;
- Si prega di NON scrivere mai questo codice che qui è riportato solo a scopo illustrativo!!

Attributi

- Diamo ora un'occhiata a come si dichiarano e inizializzano gli attributi (o le variabili di classe), oltre a come questi vengono usati;
- In C++, dichiarare variabili/attributi direttamente nella dichiarazione della classe;
- In Python, invece, aggiungi dinamicamente nuove variabili di classi all'oggetto self direttamente all'interno della funzione `__init__()`;
- Durante l'esecuzione di `__init__()` vengono inizializzati gli attributi e definiti i loro valori;

```
class Person:
    def __init__(self, firstname, lastname, age=0):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age
        self.friends = []

person = Person("Federico", "Bolelli", 20) # What do you mean I don't look 20? 😏
print(person.firstname)
print(person.lastname)

person.age = person.age + 1
print(person.age)
```

Attributi

- Come per una normale funzione, puoi assegnare valori predefiniti per qualsiasi argomento (ad es. `age=0`);
- Per accedere/aggiornare il valore degli attributi posso usare l'operatore `.` proprio come in C++;
- Gli attributi sono pubblici per impostazione predefinita;
- E gli attributi privati? Ci arriveremo più tardi!

```
class Person:
    def __init__(self, firstname, lastname, age=0):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age
        self.friends = []

person = Person("Federico", "Bolelli", 20) # What do you mean I don't look 20? 😅
print(person.firstname)
print(person.lastname)

person.age = person.age + 1
print(person.age)
```

Metodi

- I metodi in Python sono proprio come le funzioni
- L'unica differenza è che il primo parametro nella definizione del metodo deve prendere un riferimento all'istanza dell'oggetto (cioè **self**);
- Questo ti consentirà di accedere alle proprietà dell'istanza all'interno del metodo stesso;

```
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def greet(self, message):
        print(f"{self.firstname} {self.lastname} says {message}!")

    def get_fullname(self):
        return f"{self.firstname} {self.lastname}"

person = Person("Josiah", "Wang", 20) # What do you mean I don't look 20? 🤔
person.greet("I love Beer 🍺🍺!")
print(person.get_fullname())
```

Metodi

- Quando invochi `person.get_fullname()`, ciò che accade in background è che Python converte implicitamente la chiamata a `Person.get_fullname(person)`;
- Quindi `self = person`. Ovviamente, puoi anche chiamare `Person.get_fullname(person)` direttamente, ma questo renderebbe il codice meno leggibile;
- L'uso esplicito di `self` per attributi e metodi di istanza rende il codice meno ambiguo e più leggibile ed è in linea con un'altra filosofia di progettazione di Python: *"explicit is better than implicit"*.

```
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def greet(self, message):
        print(f"{self.firstname} {self.lastname} says {message}!")

    def get_fullname(self):
        return f"{self.firstname} {self.lastname}"

person = Person("Josiah", "Wang", 20) # What do you mean I don't look 20? 🤔
person.greet("I love Beer 🍺🍺!")
print(person.get_fullname())
```

Metodi Speciali o Dunder

- Abbiamo visto in precedenza il metodo di speciale `__init__()`;
- Python offre molti altri metodi *speciali* che possono essere usati per far sì che le tue classi personalizzate agiscano come i tipi *built-in* di Python;
- Ecco alcuni esempi:
 - 🐍 `__str__()` viene utilizzato durante la stampa o la conversione di un oggetto in una stringa. Infatti, `str(x)` invoca `x.__str__()`. Ad esempio, potresti volere che `print(person)` restituisca qualcosa di più carino e più informativo di `<__main__.Person object at 0x7ffe1cf127c0>`;
 - 🐍 `__add__()`, `__sub__()`, `__mul__()`, `__truediv__()`, `__pow__()` eseguono l'*overload* dei rispettivi operatori matematici; ad esempio, `x+y` invoca effettivamente `x.__add__(y)`;
 - 🐍 Per eseguire l'*overload* degli operatori booleani occorre definire i metodi: `__lt__()`, `__le__()`, `__gt__()`, `__ge__()`, `__eq__()`, `__ne__()` (`<`, `<=`, `>`, `>=`, `==`, `!=`)

Metodi Speciali o Dunder

- Collezioni:

 `__len__()` (`len(x)` invoca `x.__len__()`)

 `__contains__()` (`item in x` invoca `x.__contains__(item)`)

 `__getitem__()` (`x[key]` invoca `x.__getitem__(key)`)

 `__setitem__()` (`x[key] = item` richiama `x.__setitem__(key, item)`)

 `__iter__()` serve per consentire l'utilizzo del tipo nei cicli `for`

Metodi

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector ({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(5, 7)
v3 = v1 + v2
print(v3)
```


Ereditarietà

- Come il C++, Python supporta l'ereditarietà;
- In effetti abbiamo già detto che tutte le classi in Python3 sono sottoclassi di **object** per impostazione predefinita;
- Segue un esempio della classe **GameCharacter** e delle sua sottoclasse **Enemy**:

```
class GameCharacter:
    def __init__(self, name, health=50, strength=30, defence=20):
        self.name = name
        self.health = health
        self.strength = strength
        self.defence = defence

    def attack(self, victim):
        victim.health = victim.health - self.strength
        print(f"Bam! {self.name} attacked {victim.name}.",
              f"{victim.name}'s health is now {victim.health}")

    def defend(self, attacker):
        self.health = self.health - attacker.strength * 0.25
        print(f"{self.name} defended against {attacker.name}.",
              f"{self.name}'s health is now {self.health}")

    def __str__(self):
        return (f"{self.name} is a GameCharacter (health: {self.health}, "
                f"strength: {self.strength}, defence: {self.defence})")
```

Ereditarietà

```
class Enemy(GameCharacter):
    def __init__(self, name, health=50, strength=30, defence=20, evilness=50):
        # call the constructor of the superclass
        super().__init__(name, health, strength, defence)
        self.evilness = 50

    def evil_laugh(self):
        print("Heheheheheheeee!!")

    def __str__(self):
        return (super().__str__().replace("GameCharacter", "Enemy").replace(")", ",")
                + f" evilness: {self.evilness}")

boy = GameCharacter("Boy", 100, 20, 10)
evilman = Enemy("Voldemort", 30, 50, 40, 100)
print(boy)
print(evilman)
evilman.attack(boy)
boy.defend(evilman)
boy.evil_laugh() # You should get an error here.
```

Ereditarietà

- **Enemy** è una sottoclasse di **GameCharacter**;
- Se si definisce l'override del costruttore `__init__()`, sarà necessario richiamare esplicitamente il costruttore della superclasse **GameCharacter** con `super().__init__()`;
- Se non lo fai, **Enemy** NON erediterà attributi dalla superclasse come **name**, **health**, **strength** e **defence** in questo esempio;
- `super()` si riferisce alla superclasse, in questo caso **GameCharacter**;
- Sempre utilizzando `super()` è possibile accedere ai metodi originali della superclasse utilizzando.

Incapsulamento

- Finora abbiamo solo parlato solo di attributi pubblici, e quelli privati?
- In C++, gli attributi sono privati per impostazione predefinita. In generale, una buona regola di programmazione consiste nel nascondere tutti gli attributi ed esporre quelli necessari meutilizzando metodi *getter* o *setter*, ad es. `get_age()` o `set_age()`;
- Python si basa su una filosofia differente. Tecnicamente non puoi rendere alcun attributo veramente privato (ci sono sempre modi per accedervi), quindi viene usato il termine "non pubblico";
- Citando la guida di stile ufficiale di Python (PEP 8): «*Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed*»;
- Ci sono diverse scuole di pensiero. La guida ufficiale PEP 8 dice «*If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public*»;

Incapsulamento

- Ad ogni modo, la visione predominante al giorno d'oggi è quella di rendere tutto pubblico per impostazione predefinita a meno che non sia chiaramente non pubblico;
- Se in futuro avrai bisogno di rendere non pubblico un attributo pubblico, potrai farlo facilmente esponendolo come proprietà;
- Vediamo come con un esempio:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Federico Bolelli", 20)
print(person.age)
person.age = 70
print(person.age)
```

Incapsulamento

- Diciamo che a posteriori decidiamo di non voler esporre direttamente l'età di una persona;
- Una persona dichiarerà sempre (pubblicamente) di avere due anni in meno della sua vera età! Inoltre, consentiamo la modifica dell'età, ma solo se la nuova età è inferiore a 30 anni;
- Se dovessimo pensare come un programmatore C++, potremmo ritrovarci con un codice Python simile a questo:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def get_age(self):
        return self.__age - 2

    def set_age(self, new_age):
        if new_age < 30:
            self.__age = new_age
        else:
            print("Never! I am always under 30!")
```

Incapsulamento

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def get_age(self):
        return self.__age - 2

    def set_age(self, new_age):
        if new_age < 30:
            self.__age = new_age
        else:
            print("Never! I am always under 30!")

person = Person("Federico Bolelli", 20)
print(person.get_age())
person.set_age(10)
print(person.get_age())
person.set_age(70)
print(person.get_age())
print(person.__age)  ## this won't work!
```

Incapsulamento

- Come avrai capito, in Python per rendere un attributo (o metodo) non pubblico devi anteporre due *underscore* al nome;
- Ciò attiverà il *name mangling* che nasconderà questo attributo all'esterno della classe;
- In realtà, se sei abbastanza «disperato» puoi ancora accedervi facendo `person._Person__age`!
- Non farlo però, probabilmente c'è una buona ragione per cui lo sviluppatore originale ha voluto definire l'attributo non pubblico!
- Ignorando l'ultima riga, il codice funziona. I veri problemi (dal punto di vista di Python) sono:
 - 🐍 `person.get_age()` e `person.set_age()` rendono il codice meno leggibile rispetto al semplice `person.age` nella versione originale;
 - 🐍 Chiunque abbia utilizzato in precedenza la classe `Person` originale dovrà modificare tutte le occorrenze di `person.age` in `person.get_age()` e `person.set_age()`;
- C'è una soluzione migliore a questo?

Incapsulamento – The Pythonic Way

- I due problemi principali della precedente versione erano:
 - 🐍 I metodi getter e setter rendono il nostro codice meno leggibile;
 - 🐍 Gli utilizzatori della classe **Person** originale dovranno modificare il proprio codice;
- Proviamo ora ad affrontare lo stesso problema, ma con una soluzione più *Pythonic*:

Incapsulamento – The Pythonic Way

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def age(self):
        return self.__age - 2

    @age.setter
    def age(self, new_age):
        if new_age < 30:
            self.__age = new_age
        else:
            print("Never! I am always under 30!")

person = Person("Josiah Wang", 20)
print(person.age)
person.age = 10
print(person.age)
person.age = 70
print(person.age)
print(person.__age)  ## this won't work!
```

Incapsulamento – The Pythonic Way

- Con questa versione possiamo ancora scrivere `person.age = 10` o `print(person.age)` e il codice rimane «più leggibile»;
- Gli utilizzatori di **Person** non dovranno modificare il loro codice;
- Siamo riusciti a mantenere la nostra interfaccia, ma allo stesso tempo abbiamo un migliore controllo dei nostri attributi.
- In sostanza abbiamo introdotto il decoratore `@property`. Questo decoratore converte il metodo `age()` in una proprietà pubblica e leggibile `age`. Il metodo `age()` verrà richiamato quando si tenta di leggere la proprietà `age`, ad esempio con `print(person.age)`;
- Analogamente, il decoratore `@property_name.setter` trasforma il secondo metodo `age()` nella proprietà pubblica e modificabile `age`.
- Quindi il secondo metodo `age()` invocato quando un utente tenta di modificare la proprietà, ad esempio con `person.age = 10`;
- Per la precisione, il secondo metodo `age()` è stato usato anche nel costruttore;

Incapsulamento – The Pythonic Way

- Se dovessimo rimuovere il secondo metodo age, l'età diventerebbe una proprietà di sola lettura;
- In sintesi, mantieni pubblici i tuoi attributi in Python. E quando hai bisogno di più controllo, incapsula gli attributi convertendoli in proprietà;
- Il significato e funzionamento dei decorator verrà analizzato più avanti.

Attributi Protetti

- E per quanto riguarda l'accesso protetto?
- In Python, puoi indicare che un attributo è protetto anteponendolo al nome un singolo *underscore*;
- Tuttavia, chiunque può ancora accedere a questi attributi, indipendentemente dal fatto che siano o meno sottoclassi;
- Quindi il carattere di sottolineatura è solo un suggerimento per gli utenti: l'attributo è pensato per essere protetto (o addirittura privato) e probabilmente non dovresti usarlo;
- Piuttosto che cercare di proteggere le cose per assicurarsi che i programmatori non infrangano le regole, Python presuppone che la maggior parte dei programmatori sia responsabile e razionale 🙈;
- Quindi, se vedi qualcuno che usa il tuo attributo protetto in modo inappropriato, sentiti libero di rivolgergli un gentile rimprovero!



I File in Python

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

File - Lettura

- Python semplifica la lettura e la scrittura su file;
- Assumiamo tu abbia un file chiamato **test.txt** nella directory corrente;
- Ecco come leggeresti l'intero file in una volta sola. Non consigliato se il file è grande!

```
# Open a file in read mode, and read everything into content
with open("test.txt", "r") as infile:
    content = infile.read()
```

- **open()** restituisce un oggetto file che viene assegnato alla variabile **infile**. L'istruzione **with** chiuderà automaticamente il file una volta usciti dal blocco, quindi non devi preoccuparti di farlo;
- Puoi anche leggere il file una riga alla volta usando un ciclo:

```
# Note that each line ends with an '\n' intact.
# Use line.strip() to remove any leading/trailing whitespace
with open("test.txt", "r") as infile:
    for line in infile:
        print(line.strip())
```

File - Scrittura

- Per scrivere su un file puoi utilizzare il metodo `write()` dell'oggetto file. Ricordati anche di aprire il file in modalità scrittura!

```
# Remember to open the file in write mode
with open("output.txt", "w") as outfile:
    outfile.write("First line\n")
    outfile.write(str(2) + "\n")
    outfile.write(f"{3}rd line\n")
```


File – Lettura di un CSV

- Un file *Comma Separated Value* (CSV) è un file di testo che utilizza una struttura specifica per organizzare i dati tabulari (e.g. fogli di calcolo);
- In genere, i file CSV utilizzano una virgola (,) per separare ogni valore di dati (da qui il nome), ma è possibile utilizzare altri delimitatori: tabulazione (\t), due punti (:) e punto e virgola (;).
- La prima riga contiene (solitamente) il nome delle colonne;
- Le righe successive contengono (solitamente) i record, uno per riga;
- Supponiamo di avere un file CSV chiamato **student.csv** con il contenuto riportato sotto (basta copiare e incollare il testo in un editor e salvarlo come student.csv):

```
name,faculty,department
Alice Smith,Science,Chemistry
Ben Williams,Eng,EEE
Bob Jones,Science,Physics
Andrew Taylor,Eng,Computing
```

File – Lettura di un CSV

- Usiamo il modulo `csv` per leggere questo file:

```
import csv

with open("students.csv") as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=",")
    column_data = next(csv_reader)
    print (f"Column names are {'', '}.join(column_data)}")

    for row in csv_reader:
        print (f"Student {row[0]} is from faculty of {row[1]}, {row[2]} dept.")
```

- L'output atteso è:

```
Column names are name, faculty, department
Student Alice Smith is from faculty of Science, Chemistry dept.
Student Ben Williams is from faculty of Eng, EEE dept.
Student Bob Jones is from faculty of Science, Physics dept.
Student Andrew Taylor is from faculty of Eng, Computing dept.
```

File – Lettura di un CSV

- Tornando al codice:

🐍 con `open("students.csv") as csv_file` si apre il file CSV come file di testo, la funzione restituirà un oggetto file;

🐍 `csv_reader = csv.reader(csv_file, delimiter=",")` costruisce un oggetto `csv.reader`, passando l'oggetto file al suo costruttore. Inoltre, stiamo specificando che il separatore da utilizzare per il CSV è la virgola;

🐍 `column_data = next(csv_reader)` estrae le intestazioni di colonna dalla prima riga utilizzando la funzione `next()`;

🐍 `for riga in csv_reader:` ogni riga rimanente è un elenco di elementi `str` contenenti i dati trovati rimuovendo il delimitatore;

File – Lettura di un CSV

- Puoi anche leggere i file CSV in un dizionario. È quindi possibile accedere agli elementi utilizzando i nomi delle colonne come chiavi:

```
import csv

with open("students.csv") as csv_file:
    csv_reader = csv.DictReader(csv_file)

    for row in csv_reader:
        print(f"Student {row['name']} is from faculty of {row['faculty']}, "
              f"{row['department']} dept. ")
```

- Se il file CSV non contiene i nomi delle colonne, dovrai specificare tu le chiavi. Puoi farlo impostando come parametro **fieldnames** un elenco contenente le chiavi:

```
fieldnames = ['name', 'faculty', 'department']
csv_reader = csv.DictReader(csv_file, fieldnames=fieldnames)
```

File – Scrittura di un CSV

- È possibile scrivere dati in un file CSV usando l'oggetto **writer** nel modulo **csv**;
- Puoi usare il metodo **writerows()** che accetta più righe in una volta sola:

```
import csv

data = [{"name", "faculty", "department"}, ["Davies", "Eng", "EEE"],
        ["Smith", "Eng", "Computing"]]

with open("uni.csv", "w") as csv_file:
    writer = csv.writer(csv_file)
    writer.writerows(data)
```

- Il contenuto di **uni.csv** dovrebbe assomigliare a questo:

```
name,faculty,department
Davies,Eng,EEE
Smith,Eng,Computing
```

File – Scrittura di un CSV

- Puoi anche aggiungere una nuova (singola) riga al CSV che abbiamo generato nella slide precedente usando `.writerow()`:

```
import csv

row = ["Williams", "Eng", "EEE"]

# Note append mode "a". We are appending new items to an existing file
with open("uni.csv", "a") as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(row)
```

- Il contenuto del nuovo `uni.csv` sarà:

```
name,faculty,department
Davies,Eng,EEE
Smith,Eng,Computing
Williams,Eng,EEE
```

File – Scrittura di un CSV

- Puoi anche scrivere su un file CSV a partire da un dizionario;
- Il pezzo di codice mostra un CSV separato da | e scritto riga per riga:

```
import csv

with open("uni.csv", "w") as csv_file:
    fieldnames = ["name", "faculty", "dept"]

    writer = csv.DictWriter(csv_file, fieldnames=fieldnames, delimiter="|")
    writer.writeheader()
    writer.writerow({"name": "Davies", "faculty": "Eng", "dept": "EEE"})
    writer.writerow({"name": "Smith", "faculty": "Eng", "dept": "Computing"})
```

- Il contenuto di `uni.csv` è:

```
name|faculty|dept
Davies|Eng|EEE
Smith|Eng|Computing
```

File – Scrittura di un CSV

- Puoi anche scrivere un intero dizionario direttamente in un file CSV;

```
import csv

data = [{"name": "Davies", "faculty": "Eng", "dept": "EEE"},
        {"name": "Smith", "faculty": "Eng", "dept": "Comp"}
]

with open("uni.csv", "w") as csv_file:
    fieldnames = ["name", "faculty", "dept"]

    writer = csv.DictWriter(csv_file, fieldnames=fieldnames, delimiter="|")
    writer.writeheader()
    writer.writerows(data)
```

- Il contenuto di `uni.csv` sarà:

```
name|faculty|dept
Davies|Eng|EEE
Smith|Eng|Comp
```


Gestione dei File JSON

- Al giorno d'oggi molti dataset di Machine Learning (e non solo) organizzati i dati in formato JSON;
- Ne è un esempio il dataset COCO;
- JSON viene anche utilizzato per la comunicazione tra un server Web e la tua app Web o browser.
- Se guardi un file JSON di esempio (sotto), potrebbe sembrare terribilmente familiare. Che cosa vi ricorda? Forse un dict?
- L'oggetto radice di un JSON è generalmente un elenco o un dizionario;

```
{
  "name": "Smith",
  "interests": ["maths", "programming"],
  "age": 25,
  "courses": [
    {
      "name": "Python",
      "term": 1
    },
    {
      "name": "Soft Eng",
      "term": 2
    }
  ]
}
```

Gestione dei File JSON - Serializzazione

- Possiamo facilmente salvare la nostra struttura dati Python in una stringa JSON (serializzazione);
- Per scrivere i tuoi dati in un file JSON, puoi usare `json.dump()`. Il codice che segue serializza i dati in JSON e li salva nel file `data.json`:

```
import json

data = { "course": { "name": "Introduction to Machine Learning", "term": 2 } }

with open("data.json", "w") as f:
    json.dump(data, f)
```

- Per scrivere i tuoi dati su una stringa (e fare qualcos'altro con essa in seguito), usa `json.dumps()`:

```
json_string = json.dumps(data)
print(json_string)  ## {"course": {"name": "Introduction to Machine Learning", "term": 2}}
```

Gestione dei File JSON - Deserializzazione

- Possiamo anche convertire una stringa JSON in una struttura dati Python (deserializzazione);
- Anche in questo caso abbiamo a disposizione i metodi `json.load(fileobject)` e `json.loads(json_string)`:

```
# load JSON from file
with open("data.json", "r") as f:
    data = json.load(f)

print(data)

# this is fine too, since we are not writing to the file
data = json.load(open("data.json", "r"))

# load JSON from a string
# assuming we still have json_string from earlier
data = json.loads(json_string)
```

Pickle

- Se hai bisogno di salvare strutture di dati Python complesse e prevedi di caricarle in futuro sempre solo utilizzando Python, puoi prendere in considerazione l'utilizzo del modulo **pickle**;
- Il modulo **pickle** viene utilizzato per memorizzare gli oggetti Python in un file (e recuperarli in un secondo momento);
- Ad esempio, puoi usarlo per salvare un modello di Machine Learning durante l'addestramento;
- **pickle** salva gli oggetti Python su disco in binario (serializzazione) e viceversa (deserializzazione);
- Puoi serializzare interi, float, booleani, stringhe, tuple, liste, insiemi, dizionari (che contengono oggetti che possono essere serializzati), classi di primo livello;
- Niente cetriolini sottaceto, mi dispiace! 🥒

Pickle

- Avvertenze:

- 🐍 **pickle** è specifico di Python. È sconsigliato se prevedi di utilizzare i dati con altri linguaggi;

- 🐍 Assicurati di utilizzare la stessa versione di Python. Non è garantito che **pickle** funzioni con diverse versioni di Python;

- 🐍 Non estrarre dati non attendibili poiché potresti eseguire codice dannoso durante l'estrazione;

Pickling

- Si esegue il *pickling* con `pickle.dump(obj, file)` e l'*unpickling* con `pickle.load(file)`:

```
import pickle

courses = {70053: {"lecturer": "Josiah Wang", "title": "Python Programming"},
            70051: {"lecturer": "Robert Craven", "title": "Symbolic AI"}}

# Save courses to disk. Note binary mode!
with open("courses.pkl", "wb") as f:
    pickle.dump(courses, f)

# Load courses from disk. Again, it is a binary file!
with open("courses.pkl", "rb") as f:
    pickled_courses = pickle.load(f)

print(pickled_courses)
## {70053: {'lecturer': 'Josiah Wang', 'title': 'Python Programming'},
## 70051: {'lecturer': 'Robert Craven', 'title': 'Symbolic AI'}}

print(type(pickled_courses)) ## <class 'dict'>

print(courses == pickled_courses) ## True
```

Pickling

- Ecco un altro esempio di *pickling* di un elenco di oggetti (di una classe personalizzata):

```
import pickle

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

    def __str__(self):
        return f"Vector ({self.x}, {self.y})"

    def __repr__(self):
        """ This makes the unique string representation
            of the object instance look more readable
        """
        return str(self)

v1 = Vector(2, 3)
v2 = Vector(4, 3)
v = [v1, v2]
```

Pickling

- Ecco un altro esempio di *pickling* di un elenco di oggetti (di una classe personalizzata):

```
# Save v to disk.
with open('vectors.pkl', 'wb') as f:
    pickle.dump(v, f)

# Load pickled file from disk
with open('vectors.pkl', 'rb') as f:
    pickled_vectors = pickle.load(f)

print(pickled_vectors)  ## [Vector (2, 3), Vector (4, 3)]

print(type(pickled_vectors))  ## <class 'list'>
```




I Moduli Python

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Moduli

- Oltre alle funzioni e ai tipi *built-in*, la libreria standard di Python offre in realtà un'ampia raccolta di moduli e pacchetti. Questo è ciò che rende il Python veramente potente e versatile;
- Questi moduli vanno importati esplicitamente, proprio come faresti con `#include <iostream>` in C++;
- Proviamo ad importare il modulo matematico per usarlo. Per importare un modulo, basta... beh... importarlo!

```
>>> import math
>>> help(math)
>>> math.sqrt(9)
>>> math.log10(50)
>>> math.cos(30 * math.pi / 180)    # cosine of x radians
```

Moduli

- Puoi immaginare un modulo come uno script Python che si trova da qualche parte nella tua installazione Python (o che tu stesso hai creato);
- Importare **math** significa eseguire tutto quello che c'è all'interno dello script e quindi definire tutti i nomi e le funzioni che si trovano al suo interno. Una volta importato, è possibile accedere a tutte le funzioni/variabili/classi contenute nel modulo tramite l'operatore punto (.);
- Tutto il contenuto di **math** rimane così «incapsulato» nel *namespace* **math**;
- Puoi anche rinominare il modulo durante l'importazione. Nell'esempio seguente, abbiamo rinominato il modulo **math** in **m** per evitare di dover digitare **math** tutto il tempo; una cosa simile allo **using namespace std** del C++;

```
>>> import math as m
>>> m.sqrt(9)
>>> m.log10(50)
>>> m.cos(30 * m.pi / 180)
```

Moduli

- Volendo, è possibile importare solo specifiche funzioni/variabili/classi del modulo:

```
>>> from math import sqrt, log10, cos, pi
>>> sqrt(9)
>>> log10(50)
>>> cos(30 * pi / 180)
```

- Python ora ha le funzioni/variabili/classi direttamente nel suo spazio dei nomi (non c'è incapsulamento nel *namespace* **math**), che è possibile usare direttamente senza dover fare riferimento ad essi tramite il nome **math**.

Moduli Personalizzati

- Puoi anche scrivere il tuo modulo e importarlo da un altro script;
- Copia lo script qui sotto e salvalo come `my_module.py`:

```
FACTOR = 5

class Monster:
    def __init__(self, name="Me", food="cookies"):
        self.name = name
        self.food = food

    def talk(self):
        print(f"{self.name} love {self.food}!")

def spawn_monsters():
    return [Monster("Happy", "carrots"),
            Monster("Bashful", "ice-creams"),
            Monster("Wild", "cookies")]

def calculate_growthrate(adjustment=3):
    return 25 * FACTOR + adjustment
```

Moduli Personalizzati

- Se provi a eseguire questo script, non accadrà (quasi) nulla;
- Ora scriviamo un altro script per importare il nostro modulo. Chiamalo `my_script.py` e salvalo nella stessa directory di `my_module.py`:

```
import my_module

print(my_module.FACTOR)

print(my_module.calculate_growthrate(2))

monsters = my_module.spawn_monsters()

new_monster = my_module.Monster("Crazy", "sashimi")
monsters.append(new_monster)

for monster in monsters:
    monster.talk()
```

__name__

- Ricorda che `my_module.py` è di per sé uno script Python. Quindi potremmo scrivere:

```
FACTOR = 5

class Monster:
    def __init__(self, name="Me", food="cookies"):
        self.name = name
        self.food = food

    def talk(self):
        print(f"{self.name} love {self.food}!")

def spawn_monsters():
    return [Monster("Happy", "carrots"),
            Monster("Bashful", "ice-creams"),
            Monster("Wild", "cookies")]

def calculate_growthrate(adjustment=3):
    return 25 * FACTOR + adjustment

monster = Monster("Silly", "pizza")
monster.talk()
```

__name__

- Quando esegui il codice con `python3 my_module.py`, questo stamperà "Silly love pizza!";
- Quando esegui `python3 my_script.py` (che importa `my_module`), questo stamperà anche "Silly love pizza!".
- Tuttavia, non vogliamo che questo accada quando importiamo `my_module`: dopotutto è una libreria per utenti che utilizzano le nostre funzioni/classi/variabili;
- Vogliamo che le ultime due istruzioni vengano eseguite solo quando stiamo eseguendo `my_module` come script e non quando lo stiamo importando come modulo;
- Caso d'uso: vogliamo verificare che la nostra classe funzioni quando la implementiamo e prima di importarla come modulo;
- Per ottenere questo comportamento possiamo sfruttare il fatto che Python imposta una variabile speciale `__name__` come `__main__` quando esegui un file codice come script (ad esempio usando `python3 my_module.py`);
- Quindi è sufficiente controllare il contenuto di tale variabile ed eseguire il codice di «verifica» solo quando `__name__ == "__main__"`!

__name__

```
FACTOR = 5

class Monster:
    def __init__(self, name="Me", food="cookies"):
        self.name = name
        self.food = food

    def talk(self):
        print(f"{self.name} love {self.food}!")

def spawn_monsters():
    return [Monster("Happy", "carrots"),
            Monster("Bashful", "ice-creams"),
            Monster("Wild", "cookies")]

def calculate_growthrate(adjustment=3):
    return 25 * FACTOR + adjustment

if __name__ == "__main__":
    monster = Monster("Silly", "pizza")
    monster.talk()
```

__name__

- Ora, se esegui `python3 my_module.py`, vedrai *Silly love pizza!*
- Ma se esegui `python3 my_script.py` che importa `my_module`, non lo vedrai;
- Anche se si è tentati di pensare a questa cosa come la funzione `main()` del C++, non è così dato che Python non ha o non ha bisogno di tale funzione.






Unit Test in Python

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

La Scelta del Test Runner

- Ci sono molti *test runner* disponibili per Python. Quello integrato nella libreria standard di Python si chiama **unittest**;
- I principi di **unittest** sono facilmente trasferibili in altri framework;
- I tre test runner più popolari sono:
 -  **unittest**
 -  **naso** o **naso2**
 -  **pytest**
- La scelta del miglior test runner per le tue esigenze e il tuo livello di esperienza è importante, ma non sarà oggetto di questo corso;

unittest

- **unittest** è stato integrato nella libreria standard di Python dalla versione 2.1;
 - Viene spesso usato nelle applicazioni Python commerciali e nei progetti open source;
 - Contiene sia un framework di test che un test runner;
 - Ha alcuni requisiti importanti per la scrittura e l'esecuzione dei test:
- 🐍 Tutti i test devono essere metodi di classe;
 - 🐍 Si utilizza una serie di metodi di asserzione speciali definiti dalla classe **TestCase** (non si deve usare l'istruzione **assert** incorporata nel linguaggio)

unittest (esempio)

```
import unittest

class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

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

Come Strutturare un Semplice Test

- Prima di immergerti nella scrittura dei test, ti consigliamo di prendere prima un paio di decisioni:
 - 🐍 Cosa vuoi testare?
 - 🐍 Stai scrivendo un Unit Test o un Test di Integrazione?
- Quindi la struttura di un test dovrebbe seguire vagamente questo flusso di lavoro:
 - 🐍 Crea i tuoi input;
 - 🐍 Eseguire il codice in fase di test, catturando l'output;
 - 🐍 Confronta l'output con un risultato previsto;
- Supponiamo di dover testare la funzione `sum()` definita nel file `my_sum`. Ci sono molti comportamenti in `sum()` che potresti controllare, come ad esempio:
 - 🐍 Può sommare un elenco di numeri interi?
 - 🐍 Può sommare una tupla o un insieme?
 - 🐍 Può sommare un elenco di float?

```
def sum(arg):  
    total = 0  
    for val in arg:  
        total += val  
    return total
```

Come Strutturare un Semplice Test

- 🐍 Cosa succede quando gli fornisci un valore errato, come un singolo numero intero o una stringa?
- 🐍 Cosa succede quando uno dei valori è negativo?
- Crea il file **test.py** con il seguente codice Python:

```
import unittest

from my_sum import sum

class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
        Test that it can sum a list of integers
        """
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

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


Come Strutturare un Semplice Test

- Il codice appena visto:
 - 🐍 Importa `sum()` dal pacchetto `my_sum` che hai creato;
 - 🐍 Definisce una nuova classe test case denominata `TestSum`, che eredita da `unittest.TestCase`;
 - 🐍 Definisce un metodo di test, `.test_list_int()`, per testare un elenco di numeri interi. Il metodo `.test_list_int()`:
 - ❖ Dichiarare una variabile `data` contenente un elenco di numeri (1, 2, 3);
 - ❖ Assegna il risultato di `my_sum.sum(data)` alla variabile `result`;
 - ❖ Asserisce che il valore di `result` è uguale a 6 utilizzando il metodo `.assertEqual()` della classe `unittest.TestCase`;
 - 🐍 Definisce un punto di ingresso della riga di comando, che esegue unittest test-runner `.main()`;

Come Scrivere le Asserzioni

- L'ultimo passaggio della scrittura di un test consiste nel convalidare l'output con una risposta nota;
- Questo è noto con il nome *assertion*;
- Esistono alcune best practice generali su come scrivere asserzioni:
 - 🐍 Assicurati che i test siano ripetibili ed esegui il test più volte per assicurarti che fornisca sempre lo stesso risultato;
 - 🐍 Prova e verifica se i risultati sui dati di input sono corretti, ad esempio controllando che il risultato sia la somma effettiva dei valori nell'esempio `sum()`;
- **unittest** fornisce molti metodi per effettuare *assertion* sui valori, sui tipi e sull'esistenza delle variabili.

Come Scrivere le Asserzioni

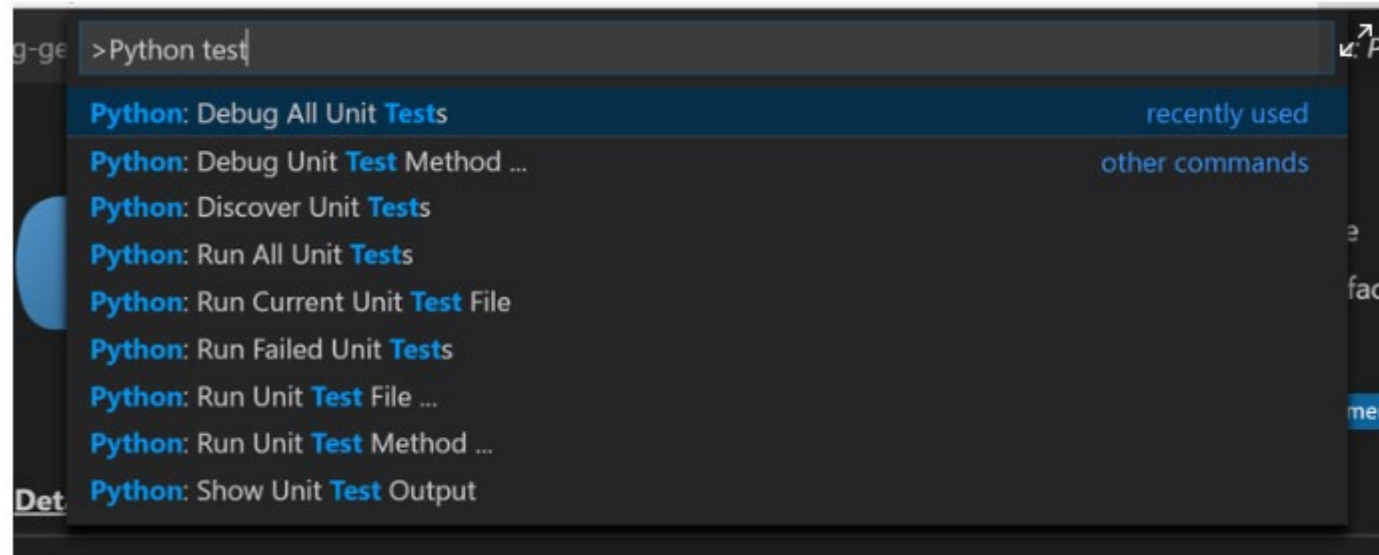
- I metodi più comunemente usati sono:

Method	Equivalent to
<code>.assertEqual(a, b)</code>	<code>a == b</code>
<code>.assertTrue(x)</code>	<code>bool(x) is True</code>
<code>.assertFalse(x)</code>	<code>bool(x) is False</code>
<code>.assertIs(a, b)</code>	<code>a is b</code>
<code>.assertIsNone(x)</code>	<code>x is None</code>
<code>.assertIn(a, b)</code>	<code>a in b</code>
<code>.assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>

- `.assertIs()`, `.assertIsNone()`, `.assertIn()` e `.assertIsInstance()` hanno tutti metodi opposti, denominati `.assertIsNot()` e così via.

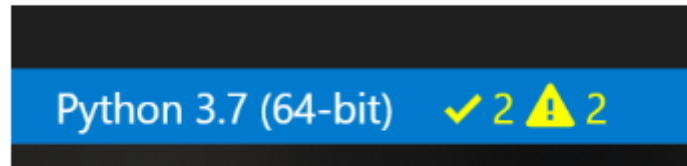
Eseguire i Test con VSCode

- Se utilizzi l'IDE di Microsoft Visual Studio Code, il supporto per l'esecuzione **unittest**, **nose** e **pytest** è integrato nel plug-in Python;
- Se hai installato il plugin Python, puoi impostare la configurazione dei tuoi test aprendo la palette dei comandi con Ctrl+Maiusc+P e digitando “Python test”. Vedrai una serie di opzioni:



Eseguire i Test con VSCode

- Scegliere *Debug All Unit Tests* e VSCode genererà quindi una richiesta per configurare il framework di test. Fare clic sull'ingranaggio per selezionare il test runner (**unittest**) e la home directory (.).
- Una volta impostato, vedrai lo stato dei tuoi test nella parte inferiore della finestra e potrai accedere rapidamente ai registri dei test ed eseguire nuovamente i test facendo clic su queste icone:



- Questo mostra che i test sono in esecuzione, ma alcuni stanno fallendo.



Oggetti Utili per i Cicli for

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

range()

- Già introdotti in precedenza, sono uno degli oggetti *built-in* spesso utilizzati assieme ai *for loops*;
- Se desideri simulare un ciclo for C++ basato sul contatore tradizionale, puoi iterare su un oggetto di tipo `range()`;

```
>>> for i in range(0, 10):  
...     print(i)
```

- Se esegui l'esempio sopra, scoprirai che `range(0, 10)` genera una sequenza di numeri da 0 (incluso) a 10 (escluso);
- Il primo parametro (start) è facoltativo e il valore predefinito è 0. `range(10)` è equivalente a `range(0, 10)`;

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

enumerate()

- A volte potresti aver bisogno sia dell'indice che dell'elemento corrente nell'elenco su cui stai iterando;
- `enumerate()` ti fornirà l'indice (a partire da 0) e l'elemento nell'elenco:

```
>>> top_hits = ["Dynamite", "WAP", "Holy", "Laugh Now Cry Later"]
>>> for (position, title) in enumerate(top_hits):
...     print(f"At number {position} we have {title}!")
...
At number 0 we have Dynamite!
At number 1 we have WAP!
At number 2 we have Holy!
At number 3 we have Laugh Now Cry Later!
```

- Come fare se volessimo partire a «contare» da 1?
- Puoi aggiungere 1 alla variabile contatore...

```
>>> for (position, title) in enumerate(top_hits):
...     print(f"At number {position + 1} we have {title}!")
...
```


enumerate()

- Oppure puoi passare ad **enumerate** un secondo argomento (facoltativo) per dirgli che l'indice deve partire da 1:

```
>>> for (position, title) in enumerate(top_hits, 1):  
...     print(f"At number {position} we have {title}!")  
...
```

zip()

- Il modo più facile per spiegare `zip()` è fare un esempio:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> print(list(zipped))
[(1, 4), (2, 5), (3, 6)]
>>> for (a, b) in zip(x, y):
...     print(f"{a}, {b}")
...
1, 4
2, 5
3, 6
```



List Comprehension

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

List Comprehension

- Potresti trovarti spesso a scrivere codice simile a quello che segue:

```
capitals = ["London", "Paris", "Rome", "Madrid"]
lengths = []
for c in capitals:
    lengths.append(len(c))
```

- In questi casi, Python consiglia di utilizzare la *list comprehension*, così da rendere il codice compatto e leggibile:

```
lengths = [len(c) for c in capitals]
```

List Comprehension

- Segue un altro esempio:

```
words = ["this", "list", "contains", "extremely", "lengthy", "expressions"]
short_words = []
for word in words:
    if len(word) <= 4:
        short_words.append(word)
```

- La versione che utilizza la *list comprehension* è molto più compatta (e forse anche più facile da leggere):

```
short_words = [word for word in words if len(word) <= 4]
```

List Comprehension

- Ecco alcune versioni più complicate della *list comprehension*:

```
[expr for var in list_name]  
[expr for var in list_name if condition]  
[expr for var1 in list_name1 for var2 in list_name2]  
[expr for var in list_name if condition1 if condition2]
```

- La linea guida generale è: usa la *list comprehension* quando questa rende il tuo codice più leggibile;

Set & Dict Comprehension

- La *comprehension* può essere applicata anche agli insiemi:

```
elements = [1, 2, 4, 2, 3, 2]
unique_elements = {element for element in elements}
```

- E ai dizionari:

```
tall_students = {key: value for (key, value) in height_dict.items() if value > 5}
```



Funzioni *Built-in*

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Funzioni Built-in

- Python fornisce molte utili funzioni integrate. Un elenco completo delle funzioni Python integrate è disponibile nella documentazione: <https://docs.python.org/3/library/functions.html>
- Abbiamo già visto alcune di queste funzioni: `len()`, `print()`, `isinstance()`, ecc.
- Python fornisce alcune funzioni matematiche integrate:

```
>>> print(sum([2, 3]))
>>> print(min([4, 1, 7]))
>>> print(max([4, 1, 7]))
>>> print(abs(-4))           # absolute value
>>> print(pow(2, 4))         # 2 to the power of 4
>>> print(round(5.6))        # round to nearest integer
>>> print(round(5.678, 2))   # round to nearest 2d.p.
```

Funzioni Built-in

- Potresti anche trovare `any()` e `all()` abbastanza utili!

```
>>> all_true = all([True, True, True, True])
>>> print(all_true)
True
>>> all_true = all([True, False, True, True])
>>> print(all_true)
False
>>> any_true = any([True, False, True, True])
>>> print(any_true)
True
```

Funzioni Built-in

- È possibile utilizzare le funzioni `sorted()` e `reversed()` per restituire una sequenza ordinata/invertita:

```
>>> numbers = [1, 5, 3, 7, 8, 2]
>>> sorted_numbers = sorted(numbers)
>>> print(sorted_numbers)
[1, 2, 3, 5, 7, 8]
>>> reversed_numbers = reversed(numbers) # reversed returns an `iterator`.
>>> print(list(reversed_numbers)) # you will need to convert it to a list if you need a list
[2, 8, 7, 3, 5, 1]
```

Funzioni Built-in

- Ricorda che tutto è un oggetto in Python, anche le funzioni!

```
>>> print(type(len))
<class 'builtin_function_or_method'>
>>> print(type(print))
<class 'builtin_function_or_method'>
```

- Ciò significa che puoi assegnare funzioni a una variabile, sia che si tratti della tua funzione o di una funzione *built-in*. Una specie di puntatore a funzione in C++, ma senza dover gestire i puntatori:

```
>>> my_len = len
>>> print(type(my_len))
>>> print(my_len([1, 2, 3]))
3
```



Eccezioni

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Eccezioni vs Errori di Sintassi

- Gli errori di sintassi si verificano quando il parser rileva un'istruzione errata:

```
>>> print( 0 / 0 )  
File "<stdin>", line 1  
    print( 0 / 0 )  
            ^  
SyntaxError: invalid syntax
```

- La freccia indica dove il parser ha riscontrato l'errore di sintassi. In questo esempio, c'era una parentesi di troppo. Rimuovila ed esegui di nuovo il codice:

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

- Questa volta, ti sei imbattuto in un'eccezione. Questo tipo di errore si verifica ogni volta che il codice Python sintatticamente corretto genera un errore;

Lanciare un Eccezione

- L'ultima riga del messaggio indica il tipo di eccezione in cui ti sei imbattuto;
- In questo caso, si tratta di un **ZeroDivisionError**. Python viene fornito con varie eccezioni integrate e la possibilità di creare eccezioni autodefinito;
- Possiamo usare **raise** per lanciare un'eccezione. L'istruzione può essere integrata con un messaggio personalizzato:

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

- Quando esegui questo codice, l'output sarà il seguente:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

- Il programma si ferma e mostra la nostra eccezione sullo schermo, offrendo indizi su cosa è andato storto.

L'eccezione AssertionError

- Invece di aspettare che un programma vada in crash a metà strada, è possibile eseguire un **assertion**;
- Ovvero, affermare che una certa condizione deve essere soddisfatta. Se questa condizione risulta essere effettivamente vera, allora tutto ok, il programma può continuare;
- Se la condizione risulta essere false il programma lancerà un'eccezione **AssertionError**:

```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

- Se esegui questo codice su una macchina Linux, l'asserzione ha esito positivo;
- Se dovessi eseguire questo codice su una macchina Windows, il risultato dell'asserzione sarebbe false e il risultato sarebbe il seguente:

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

- In questo esempio, lanciare un'eccezione **AssertionError** sarà l'ultima cosa che farà il programma prima di fermarsi. E se non fosse quello che vuoi?

Gestione delle Eccezioni – Try and Except

- Il blocco **try** e **except** in Python viene utilizzato per rilevare e gestire le eccezioni;
- Python esegue il codice seguendo l'istruzione **try** come flusso "normale" di esecuzione del programma;
- Il codice che segue l'istruzione **except** è la risposta del programma a qualsiasi eccezione che si verifica nella clausola **try** precedente;

```
def linux_interaction():  
    assert ('linux' in sys.platform), "Function can only run on Linux systems."  
    print('Doing something.')
```

- **linux_interaction()** può essere eseguita solo su un sistema Linux. L'asserzione in questa funzione genererà un'eccezione **AssertionError** se la chiami su un sistema operativo diverso da Linux;
- Puoi provare la funzione usando il seguente codice:

```
try:  
    linux_interaction()  
except:  
    pass
```

Gestione delle Eccezioni – Try and Except

```
try:
    linux_interaction()
except:
    pass
```

- In questo caso l'errore non viene gestito, ma solo ignorato. Se dovessi eseguire questo codice su una macchina Windows, otterresti il seguente output:

- Niente;
- La cosa buona qui è che il programma non è andato in crash, ma sarebbe bello vedere se si è verificato qualche tipo di eccezione ogni volta che hai eseguito il tuo codice;
- A tal fine, puoi modificare il **pass** in qualcosa che genererebbe un messaggio informativo, ad esempio:

```
try:
    linux_interaction()
except:
    print('Linux function was not executed')
```

Gestione delle Eccezioni – Try and Except

- Quello che non sei riuscito a vedere è stato il tipo di errore che è stato generato come risultato della chiamata di funzione;
- Per vedere esattamente cosa è andato storto, dovresti rilevare l'errore generato dalla funzione
- Il codice seguente è un esempio in cui acquisisci l'**AssertionError** e visualizzi il messaggio sullo schermo:

```
try:  
    linux_interaction()  
except AssertionError as error:  
    print(error)  
    print('The linux_interaction() function was not executed')
```

- L'esecuzione di questa funzione su un computer Windows restituisce quanto segue:

```
Function can only run on Linux systems.  
The linux_interaction() function was not executed
```

Gestione delle Eccezioni – Try and Except

- Ecco un altro esempio in cui si apre un file e si utilizza un'eccezione *built-in*:

```
try:
    with open('file.log') as file:
        read_data = file.read()
except:
    print('Could not open file.log')
```

- Se **file.log** non esiste, questo blocco di codice restituirà quanto segue:

```
Could not open file.log
```

- Leggendo la documentazione, possiamo facilmente notare che esiste l'eccezione *built-in* **FileNotFoundError**, quindi potrebbe anche scrivere:

```
try:
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

Gestione delle Eccezioni – Try and Except

- È possibile catturare diverse tipi di eccezione che potrebbero essere generate all'interno della clausola **try**, semplicemente utilizzando più volte la clausola **except**;
- In questo caso, l'esecuzione del blocco **try** – **except** terminerà appena verrà generata (e risolta) la prima eccezione:

```
try:
    linux_interaction()
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
except AssertionError as error:
    print(error)
print('Linux linux_interaction() function was not executed')
```

- **Attenzione:** la clausola **except** «nuda» cattura tutte le eccezioni, anche quelle inaspettate, quindi andrebbe sempre evitata,

La Clausola else

- In Python, usando l'istruzione **else**, puoi istruire un programma ad eseguire un certo blocco di codice solo in assenza di eccezioni:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    print('Executing the else clause.')
```

- Se dovessi eseguire questo codice su un sistema Linux, l'output sarebbe il seguente:

```
Doing something.
Executing the else clause.
```

La Clausola else

- Puoi anche provare inserire il blocco **try-except** all'interno della clausola **else** e per rilevare altre possibili eccezioni:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
```

- Se dovessi eseguire questo codice su una macchina Linux, otterresti il seguente risultato:

```
Doing something.
[Errno 2] No such file or directory: 'file.log'
```

La Pulizia Finale con `finally`

- Immagina di dover sempre implementare una sorta di azione per «ripulire» dopo aver eseguito il tuo codice. Python ti consente di farlo usando la clausola `finally`:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

- In sostanza, non importa se incontri un'eccezione da qualche parte nelle clausole `try` o `else`;
- L'esecuzione del codice precedente su una macchina Windows genererebbe quanto segue:

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```


Riassunto

- La clausola **try-catch** può essere riassunta come segue:

```
try:
    # Run this code
except:
    # Run this code when there is an exception
else:
    # No exceptions? Run this code
finally:
    # Always run this code
```



Iteratori e Generatori

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Iteratori

- In Python, un iteratore è un oggetto che consente di iterare su raccolte di dati, come elenchi, tuple, dizionari e insiemi;
- Gli iteratori Python consentono di attraversare un contenitore e accedere ai suoi elementi disaccoppiando gli algoritmi di iterazione dal contenitore;
- Gli iteratori si assumono la responsabilità di due azioni principali:
 - 🐍 **Restituzione** dei dati di un contenitore **un elemento alla volta**;
 - 🐍 Tenere traccia dell'elemento corrente corrente e di quelli già visitati;
- In sintesi, un iteratore conterrà ogni elemento di un contenitore (o di un flusso di dati) mentre esegue tutta le operazioni interne necessaria per mantenere lo stato del processo di iterazione;

Iteratori – Protocollo Python

- Un oggetto Python è considerato un iteratore quando implementa due metodi speciali:

Method	Description
<code>__iter__()</code>	Called to initialize the iterator. It must return an iterator object.
<code>__next__()</code>	Called to iterate over the iterator. It must return the next value in the data stream.

- Il metodo `__iter__()` di un iteratore in genere restituisce **self**, che contiene un riferimento all'oggetto corrente: l'iteratore stesso. Questo metodo è semplice da scrivere e, il più delle volte, assomiglia a questo:

```
def __iter__(self):  
    return self
```

Iteratori – Protocollo Python

- Il metodo `.__next__()` può essere più complesso o meno complesso a seconda di cosa stai cercando di fare con il tuo iteratore;
- Questo metodo deve restituire l'elemento successivo nel flusso di dati;
- Dovrebbe anche sollevare l'eccezione **StopIteration** quando non sono più disponibili elementi nel flusso di dati;
- Questa eccezione farà terminare l'iterazione: l'iterazione usa le eccezioni per il controllo di flusso!

Iteratori – Quando Usarli?

- Python utilizza gli iteratori per supportare ogni operazione che richiede l'iterazione, inclusi cicli *for*, *comprehension*, *iterable unpacking* e altro ancora;
- Quindi, gli iteratori vengono usati continuamente senza esserne consapevoli (almeno fino a questo momento);
- Nella programmazione quotidiana, gli iteratori sono utili quando devi eseguire l'iterazione su un set di dati o un flusso di dati con un numero sconosciuto di elementi;
- Questi dati possono provenire da fonti diverse, ad esempio il disco locale (file), un database e una rete;
- In queste situazioni, gli iteratori consentono di elaborare i dati un elemento alla volta, senza esaurire le risorse di memoria del sistema, che è una delle caratteristiche più interessanti degli iteratori

Creazione di Diversi Tipi di Iteratori

- Usando i due metodi che definiscono il protocollo iteratore nelle tue classi, puoi scrivere almeno tre diversi tipi di iteratori personalizzati. Puoi avere iteratori che:
 - 🐍 Prendono un flusso di dati e restituiscono elementi di dati così come appaiono nella sorgente;
 - 🐍 Prendono un flusso di dati, trasforma ogni elemento e restituiscono gli elementi trasformati;
 - 🐍 Non prendono alcun input, ma generano e restituiscono nuovi dati sulla base di specifici calcoli;
- Il primo tipo di iteratore è definito *classic iterator* perché implementa il modello di iteratore «originale»;
- Il secondo e il terzo tipo di iteratori aggiungono nuove funzionalità sfruttando il paradigma degli iteratori;

Restituire i Dati Originali

```
class SequenceIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            item = self._sequence[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

- Questo **SequenceIterator** prende una sequenza di valori al momento della creazione. Il costruttore della classe, `.__init__()`, crea gli attributi di istanza:

🐍 La sequenza di input;

🐍 Un attributo `._index` che verrà utilizzato per scorrere gli elementi.

Restituire i Dati Originali

```
class SequenceIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            item = self._sequence[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

- Il metodo `.__iter__()` restituisce l'oggetto corrente, `self`;
- Il metodo `.__next__()` definisce un'istruzione condizionale per verificare se l'indice corrente è inferiore al numero di elementi nelle sequenze. Questo controllo consente di interrompere l'iterazione quando i dati sono finiti, nel qual caso la clausola **else** solleverà l'eccezione **StopIteration**.

Restituire i Dati Originali

```
class SequenceIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            item = self._sequence[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

- Nella clausola `if` viene estratto e ritornato l'elemento corrente dalla sequenza originale, usando l'attributo `index`;
- L'attributo `._index` deve essere anche incrementato opportunamente;

Restituire i Dati Originali

- Ecco come funziona il tuo iteratore quando lo usi in un ciclo for:

```
>>> for item in SequenceIterator([1, 2, 3, 4]):  
...     print(item)  
...  
1  
2  
3  
4
```

Restituire i Dati Originali

- Prima di addentrarci in un altri esempi, vediamo come funzionano internamente i cicli for di Python. Il codice seguente simula il processo completo:

```
>>> sequence = SequenceIterator([1, 2, 3, 4])

>>> # Get an iterator over the data
>>> iterator = sequence.__iter__()
>>> while True:
...     try:
...         # Retrieve the next item
...         item = iterator.__next__()
...     except StopIteration:
...         break
...     else:
...         # The loop's code block goes here...
...         print(item)
...
1
2
3
4
```

Trasformare i Dati di Input Durante l'Iterazione

- Supponiamo ora di scrivere un iteratore che prende in input una sequenza di numeri, calcola il quadrato di ognuno e fornisce quei valori su richiesta. In questo caso, puoi possiamo scrivere la seguente classe:

```
class SquareIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            square = self._sequence[self._index] ** 2
            self._index += 1
            return square
        else:
            raise StopIteration
```

Trasformare i Dati di Input Durante l'Iterazione

- La prima parte di questa classe è identica alla precedente;
- Anche il metodo `.__next__()` è piuttosto simile. L'unica differenza è che prima di restituire l'elemento corrente, il metodo ne calcola il quadrato:

```
class SquareIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            square = self._sequence[self._index] ** 2
            self._index += 1
            return square
        else:
            raise StopIteration
```

Trasformare i Dati di Input Durante l'Iterazione

- L'utilizzo non cambia:

```
>>> from square_iter import SquareIterator  
  
>>> for square in SquareIterator([1, 2, 3, 4, 5]):  
...     print(square)  
...  
1  
4  
9  
16  
25
```

Trasformare i Dati di Input Durante l'Iterazione

- La possibilità di eseguire la trasformazione dei dati direttamente all'interno di un iteratore può rendere il tuo codice abbastanza efficiente in termini di consumo di memoria;
- Immagina per un momento che gli iteratori non esistessero. In tal caso, se si desidera eseguire iterazioni sui valori quadrati dei dati originali, è necessario creare un nuovo elenco per memorizzare i quadrati calcolati;
- In alternativa si potrebbe inserire il calcolo all'interno della routine più complessa che ne fa uso, ma questo rendere il codice meno leggibile (e quindi meno *Pythonic*) e richiederebbe di re-implementare la funzione che fa «il calcolo» tutte le volte che questa deve essere utilizzata;
- Ovviamente quello dei quadrati è solo un esempio, l'iteratore potrebbe fare cosa anche più sofisticate;
- Importante sottolineare che, se utilizzi un iteratore, il tuo codice richiederà solo memoria per un singolo elemento alla volta;
- L'iteratore calcolerà i seguenti elementi su richiesta senza archivarli in memoria;
- A questo proposito, gli iteratori sono detti *lazy objects*.

Generazione di Nuovi Dati Durante l'Iterazione

- È anche possibile creare iteratori personalizzati che generano un flusso di nuovi dati da un determinato calcolo senza prendere di dati come input;
- Di seguito è riportato un esempio per la generazione di numeri di Fibonacci:

```
class FibonacciIterator:
    def __init__(self, stop=10):
        self._stop = stop
        self._index = 0
        self._current = 0
        self._next = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < self._stop:
            self._index += 1
            fib_number = self._current
            self._current, self._next = self._next, self._current + self._next
            return fib_number
        else:
            raise StopIteration
```

Generazione di Nuovi Dati Durante l'Iterazione

- L'iterato appena visto non accetta di dati di input;
- Al contrario, genera ogni elemento eseguendo il calcolo che produce ogni volta nuovi valori dalla sequenza di Fibonacci;
- Questo calcolo viene effettuato all'interno del metodo `.__next__()`, sfruttando lo stato delle variabili `self._current` e `self._next`;
- Si avvia questo metodo con un condizionale che controlla se l'indice della sequenza corrente non ha raggiunto il valore `._stop`, nel qual caso si incrementa l'indice corrente per controllare il processo di iterazione. Quindi calcoli il numero di Fibonacci che corrisponde all'indice corrente, restituendo il risultato al chiamante di `.__next__()`.
- Quando `._index` cresce fino al valore di `._stop`, si genera `StopIteration`, che termina il processo di iterazione. Si noti che è necessario fornire un valore di arresto quando si chiama il costruttore della classe per creare una nuova istanza. L'argomento `stop` è predefinito a 10, il che significa che la classe genererà dieci numeri di Fibonacci se crei un'istanza senza argomenti.

Generazione di Nuovi Dati Durante l'Iterazione

- Ecco come puoi utilizzare la classe `FibonacciIterator` nel tuo codice:

```
>>> from fib_iter import FibonacciIterator

>>> for fib_number in FibonacciIterator():
...     print(fib_number)
...
0
1
1
2
3
5
8
13
21
34
```

Codifica di Iteratori Potenzialmente Infiniti

- Una caratteristica interessante degli iteratori Python è che possono gestire flussi di dati potenzialmente infiniti. Sì, possono infatti creare iteratori che producono valori senza mai raggiungere la fine! Per fare questo, è sufficiente saltare la parte di **StopIteration**;
- Ad esempio, supponi di voler creare una nuova versione della tua classe **FibonacciIterator** in grado di produrre numeri di Fibonacci potenzialmente infiniti. Per farlo è sufficiente:

```
class FibonacciInfIterator:
    def __init__(self):
        self._index = 0
        self._current = 0
        self._next = 1

    def __iter__(self):
        return self

    def __next__(self):
        self._index += 1
        self._current, self._next = (self._next, self._current + self._next)
        return self._current
```

Codifica di Iteratori Potenzialmente Infiniti

- Per verificare se il tuo `FibonacciIterator` funziona come previsto, esegui il seguente ciclo;
- Ma ricorda, sarà un ciclo infinito:

```
>>> from inf_fib import FibonacciInfIterator

>>> for fib_number in FibonacciInfIterator():
...     print(fib_number)
...
0
1
1
2
3
5
8
13
21
34
KeyboardInterrupt
Traceback (most recent call last):
...
```



Lambda, map, filter, reduce

Docenti:

Federico Bolelli <federico.bolelli@unimore.it>

Costantino Grana <costantino.grana@unimore.it>

Lambda

- La funzione identità, una funzione che restituisce il suo argomento, può essere definita come:

```
>>> def identity(x):  
...     return x
```

- L'equivalente utilizzando una lambda Python sarebbe:

```
>>> lambda x: x
```

- Nell'esempio precedente, l'espressione è composta da:

-  La parola chiave lambda

-  Una variabile x

-  Un corpo x

- Puoi scrivere un esempio più articolato ad esempio sommando 1 a x:

```
>>> lambda x: x + 1
```

Lambda

- Puoi eseguire la funzione lambda racchiudendo la funzione tra parentesi e specificando i suoi eventuali argomenti:

```
>>> (lambda x: x + 1)(2)
3
```

- Poiché una funzione lambda è un'espressione, le si può assegnare un nome;
- Quindi potresti scrivere il codice precedente come segue:

```
>>> add_one = lambda x: x + 1
>>> add_one(2)
3
```

- La funzione lambda sopra equivale a scrivere:

```
def add_one(x):
    return x + 1
```


Map

- Insieme a **Filter** e **Reduce** è una delle funzioni che «avvicinano» il Python all'approccio tipico della programmazione funzionale
- **Map** applica una funzione a tutti gli elementi in un **input_list**:

```
map(function_to_apply, list_of_inputs)
```

- Molto spesso ci capita di dover passare tutti gli elementi di un elenco ad a una funzione (uno per uno) e quindi di sfruttare l'output. Ad esempio:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

- Map ci consente di implementarlo ciò in modo molto semplice:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Filter

- Come suggerisce il nome, **Filter** crea un elenco di elementi per i quali una funzione specificata restituisce **True**. Ecco un esempio breve e conciso:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

- Il filtro assomiglia a un ciclo for, ma è una funzione incorporata ed è più veloce.

Reduce

- Riduci è una funzione davvero utile per eseguire calcoli su un elenco e restituire il risultato. Ad esempio, se si desidera calcolare il prodotto di un elenco di numeri interi, senza **Reduce** scriveremmo:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

- Con la **Reduce** possiamo scrivere:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```