\textbf{\huge Introduzione alla programmazione Python}

Prime nozioni su Python e Jupyter Notebook

Rights & Credits

Questo notebook è stato creato da Agostino Migliore; la parte iniziale sfrutta alcuni materiali di un notebook di:

- Simone Campagna (CINECA)
- Mirko Cestari (CINECA)
- Nicola Spallanzani (CNR-NANO)

La filosofia di Python

Come abbiamo visto, il linguaggio di programmazione Python mira alla massima semplicità, tanto da sembrare uno pseudo-codice. Esso consente

- semplice apprendimento
- semplice uso
- eccellente leggibilità
- eccellente portabilità.

Python è anche un linguaggio di *scripting* (per *script* si intende un insieme, di solito relativamente semplice, di istruzioni che vengono eseguite direttamente dal computer).

Python è un liguaggio moderno e completo con il quale si possono costruire programmi complessi. Lo si può utilizzare per produrre un programma di elaborazione scientifica, costruire un sito web, sviluppare un gioco, implementare machine learning e processi automatici, raccogliere dati da siti web, ecc.
Come abbiamo visto, è un linguaggio interpretato (abbiamo spiegato la funzione di un *interpreter* nella prima lezione) e lo si può definire ancora più precisamente come un *linguaggio dinamico*, in cui i controlli, la definizione dei tipi di dati, ecc. vengono effettuati al tempo dell'esecuzione (*runtime* ovvero *run-time*).

La comunità di utenti di Python è molto vasta e collaborativa. Di conseguenza, è facile trovare in rete la soluzione a qualsiasi eventuale problema che si presenta nell'utilizzo di Python.

Infine, Python è un linguaggio multi-paradigma, nel senso che permette l'implementazione di diverse strategie di programmazione: imperativa, funzionale, orientata agli oggetti (*object-oriented*), ecc. Capiremo meglio questo punto in seguito.

Modulo, Pacchetto e Libreria

Nell'usare un linguaggio di programmazione orientato agli oggetti e che si presta alla divisione in blocchi come Python è più che mai utile conoscere i concetti di modulo (*module*), pacchetto (*package*) e libreria (*library*).

Un modulo è un file che può contenere diverse istruzioni e funzioni (o *routines*) e può essere utilizzato da qualsiasi programma.

Un pacchetto è un set di moduli.

Una libreria di solito contiene un grouppo di moduli e pacchetti connessi. Tuttavia, spesso le parole libreria e pacchetto sono usate in modo intercambiabile.

Python ha una cosiddetta libreria standard. Per usarla bisogna importare i moduli necessari. Facciamo un esempio.

In [3]:
```
import math
print(math.pi)
```

```
3.141592653589793
```

Con la prima riga di comando di sopra abbiamo importato il modulo `math`, che fornisce accesso a moltissime funzioni matematiche. Con la seconda riga, abbiamo usato la funzione incorporata (*built-in*) `print` per stampare `math.pi`, che è il valore (costante) π = 3.14159... immagazzinato nella variabile `pi` definita nel modulo `math`.

Svantaggio

Python è un linguaggio di programmazione relativamente lento. Tuttavia bisogna considerare i due punti seguenti:

- ci sono approcci opportuni per velocizzare le parti più critiche dei codici;
- la velocità del codice in sé non sempre è il punto fondamentale: la gestione della complessità del codice, la semplicità della programmazione e la leggibilità del codice sono spesso punti non meno importanti per l'efficacia di un progetto informatico.

Creazione e gestione di ambienti virtuali con Python

Il modulo `venv`, disponibile a partire dalla versione 3.3 di Python, crea un ambiente virtuale attraverso l'istruzione

```
> python -m venv ENV
```

dove `ENV` è una directory o il path a una directory che conterrà l'ambiente virtuale. La struttura di questo ambiente virtuale è un po' diversa da quella dell'ambiente creato con conda, ma non per aseottu rilevanti.
Nell'istruzione di sopra, il simbolo `>` sta ad indicare il prompt dei comandi di Anaconda in Windows. La `m` dell'opzione sta per `module` e la riga di comando dice infatti di usare il modulo `venv` per creare il nuovo ambiente. Lavorando in un sistema Linux, `python` verrebbe sostituito da `python3`.

Prima di usare il l'ambiente `ENV`, bisogna attivarlo tramite la riga di comando

```
> ENV\Scripts\activate
```

Infatti, il file di attivazione, `activate` si trova nella sottocartella `Scripts` di ENV. Per disattivare l'ambiente virtuale, si usa l'istruzione

```
> ENV\Scripts\deactivate
```

Se uno si trovasse nel sistema operativo Linux piuttosto che in Windows, dovrebbe usare la riga di comando

```
$ source ENV/bin/activate
```

per attivare l'ambiente virtuale. Notate che abbiamo cambiato pure il simbolo che indica il prompt del terminale.

Installazione di pacchetti Python con pip

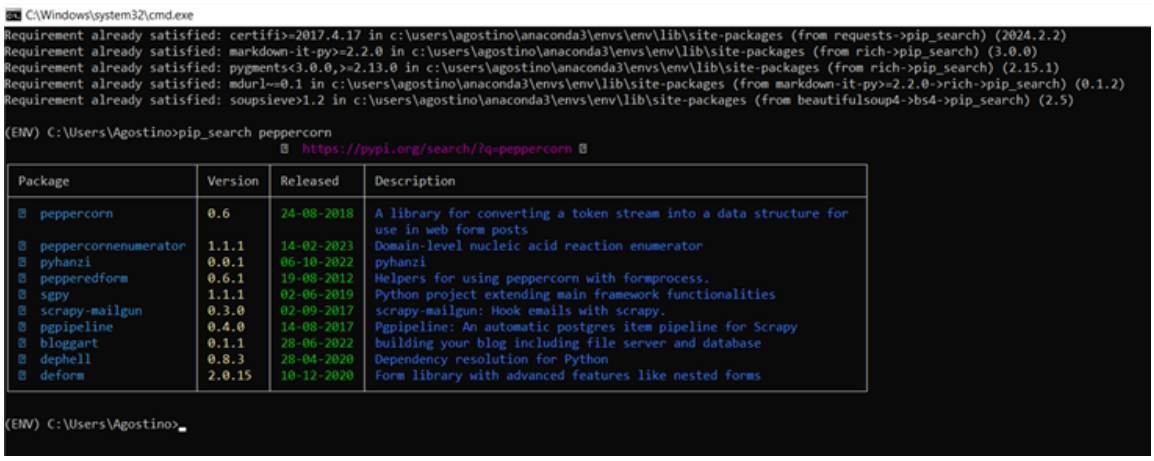
Ricerca e Installazione di pacchetti con pip

pip (preferred installer program) è lo strumento (*tool*) consigliato dalla Python Packaging Authority (PyPA) per installare pacchetti Python dal Python Packege Index (PyPI).

Prima di tutto usiamo `pip` per installare `pip_search`, che ci consente di cercare i pacchetti software e le loro versioni. Per farlo dal notebook (in modo tale da mostrarvi il risultato in questo documento), come al solito devo aggiungere il punto esclamativo prima di `pip` (voi fatelo dal prompt dei comandi, quindi senza punto esclamativo, e nell'ambiente virtuale che avete creato). L'istruzione di installazione risulta, quindi, come segue:

```
In [ ]: !pip install pip_search
```

(non mostro qui il risultato dell'installazione). Adesso usiamo il software appena installato dal prompt dei comandi per cercare una certa libreria `peppercorn` e vedere quali versioni sono disponibili. Vien fuori quanto segue:



Possiamo quindi procedere a installare `peppercorn` mediante l'istruzione

```
(ENV) > pip install peppercorn
```

che, di default, installa l'ultima versione disponibile (tra l'altro, l'unica visibile nella schermata di sopra) e anche i pacchetti ausiliari. Qualora fosse disponibile una versione precedente di `peppercorn`, diciamo la 0.4, e fossimo interessati a quella, potremmo usare la seguente riga di comando per installarla:

```
(ENV) > pip install peppercorn==0.4
```

Altrimenti, si può effettuare l'installazione usando `conda` come mostrato precedentemente.

Installazione di pacchetti scaricati dall'utente

Con `pip` è possibile installare anche pacchetti scaricati dall'utente. A tal fine, installiamo prima i programmi `unzip` e `wget`. Il primo ci consente di estrarre i materiali da pacchetti compressi. Il secondo consente di ricavare contenuti dal web.

```
(ENV) > pip install unzip
(ENV) > pip install wget
```

A questo punto, possiamo usare tali programi e pip come segue per scaricare e installare un certo contenuto dalla rete:

```
python -m wget URL/package_name.zip
python -m unzip package_name.zip
```

```
pip install -e package_name
```

L'opzione `-e` permette di installare il pacchetto in modalità editabile.

L'interprete di Python (Python interpreter)

Quando scriviamo uno codice in linguaggio Python e lo lanciamo, entra in azione l'interprete di Python (Python interpreter), cioè un programma che opportunamente converte (compila), run-time, le istruzioni del nostro script in linguaggio macchina. Supponiamo di scrivere un programma in linguaggio Python in un file e di dargli il nome `nome_programma.py` (si noti che viene usata l'estensione `.py` per i nomi di programmi Python). Lanciamo il programma con la riga di comando

```
(ENV) > python nome_programma.py
```

Nel momento in cui lo lanciamo, eseguiamo l'interprete di Python.
Creiamo adesso (giusto per esercitarci) il programma banale `somma_di_3_e_4.py` con il seguente contenuto:

```
S = 3 + 4
print(S)
```

Eseguendo la riga di comando

```
(ENV) > python somma_di_3_e_4.py
```

otteniamo sullo standard output del terminale o prompt il risultato

```
7
```

Si noti che per visualizzare il risultato sullo schermo (lo *standard output*) bisogna aggiungere l'istruzione di stampa, che fa uso della funzione print(). Tale funzione può prendere più argomenti. Per esempio, scriviamo il programma `somme.py` fatto come segue:

```
R = 1 + 3
S = 3 + 4
T = 7 + 12
print(R, S)
print(T)
```

Eseguendolo come sopra, otteniamo

```
4 7
19
```

Se nel programma `somme.py` avessimo scritto la prima istruzione di stampa nella forma

```
print(R, S, end=' ')
```

avremmo ottenuto

```
4 7 19
```

Per capirlo, approfondiamo un po' la nostra conoscenza della funzione built-in `print()`. Essa contiene anche un parametro `end` con il valore di default `'\n'`, che è un cosiddetto carattere di fuga (*escape character*). Si tratta di un carattere di controllo che dice all'interprete di Python di andare a capo. Così, la prima istruzione di stampa detta di stampare `4 7` e andare a capo, dove poi viene stampato il numero `19`. Con la modifica della prima istruzione di stampa di cui sopra, noi assegniamo il valore (argomento) spazio vuoto al parametro `end` (`end=' '`). Di conseguenza, adesso la prima istruzione di stampa dice all'interprete di scrivere `4 7` e poi aggiungere uno spazio vuoto nella stessa riga. Così, la seconda istruzione di stampa scriverà `19` sulla stessa riga. Ovviamente, abbiamo fatto questo esempio a fini istruttivi, ma averemmo potuto ottenere semplicemente lo stesso risultato con la singola istruzione di stampa

```
print(R, S, T)
```

Modalità interattiva

Nel semplice caso di sopra (e non solo), in cui dobbiamo sommare due numeri, è molto più semplice usare l'interprete in modo interattivo. A tal fine, eseguiamo prima

```
(ENV) > python
```

Notiamo subito una variazione del prompt, che assume la forma `>>>`. Ora i comandi sono passati direttamente all'interprete e non racchiusi in un programma da eseguire. In pratica, i comandi sono eseguiti man mano che vengono impartiti e, se un'espressione ha un valore, esso viene stampato. Per esempio,

```
>>> 3 + 4
7
```

Se un'espressione contiene un errore, l'errore viene segnalato ma l'interprete rimane attivo e continua a funzionare (non è come l'esecuzione di un programma che si ferma a seguito di un errore):

```
>>> 7/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Per uscire dall'interprete, si esegue il comando `exit()` o `quit()`.

IPython interpreter

Oltre al Python interpreter, esiste un interactive Python interpreter chiamato IPython. Esso è un Python interpreter con proprietà aggiuntive (per esempio, consente il completamento automatico dei comandi, contiene `comandi magici`, di cui parleremo, ecc.) e che funziona in modo interattivo, da linea di comando.

IPython può essere usato da terminale o prompt digitando `ipython` invece di `python`.

Noi lavoreremo molto spesso col notebook di Jupyter (Jupyter notebook), che usa IPython. In particolare, useremo l'ipykernel Python 3. Si noti che il nome `ipykernel` esprime il fatto che IPython è il programma cuore, nocciolo (*kernel*) che rende possibili la programmazione in linguaggio Python e l'esecuzione interattiva nei notebook di Jupyter.

Jupyter Notebook (e JupyterLab)

Jupyter è un progetto open-source creato per sostenere la scienza dei dati (*data science*) e il calcolo scientifico interattivi. Esso offre un ambiente basato sul web per lavorare con notebook che contengono codice, dati, grafici, ecc. I notebook di Jupyter sono lo spazio di lavoro (*workspace*) standard per la maggior parte degli scienziati che si occupano di gestione dei dati usando Python. Tali notebook possono essere agevolmente usati per creare e condividere codice, testo, equazioni e visualizzazioni. Google, Microsoft, IBM, ecc. usano ampiamente i notebook di Jupyter.

Come avviare Jupyter Notebook e JupyterLab

Jupyter Notebook e JupyterLab possono essere lanciati direttamente dalla finestra di Anaconda Navigator mostrata sopra (dove possono essere anche aggiornati cliccando il simbolo di ingranaggio floreale o, in ingelese, *flour gear*) oppure da uno dei suoi prompt, mediante le righe di comando

```
> conda jupyter notebook
```

e

```
> conda jupyter-lab
```

rispettivamente. Tuttavia, prima di connettersi, è bene creare un ambiente virtuale in cui si possa usare IPython e quindi Jupyter. Per esempio, chiamiamo l'ambiente virtuale `IENV`. Eseguiamo l'istruzione

```
python -m ipykernel install --user --name=IENV
```

A questo punto, aprendo Jupyter notebook (sui ci soffermeremo principalmente d'ora in poi), notiamo che tale ambiente virtuale è collocato in `C:\Users\Agostino\`; quindi, possiamo adesso creare i nostri notebook in the directory `C:\Users\Agostino\IENV`.

Widgets in Jupyter Notebook

Una volta aperto Jupyter Notebook, dal menu a discesa (*drop-down menu*) in alto a destra possiamo scegliere di aprire un nuovo notebook, un terminale, ecc. Soffermiamoci sul notebook. Scegliendo di aprire un notebook, ci viene chiesto quale ipykernel vogliamo usare. Poi possiamo rinominare il file con un clic sul nome `Untitled` assegnato inizialmente ad esso. L'estensione del nome del file è `ipynb`, dove, chiaramente, `ipy` sta per `IPython` e `nb` per `notebook`. Mentre i files con estensione `.py` sono file di testo standard che contengono codice scritto in linguaggio Python, i file con estensione `.ipynb` sono IPython notebook file che si possono appoggiano al web e possono contenere codice unitamente ai risultati della sua escuzione, a grafici e altro.

I notebook possono essere arricchiti importando `widgets` (*widget* è un termine inglese che in italiano significa *aggeggio*) come segue:

In [1]:
```
import ipywidgets as widgets
```

Nell'istruzione di sopra, `as widgets` permette di chiamare `ipywidgets` col nome abbreviato `widgets` e avremmo potuto scegliere anche un qualsiasi nome più breve.

I Jupyter widgets sono componenti di software interattivi che consentono di controllare l'esecuzione e la visualizzazione di oggetti Python (cioè creati col linguaggio Python) in notebook di Jupyter. Consideriamo l'esempio di sotto, senza entrare, per il momento, nel dettaglio della sintassi di una funzione, ma sfruttandone l'aspetto intuitivo nel linguaggio Python, come abbiamo fatto nella prima lezione.

In [2]:
```
def f(x):
    return x**2
widgets.interact(f, x=(0,4,0.5));
```

```
interactive(children=(FloatSlider(value=2.0, description='x', max=4.0, step=0.5), Ou
tput()), _dom_classes=('wi…
```

La funzione `interact` di `widgets` genera strumenti di controllo che consentono all'utente di interfacciarsi attivamente col notebook, per controllare gli argomenti passati a una funzione e invocare la funzione con quegli argomenti. Sopra, infatti, appare un cursore (*slider*) grazie al quale l'utente può selezionare il valore di `x` (in altri termini, decidere, avere il controllo del valore di `x`) e richiedere alla funzione `f` di renderci (*return*) il suo valore corrispondente.

Passando `f` come primo argomento, `interact` genera il cursore e lega il valore sul cursore al parametro della funzione: l'argomento passato alla funzione sarà quello selezionato col cursore. Il secondo argomento passato ad `interact` indica il range di valori di x, che va da 0 a 4, e il terzo argomento rappresenta l'incremento tra un valore e l'altro, che è 0.5 in questo caso.

Per riutilizzare in seguito i widgets o accedere ai dati connessi, si può usare la funzione `interactive`. Tuttavia, usando tale funzione, i valori ottenuti dalla funzione non sono mostrati direttamente e bisogna usare la funzione `display` di `IPython` per visualizzare l'output della funzione corrispondente al valore di `x` selezionato dall'utente, come mostrato sotto.

In [3]:
```
def f(x):
    return display(x**2)
widgets.interactive(f, x=(0,4,0.5))
```

Out[3]:
```
interactive(children=(FloatSlider(value=2.0, description='x', max=4.0, step=0.5),
Output()), _dom_classes=('wi…
```

Nell'esempio seguente, usiamo una funzione della libreria grafica di Python chiamata **Matplotlib** (ne parleremo in seguito) per raggiungere il nostro obiettivo.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def power(n):
    x = np.array([0,1,2,3,4,5,6])
    y = x**n+5
    return plt.plot(x,y,'-o')
    # plt.show()
widgets.interactive(power, n=(0,7))
```

Out[2]:
```
interactive(children=(IntSlider(value=3, description='n', max=7), Output()), _dom_
classes=('widget-interact',)…
```

Markdown, codici e altro in Jupyter Notebook

Il *markdown* ha regole sintattiche molto semplici. Per esempio, nella prima riga di questa cella abbiamo usato tre simboli di diesis per creare un secondo livello di sottotitolo a partire dal titolo principale, introdotto per mezzo di un solo simbolo di diesis.
Racchiudendo una parola tra asterischi, la si scrive in corsivo; racchiudendola tra due doppi

asterischi, la si rende in grassetto. Alla pagina web learn markdown language si trova un tutorial sull'uso del *markdown*. Il collegamento ipertestuale sul quale potete cliccare per andare al tutorial è, esso stesso, un esempio delle cose che si possono fare con il *markdown*. Potete imparare molte altre cose sul *markdown* confrontando i formati `.pdf` e `.ipynb` di questo documento e di quelli con le altre lezioni.

Si possono scrivere codici nelle celle di un notebook ed eseguirli. Quello di sotto è un esempio di un codice che consiste di una sola istruzione.

In [9]:
```
3+4
```

Out[9]:
```
7
```

Abbiamo fatto la stessa cosa che avremmo potuto fare usando il Python interpreter in modo interattivo su un prompt o su un terminale.

Si può usufruire dell'autocompletamento dei comandi. Inoltre, pressando `shift + tab` lentamente e ripetutamente dopo aver posizionato il cursore alla fine del nome di una funzione (o comando), appaiono informazioni inerenti alla funzione a uno più livelli crescenti di dettaglio. Questo è un tipo di *introspezione* in termini informatici, cioè di capacità di esaminare le proprietà inerenti ad un oggetto. Provate per esempio con `sum`. Un altro modo di ottenere tali informazioni su un oggetto consiste nell'usare la funzione incorporata help() di Python, fornendo come argomento l'oggetto in questione:

```
help(object)
```

Sotto mostriamo un esempio in cui vogliamo avere informazioni sulla funzione incorporata `sum`.

In [10]:
```
help(sum)
```

```
Help on built-in function sum in module builtins:

sum(iterable, /, start=0)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

Concludiamo questa sezione con qualche considerazione sulla stampa nel notebook. Supponiamo di definire una variabile assegnandole, come valore, una stringa di caratteri:

In [11]:
```
A = 'John'
```

Vediamo che

In [12]:
```
A
```

Out[12]:
```
'John'
```

mentre

In [13]:
```
print(A)
```

```
John
```

stampa il valore intrinseco della variabile, il nome *John*, senza l'indicazione, tramite apici, che si tratta di una stringa di caratteri.

Funzioni (o comandi) magici in Jupyter Notebook

IPython mette a disposizione dell'utente diverse funzioni magiche.
Nel contesto informatico, *magic* è un termine informale per *astrazione* ed è usato con riferimento a una funzione/comando (ovvero al codice che vi sta dietro) in grado di gestire un compito complesso nascondendone la complessità e presentando, invece, una semplice interfaccia per il suo uso da parte dell'utente.
Potete ottenere una lista delle funzioni magiche disponibili nel notebook eseguendo la riga di comando seguente:

In [ ]:
```
%quickref
```

Consideriamo, per esempio, la funzione magica `%run`. La riga di comando

```
%run nome_file.py
```

consente di lanciare il file `nome_file.py` come un programma nell'ambito dell'IPython di Jupyter. Per fare un esempio di utilizzo di tale comando magico, vediamo prima se vi è un file di programma Python al quale possiamo applicarlo nel presente direttorio:

In [27]:
```
!dir *py
```

```
 Il volume nell'unità C è OS
 Numero di serie del volume: 26B8-B399

 Directory di C:\Users\Agostino\Desktop\New_Life\insegnamento\insegnamento_2023_2024
\metodi_di_calcolo_2023_2024\Agostino_Migliore_MCC_lectures_2024

27/01/2024  00:51            32,868 01-Introduction-Environment-Interpreter.ipynb
03/03/2024  05:32           492,535 2_Introduzione_ad_Anaconda_e_Conda.ipynb
04/03/2024  03:46           262,902 3_prime_nozioni_su_Python_e_Jupyter.ipynb
               3 File        788,305 byte
               0 Directory  116,782,198,784 byte disponibili
```

Non ne vediamo. Tuttavia, nella directory `C:\Users\Agostino` si trova il programma `somme.py` che abbiamo discusso prima. Quindi, includendo il `path` al programma nella

riga di comando, la eseguiamo come

```
In [28]: %run C:\Users\Agostino\somme.py
```

```
4 7
19
```