Laboratorio di WebScraping

Anno Accademico 2024-2025

Docente: Laura Ricci

Lezione 3

Python:

Set, List, Tuple, Dictionary, Functions

3 Febbraio 2025

L'ambiente di sviluppo

- lo sviluppo del progetto finale deve essere effettuato mediante un **Notebook**
- ambienti consigliati, entrambi basati su **IPython** (un ambiente interattivo per Python), reperibili in **https://jupyter.org/**
    - **Jupyter Notebook**: classico notebook
    - **Jupyter Lab**: versione più avanzata
- facilitano il reperimento, l'esplorazione e la visualizzazione dei dati.
- consentono di scrivere
    - codice in blocchi separati ("celle") e di eseguire individualmente le singole celle di codice, piuttosto che scrivere l'intero programma e poi eseguirlo
    - blocchi di testo, formattato mediante linguaggio di markdown

Struttura delle prossime lezioni

- una serie di esercizi proposti
- risoluzione incrementale di ogni esercizio
- le slide con titolo "Es. x" contengono fragmenti di codice che fanno parte della soluzione dell'esercizio
- le slide con titolo "Approfondimento" contengono approfondimenti sul costrutto Python usato in quell'esercizio

Cosa vedremo: Strutture e dati e librerie di base

- programmazione di semplici problemi di calcolo delle probabilità
    - famiglia numerosa
    - fair dice roll
    - biased dice roll e multiple coin flipping
- costrutti **Python** analizzati
    - funzioni e passaggio di parametri
    - dati strutturati
        - set
        - list
        - tuple
        - dictionaries
- librerie *Python* utilizzate
    - *Matplotlib*
    - *Numpy*

Zen of Python: 19 aforsimi contenenti dei suggerimenti per i programmatori Python, un easter egg

In [1]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Es 1: Una famiglia numerosa

- consideriamo le famiglie con esattamente 4 figli
- ipotesi: avere un maschio o una femmina è equiprobabile
- spazio campionario: è costituito dall'esito delle 4 nascite e ogni esito può essere rappresentato con una tupla
    - rappresenta una sequenza dei 4 bambini nati
- calcoleremo diverse probabilità
    - esempio: quale è la probabilità che **esattamente due figli** siano maschi?

Es. 1: Spazio degli eventi: liste, tuple, set

In [2]:

```
possible_children = ['Boy', 'Girl']
sample_space = set()
for child1 in possible_children:
    for child2 in possible_children:
        for child3 in possible_children:
            for child4 in possible_children:
                outcome = (child1, child2, child3, child4)
                sample_space.add(outcome)
print(sample_space)
```

```
{('Girl', 'Girl', 'Girl', 'Girl'), ('Girl', 'Girl', 'Girl', 'Boy'), ('Boy', 'Bo
y', 'Girl', 'Girl'), ('Boy', 'Girl', 'Boy', 'Girl'), ('Girl', 'Boy', 'Boy', 'Gi
rl'), ('Boy', 'Boy', 'Girl', 'Boy'), ('Boy', 'Girl', 'Boy', 'Boy'), ('Girl', 'B
oy', 'Boy', 'Boy'), ('Boy', 'Girl', 'Girl', 'Boy'), ('Boy', 'Girl', 'Girl', 'Gi
rl'), ('Girl', 'Girl', 'Boy', 'Girl'), ('Girl', 'Girl', 'Boy', 'Boy'), ('Boy',
'Boy', 'Boy', 'Girl'), ('Girl', 'Boy', 'Girl', 'Girl'), ('Boy', 'Boy', 'Boy',
'Boy'), ('Girl', 'Boy', 'Girl', 'Boy')}
```

Approfondimento: gli insiemi

```
a={3.5, 'Web', 22, True}
```

- collezione **non ordinata** di uno o più riferimenti ad oggetti **non duplicati**
- **eterogenei**: possono contenere valori di tipo diverso
- struttura **mutabile**, ma gli elementi devono essere **hashable**, quindi **immutabili**
    - si possono aggiungere e togliere elementi, ma non gli elementi devono essere immutabili
- non indicizzato
- no slicing o striding
- definite operazioni classiche su insiemi
- implementati con hash table set membership: complessità media O(1), complessità caso pessimo O(N)

Approfondimento: gli insiemi, creazione {}

In [3]:

```
a = {3.5, 'Web', 22, True, ("x",11)}
a
```

Out[3]:

```
{('x', 11), 22, 3.5, True, 'Web'}
```

In [4]:

```
a= {3.5, 'Web', 22, True, ["x",11]}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 a= {3.5, 'Web', 22, True, ["x",11]}

TypeError: unhashable type: 'list'
```

- gli oggetti mutabili come le liste, non sono hashable

Approfondimento: tipi "hashable"

- hashable objects definiscono un metodo **__hash__()** che restituisce un identificatore dell'oggetto di lunghezza predefinita
- tutti i tipi di dato primitivi sono immutabile e *hashable*
- i tipi di dato mutabili non sono **hashable**, poichè il valore dell'hash cambierebbe a seconda dei dati che contengono

Approfondimento: gli insiemi, creazione Set() e shallow copy

In [5]:

```
b = set('scraping')
b
```

Out[5]:

```
{'a', 'c', 'g', 'i', 'n', 'p', 'r', 's'}
```

In [6]:

```
c = b
c
```

Out[6]:

```
{'a', 'c', 'g', 'i', 'n', 'p', 'r', 's'}
```

In [7]:

```
b.add('x')
b
```

Out[7]:

```
{'a', 'c', 'g', 'i', 'n', 'p', 'r', 's', 'x'}
```

In [8]:

```
c
```

Out[8]:

```
{'a', 'c', 'g', 'i', 'n', 'p', 'r', 's', 'x'}
```

- tuttavia il costruttore set crea sempre un nuovo oggetto!

Approfondimento: Le tuple

```
outcome = (child1, child2, child3, child4)
```

- una sequenza **ordinata** di zero o più riferimenti ad oggetti
- **immutabili** - non si possono aggiungere o togliere elementi
- eterogenee: possono contenere dati di qualsiasi tipo
- può contenere duplicati
- elementi indicizzati
- slicing e striping
- iterable
- hashable
- creazione
    - parentesi tonde
    - il costruttore *tuple()*

Approfondimenti: le tuple

- versione **immutabile** delle liste
    - impossibile cambiare un elemento di una tupla
- quali usi?
    - possono essere usate come **chiavi** in dizionari

In [9]:

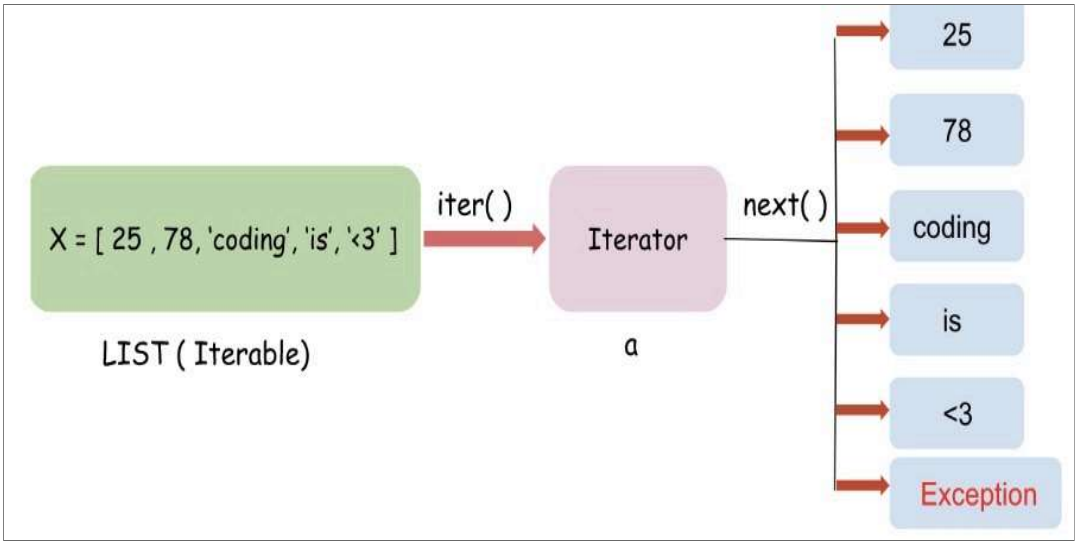
```
paris = (33.66, -95.54)
athens = (32.20, -95.85)
texas_cities = {
    paris: 'Paris',
    athens: 'Athens',
}
texas_cities
```

Out[9]:

```
{(33.66, -95.54): 'Paris', (32.2, -95.85): 'Athens'}
```

Gli iteratori

- iterables: tutti le collezioni di oggetti in grado di restituire i loro membri uno alla volta
    - liste, tuple, insiemi,stringhe, dictionaries,
- iterators
    - costruiti a partire da Iterables mediante l'operatore *iter()*
    - operatore *next()*



Es 1: Pretty printing dello spazio campionario: gli iteratori

In [10]:

```
possible_children = ['Boy', 'Girl']
sample_space = set()
for child1 in possible_children:
    for child2 in possible_children:
        for child3 in possible_children:
            for child4 in possible_children:
                outcome = (child1, child2, child3, child4)
                sample_space.add(outcome)
iter_sample_space=iter(sample_space)
```

Es 1: Pretty printing dello spazio campionario: gli iteratori

In [11]:

```
while True:
    # item will be "end" if iteration is complete
    item = next(iter_sample_space, "end")
    if item == "end":
        break
    print(item)
```

```
('Girl', 'Girl', 'Girl', 'Girl')
('Girl', 'Girl', 'Girl', 'Boy')
('Boy', 'Boy', 'Girl', 'Girl')
('Boy', 'Girl', 'Boy', 'Girl')
('Girl', 'Boy', 'Boy', 'Girl')
('Boy', 'Boy', 'Girl', 'Boy')
('Boy', 'Girl', 'Boy', 'Boy')
('Girl', 'Boy', 'Boy', 'Boy')
('Boy', 'Girl', 'Girl', 'Boy')
('Boy', 'Girl', 'Girl', 'Girl')
('Girl', 'Girl', 'Boy', 'Girl')
('Girl', 'Girl', 'Boy', 'Boy')
('Boy', 'Boy', 'Boy', 'Girl')
('Girl', 'Boy', 'Girl', 'Girl')
('Boy', 'Boy', 'Boy', 'Boy')
('Girl', 'Boy', 'Girl', 'Boy')
```

- vogliamo stampare ognuna delle tuple dello spazio campionario su una riga diversa
- definiamo un **iteratore** sull'insieme sample_space
- primo parametro: la struttura da scorrere
- secondo parametro: un *valore di default* che deve essere ritornato se l'iteratore termina

Es 1: Calcolare le probabilità degli eventi

- **evento**: un qualsiasi sottoinsieme dello spazio campionario (composto da tuple, che indicano l'esito dell'esperimento), che soddisfa una certa condizione
- per ogni evento, definiamo una **funzione boolena** che dato un elemento dello spazio campionario, verifica se l'esito dell'esperimento verifica l'evento corrispondente
- ad esempio, se vogliamo calcolare la probabilità di avere esattamente due figli maschi sui quattro figli, possiamo definire l'evento **has two boys**: verifica se una singola tupla dello spazio campionario contiene esattamente due valori uguale a **Boy**

In [12]:

```
def has_two_boys(outcome):
    return len([child for child in outcome
                      if child == 'Boy']) == 2
```

Ex 1: Calcolare la probabilità degli eventi con higher order function

- vogliamo calcolare la probabilità di diversi eventi sullo spazio campionario definito in precedenza (spazio delle tuple contenente i 4 figli di una famiglia)
- come strutturiamo le funzioni?
    - una funzione *get_matching_event* che dato un evento e lo spazio campionario calcola il numero di elementi dello spazio campionario che soddisfano quell'evento
        - passeremo a questa funzione la funzione booleana definita nella slide precedente che controlla che l'evento sia verificato
        - uso di higher order functions
    - una funzione *compute_event_probability* che dato il risultato della funzione precedente, lo divide per la dimensione dello spazio campionario, restituendo così la probabilità
    - useremo l'higher order per verificare se l'evento è soddisfatto da un elemento dello spazio campionario

Ex 1: Calcolare la probabilità degli eventi con higher order function

In [13]:

```
def get_matching_event(event_condition, generic_outcome_space):
    return set([outcome for outcome in generic_outcome_space
                if event_condition(outcome)])

def compute_event_probability(event_condition, generic_sample_space):
    samples = get_matching_event(event_condition, generic_sample_space)
    return len(samples) / len(generic_sample_space)
```

Ex 1: Calcolare la probabilità degli eventi

In [14]:

```
prob = compute_event_probability(has_two_boys, sample_space)

print(f"Probability of 2 boys is {prob}")
```

```
Probability of 2 boys is 0.375
```

- quali costrutti abbiamo utilizzato?
    - definizione di funzioni
    - liste e list comprehension
    - higher order
- approfondiamo questi concetti

Approfondimento: liste

```
L = [20, 'Laura', 3.14, [10,20,30]]
```

- collezione **ordinata** di riferimenti ad oggetti
- **eterogenei**: gli elementi possono essere di tipo diverso
- può contenere **duplicati**
- **mutabili** si possono modificare gli elementi della lista
- creata utilizzando
    - il costruttore list
    - le parentesi quadre
- accesso per indice
- slicing, striping

Approfondimento: liste assegnamento oggetti mutabili

In [15]:

```
fruits = ["Apple", "Banana", "Melon"]
gifts = fruits
fruits = ["Kiwi", "Charry"]
print(fruits)
print(gifts)
```

```
['Kiwi', 'Charry']
['Apple', 'Banana', 'Melon']
```

- il nuovo assegnamento ha creato un nuovo legame per la variabile *fruits*
- *gifts* rimane invece legato al vecchio valore

Approfondimento: liste assegnamento oggetti mutabili

In [ ]:

```
books = ["MemoriediAdriano", "IlMaestroeMargherita"]
myPrefBooks = books
myPrefBooks.append("LaRecerche")
print(books)
print(myPrefBooks)
```

- modificato l'oggetto myPrefBook senza creare il legame con una nuova variabile
- le due variabili puntano ancora allo stesso oggetto
- la modifica effettuata su myPrefBooks ha avuto impatto anche su books
- questo comportamento è vero per ogni *oggetto mutabile*

Approfondimenti: liste, assegnamento oggetti mutabili

In [16]:

```
L1 = ["orange","peach","banana"]
L2 = ["blueberry","strawberry"]
JoinL= [L1,L2]
print(JoinL)
```

```
[['orange', 'peach', 'banana'], ['blueberry', 'strawberry']]
```

In [17]:

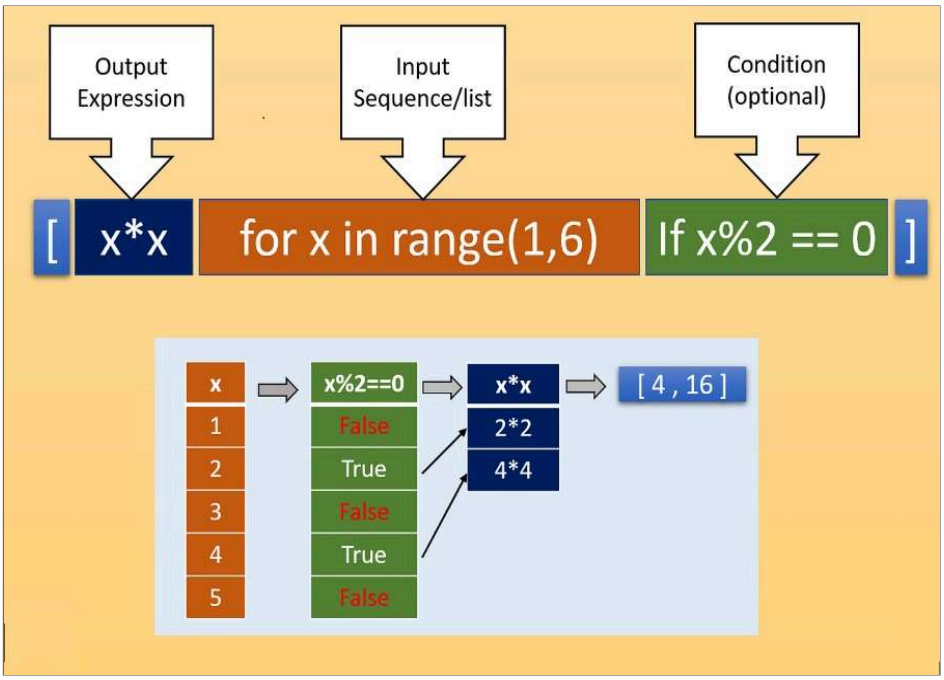
```
L1[2] = "apple"
print(JoinL)
L2.append("tangerines")
print(JoinL)
JoinL[1][1] = "pear"
print(L2)
```

```
[['orange', 'peach', 'apple'], ['blueberry', 'strawberry']]
[['orange', 'peach', 'apple'], ['blueberry', 'strawberry', 'tangerines']]
['blueberry', 'pear', 'tangerines']
```

Approfondimenti: liste, il meccanismo della "comprehension"

- il meccanismo di comprehension, nel caso delle liste
    - permette di filtrare e/o trasformare una lista, creandone una nuova
    - la lista originale non viene modificata
    - utilizzabile non solo per filtrare elementi da liste, ma da qualsiasi oggetto iterabile (insiemi, tuple, dizionari)
- vantaggi
    - leggibilità
    - performance
- più **"pythonic"** rispetto al classico for loop

Approfondimenti: liste, il meccanismo della "comprehension"



Approfondimenti: il meccanismo della "comprehension": filtrare

- permette di creare una nuova lista a partire da un oggetto iterabile **O**
- la nuova lista contiene solo gli elementi di **O** che soddisfano una certa condizione

In [18]:

```
filastrocca = 'trentatré trentini entrarono a Trento, tutti e trentatré trotterellando?'.split()
print(filastrocca)
listaparole = [x
               for x in filastrocca if 'tr' in x]
print(listaparole)
```

```
['trentatré', 'trentini', 'entrarono', 'a', 'Trento,', 'tutti', 'e', 'trentatr
é', 'trotterellando?']
['trentatré', 'trentini', 'entrarono', 'trentatré', 'trotterellando?']
```

Approfondimenti: il meccanismo della "comprehension": confronto con costrutti iterativi

In [19]:

```
names = ["Isac Newton", "Albert Einstein", "Niels Bohr", "Marie Curie", "Charles Darwin", "Louis C
result = []
for name in names:
    split_name = name.split(" ")
    last_name_first = split_name[1] + ", " + split_name[0]
    result.append(last_name_first)
print (result)
```

```
['Newton, Isac', 'Einstein, Albert', 'Bohr, Niels', 'Curie, Marie', 'Darwin, Ch
arles', 'Oasteur, Louis', 'Galilei, Galileo']
```

In [20]:

```
split_names = [name.split(" ") for name in names]
last_names_first = [sn[1] + ", " + sn[0]   for sn in split_names]
print(last_names_first )
```

```
['Newton, Isac', 'Einstein, Albert', 'Bohr, Niels', 'Curie, Marie', 'Darwin, Ch
arles', 'Oasteur, Louis', 'Galilei, Galileo']
```

Approfondimenti: il meccanismo della "comprehension": trasformare

In [21]:

```
txns = [1.09, 23.56, 57.84, 4.56, 6.78]
TAX_RATE = .08
def get_price_with_tax(txn):
    return txn * (1 + TAX_RATE)
final_prices = [get_price_with_tax(val) for val in txns if val>2]
print(final_prices)
```

```
[25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

Approfondimenti: il meccanismo della "comprehension": iteratori multipli

In [22]:

```
nums_list = [[7, 2], [6], [1, 4, 5], [-2, 8, 0]]
small_nums_squared = [n ** 2
                      for li in nums_list
                          for n in li
                            if n < 5]
small_nums_squared
```

Out[22]:

```
[4, 1, 16, 4, 0]
```

Approfondimenti: liste, ”comprehension” o ”for loop”?

In [23]:

```
import time
MILLION_NUMBERS = list(range(70000000))
def for_loop():
    output = []
    for element in MILLION_NUMBERS:
        if not element % 2:
            output.append(element)
    return output
def list_comprehension():
    return [number for number in MILLION_NUMBERS if not number % 2]
```

Approfondimenti: liste, ”comprehension” o ”for loop”?

In [24]:

```
st = time.time()
for_loop()
et = time.time()
elapsed_time = et - st
print(f"tempo del for_loop {elapsed_time}")

st = time.time()
list_comprehension()
et = time.time()
elapsed_time = et - st
print(f"tempo list_comprehension {elapsed_time}")
```

```
tempo del for_loop 3.8530757427215576
tempo list_comprehension 3.26108717918396
```

Approfondimento: importare una libreria

- possibilità di importare intere librerie o moduli
    - simile a *#include* di C o *import* di *JAVA*
- *import* importa l'intera libreria
- *from ...import...* importa solo un modulo della libreria
- *time*
    - diverse funzioni in relazione con il tempo
    - altro modulo **datetime**

Approfondimento: differenza tra liste e tuple, spazio di memoria

In [25]:

```
import sys
a_list = ['abc', 'xyz', 123, 231, 13.31, 0.1312]
a_tuple = ('abc', 'xyz', 123, 231, 13.31, 0.1312)
print('The list size:', sys.getsizeof(a_list), 'bytes')
print('The tuple size:', sys.getsizeof(a_tuple), 'bytes')
```

```
The list size: 104 bytes
The tuple size: 88 bytes
```

- minore occupazione di memoria di tuple rispetto alle liste
- più marcato nel caso di dati di grosse dimensioni
- immutabilità consente al compilatore di fare un maggior numero di ottimizzazioni

Approfondimento: differenza tra liste e tuple, tempo di esecuzione

In [26]:

```
import sys, platform
import time

start_time = time.time()
b_list = list(range(70000000))
end_time = time.time()
print("Instantiation time for LIST:", end_time - start_time)

start_time = time.time()
b_tuple = tuple(range(70000000))
end_time = time.time()
print("Instantiation time for TUPLE:", end_time - start_time)
```

```
Instantiation time for LIST: 2.2152633666992188
Instantiation time for TUPLE: 7.480693578720093
```

Approfondimento: differenza tra liste e tuple, tempo di esecuzione

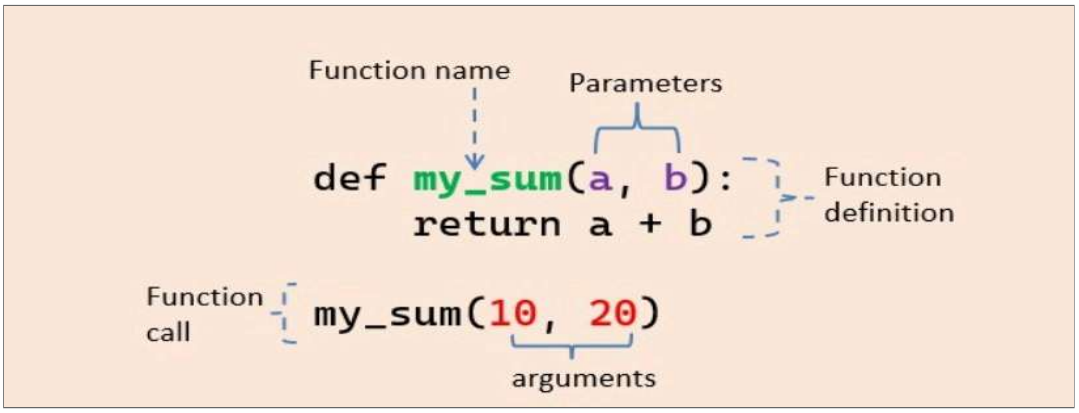
In [27]:

```
start_time = time.time()
for item in b_list:
  aa = b_list[20000]
end_time = time.time()
print("Lookup time for LIST: ", end_time - start_time)
start_time = time.time()
for item in b_tuple:
  aa = b_tuple[20000]
end_time = time.time()
print("Lookup time for TUPLE: ", end_time - start_time)
```

```
Lookup time for LIST:  6.609704494476318
Lookup time for TUPLE:  4.092849016189575
```

Approfondimento: Funzioni

```
def has_two_boys(outcome):
    return len([child for child in outcome
                                    if child == 'Boy']) == 2
```



Approfondimento: Funzioni

- come ogni elemento di **Python**, le funzioni sono oggetti
    - quando viene creata una funzione, viene creato un oggetto e memorizzato un riferimento all'oggetto nella variabile con il nome della funzione
    - possono essere memorizzate in una collezione
    - possono essere passati come argomenti ad altre funzione o restituite come risultati (higher order)
        - permettono uno stile di programmazione funzionale
- creano un nuovo namespace
- restituiscono sempre un valore
    - può essere **None** se non c'è un return esplicito

Approfondimento: Funzioni come oggetti

In [28]:

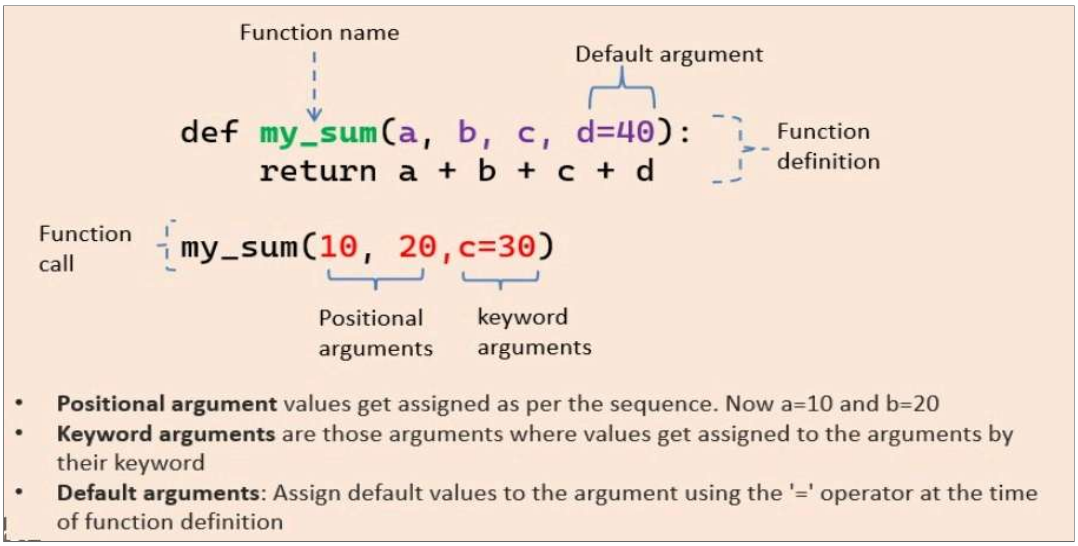
```
def echo (arg):
    return arg
print(type(echo))
isinstance(echo, object)
```

```
<class 'function'>
```

Out[28]:

```
True
```

Approfondimento: Funzioni, passaggio di parametri



- **Positional argument** values get assigned as per the sequence. Now a=10 and b=20
- **Keyword arguments** are those arguments where values get assigned to the arguments by their keyword
- **Default arguments**: Assign default values to the argument using the '=' operator at the time of function definition

Approfondimento: Funzioni, positional and keyword parameters

In [29]:

```
def greet_person(person, number):
    for greeting in range(number):
        print(f"Hello {person}! Come stai?")

greet_person("Maria", 4)
greet_person("Paolo", number=2)
greet_person(person="Matteo", number=4)
```

```
Hello Maria! Come stai?
Hello Maria! Come stai?
Hello Maria! Come stai?
Hello Maria! Come stai?
Hello Paolo! Come stai?
Hello Paolo! Come stai?
Hello Matteo! Come stai?
Hello Matteo! Come stai?
Hello Matteo! Come stai?
Hello Matteo! Come stai?
```

Approfondimento: Funzioni, positional and keyword parameters

In [30]:

```
greet_person(person="Stephen", 10)
```

```
  Cell In[30], line 1
    greet_person(person="Stephen", 10)
                                       ^
SyntaxError: positional argument follows keyword argument
```

- i parametri posizionali devono sempre precedere i parametri keyword

Approfondimento: Funzioni, positional and keyword parameters in funzioni predefinite

In [31]:

```
print(2, 1, 3, 4, sep='-')
print(2, 1, 3, 4, sep=', ')
```

```
2-1-3-4
2, 1, 3, 4
```

In [32]:

```
print(2, 1, 3, 4, sep=',', end='!!\n')
```

```
2,1,3,4!!
```

In [33]:

```
print(2, 1, 3, 4, end='!!\n', sep=',' )
```

```
2,1,3,4!!
```

In [34]:

```
sum([2, 1, 3 ,4], start=1)
```

Out[34]:

```
11
```

Approfondimento: Funzioni, default parameters

- parametri il cui valore attuale è opzionale
- se non viene passato alcun valore del parametro attuale, viene utilizzato il valore di default, associato al parametro formale
- altrimenti il valore passato sovrascrive quello di default

In [44]:

```
#default parameters
def add_tax(x, vat=20):
    return x*(1+vat/100.)
print(add_tax(1000))
print(add_tax(1000,5))
```

```
1200.0
1050.0
```

Approfondimento: Higher Order Functions

- le funzioni in Python sono oggetti e come tali possono essere passate come parametri ad altre funzioni
- è sufficiente passare il nome della funzione
- possibile anche in JAVA; C++, ma molto più complicato

In [ ]:

```
def get_matching_event(event_condition, generic_outcome_space):
    return set([outcome for outcome in generic_outcome_space
                if event_condition(outcome)])
```

- higher order: viene passata alla *compute_event_probability* il **nome della funzione** *has_two_boys*
- la funzione verrà poi invocata nel corpo della *compute_event_probability* per verificare se un evento è verificato per un certo elemento dello spazio campionario

Approfondimento: Higher Order Functions

In [35]:

```
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def greet(func):
    # storing the function in a variable
    greeting = func("Hi, I am created by a function \
    passed as an argument.")
    print(greeting)

greet(shout)
greet(whisper)
```

```
HI, I AM CREATED BY A FUNCTION     PASSED AS AN ARGUMENT.
hi, i am created by a function     passed as an argument.
```

Es 2: Lanci Multipli di un dado

- un dado lanciato per 6 volte
- quale è la probabilità di ottenere un risultato = 21?
- primo passo: rappresentazione dello lo spazio campionario

In [36]:

```
from itertools import product
possible_rolls = list(range(1, 7))
print(possible_rolls)
sample_space = set(product(possible_rolls, repeat=6))
print(len(sample_space))
```

```
[1, 2, 3, 4, 5, 6]
46656
```

Approfondimento: La funzione Range

- **range(start, stop, step)**
- restituisce una sequenza di numeri, iterabile
- inizia da start (0, di default)
- si incrementa ad ogni passo di step (1, di default)
- termina a stop, l'ultimo numero della sequenza (stop) non è incluso nel risultato
- forma sintetica *range(stop)*

Es 2: Probabilità dell'evento "Somma=21"

In [37]:

```
def has_sum_of_21(outcome):
    return sum(outcome) == 21

prob = compute_event_probability(has_sum_of_21, sample_space)

print(f"6 rolls sum to 21 with a probability of {prob}")
```

```
6 rolls sum to 21 with a probability of 0.09284979423868313
```

- possiamo riutilizzare la funzione **compute_event_probability** con uno spazio campionario diverso e con una condizione diversa
- outcome è un elemento del nuovo spazio campionario
    - un insieme di 6 elementi che rappresentano le 6 facce dei 6 dadi lanciati
    - la funzione sum applicata ad un insieme restituisce la somma dei suoi valori

Es 2: Probabilità dell'evento "Somma=21"

- invece di definire la funzione **has_sum_21**, si passa direttamente la funzione come parametro
- uso delle lambda expressions

In [38]:

```
prob = compute_event_probability(lambda x: sum(x) == 21, sample_space)
print(f"6 rolls sum to 21 with a probability of {prob}")
assert prob == compute_event_probability(has_sum_of_21, sample_space)
```

```
6 rolls sum to 21 with a probability of 0.09284979423868313
```

Approfondimento: Dizionari

- per la soluzione del prossimo assignment utile è utile utilizzare un **dizionario**
- collezione di coppie *chiave:valore*
    - chiavi sono riferimenti aoggetti hashabili e quindi immutabili
    - valori sono riferimenti ad oggetti di qualsiasi tipo, anche altre collezioni
- unicità delle chiavi, ma non dei valori
- mutabili
    - è possibile aggiungere/rimuovere/cambiare le associazioni tra chiave e valore
- elementi sono
    - ordinati da **Python 3.7** in poi
    - non ordinati nelle versioni precedenti di Python
- no indexing, no striping
- creazione
    - insieme di coppie (chiave, valore) in **parentesi graffe**
    - costruttire **dict()**

Approfondimento: Dizionari, views

- possibilità di **iterare** su tutti gli eleenti del dizionario, sulle chiavi, oppure sui valori

In [39]:

```
mydict = {
  "Course": "WebScraping",
  "Teacher": "Laura Ricci",
  "Semester": 2
}
print (mydict["Teacher"])
print (list(mydict.items()))
print (list (mydict.keys()))
print (list(mydict.values()))
```

```
Laura Ricci
[('Course', 'WebScraping'), ('Teacher', 'Laura Ricci'), ('Semester', 2)]
['Course', 'Teacher', 'Semester']
['WebScraping', 'Laura Ricci', 2]
```

Approfondimento: Dictionary Comprehension

In [45]:

```
inverted_my_dict={v:k for k,v in mydict.items()}
inverted_my_dict
```

Out[45]:

```
{'WebScraping': 'Course', 'Laura Ricci': 'Teacher', 2: 'Semester'}
```

- possibile se i valori contenuti nel dizionario sono hashabili e unici
- altrimenti solleva un'eccezione

Approfondimento: Dictionary Comprehension

In [40]:

```
def filter_data(students):
    result = {k: s for k, s in students.items() if s[0] >= 170 and s[1] >= 70}
    return result
students = {
    'Mario Rossi': (165, 70),
    'Giovanni Bianchi': (173, 75),
    'Antonio Verdi': (180, 88),
    'Marco Gialli': (169, 68)
}
print("Original Dictionary:")
print(students)
print("\nHeight > 170 and Weight > 70kg:")
print(filter_data(students))
```

```
Original Dictionary:
{'Mario Rossi': (165, 70), 'Giovanni Bianchi': (173, 75), 'Antonio Verdi': (18
0, 88), 'Marco Gialli': (169, 68)}

Height > 170 and Weight > 70kg:
{'Giovanni Bianchi': (173, 75), 'Antonio Verdi': (180, 88)}
```

Assignment

- modificare l'esercizio 2 considerando una **spazio campionario pesato**
- rappresentare ogni elemento dello spazio campionario come una coppia (valore della somma dei sei dadi, peso), dove il peso è numero di volte in cui quell'evento (quella somma) si può presentare
- ricalcolare la probabilità dell'evento **somma=21** prendendo in considerazione lo spazio campionario modificato
- utilizzare un **Python dictionary**, che associ il valore delle possibili somme al numero di volte che quella somma si presenta

Approfondimento: Python Multithreading

In [41]:

```
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds -', end - start)
```

```
Time taken in seconds - 1.4441859722137451
```

Approfondimento: Multithreading non effettivo per task CPU bound

In [43]:

```
import time
from threading import Thread
COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1
t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))
start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()
print('Time taken in seconds -', end - start)
```

```
Time taken in seconds - 1.3925385475158691
```

In [ ]:

```

```