

Python 3

Laboratorio di Intelligenza Artificiale

Vincenzo Bonnici

Corso di Laurea Magistrale in Scienze Informatiche

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Università degli Studi di Parma

2025-2026

Perchè python?

- E' un linguaggio **easy to use**
- fornisce **costrutti** sia semplice che sofisticati, in base alle necessità dell'utente
- fornisce delle **strutture dati native** efficienti, e anche un sempre approccio **OOP**
- è **interpretato** ed è **portatile**, ma può anche essere **compilato**
- si presenta anche con interfacce interattive: **IPython** o **Jupyter**
- è al momento il linguaggio principalmente usate per il **machine learning**

Attenzione: noi non utilizzeremo **Python2** bensì **Python3** che **NON** è **retrocompatibile**, es: `python2 print something`; `python3 print(something)`

Come usare python3?

- lanciare l'**interprete** python su uno **script** scritto in precedenza
`python3 myscript.py`
- usare python in modo interattivo
 - il comando `python3` apre un interprete **interattivo** da linea di comando
 - `ipython3` è una versione più **usabile**
 - `jupyter` consente di scrivere nei **notebook** che contengono **celle** interattive di vario tipo: es. **codice** e **markdown**
 - **PERICOLO**: l'ordine degli elementi (celle) del notebook potrebbe non essere quello di interpretazione ed esecuzione del codice scritto in essi

In questo corso utilizzeremo **Jupyter**, in particolare la versione **lab**, per la creazione di **notebook** dentro cui scriveremo il codice e faremo le nostre analisi.

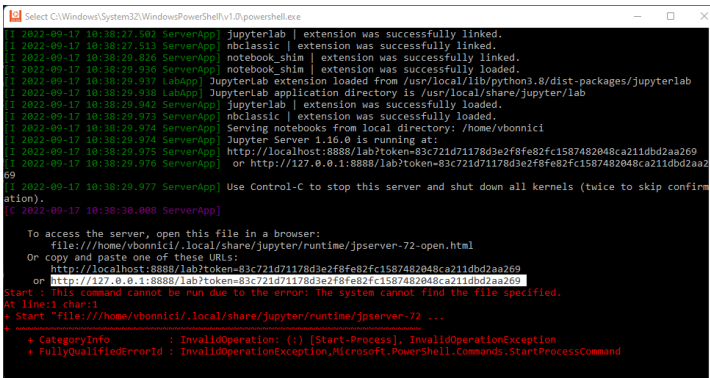
Jupyter-lab è disponibile sui principali sistemi operativi. Tuttavia, le indicazioni fornite per questo corso riguardano distribuzioni Ubuntu fornite tramite **bash on windows**.

La documentazione ufficiale di **Jupyter** è disponibile al sito <https://jupyter.org/>, mentre quella specifica della versione **Jupyter-lab** è al sito <https://jupyterlab.readthedocs.io/en/stable/>.

Per una installazione veloce:

```
sudo apt-get update
sudo apt install python3
sudo apt install python3-pip
sudo pip install jupyterlab
```

Per eseguire lanciare il comando `jupyter-lab` ed aprire il browser al link specificato:



```
Select C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
I 2022-09-17 10:38:27.502 ServerApp] jupyterlab | extension was successfully linked.
I 2022-09-17 10:38:27.513 ServerApp] nbclassic | extension was successfully linked.
I 2022-09-17 10:38:29.826 ServerApp] notebook_shim | extension was successfully linked.
I 2022-09-17 10:38:29.936 ServerApp] notebook_shim | extension was successfully loaded.
I 2022-09-17 10:38:29.937 LabApp] JupyterLab extension loaded from /usr/local/lib/python3.8/dist-packages/jupyterlab
I 2022-09-17 10:38:29.938 LabApp] JupyterLab application directory is /usr/local/share/jupyter/lab
I 2022-09-17 10:38:29.942 ServerApp] jupyterlab | extension was successfully loaded.
I 2022-09-17 10:38:29.973 ServerApp] nbclassic | extension was successfully loaded.
I 2022-09-17 10:38:29.974 ServerApp] Serving notebooks from local directory: /home/vbonnici
I 2022-09-17 10:38:29.974 ServerApp] JupyterLab application directory is /usr/local/share/jupyter/lab
I 2022-09-17 10:38:29.975 ServerApp] Jupyter Server 1.16.0 is running at:
I 2022-09-17 10:38:29.976 ServerApp] http://localhost:8888/lab?token=83c721d71178d3e2f8fe82fc1587482048ca211dbd2aa269
I 2022-09-17 10:38:29.976 ServerApp] or http://127.0.0.1:8888/lab?token=83c721d71178d3e2f8fe82fc1587482048ca211dbd2aa269
I 2022-09-17 10:38:29.977 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirm
[C 2022-09-17 10:38:30.008 ServerApp]
To access the server, open this file in a browser:
file:///home/vbonnici/.local/share/jupyter/runtime/jpserver-72-open.html
Or copy and paste one of these URLs:
http://localhost:8888/lab?token=83c721d71178d3e2f8fe82fc1587482048ca211dbd2aa269
or http://127.0.0.1:8888/lab?token=83c721d71178d3e2f8fe82fc1587482048ca211dbd2aa269
Start: This command cannot be run due to the error: The system cannot find the file specified.
At line:1 char:1
+ Start "file:///home/vbonnici/.local/share/jupyter/runtime/jpserver-72 ...
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Start-Process], InvalidOperationException
+ FullyQualifiedErrorId : InvalidOperationException,Microsoft.PowerShell.Commands.StartProcessCommand
```

In python il **tipo** di una **variabile** può essere:

- forzato dall'utente in fase di **dichiarazione** oppure tramite **casting esplicito**
- inferito tramite **contesto** (questo permette all'interprete di non necessitare dichiarazione esplicita di una variabile!)
- il **valore nullo** è definito dalla parola chiave **None**

Tipi built-in (primitivi) di variabile:

- `int(a [,base])` interi, es `10`, `0xFF`, `0b110101`
- `float(a)` numeri in virgola mobile, es `1.123` oppure `1.82e308`
- `chr(a)` singolo carattere, es `'a'` (non corrispondono agli interi)
- `str(a)` stringa di caratteri, es `'stringa'`, `"stringa"` o multilinea aprendo e chiudendo con `"""`
- `bool(a)` booleani, che possono essere **True** o **False**
- `complex(a [,image])` numeri complessi

Python3 - funzioni primitive

Stampare `print(a1, [a2, a3, ...], sep='\t ', end='\n ')`

```
>>>a = 1
>>>print(a,2,sep='-', end='|')
1-2|
```

Leggere da standard input (console):

```
>>>a = input('please input the value of a')
5
>>>print(a)
5
```

Valutare espressioni aritmetiche:

```
>>>a = eval(input('please input the value of a'))
5 + 5
>>>print(a)
10
>>>eval('5+5')
10
```

Operatori **numerici**:

- `+, -, *, /` classici operatori aritmetici
- `%` modulo, ovvero resto della divisione tra due numeri
- `//` quoziente della divisione
- `**` esponente, es `2**4 = 16`

Operatori **bitwise** (bit a bit)

- `&, |, ^` rispettivamente AND, OR e XOR
- `~` complemento a 1 (inverte i bit)
- `<<., >>.` shift binario a sinistra e destra di n bit, es `0b1 << 2 = 100`

Operatori di **assegnamento**

- `=, +=, -=, *=, /=, **=, &=, ...`

NON ci sono gli operatori di **incremento** (e decremento) `++` e `--`

Operatori di **confronto**:

- `==` **True** se e solo se i due operandi sono **uguali**, altrimenti **False**
 - **ATTENZIONE**: in python ci sono i puntatori (nascosti) e a volte questo operatore confronta puntatori e non valori.
- `!=` **True** se e solo se i due operandi **NON** sono **uguali**
- `<, >, <=, >=` classici operatori di confronto sull'**ordine**

Operatori **logici**:

- `and, or, not` sono utilizzati come operatori logici nelle espressioni booleane

```
>>> a, b, c = True, True, False # NOTE this is a multiple assignment
>>> print( (a and b) or not c )
True
```

Python è molto simile a Java. I tipi di dato primitivi sono passati per copia, mentre gli altri tipi di variabili/oggetti sono passati per riferimento.

Data una variabile qualsivoglia `x`, la funzione primitiva `id(x)` ritorna il puntatore (indirizzo fisico) all'oggetto, ovvero:

the identity of the object (which is a unique integer for a given object).

Operatori **identità**:

- `is`, `is not` confrontano le locazioni di memoria (i riferimenti) di due variabili/oggetti tramite `id`

Python3 - operatori su stringhe

In python le stringhe sono oggetti **immutabili**, ovvero il loro contenuto non può essere modificato.

Concatenazione tra stringhe `+`

```
>>>'hello' + 'world'  
'hello world'
```

Ripetizione `*`

```
>>>'AG' * 3  
'AGAGAG'
```

Lunghezza `len(str)`

```
>>>len('AGAGAG')  
6
```

Python3 - operatori su stringhe

L'operatore di **slicing** serve a ottenere sottostringhe (o sottosequenze).

`s[index]` per estrarre una **singola posizione**

```
>>>s='my string'  
>>>s[0]  
'm'
```

`s[start:end]` per estrarre una **sottostringa**, dalla posizione `start` alla posizione `end`

```
>>>s[0:5]  
'my st'
```

`s[start:end:step]` per estrarre una **sottosequenza**, dalla posizione `start` alla posizione `end` saltando di `step` posizioni alla volta

```
>>>s='0123456789'  
>>>s[0:len(s):3]  
'0369'
```

Python3 - operatori su stringhe

Alcuni esempi utili sull'operatore `slice`.

Le posizioni possono essere negative. Ad esempio si può estrarre l'ultimo elemento in questo modo

```
>>>s='0123456789'  
>>>s[-1]  # equivalent to s[-1:] or s[len(s) - 1:len(s)]  
'9'
```

Oppure per avere la stringa senza l'ultimo elemento

```
>>>s[:-1]  
'012345678'
```

Oppure per leggere la stringa al contrario

```
>>>s[::-1]  
'9876543210'
```

Python3 - funzioni built-in stringhe

`str.find(sub[, start, end])` ritorna la posizione più piccola posizione dove occorre la sottostringa `sub` nell'intervallo `[start, end]`.

`str.index(sub[, start, end])` simile a `find`, ma se non esiste almeno una occorrenza allora torna un `ValueError`.

`str.replace(old, new [,count=1])` torna una copia della stringa dove le prime `count` (di default solo la prima) occorrenze della sottostringa `old` sono sostituite con al sottostringa `new`.

`str.count(sub[, start, end])` torna il numero di occorrenze della sottostringa `sub` nella stringa `str`.

ATTENZIONE: ricerca e sostituzione non prevedono overlap:

```
>>>s = 'TTTATATATTT'  
>>>print(s.replace('ATA','N'))  
TTTNTATTT  
>>>print(s.count('ATA'))
```

1

Python3 - funzioni built-in stringhe

`str.upper()`, `str.lower()` ritornano una copia della stringa in maiuscole, minuscole.

`str.isupper()`, `str.islower()`

`str.isalpha()`, `str.isdigit()`, `str.isprintable()`, ...

`str.strip([chars])` ritornano una copia della stringa a cui sono tolti i caratteri `chars` (di default i bianchi) a inizio e fine stringa.

```
>>>s = '   TTTATATATTT\t\n'
```

```
>>>print(s)
```

```
   TTTATATATTT
```

```
>>>print(s.strip())
```

```
TTTATATATTT
```

`str.startswith(sub)`, `str.endswith(sub)` ritornano `True` se la stringa inizia (finisce) con la sottostringa `start` (`end`)

Python3 - costrutti di controllo del flusso

In python i blocchi di codice non sono delimitati da **parentesi**.
Inoltre, il corpo di un costrutto è identificato tramite **indentazione**.

```
if <condition>:  
    <code block>  
elif <condition>:  
    <code block>  
else:  
    <code block>
```

`condition` è una espressione booleana.

`<code block>` può essere una o più righe di codice.

Per avere un corpo vuoto si deve usare `pass`.

```
if a>b:  
    print(a)  
elif a<b:  
    pass  
else:  
    print(b)  
    b += 1
```


Il costrutto `while`

```
while <condition>:  
    <code block>  
else:  
    <code block>
```

Il corpo del costrutto `else` viene eseguito non appena la condizione del `while` diventa `False`, a meno che non si esce dal `while` con una istruzione di `break`.

```
>>>a,i = 3,0  
>>>while i < a :  
>>>     print(i,' ', sep='',end='')  
>>>     i += 1  
>>>else:  
>>>     print('finish')  
0 1 2 finish
```

Il costrutto `for`

```
for object in list:
    <code block>
else:
    <code block>
```

```
>>>for i in range(3):
>>>     print(i,sep=' ', end='')
0 1 2
```

La funzione `range` permette di generare liste di numeri *al volo*

```
range([start=0, ] stop [, step=1])
```

al volo vuol dire che `range` costruisce un **iteratore** ad una lista virtuale.

Infatti, `range(1000)` non costruisce una lista di 1000 elementi, ma itera in modo virtuale tra i numeri da 0 a 999.

Python fornisce alcune strutture dati built-in che sono a disposizione del programmatore senza caricare librerie aggiuntive.

Le più importanti sono `list`, `tuple`, `set`, `frozenset` e `dict`.

`list` e `tuple` sono sequenze ordinate di oggetti eterogenei (oggetti di diverso tipo all'interno della stessa struttura dati).

`list` e `tuple` sono indicizzate tramite indici numerici, come degli array (non esistono array built-in).

`list` è una lista mutabile, `tuple` è immutabile.

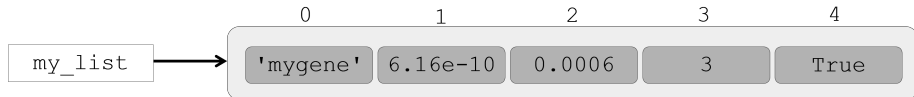
`set` è un insieme (non ordinato) di oggetti eterogenei.

`frozenset` è un insieme immutabile, che permette prestazioni migliori rispetto a `set`.

`dict` (dizionario) è una mappa mutabile di elementi eterogenei del tipo chiave-valore.

Una lista è una sequenza di valori, possibilmente di tipo eterogeneo, ordinata e mutabile.

```
my_list = ['mygene', 6.16e-10, 0.0006, 3, True]
```



`my_list` è un **riferimento** alla lista reale e può essere usato per accedere agli elementi della lista tramite l'operatore di slicing.

```
>>>print(my_list[0])  
'mygene'  
>>>print(my_list[-1])  
True
```

L'operatore di slicing è simile a quello utilizzato per le lista con l'eccezione che può essere utilizzato per **modificare** il contenuto della lista.

```
>>>print(my_mylist[0:2])
['mygene', 6.16e-10]

>>>my_list[4] = False
>>>print(my_list)
['mygene', 6.16e-10, 0.0006, 3, False]
```

Ogni operazione di slicing non di assegnamento crea una **nuova copia** della lista ottenuta copiando i valori estratti dalla lista originaria

```
>>>my_list_2 = my_list[:]
>>>print(my_list_2)
['mygene', 6.16e-10, 0.0006, 3, False]
>>>my_list[4] = True
>>>print(my_list)
['mygene', 6.16e-10, 0.0006, 3, True]
>>>print(my_list_2)
['mygene', 6.16e-10, 0.0006, 3, False]
```

Operatori principali per le liste:

`len(lst)` lunghezza della lista.

`[lst1] + [lst2]` concatenazione.

`[lst]*nof_times` ripetizione, es. `[1,2]*3` \mapsto `[1,2,1,2,1,2]`.

`lst.append(x)` aggiunge un elemento in coda a `lst`.

`lst.extend(lst2)` appende tutti gli elementi di `lst2` in coda a `lst` (concatenazione).

`lst.insert(position,x)` inserisce il valore `x` in posizione `position`.

`lst.clear()` pulisce la lista.

`lst.count(x)` conta il numero di occorrenze di `x` nella lista.

`lst.reverse()` restituisce una copia della lista con gli elementi in ordine inverso.

Operatori principali per le liste (continua):

`lst.remove(x)` rimuove la prima occorrenza di `x`. Lancia un errore se `x` non è in `lst`.

`lst.pop([position])` rimuove e restituisce l'elemento alla posizione `position` che di default è la coda della lista.

operatore `del`

L'operatore `del` elimina una particolare variabile.

Nel caso delle liste può essere utilizzato per eliminare un singolo elementi dalla lista o per eliminare una intera porzione grazie alla congiunzione con l'operatore di slicing.

`del lst[3]` equivale a `lst = lst[0:3] + lst[4:]`. Tuttavia ci sono delle considerazioni in termini di memoria utilizzata e passaggi nascosti effettuati. Infatti `lst = lst[0:3] + lst[4:]` genera due liste per copia e poi le concatena. Invece, `del lst[3]` non genera copie.

Una tupla è una sequenza ordinata immutabile di elementi.

```
>>>my_tuple = ('mygene', 6.16e-10, 0.0006, 3, False)
>>>print(my_tuple)
('mygene', 6.16e-10, 0.0006, 3, False)
```

Valgono gli stessi operatori delle liste tranne quelli per la modifica.

```
>>>print(my_tuple[0])
'mygene'

>>>print(my_tuple[0:2])
('mygene', 6.16e-10)

>>>print( (1,2)+(3,4) )
(1,2,3,4)
```


Python3 - strutture dati innestate e loro passaggio

Le **tuple** sono simili a tipi di dati primitivi, quindi vengono sempre gestite **per copia** e mai per riferimento. A differenza delle tuple, le **liste** invece sono sempre gestite **per riferimento**.

Possiamo creare liste di tuple

```
>>>my_list = [ (1,2), (3,4), (5,6) ]
```

così come possiamo creare liste di liste (non esistono le matrici built-in)

```
>>>my_list = [ [1,2], 3, 4 ]
```

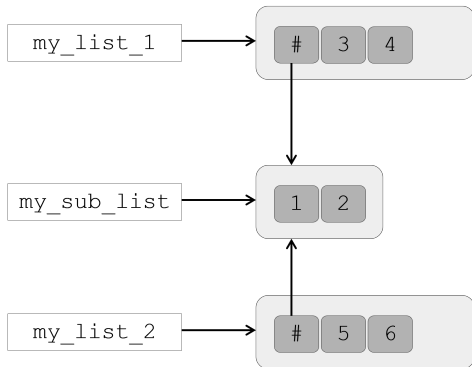
ma dobbiamo stare attenti a quando le maneggiamo:

```
>>>my_sub_list = [1,2]
>>>my_list_1 = [ my_sub_list, 3, 4]
>>>my_list_2 = [ my_sub_list, 5, 6]
```

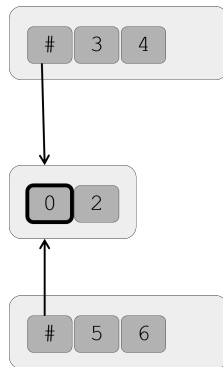
```
>>>my_sub_list[0] = 0
>>>print(my_list_1)
[[0,2],3,4]
>>>print(my_list_2)
[[0,2],5,6]
```

Python3 - strutture dati innestate e loro passaggio

```
>>>my_sub_list = [1,2]
>>>my_list_1 = [ my_sub_list, 3, 4]
>>>my_list_2 = [ my_sub_list, 5, 6]
>>>my_sub_list[0] = 0
```



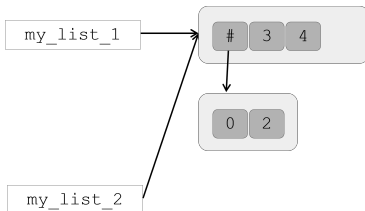
`my_sub_list[0] = 0`



Python3 - strutture dati innestate e loro passaggio

L'operatore di assegnamento delle liste, che utilizza l'operatore `copy` (`lst.copy()` o `copy(lst)`) è uno **shallow copy** (copia poco profonda). questo vuol dir che vengono copiati i riferimenti/puntatori ma non viene copiato in modo ricorsivo il contenuto della variabili da essi puntati.

```
>>>my_list_1 = [ [1,2], 3, 4]
>>>my_list_2 = my_list_1
>>>my_list_1[0][0] = 0
>>>print(my_list_1)
[[0,2],3,4]
>>>print(my_list_2)
[[0,2],3,4]
```



Python3 - strutture dati innestate e loro passaggio

Python fornisce un operatore addizionale per la **deep copy**, che deve essere importato dalla libreria `deepcopy`.

```
>>>from copy import deepcopy
>>>my_list_2 = deepcopy(my_list_1)
>>>my_list_1[0][0] = 0
>>>print(my_list_1)
[[0,2],3,4]
>>>print(my_list_2)
[[1,2],3,4]
```



Un **insieme** è un **collezione non ordinata** di elementi eterogenei che **non ammette duplicati**.

```
>>>my_set = {'DNA repair', 2, 0.003}
>>>print(my_set)
{'DNA repair', 2, 0.003}

>>>my_set = set()
>>>my_set.add('DNA repair')
>>>my_set.add(2)
>>>my_set.add(0.003)
>>>print(my_set)
{'DNA repair', 2, 0.003}

>>>my_set.remove(0.003)
>>>print(my_set)
{'DNA repair', 2}
```

Un **insieme** è un **collezione non ordinata** di elementi eterogenei che **non ammette duplicati**.

```
>>>my_set = {'DNA repair', 2, 0.003}
>>>print(my_set)
{'DNA repair', 2, 0.003}
```

Aggiungere elementi

```
>>>my_set = set()
>>>my_set.add('DNA repair')
>>>print(my_set)
{'DNA repair'}
```

Rimuovere elementi

```
>>>my_set = {2, 'DNA repair', 0.003}
>>>my_set.remove(0.003)
>>>print(my_set)
{'DNA repair', 2}
```

```
>>>A = { 1, 2, 3}  
>>>B = { 1, 4, 5}
```

Unione (OR)

```
>>>A | B  
{1, 2, 3, 4, 5}
```

Intersezione (AND)

```
>>>A & B  
{1}
```

Differenza

```
>>>A - B  
{2,3}
```

Differenza simmetrica (XOR)

```
>>>A ^ B  
{2, 3, 4, 5}
```

tutti questi operatori creano copie, così come

```
&=, |=, -=, ^= .
```

Operatori di confronto:

`A <= B` verifica se A è contenuto in B, ovvero se ogni elemento di A è in B.

`A >= B` verifica se A contiene B, ovvero se ogni elemento di B è in A.

`x in A` restituisce `True` se e solo se il valore x è presente in A, `False` altrimenti.

`x not in A` restituisce `True` se e solo se il valore x non è presente in A, `False` altrimenti.

Un dizionario è un insieme mutabile di coppie chiave-valore.

Chiavi e valori possono essere di qualsiasi tipo. Ogni chiave è univoca. I valori possono essere duplicati.

```
>>>my_dict = { 1:'one', 'two':2 }
>>>print(my_dict)
{ 1:'one', 'two':2 }
```

L'operatore di slicing singolo può essere utilizzato per accedere ad un valore tramite la sua chiave.

```
>>>my_dict = dict()
>>>my_dict[1] = 'one'
>>>print(my_dict)
{1:'one'}
>>>my_dict['two'] = 2
>>>print(my_dict)
{1:'one', 'two':2}
```

Le strutture dati di python sono anche delle **collezioni iterabili**. Questo vuol dire che si può iterare tra gli elementi contenuti nelle strutture dati. Se la struttura dati è ordinata, l'iterazione segue l'**ordine** della struttura dati, altrimenti l'ordine dell'iterazione non è assicurato.

```
>>>my_list = [1,2,3]
>>>for x in my_list:
>>>    print(x, end=' ')
1 2 3
```

```
>>>my_tuple = (1,2,3)
>>>for x in my_tuple:
>>>    print(x, end=' ')
1 2 3
```

```
>>>my_set = {1,2,3}
>>>for x in my_set:
>>>    print(x, end=' ')
1 2 3
```

```
>>>my_dict = { 1:'one', 'two':2 }
>>>for x in my_dict.values():
>>>     print(x, end='')
'one' 2
```

```
>>>for p in my_dict.items():
>>>     print(p, end='')
(1,'one')(2,'two')
```

```
>>>for p in my_dict.items():
>>>     print(p[0], p[1])
1 'one'
2 'two'
```

```
>>>for k,v in my_dict.items():
>>>     print(k,v)
1 'one'
2 'two'
```

Python3 - generazione di sequenze casuali

Python mette a disposizione una libreria per la generazione di sequenze casuali di numeri e per la selezione casuale di elementi da una collezione.

<https://docs.python.org/3/library/random.html>

La libreria va importata tramite l'istruzione `import random`.

del generatore random

Per settare manualmente il seme del generatore random si può usare la funzione `random.seed([seed=None])`.

NOTA BENE: settare manualmente il seme del generatore consente di implementare parte delle procedure necessarie alla **riproducibilità** dei risultati. Utilizzare un seme automatico, e quindi diverso, ad ogni esecuzione implica di poter ottenere risultati diversi ad ogni esecuzione con conseguente invalidazione dei risultati ottenuti precedentemente.

Python3 - generazione di sequenze casuali

La libreria utilizza un generatore Mersenne Twister a 623 dimensionalità per la generazione di sequenze casuali che seguono una distribuzione e uniforme.

Tra le funzionalità principali:

`random.randint(a,b)` **genera** un numero intero casuale nell'intervallo `[a , b]`, entrambi inclusi.

`random.shuffle(lst)` **mescola** in modo casuale una lista.

`random.choice(lst)` **seleziona** un elemento a caso da una lista data.

`random.sample(population, k)` **seleziona** `k` elementi a caso, possibilmente **con ripetizione**, dalla popolazione data.

Caso d'uso: metodi *bag of words* per DNA

Come caso d'uso implementeremo dei metodi *bag of words* per l'analisi di sequenze (stringhe) di DNA.

Una metodologia **bag of words** scompone un oggetto X in una collezione di oggetti $\mathbb{X} = \{x_1, x_2, \dots, x_n\}$ che lo compongono. Quindi, utilizza \mathbb{X} per analizzare X .

Una **sequenza di DNA** è una stringa nell'**alfabeto nucleotidico** $\Gamma = \{A, C, G, T\}$.

Indichiamo con Γ^k le parole di lunghezza k costruite sull'alfabeto Γ , chiamate anche **k-meri** (k-mers).

Un **dizionario** su un alfabeto Γ è una collezione di parole/fattori, di lunghezza a piacere, sull'alfabeto dato, ovvero in Γ^* .

Caso d'uso: metodi *bag of words* per DNA

Sia s una stringa su Γ di lunghezza $|w|$, indichiamo con $s[i]$ l' i -esimo carattere di s e con $s[i, j]$, con $1 \leq i \leq j \leq |s|$, la sottostringa da i a j (entrambi inclusi) di s .

Data una stringa di DNA $w = a_1 a_2 \dots a_n$, con $a_i \in \Gamma$, il **dizionario k -esimo** (k -dictionary) di w , che indichiamo con $D_k(w)$, è dato tutti e soli i k -meri che occorrono almeno una volta in w :

$$D_k(w) = \{s \in \Gamma^k : \exists i, 1 \leq i \leq n | w[i, i + k - 1] = s\}$$

. Indichiamo con $D(w)$ il dizionario di tutti e soli i fattori di w ovvero:

$$D(w) = \bigcup_{1 \leq k \leq |w|} D_k(w)$$

Sia s una stringa su Γ ed un suo fattore s , l'**insieme delle posizioni** in w dove s occorre è dato da

$$pos(w, s) = \{1 \leq i \leq |w| : w[i, i + |s|] = s\}$$

La **molteplicità** di s in w è data dal numero di occorrenze di s in w :

$$mult(w, s) = |pos(w, s)|$$

Esercizio 1.1

Generare una sequenza di DNA casuale con distribuzione uniforme dei nucleotidi.

Esercizio 1.2

Verificare che la distribuzione dei nucleotidi della stringa generata nell'esercizio precedente sia uniforme.

Esercizio 1.3

Generare una sequenza di DNA casuale con distribuzione non uniforme dei nucleotidi. Nello fattispecie, *A* e *T* occorrono entrambi 30 volte su 100, mentre *C* e *G* 20 volte su 100. Verificarne la distribuzione.

Esercizio 1.4

Generare un sequenza casuale di DNA w ed un fattore s verificare che s sia contenuto in w .

Esercizio 1.5

Generare un sequenza casuale di DNA w ed un fattore s contare il numero di occorrenze di s in w , tenuto conto che possibili occorrenze di s in sovrapposizione tra di loro non interferiscono l'un l'altra. Es.

$w = AGAGA$, $count(w, 'AGA') = 2$.

Si confronti il risultato con l'istruzione `count('AGAGA', 'AGA')`.

Esercizio 1.6

Generare una sequenza casuale di DNA w ed estrarre $D_2(w)$.

Esercizio 1.7

Generalizzare l'esercizio precedente per estrarre $D_k(w)$ per un qualsivoglia valore k

Esercizio 1.8

Scrivere una procedura per estrarre $D_k(w)$ tale che ad ogni k -mer in $D_k(w)$ è associata la sua molteplicità.

Esercizio 1.9

Siano A e B due insiemi, l'indice di Jaccard è definito come

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

Date due sequenze di DNA, w_1 e w_2 , si calcoli $J(D_k(w_1), D_k(w_2))$ per k a piacere.

Esercizio 1.10

Un multi-insieme è una struttura che associa ad ogni elemento x dell'insieme A una molteplicità $m_A(x)$. Siano A e B due multi-insiemi, il coefficiente di Jaccard generalizzato è definito come

$$GJ(A, B) = \frac{\sum_{x \in A \cup B} \min(m_A(x), m_B(x))}{\sum_{x \in A \cup B} \max(m_A(x), m_B(x))}.$$

Date due sequenze di DNA, w_1 e w_2 , si calcoli $GJ(D_k(w_1), D_k(w_2))$ per k a piacere.

Verificare che tale misura sia sempre in $[0, 1]$.

Per ordinare una collezione di dati **iterabile** è possibile chiamare la funzione built-in `sorted(iterable[,key=None, reverse=False])` che ritorna una **lista ordinata** degli elementi della collezione.

```
>>>my_set = {2,4,3,1}
>>>for x in sorted(my_set):
>>>    print(x, end=' ')
1 2 3 4
```

```
>>>my_set = {2,4,3,1}
>>>x = sorted(my_set):
>>>print(type(x))
<class 'list'>
```

```
>>>my_list = [2,4,3,1]
>>>for x in sorted(my_list):
>>>    print(x, end=' ')
1 2 3 4
```

La struttura dati `list` ha un metodo interno per ordinare la lista in loco senza crearne una copia `lst.sort(key=None, reverse=False)`.

E' anche possibile ordinare le coppie chiave-valore di un `dict`, es:

```
>>>my_dict = { 1:'one', 2:'two' }
>>>for k,v in sorted(my_dict.items()):
>>>    print(k,v)
1 'one'
2 'two'
```

Esercizio 1.11

Esaminare il valore di ritorno di `my_dict.items()`.

Tuttavia, questo è possibile solo se le chiavi sono tutte dello stesso tipo.

Provare ad ordinare `my_dict = { 1:'one', 'two':2 }`.

Questo non è possibile perché non può essere trovato un ordine tra `1` e `'two'`.

Python3 - ordinare tuple

Immaginate di aver immagazzinato i vostri dati in uno foglio di calcolo, tale che ogni tupla/record è una riga, e di ordinare il foglio per colonne seguendo una specifica preferenza per quale colonna prendere prima in considerazione. Immaginiamo di avere le nostre tuple tutte nel formato (nome, gruppo sanguigno, età) .

Di default se proviamo ad usare la funzione `sorted` questa prenderà le colonne in ordine da sinistra verso destra. Tuttavia, potremmo voler ordinare i nostri pazienti in base all'età, poi per nome e poi per gruppo sanguigno. Possiamo, quindi, utilizzare `itemgetter` del modulo `operator` :

```
>>>from operator import itemgetter
my_tuples={('john', 'A', 10),('dave', 'B', 15),('jane', 'A', 15)}
>>>print(sorted(my_tuples))
[('dave', 'B', 15), ('jane', 'A', 15), ('john', 'A', 10)]
>>>print(sorted(my_tuples, key=itemgetter(2)))
[('john', 'A', 10), ('jane', 'A', 15), ('dave', 'B', 15)]
>>>print(sorted(my_tuples, key=itemgetter(2,0,1)))
[('john', 'A', 10), ('dave', 'B', 15), ('jane', 'A', 15)]
```

Python3 - comprensione di lista

Python implementa alcune funzionalità della programmazione funzionale. Tra queste c'è la comprensione di lista che permette di creare una lista basandosi su altre liste e applicando eventualmente degli operatori condizionali di filtraggio.

```
[ <generated values> <for, one or more> <filters> ]
```

per generare semplici liste di elementi

```
>>>[ True for i in range(3)]  
[True, True, True]
```

```
>>>[ i for i in range(3)]  
[0,1,2]
```

anche con specifiche proprietà, come i multipli del 2

```
>>>[ i for i in range(10) if i % 2 == 0]  
[0,2,4,6,8]
```


o per generare pattern combinatoriali

```
>>>[ (i,j) for i in range(3) for j in range(3) if i != j]  
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

e anche liste innestate

```
>>>[ [j * i for j in range(3) ] for i in range(3)]  
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

o per estrarre elementi da una lista

```
>>>my_list = [i for i in range(10)]  
>>>[ i for i in my_list if i % 2 == 0]  
[0,2,4,6,8]
```

La comprensione può anche essere usata per generare **insiemi** e **dizionari**

```
>>>{ i for i in range(3)}  
{0, 1, 2}
```

```
>>>{ i:i*2 for i in range(3)}  
{0:0, 1:2, 2:4}
```

Esercizio 1.12

Generare tramite comprensione di lista una lista di lista contenente tutte le tabellina dal 2 al 9.

Esercizio 1.13

Generare tramite comprensione di dizionario e lista un dizionario cui le chiavi sono i numero di 2 a 9 e i valori ad esse associate sono le rispettive tabelline.

Esercizio 1.14

Dato un insieme a generare tramite comprensione di lista tutte le possibili combinazioni degli elementi di a con e senza ripetizioni.

La funzione built-in `zip` consente di creare un iteratore per aggregare due o più liste.

Può esser usata, per esempio, per creare un dizionario a partire da due liste, una contenente le chiavi e l'altra contenente i valori.

```
>>>a, b = ['a','b','c'], [0,1,2]
>>>for i in zip(a,b): print(i, end=' ')
('a', 0) ('b', 1) ('c', 2)
```

Python fornisce un intero modulo aggiuntivo, `itertools`, di strumenti efficienti per la combinatoria che permettono, ad esempio, di calcolare prodotti cartesiani, permutazioni, combinazioni, etc..., nella forma di iteratori. <https://docs.python.org/3/library/itertools.html>

```
>>>from itertools import *
>>>for i in permutations('ABCD', 2): print(''.join(i),end=' ')
AB AC AD BA BC BD CA CB CD DA DB DC
>>>for i in permutations(range(3)): print(''.join(i),end=' ')
012 021 102 120 201 210
```

Esercizio 1.15

Data una lunghezza di parola k , generare tutte le possibili parole di tale lunghezza sull'alfabeto nucleotidico. Ovvero, generare Γ^k .

Esercizio 1.16

Data una stringa di DNA, s , un nullomero è una parola che appartiene a Γ^* ma che non è presente in s .

Data una lunghezza di parola k , si stampino tutti e soli i nullomeri di una stringa di DNA (scelta a piacere).

Esercizio 1.17

Data un k -mer w sull'alfabeto nucleotidico $\Gamma = \{A, C, G, T\}$, una elongazione di w è un $(k + 1) - kmer$ nella forma $w \cdot x$ con $x \in \Gamma$.

Data una collezione di n stringhe di DNA scelte a piacere, si stampino tutti e soli i k -meri che non compaiono in nessuna delle stringhe della collezione, per k a piacere.

Dichiarazione:

```
def function_name(<function parameters>):  
    "string documenting the function"  
    <code block>  
    [return values]
```

Il **corpo** della funzione è individuato grazie alla indentazione

```
def count_A(dna)  
    """this function returns the number of occurrences of  
        A within a DNA sequence"""  
    count_a = dna.count('A') + dna.count('a')  
    return count_a
```

Valore di ritorno

- il valore di ritorno è opzionale
- il tipo del valore di ritorno non è specificato
- la funzione può tornare più valori di ritorno
 - internamente costruita una tupla di valore id ritorno che verrà poi scomposta per assegnarli alle variabili

```
def count_AT(dna)
    """this function returns the number of occurrences of
        A and T within a DNA sequence"""
    count_a = dna.count('A') + dna.count('a')
    count_t = dna.count('T') + dna.count('t')
    return count_A, count_T

my_sequence = 'ACcccGTtGAaCGggcTTaatGAC'
count_A, count_T = count_AT(my_sequence)
```

Parametri

- il tipo dei parametri non è specificato/necessario
- i parametri sono passati per copia (tipi primitivi) o per riferimento (simile a Java)
- i parametri possono essere opzionali se un valore di default è specificato

```
def count_AT(dna, lower_case=True)
    """this function returns the number of occurrences of
        A and T within a DNA sequence"""
    count_a = dna.count('A') + dna.count('a')
    count_t = dna.count('T') + dna.count('t')
    if lower_case:
        count_a += dna.count('a')
        count_t += dna.count('t')
    return count_A, count_T
```


Se un parametro opzionale non è specificato nella chiamata a funzione, allora viene usato il valore di default

```
def count_AT(dna, lower_case=True)
    ...

my_sequence = 'AccccGTtGAaCGggcTTaatGAC'
print( count_AT(my_sequence) )
```

E' possibile specificare un valore per una specifico parametro senza seguire l'ordine di dichiarazione dei parametri

```
print( count_AT(my_sequence, lower_case=True) )
```

I parametri senza valore di default devono sempre essere i primi nella chiamata e seguire l'ordine dichiarato

```
print( count_AT(lower_case=False, dna=my_sequence) )
```

Per questi motivi è essenziale scrivere una buona stringa di documentazione, anche detta **docstring**

```
def count_CG(s):  
    """Count the number of c and g  
        in a given string and return the countings  
        -----  
        Parameters:  
            s (str) : the input string  
        -----  
        Returns:  
            int : the count of c  
            int : the count of g  
        """  
    count_c = dna.count('C') + dna.count('c')  
    count_g = dna.count('G') + dna.count('g')  
    return count_c, count_g
```

La docstring verrà visualizzata se richiama la documentazione tramite la funzione built-it `help`.

```
>>>help(count_CG(s))

Help on function count_CG in module __main__:

count_CG(s)
    Count the number of c and g
    in a given string and return the countings
    -----
    Parameters:
        s (str) : the input string
    -----
    Returns:
        int : the count of c
        int : the count of g
```

Per dettagli su variabili e funzioni è anche possibile usare l'operatore unario `?`

```
>>>?count_CG
Signature: count_CG(s)
Docstring:
Count the number of c and g
    in a given string and return the countings
-----
Parameters:
    s (str) : the input string
-----
Returns:
    int : the count of c
    int : the count of g
File:      /tmp/ipykernel_342/1112089259.py
Type:      function
```

Visibilità delle variabili:

- le variabili dichiarate dentro la funzione non sono visibili all'esterno
- le variabili dichiarate fuori dalla funzione sono visibili all'interno della funzione!!!

```
>>>b = 2
>>>def foo():
>>>    a = 10
>>>    print(a,b)
>>>foo()
10 2
>>>print(a,b)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [21], in <cell line: 9>()
      5     print(a,b)
      7 foo()
----> 9 print(a,b)

NameError: name 'a' is not defined
```

La funzione `open(filename, mode)` è una funzione built-in per aprire i file in lettura/scrittura. La funzione restituisce un **oggetto descrittore** di file.

Per **aprire** i file in **sola lettura** `f = open('mypath/myfile', 'r')`

Di default i file sono aperti in sola lettura: `f = open('mypath/myfile')`.

Di default i file sono aperti in **modalità testuale**. Tuttavia, è possibile aprire i file in modalità binaria: `f = open('mypath/myfile', 'rb')`.

Per aprire un file in **scrittura**, tale che se il file esiste già il suo contenuto viene **sovrascritto**: `f = open('mypath/myfile', 'w')`.

Per aprire un file in **scrittura**, tale che il nuovo contenuto viene **aggiunto** alla fine del file `f = open('mypath/myfile', 'a')`.

ATTENZIONE!!!

Per i file aperti in scrittura, ricordarsi sempre di fare **flush**: `f.flush()` !!!

Funzioni principali del descrittore di file:

`f.read()` legge l'intero contenuto del file e lo restituisce come singola stringa multilinea.

`f.read(N)` legge solo i primi `N` byte dal file.

`f.readline()` legge la prossima riga dal file.

`f.readlines()` legge tutte le righe del file e le restituisce come lista ordinata di stringhe.

`f.write(str)` scrive la stringa `str` sul file.

`f.write(str_list)` scrive la lista di stringhe `str_list` sul file come linee consecutive.

Ricordarsi sempre di **chiudere** sempre i file, sia in lettura che in scrittura:

`f.close()` .

Un esempio di lettura e scrittura di righe da e su file:

```
f_in = open('my_file_in', 'r')
f_out = open('my_file_out', 'w')
for line in f_in.readlines():
    f_out.write(line)
f_out.flush()
f_out.close()
f_in.close()
```

ATTENZIONE!!!

Per i file aperti in scrittura, ricordarsi sempre di fare **flush**: `f.flush()` !!!

Il descrittore di file è un **oggetto iterabile**, e gli elementi su cui itera sono le righe. Possiamo quindi utilizzarlo per iterare le righe del file:

```
for line in open('my_file', 'r'):  
    print(line)
```

In questo esempio non abbiamo bisogno di chiudere il file perché, quando `open` è utilizzato come argomento di un ciclo `for`, il descrittore è chiuso automaticamente.

Nel caso di file aperti in scrittura, può venirci in aiuto il costrutto `with` che in pratica implementa questi **automatismi** (e la gestione delle eccezioni).

```
with open('my_file_out', 'w') as f_out:  
    for line in open('my_file_in', 'r'):  
        f_out.write(line)
```

Il costrutto serve anche in altri contesti dove abbiamo bisogno di implementare degli automatismi legati appunto al contesto. Vedere https://docs.python.org/3/reference/compound_stmts.html#

In python abbiamo il seguente costrutto per gestire gli errori e le eccezioni:

```
try:
    <code block that can arise an exception>
except <error type> [as err]:
    <executed if an exception is thrown>
else:
    <executed if exceptions did not arise>
finally:
    <executed anyway>
```

La parte di `except` può essere ripetuta più volte.

Se nessuno degli `except` dichiarati cattura il tipo di errore che è effettivamente lanciato, allora l'errore non viene catturato.

Un metodo per catturare tutti gli errori è `except Error: .` Infatti `except` cattura qualsiasi errore che sia della classe `Error` o di una sua sottoclasse.

Un esempio

```
try:
    f = open('myfile', 'w')
    try:
        f.write(mystring)
    except IOError:
        print('non riesco a scrivere sul file')
    else:
        print('ho scritto correttamente sul file')
    finally:
        f.close()
except OSError as err:
    print('non posso aprire/chiudere il file')
    print(err)
```

Per lanciare una eccezione/errore si usa il costrutto `raise`. Esempio:

```
raise ValueError("Date provided can't be in the past") .
```

Per una lista completa delle eccezioni built-in si veda

<https://docs.python.org/3/library/exceptions.html>.

In python inoltre è possibile utilizzare la funzione `assert`.

```
import sys
assert sys.version_info >= (3, 7), "Versione minima supportata 3.7."
```

Esercizio 1.18

Il formato FASTA è il principale formato di file testuale per la memorizzazione di sequenze biologiche (nucleotidiche o aminoacidiche). All'interno di ogni file ci possono essere una o più sequenze biologiche. Ogni sequenza è preceduta da una linea di descrizione che inizia per `>`. Dopo tale linea, la sequenza viene scritta nel file tale che ogni 80 caratteri si va a capo.

Scrivere una funzione per la lettura dei file FASTA che possa gestire tutte le casistiche.

```
>sp|P26367|PAX6_HUMAN Paired box protein Pax-6 OS=Homo sapiens OX=9606 GN=PAX6 PE=1 SV=2
MQNSHSGVNLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRILQVSNCGVSKILGRY
YETGSIRPRAIGGSKPRVATPEVVS KIAQYKRECP SIFAW EIRDRL LSEGVC TNDNIP SV
SSINRVLRNLASEKQ QMGADGMYDKLRMLNGQTGSW GTRPGWYPGTSVPGQPTQDGCQQQ
LQ
>sp|P26367-2|PAX6_HUMAN Isoform 5a of Paired box protein Pax-6 OS=Homo sapiens OX=9606 GN=PAX6
MQNSHSGVNLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRILQTHADAKVQVLDNQ
NVNCGVSKILGRYYETGSIRPRAIGGSKPRVATPEVVS KIAQYKRECP SIFAW EIRDRL
LSEGVC TNDNIP SVSSINRVLRNLASEKQ QMGADGMYDKLRMLNGQTGSW GTRPGWYPGT
VPGSEPDMSQYWPR LQ
```

Il file `Student.py` illustra brevemente le basi della programmazione ad oggetti in python, in particolare:

- come dichiarare una classe
- la parola chiave `self`
- variabili pubbliche e private
- funzioni di distanza e statiche
- overriding delle funzioni built-in
- overloading degli operatori
- ordinare oggetti di classe custom

Per creare una classe che sia **iterabile** si deve ridefinire la funzione `__iter__(self)` tale che essa ritorni un oggetto iteratore appositamente costruito.

All'interno della classe iteratrice si deve ridefinire la funzione `__next__` tale funzione lancia una eccezione di tipo `StopIteration` quando si è giunti alla fine della iterazione.

La classe principale e la classe iteratrice possono coincidere.

```
class irange:
    def __init__(self, end, start=0):
        self.c = start
        self.end = end

    def __iter__(self):
        return irange(self.end, self.c)

    def __next__(self):
        self.c += 1
        if self.c >= self.end:
            raise StopIteration
        else:
            return self.c

for i in irange(10): print(i)
```


Esercizio 1.19

Scrivere una op più classi per iterare i k -meri di una stringa, indipendentemente dall'alfabeto su cui è costruita la stringa.

Esercizio 1.20

Scrivere una op più classi per iterare i k -nullomeri di una stringa, indipendentemente dall'alfabeto su cui è costruita la stringa.