Laboratorio di programmazione

Python

A.A. 2020-2021

Lezione 3

Funzioni

Una **funzione** è un nome che corrisponde ad **gruppo di istruzioni**.

Lo **scopo** è dividere in **blocchi logigi** il programma

La sintassi di una funzione è:

```
#inizio funzione
def snake_case_name( parametro_1, parametro_2 ):
    STATEMENTS

# fine funzione
```

- la keyword **def**
- nome in snake case con stesse regole delle variabili
- parentesi tonde che racchiudono i nomi dei parametri se ci sono
- il token `:` che indica l'inizio del blocco **body** della funzione
- tutti gli statements che vanno eseguiti quando si **chiama** la funzione
- gli statements **devono essere indentati**
- la funzione fiunisce quando rientro dall'indentazione

si possono creare quante funzioni si vuole in un programma

Raffaello e le funzioni

Prendiamo il `turtle_program` e ragioniamo con le funzioni:

- dobbiamo disegnare dei quadrati
- "disegna quadrato" è una *astrazione* dell'attività che va svolta nel programma
- possiamo scrive il programma con una funzione che riproduce il ragionamento

```python
import turtle

def draw_square(my_turtle, steps):
    """Make my_turtle draw a square of side 'steps'"""
    for i in range(4):
        my_turtle.forward(steps)
        myturtle.left(90)
    # fine del blocco 'for'
# fine della funzione

# programma 'main'
window = turtle.Screen()        # Set up the window and its attributes
window.bgcolor("lightgreen")
window.title("Raffaello e le funzioni")

raffaello = turtle.Turtle()  # Creo la tartatuga
draw_square(raffaello, 50)   # Chiamo la funzione per disegnare un quadrato con raffaello
window.mainloop()
```

> *La funzione draw_square puo' funzionare con ogni tartaruga*

Considerazioni

Analizziamo la funzione:

```
def draw_square(my_turtle, steps):                          # 1
    '''my_turtle' disegna un quadrato di lato 'steps' '''   # 2
    for i in range(4):                                      # 3
        my_turtle.forward(steps)                            # 4
        myturtle.left(90)                                   # 5
    # fine del blocco 'for'                                 # 6
# fine della funzione                                       # 7
```

- in 1 definisce il nome funzione e le variabili (**argomenti**) che accetta come parametri
- in 2 c'è la descrizione **docstring**. Attenzione non è un commento!!
    - la docstring genera codice: è parte della funzione
    - provate a scrivere *help(draw_square)*
- dalla linea 2 alla 6 c'è il corpo (**body**) della funzione
- la linea 7 conclude la funzione perchè rientra dell'indentazionie
- **posso usare (chiamare) una funzione solo DOPO che è stata definita** (cioè sotto)

Esercizi

1. Utilizzando come spunto il programma appena analizzato, semplificate il `turtle_program` per disegnare 2 quadrati (`draw_square()`) e triangoli (`draw_triangle()`)

Funzioni chiamano Funzioni

Il **body** di una funzione può contenere la chiamata ad un'altra funzione.

Ad esempio: voglio disegnare un rettangolo con la tartaruga.

1. Mi serve una funzione con 3 parametri: tartaruga, altezza rettangolo, lunghezza rettangolo

```
def draw_rectangle(tartaruga, width, height):
 ''''tartaruga' disegna un rettangolo di lato 'width' e altezza 'height''''
 for i in range(2):     # draw 2 times both sides
     tartaruga.forward(width)
     tartaruga.left(90)
     tartaruga.forward(height)
     tartaruga.left(90)
```

2. Ma adesso che ho il rettangolo so che il quadrato è una *specializzazione* del rettangolo

```
def draw_square(my_turtle, side):
 '''Specializzo 'draw_rectangle' per il quadrato'''
 draw_rectangle(my_turtle, side, side)
```

Considerazioni

- Funzioni possono chiamare altre funzioni
- Se scrivo `draw_square()` in questo modo, sto evidenziando la **relazione** tra rettangolo (generico) e quadrato (specifico)
- Quando *chiamo* la funzione `draw_square(raffaello, 20)`, `raffaello` è assegnato a `my_turtle`, `20` a `side`. Poi posso continuare ad usarli come variabili dentro il *body* della funzione
- La stessa cosa succede con `my_turtle` e `side` in `draw_rectangle(my_turtle, side, side)`
- i nomi delle variabili nelle funzioni sono **indipendenti** da quelli fuori dalle funzioni
- Perchè dovrei creare tutte queste funzioni?
    - Ogni funzione permette di raggruppare delle operazioni
    - Posso semplificare il ragionamento nascondendo le operazioni complesse
    - Ogni funzione (incluso il nome) aiuta a visualizzare la nostra astrazione mentale del problema
    - Le funzioni possono accorciare il numero di righe nei programmi ripetitivi (leggibilità!)

Funzioni che ritornano valori

Finora abbiamo visto **funzioni** `void`: *si eseguono perchè sono utili, non per ottenere un risultato*.

Se vogliamo un produrre un **risultato**, dobbiamo usare la keyword `return`

In [1]:
```
def somma (a, b) :      # header della funzione
    risultato = a + b   # risultato è variabile temporanea
    return risultato    # restituisco risultato
```

Funzioni che ritornano valori

Finora abbiamo visto **funzioni** `void`: *si eseguono perchè sono utili, non per ottenere un risultato*.

Se vogliamo un produrre un **risultato**, dobbiamo usare la keyword `return`

In [1]:
```
def somma (a, b) :      # header della funzione
    risultato = a + b   # risultato è variabile temporanea
    return risultato    # restituisco risultato
```

> *Le **variabili locali** create nelle funzioni, esistono solo **dentro** le funzioni*

Funzioni che ritornano valori

Finora abbiamo visto **funzioni** `void`: *si eseguono perchè sono utili, non per ottenere un risultato*.

Se vogliamo un produrre un **risultato**, dobbiamo usare la keyword `return`

In [1]:
```
def somma (a, b) :      # header della funzione
    risultato = a + b   # risultato è variabile temporanea
    return risultato    # restituisco risultato
```

> *Le **variabili locali** create nelle funzioni, esistono solo **dentro** le funzioni*

In [16]:
```
risultato
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-65ab1212a9e0> in <module>
----> 1 risultato

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

Funzioni che ritornano valori

Finora abbiamo visto **funzioni** `void`: *si eseguono perchè sono utili, non per ottenere un risultato*.

Se vogliamo un produrre un **risultato**, dobbiamo usare la keyword `return`

In [1]:
```
def somma (a, b) :      # header della funzione
    risultato = a + b   # risultato è variabile temporanea
    return risultato    # restituisco risultato
```

> *Le **variabili locali** create nelle funzioni, esistono solo **dentro** le funzioni*

In [16]:
```
risultato
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-65ab1212a9e0> in <module>
----> 1 risultato

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

> *Anche gli **argomenti** delle funzioni sono **variabili locali***

`turtle_program` vol.2

Con le **funzioni** possiamo ristrutturare (**refactoring**) il `turtle program` per raggrupparlo in **blocchi logici più vicini al nostro modo di pensare**

```
import turtle

def make_window(bkg_color, title):
    '''
      Crea una finestra con backgroud e titolo e ritorna la nuova finestra
    '''
    w = turtle.Screen()
    w.bgcolor(bkg_color)
    w.title(title)
    return w

def make_turtle(pen_color, pen_size):
    """
      Create a turtle with the given color and pensize. Returns the new turtle.
    """
    t = turtle.Turtle()
    t.color(pen_color)
    t.pensize(pen_size)
    return t

# main
window = make_window("lightgreen", "Raffaello e Donatello")
raffaello = make_turtle("red", 3)
donatello = make_turtle("violet", 2)
...
```

Esercizi

1. Con l'esempio `turtle_program` "vol.2" fare il refactoring del vostro `turtle_program` per utilizzare le funzioni

2. Aggiungete al turtle program una funzione `draw_poly` per disegnare qualunque poligono regolare dati il numero di lati e il numero di passi

3. Create un programma che disegna 5 quadrati allineati ed equispaziati. la dimensione dei quadrati la fornisce l'utente

4. Disegnate 5 quadrati concentrici, ognuno 20 passi piu' grande del precedente

Funzioni ricorsive

I linguaggi di programmazione in genere supportano la ricorsione

Una funzione può chiamare **qualunque** altra funzione nel suo *body*, anche se stessa

Questo significa che per risolvere un problema, le funzioni possono chiamare se stesse per risolvere un sotto-problema più piccolo

Le funzioni ricorsive richiedono:

- Un calcolo ripetibile basato sulla funzione stessa
- Un caso base che *termina* la ricorsione

Fibonacci

La serie di Fibonacci è un esempio in cui si può usare la ricorsione:

1) calcolo ripetibile: `fib(n) = fib(n-1) + fib(n-2)` .. per `n >= 2`

2) caso base: `fib(0) = 0` e `fib(1) = 1`

Fibonacci

La serie di Fibonacci è un esempio in cui si può usare la ricorsione:

1) calcolo ripetibile: `fib(n) = fib(n-1) + fib(n-2)` .. per `n >= 2`

2) caso base: `fib(0) = 0` e `fib(1) = 1`

In [7]:
```
def fib(n):
    if n <= 1:
        return n
    t = fib(n-1) + fib(n-2)
    return t

fib(4)
```

Out[7]: 3

Fibonacci

La serie di Fibonacci è un esempio in cui si può usare la ricorsione:

1) calcolo ripetibile: `fib(n) = fib(n-1) + fib(n-2)` .. per `n >= 2`

2) caso base: `fib(0) = 0` e `fib(1) = 1`

In [7]:
```
def fib(n):
    if n <= 1:
        return n
    t = fib(n-1) + fib(n-2)
    return t

fib(4)
```

Out[7]:
```
3
```

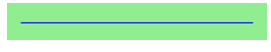
Un altro esempio per utilizzare le funzioni ricorsive è il calcolo del fattoriale.

Frattali

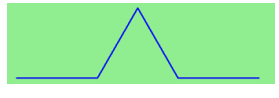
Un altro esempio della ricorsione si può usare per il disegno dei frattali

Proviamo a sidegnare iol frattale di Koch:

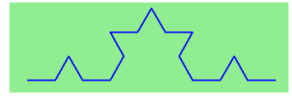
- Il frattale di ordine `0` è una linea di lunghezza `size`.



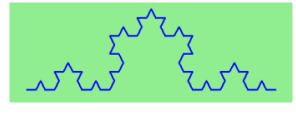
- Il frattale di ordine 1 si ottiene disegnando 4 segmenti:



- Il frattale di ordine 2 si ottiene disegnando frattali di ordine 1 su ognuno dei segmenti



- ripetendo in modo *ricorsivo* su ogni segmento ottenuto dall'ordine 2, si ottiene il frattale di ordine 3



..e così via

In [8]:

```
def koch(t, order, size):
    '''
       La tartaruga tur disegna frattale Koch di 'order' and 'size'.
       lasciando la tartaruga nella direzioni iniziale
    '''

    if order == 0:
        # caso base che termina la ricorsione
        t.forward(size)
    else:
        # caso ricorsivo (ordine 1) in cui disegna i 4 segmenti
        koch(t, order-1, size/3)
        t.left(60)
        koch(t, order-1, size/3)
        t.right(120)
        koch(t, order-1, size/3)
        t.left(60)
        koch(t, order-1, size/3)
```

.. o meglio

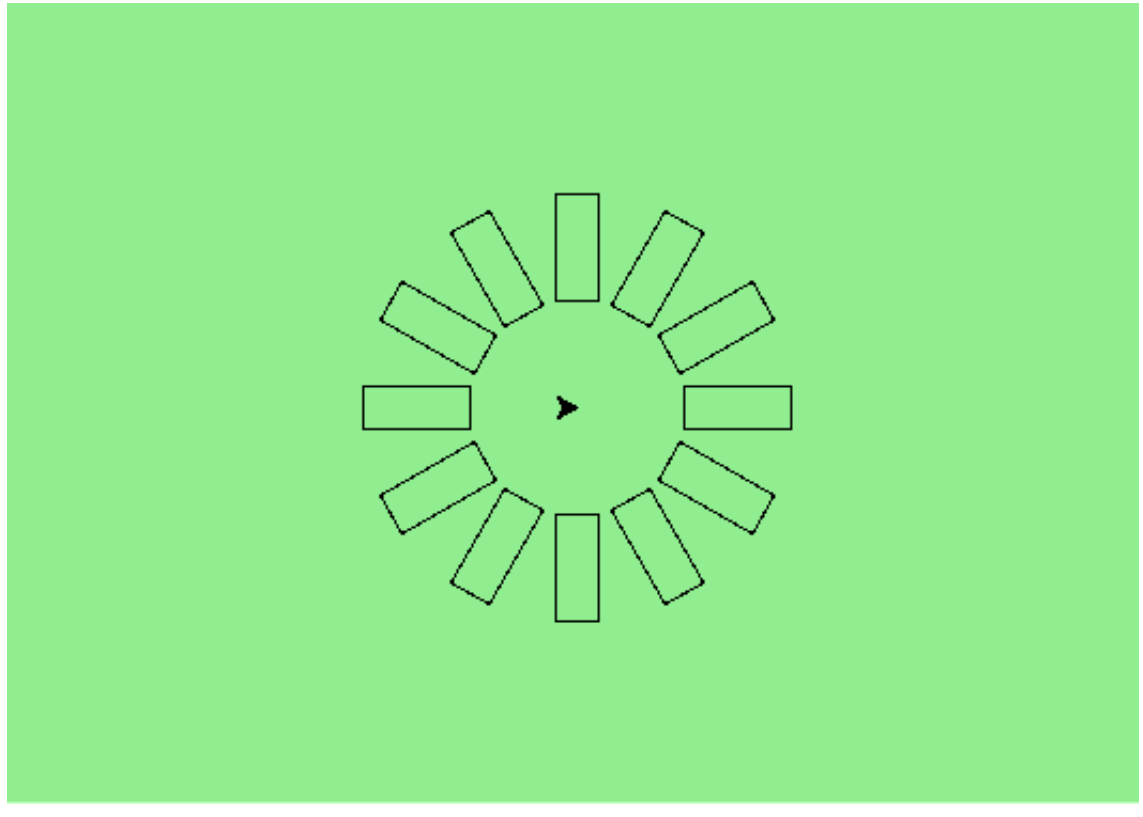
.. o meglio

In [10]:

```
def koch(t, order, size):
    if order == 0:
        t.forward(size)
    else:
        for angle in [60, -120, 60, 0]:
           koch(t, order-1, size/3)
           t.left(angle)
```

Esercizi

Utilizzando funzioni e loop control, disegnate questa figura "orologio"



e modificate il vostro programma per disegnare un poligono diqualsiasi dimensione in modo che disegni un 'orologio' con un numero qualsiasi di tacche per l'ora

Riprendiamo in mano le stringhe

Nelle lezioni precedenti abbiamo familiarizzato con il concetto di **tipo** di dati e siamo partiti dai tipi *built-in*:

- `int`
- `float`
- `str`
- `bool`

Abbiamo visto che la funzione *built-in* `len()` ci mostra la lunghezza di una stringa, ovvero il numero di caratteri che la compone:

Riprendiamo in mano le stringhe

Nelle lezioni precedenti abbiamo familiarizzato con il concetto di **tipo** di dati e siamo partiti dai tipi *built-in*:

- `int`
- `float`
- `str`
- `bool`

Abbiamo visto che la funzione *built-in* `len()` ci mostra la lunghezza di una stringa, ovvero il numero di caratteri che la compone:

In [4]:
```
len('Michelangelo')
```

Out[4]:
```
12
```

La stessa funzione, ci ritorna un `TypeError` se proviamo a passare un `int` o un `float` come argomento:

La stessa funzione, ci ritorna un `TypeError` se proviamo a passare un `int` o un `float` come argomento:

In [2]:
```
len(11)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-54a18e5e7c89> in <module>
----> 1 len(11)

TypeError: object of type 'int' has no len()
```

Gli **oggetti** di tipo `str` sono diversi rispetto agli altri tipi *built-in*: `int`, `float` o `bool`. Le stringhe rappresentano un tipo di dati **composto**: la singola **istanza** dell'**oggetto** di tipo `str` è composta da diverse unità, ognuna delle quali è rappresentata dal singolo carattere.

A seconda di quello che vogliamo fare possiamo trattare i nosri **tipi composti** (*compuond data type*) come singole entintà oppure possiamo accedere alle singole unità che li compongono, attraverso l'operatore di indicizzazione `[]`.

Se volessimo, ad esempio, sapere qual'è il primo elemento che compone la stringa `Michelangelo`:

Gli **oggetti** di tipo `str` sono diversi rispetto agli altri tipi *built-in*: `int`, `float` o `bool`. Le stringhe rappresentano un tipo di dati **composto**: la singola **istanza** dell'**oggetto** di tipo `str` è composta da diverse unità, ognuna delle quali è rappresentata dal singolo carattere.

A seconda di quello che vogliamo fare possiamo trattare i nosri **tipi composti** (*compuond data type*) come singole entintà oppure possiamo accedere alle singole unità che li compongono, attraverso l'operatore di indicizzazione `[]`.

Se volessimo, ad esempio, sapere qual'è il primo elemento che compone la stringa `Michelangelo`:

In [5]:
```
nome = "Michelangelo"
nome[0]
```

Out[5]:
```
'M'
```

Gli **oggetti** di tipo `str` sono diversi rispetto agli altri tipi *built-in*: `int`, `float` o `bool`. Le stringhe rappresentano un tipo di dati **composto**: la singola **istanza** dell'**oggetto** di tipo `str` è composta da diverse unità, ognuna delle quali è rappresentata dal singolo carattere.

A seconda di quello che vogliamo fare possiamo trattare i nosri **tipi composti** (*compuond data type*) come singole entintà oppure possiamo accedere alle singole unità che li compongono, attraverso l'operatore di indicizzazione `[]`.

Se volessimo, ad esempio, sapere qual'è il primo elemento che compone la stringa `Michelangelo`:

In [5]:
```
nome = "Michelangelo"
nome[0]
```

Out[5]:
```
'M'
```

> *Ricordatevi che Python è zero-based:* ***si comincia a contare da 0!***
>
> *Un piccolo trucco: Python sa contare anche al contrario, utilizzando indici negativi. Provate a chiedere all'interprete il risultato di:* `nome[-1]`

Ed essendo **istanze** del tipo `str`, hanno a disposizione una serie di metodi ed attributi utili, proprio come `raffaello` aveva a disposizione il metodo `turn()` o l'attributo `color()`, per il fatto di essere un'**istanza** del tipo `Turtle`.

Il tipo `str` ci mette a disposizine tutta una serie di metodi per lavorare con il formato del testo, come ad esempio:

Ed essendo **istanze** del tipo `str`, hanno a disposizione una serie di metodi ed attributi utili, proprio come `raffaello` aveva a disposizione il metodo `turn()` o l'attributo `color()`, per il fatto di essere un'**istanza** del tipo `Turtle`.

Il tipo `str` ci mette a disposizine tutta una serie di metodi per lavorare con il formato del testo, come ad esempio:

In [4]:
```
saluto = "Hello World!"
salutone = saluto.upper()
salutone
```

Out[4]:
```
'HELLO WORLD!'
```

Ed essendo **istanze** del tipo `str`, hanno a disposizione una serie di metodi ed attributi utili, proprio come `raffaello` aveva a disposizione il metodo `turn()` o l'attributo `color()`, per il fatto di essere un'**istanza** del tipo `Turtle`.

Il tipo `str` ci mette a disposizine tutta una serie di metodi per lavorare con il formato del testo, come ad esempio:

In [4]:
```
saluto = "Hello World!"
salutone = saluto.upper()
salutone
```

Out[4]:
```
'HELLO WORLD!'
```

In [5]:
```
salutone = saluto.swapcase()
salutone
```

Out[5]:
```
'hELLO wORLD!'
```

Ed essendo **istanze** del tipo `str`, hanno a disposizione una serie di metodi ed attributi utili, proprio come `raffaello` aveva a disposizione il metodo `turn()` o l'attributo `color()`, per il fatto di essere un'**istanza** del tipo `Turtle`.

Il tipo `str` ci mette a disposizine tutta una serie di metodi per lavorare con il formato del testo, come ad esempio:

In [4]:
```
saluto = "Hello World!"
salutone = saluto.upper()
salutone
```

Out[4]:
```
'HELLO WORLD!'
```

In [5]:
```
salutone = saluto.swapcase()
salutone
```

Out[5]:
```
'hELLO wORLD!'
```

> *In tutto questo, il nostro **oggetto** `saluto` è rimasto immutato.*
> *`saluto` e `salutone` sono due **istanze** diverse e noi abbiamo modificato `salutone`*

Essendo scomponibili nelle loro singoli componenti, le stringhe possono essere utilizzate come sequenza nei loop `for`:

Essendo scomponibili nelle loro singoli componenti, le stringhe possono essere utilizzate come sequenza nei loop `for`:

In [6]:
```
for lettera in nome:
    print(lettera)
```

```
M
i
c
h
e
l
a
n
g
e
l
o
```

Avevamo già visto nel *turtle_program* un loop su di un **tipo** particolare di raccolta di valori:

```
for c in ['red', 'purple', 'yellow', 'blue']:
    raffaello.color(c)
```

Avevamo già visto nel *turtle_program* un loop su di un **tipo** particolare di raccolta di valori:

```
for c in ['red', 'purple', 'yellow', 'blue']:
    raffaello.color(c)
```

L'**oggetto**:

```
['red', 'purple', 'yellow', 'blue']
```

è una **lista**, ed è una *Data Structure*: un contenitore in cui possiamo organizzare dati: possiamo accedere facilmente ai diversi elementi ed intervenire per fare delle modifiche (se il tipo di *Data Structure* che stiamo usando lo consente.

In questo corso, vedremo le quattro principali *Data Structures* che Python mette a disposizione:

- La lista: `list`
- La tupla: `tuple`
- Il set: `set`
- Il dizionario: `dict`

Esercizi

1. Create una variabile che contenga la scritta "Ronaldo,Zlatan,Lukaku" e e chiamate il metodo `split()` passando l'argomento `,`
2. Date le espressioni:

   ```
   compatto = "Ronaldo,Zlatan,Lukaku"
   diviso = compatto.split(',')
   ```

   che cosa ottengo applicando l'operatore di indicizzazione `[]` su `compatto` e `diviso`?

Liste

La lista è una **collezione ordinata** di valori. I valori che compongono la lista vengono chiamati **elementi** (*item*) della lista.

Un **oggetto** di tipo `list` si definisce racchiudendo gli elementi che lo compongono tra parentesi quadre `[ ]`, separati da `,`.

Gli **elementi** di una lista possono essere di **qualunque tipo**:

```
lista_numeri = [7,11,9]                        # Questa è una lista di interi
lista_nomi   = ['Ronaldo', 'Zlatan', 'Lukaku'] # Questa è una lista di stringhe
```

Liste

La lista è una **collezione ordinata** di valori. I valori che compongono la lista vengono chiamati **elementi** (*item*) della lista.

Un **oggetto** di tipo `list` si definisce racchiudendo gli elementi che lo compongono tra parentesi quadre `[ ]`, separati da `,`.

Gli **elementi** di una lista possono essere di **qualunque tipo**:

```
lista_numeri = [7,11,9]                          # Questa è una lista di interi
lista_nomi   = ['Ronaldo', 'Zlatan', 'Lukaku']   # Questa è una lista di stringhe
```

Non tutti gli elementi della lista devono essere dello stesso tipo:

```
lista_mista = [11,'Zlatan',1.95, True]   # Questa è una lista di valori di diverso tipo
```

Liste

La lista è una **collezione ordinata** di valori. I valori che compongono la lista vengono chiamati **elementi** (*item*) della lista.

Un **oggetto** di tipo `list` si definisce racchiudendo gli elementi che lo compongono tra parentesi quadre `[ ]`, separati da `,`.

Gli **elementi** di una lista possono essere di **qualunque tipo**:

```
lista_numeri = [7,11,9]                        # Questa è una lista di interi
lista_nomi   = ['Ronaldo', 'Zlatan', 'Lukaku'] # Questa è una lista di stringhe
```

Non tutti gli elementi della lista devono essere dello stesso tipo:

```
lista_mista = [11,'Zlatan',1.95, True]   # Questa è una lista di valori di diverso tipo
```

> *Gli oggetti di tipo `str` sono collezioni ordinate di caratteri. Nel caso delle stringhe **tutti** gli elementi della collezione sono dello stesso tipo*

Liste annidate

`list` è un tipo, quindi nulla vieta ad un **oggetto** di tipo `list` di essere un elemento di una lista:

```
lista_liste = [11,'Zlatan',[1.95, 95]]   # Questa è una lista che contiene
                                         # un oggetto di tipo list
```

Una lista contenuta da un'altra lista viene definita **annidata** (*nested list*).

Liste annidate

`list` è un tipo, quindi nulla vieta ad un **oggetto** di tipo `list` di essere un elemento di una lista:

```
lista_liste = [11,'Zlatan',[1.95, 95]]   # Questa è una lista che contiene
                                         # un oggetto di tipo list
```

Una lista contenuta da un'altra lista viene definita **annidata** (*nested list*).

Liste vuote

Possiamo anche creare un oggetto `list` senza elementi:

```
lista_vuota = []   # Questa è una lista vuota: non contiene nessun elemento
```

Una lista che non contiene elementi viene defintia **vuota** (*empty list*).

Accedere agli elementi

Come le stringhe, anche le liste sono un **tipo composto**. Significa che possiamo accedere ai singoli elementi della lista con l'operatore di indicizzazione `[]`:

Accedere agli elementi

Come le stringhe, anche le liste sono un **tipo composto**. Significa che possiamo accedere ai singoli elementi della lista con l'operatore di indicizzazione `[]`:

In [6]:
```
lista_nomi = ['Ronaldo', 'Zlatan', 'Lukaku']
lista_nomi[1]
```

Out[6]:
```
'Zlatan'
```

Possiamo anche "ritagliare" una sezione (*slice*) della lista che abbiamo definito. Per selezionare (*slicing*) gli elementi che ci interessano, possiamo usare l'operatore di indicizzazione definendo l'indice di partenza e l'indice di arrivo separati da `:`:

Possiamo anche "ritagliare" una sezione (*slice*) della lista che abbiamo definito. Per selezionare (*slicing*) gli elementi che ci interessano, possiamo usare l'operatore di indicizzazione definendo l'indice di partenza e l'indice di arrivo separati da `:`:

In [9]:
```
lista_nomi = ['Ronaldo', 'Zlatan', 'Lukaku', 'Immobile', 'Belotti', 'Dzeko']
lista_nomi[0:3]     # Slice contenente i primi tre elementi della lista
```

Out[9]:
```
['Ronaldo', 'Zlatan', 'Lukaku']
```

Possiamo anche "ritagliare" una sezione (*slice*) della lista che abbiamo definito. Per selezionare (*slicing*) gli elementi che ci interessano, possiamo usare l'operatore di indicizzazione definendo l'indice di partenza e l'indice di arrivo separati da `:`:

In [9]:
```
lista_nomi = ['Ronaldo', 'Zlatan', 'Lukaku', 'Immobile', 'Belotti', 'Dzeko']
lista_nomi[0:3]     # Slice contenente i primi tre elementi della lista
```

Out[9]:
```
['Ronaldo', 'Zlatan', 'Lukaku']
```

In [10]:
```
lista_nomi[2:4]     # Slice contenente i due elementi centrali della lista
```

Out[10]:
```
['Lukaku', 'Immobile']
```

Possiamo anche "ritagliare" una sezione (*slice*) della lista che abbiamo definito. Per selezionare (*slicing*) gli elementi che ci interessano, possiamo usare l'operatore di indicizzazione definendo l'indice di partenza e l'indice di arrivo separati da `:`:

In [9]:
```
lista_nomi = ['Ronaldo', 'Zlatan', 'Lukaku', 'Immobile', 'Belotti', 'Dzeko']
lista_nomi[0:3]     # Slice contenente i primi tre elementi della lista
```

Out[9]:
```
['Ronaldo', 'Zlatan', 'Lukaku']
```

In [10]:
```
lista_nomi[2:4]     # Slice contenente i due elementi centrali della lista
```

Out[10]:
```
['Lukaku', 'Immobile']
```

> *L'intervallo definito con questa sintassi è **aperto a destra**!*
>
> `[0:3]` *equivale all'intervallo* $0 \leq indice < 3$
>
> *L'indice di arrivo è escluso!*

Esercizi

1. Create una lista e provate a passare un `float` all'operatore di indicizzazione `[]`
2. Create una lista e provate a passare il valore di un elemento all'operatore di indicizzazione `[]`

Le liste nei loop for

Come abbiamo visto nel `turtle_program` possiamo usare una lista come sequenza su cui costruire un loop `for`:

Le liste nei loop for

Come abbiamo visto nel `turtle_program` possiamo usare una lista come sequenza su cui costruire un loop `for`:

In [19]:
```
for bomber in lista_nomi:
    print(bomber)
```

```
Ronaldo
Zlatan
Lukaku
Immobile
Belotti
Dzeko
```

Le liste nei loop for

Come abbiamo visto nel `turtle_program` possiamo usare una lista come sequenza su cui costruire un loop `for`:

In [19]:
```
for bomber in lista_nomi:
    print(bomber)
```

```
Ronaldo
Zlatan
Lukaku
Immobile
Belotti
Dzeko
```

In questo modo stiamo stiamo costruendo il loop accedendo direttamente ai singoli elementi della lista:

> *per ogni **elemento** della **lista**: stampa elemento*

Possiamo anche "complicare" la sintassi ed accedere ai singoli elementi tramite il loro indice:

Possiamo anche "complicare" la sintassi ed accedere ai singoli elementi tramite il loro indice:

In [22]:
```
lunghezza_lista = len(lista_nomi)
for i in range(lunghezza_lista):
    print(lista_nomi[i])
```

```
Ronaldo
Zlatan
Lukaku
Immobile
Belotti
Dzeko
```

Possiamo anche "complicare" la sintassi ed accedere ai singoli elementi tramite il loro indice:

In [22]:
```
lunghezza_lista = len(lista_nomi)
for i in range(lunghezza_lista):
    print(lista_nomi[i])
```

```
Ronaldo
Zlatan
Lukaku
Immobile
Belotti
Dzeko
```

In questo caso, stiamo dicendo una cosa leggermente diversa, sia per sintassi che per semantica:

> *per ogni **indice** nell'**intervallo** che va da 0 a lunghezza della **lista**: stampa l'elemento che corrisponde all'indice*

Modificare le liste

Le liste sono una *Data Structure* **modificabile**. Significa che possiamo modificarne il contenuto, aggiornando ad esempio un **elemento** della lista, usando l'operatore di indicizzazione `[]`:

Modificare le liste

Le liste sono una *Data Structure* **modificabile**. Significa che possiamo modificarne il contenuto, aggiornando ad esempio un **elemento** della lista, usando l'operatore di indicizzazione `[]`:

In [3]:
```
lista_numeri = [7, 11, 9, 17]
lista_numeri[2] = 99
lista_numeri
```

Out[3]:
```
[7, 11, 99, 17]
```

Modificare le liste

Le liste sono una *Data Structure* **modificabile**. Significa che possiamo modificarne il contenuto, aggiornando ad esempio un **elemento** della lista, usando l'operatore di indicizzazione `[]`:

In [3]:
```
lista_numeri = [7, 11, 9, 17]
lista_numeri[2] = 99
lista_numeri
```

Out[3]:
```
[7, 11, 99, 17]
```

In questo esempio abbiamo (ri)assegnato all'elemento `2` di `lista_numeri` il valore `99`. Assegnare un valore ad un elemento di una *Data Structure* prende il nome di **item assignment**.

Non tutte le *Data Structure* permettono un operazione di **item assignment**. Il tipo `str`, ad esempio, **NON** è modificabile:

Non tutte le *Data Structure* permettono un operazione di **item assignment**. Il tipo `str`, ad esempio, **NON** è modificabile:

In [4]:
```
nome = 'Michelangelo'
nome[-1] = 'a'
nome
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-4928e6e8fe7d> in <module>
      1 nome = 'Michelangelo'
----> 2 nome[-1] = 'a'
      3 nome

TypeError: 'str' object does not support item assignment
```

Possiamo anche utilizzare l'operazione di *slicing* per modificare intere sezioni della nostra lista, senza dover modificare un elemento all volta:

Possiamo anche utilizzare l'operazione di *slicing* per modificare intere sezioni della nostra lista, senza dover modificare un elemento all volta:

In [5]:
```
lista_lettere = ['P', 'E', 'N', 'N', 'A', 'R', 'E', 'L', 'L', 'O']
lista_lettere[0:5] = ['M', 'A', 'T', 'T', 'A']
lista_lettere
```

Out[5]:
```
['M', 'A', 'T', 'T', 'A', 'R', 'E', 'L', 'L', 'O']
```

Possiamo anche utilizzare l'operazione di *slicing* per modificare intere sezioni della nostra lista, senza dover modificare un elemento all volta:

In [5]:
```
lista_lettere = ['P', 'E', 'N', 'N', 'A', 'R', 'E', 'L', 'L', 'O']
lista_lettere[0:5] = ['M', 'A', 'T', 'T', 'A']
lista_lettere
```

Out[5]:
```
['M', 'A', 'T', 'T', 'A', 'R', 'E', 'L', 'L', 'O']
```

Grazie a questo "trucchetto" e a un po' di furbizia, possiamo eliminare determinati elementi dalla lista, riassegnado un oggetto di tipo **lista vuota** agli elementi nelle posizini desiderate:

Possiamo anche utilizzare l'operazione di *slicing* per modificare intere sezioni della nostra lista, senza dover modificare un elemento all volta:

In [5]:
```
lista_lettere = ['P', 'E', 'N', 'N', 'A', 'R', 'E', 'L', 'L', 'O']
lista_lettere[0:5] = ['M', 'A', 'T', 'T', 'A']
lista_lettere
```

Out[5]:
```
['M', 'A', 'T', 'T', 'A', 'R', 'E', 'L', 'L', 'O']
```

Grazie a questo "trucchetto" e a un po' di furbizia, possiamo eliminare determinati elementi dalla lista, riassegnado un oggetto di tipo **lista vuota** agli elementi nelle posizini desiderate:

In [7]:
```
lista_lettere[5:10] = []
lista_lettere
```

Out[7]:
```
['M', 'A', 'T', 'T', 'A']
```

Esercizi

1. Partendo dalla lista che abbiamo appena costruito:

   ```
   lista_lettere = ['M','A', 'T', 'T', 'A']
   ```

   cosa otteniamo dal seguente **item assignment**?

   ```
   lista_lettere[4:4] = ['I','N']
   ```

Operazioni con le liste

Come per le stringhe, anche nel caso delle liste gli operatori `+` e `*` assumono il significato di **concatenazione**:

Operazioni con le liste

Come per le stringhe, anche nel caso delle liste gli operatori `+` e `*` assumono il significato di **concatenazione**:

In [1]:
```
tartarughe = ["Raffaello", "Donatello"]
nuove_tartarughe = ["Michelangelo", "Leonardo"]
tartarughe + nuove_tartarughe
```

Out[1]:
```
['Raffaello', 'Donatello', 'Michelangelo', 'Leonardo']
```

Operazioni con le liste

Come per le stringhe, anche nel caso delle liste gli operatori `+` e `*` assumono il significato di **concatenazione**:

In [1]:
```
tartarughe = ["Raffaello", "Donatello"]
nuove_tartarughe = ["Michelangelo", "Leonardo"]
tartarughe + nuove_tartarughe
```

Out[1]:
```
['Raffaello', 'Donatello', 'Michelangelo', 'Leonardo']
```

e **ripetizione**:

Operazioni con le liste

Come per le stringhe, anche nel caso delle liste gli operatori `+` e `*` assumono il significato di **concatenazione**:

In [1]:
```
tartarughe = ["Raffaello", "Donatello"]
nuove_tartarughe = ["Michelangelo", "Leonardo"]
tartarughe + nuove_tartarughe
```

Out[1]:
```
['Raffaello', 'Donatello', 'Michelangelo', 'Leonardo']
```

e **ripetizione**:

In [2]:
```
tartarughe * 3
```

Out[2]:
```
['Raffaello', 'Donatello', 'Raffaello', 'Donatello', 'Raffaello', 'Donatello']
```

Cancellare elementi

Fortunatamente Python mette a disposizione un metodo un po' più intuitivo per eliminare elementi di una lista. Possiamo usare la **keyword** `del`:

Cancellare elementi

Fortunatamente Python mette a disposizione un metodo un po' più intuitivo per eliminare elementi di una lista. Possiamo usare la **keyword** `del`:

In [9]:
```
lista_numeri = [7, 11, 9, 17]
del lista_numeri[2]
lista_numeri
```

Out[9]:
```
[7, 11, 17]
```

Cancellare elementi

Fortunatamente Python mette a disposizione un metodo un po' più intuitivo per eliminare elementi di una lista. Possiamo usare la **keyword** `del`:

In [9]:
```
lista_numeri = [7, 11, 9, 17]
del lista_numeri[2]
lista_numeri
```

Out[9]:
```
[7, 11, 17]
```

Metodi e attributi

Anche il tipo `list` fornisce una serie di metodi ed attributi utili, come `append()`, che consente di appendere un elemento in fondo alla lista:

Cancellare elementi

Fortunatamente Python mette a disposizione un metodo un po' più intuitivo per eliminare elementi di una lista. Possiamo usare la **keyword** `del`:

In [9]:
```
lista_numeri = [7, 11, 9, 17]
del lista_numeri[2]
lista_numeri
```

Out[9]:
```
[7, 11, 17]
```

Metodi e attributi

Anche il tipo `list` fornisce una serie di metodi ed attributi utili, come `append()`, che consente di appendere un elemento in fondo alla lista:

In [8]:
```
tartarughe = ['Raffaello', 'Donatello', 'Michelangelo']
tartarughe.append('Leonardo')
tartarughe
```

Out[8]:
```
['Raffaello', 'Donatello', 'Michelangelo', 'Leonardo']
```

Esercizi

1. Create una lista e provate a usare l'operatore `del` su una *slice*
2. Create una lista e provate ad utilizzare i metodi:
    - `insert()`
    - `count()`
    - `extend()`
    - `index()`
    - `sort()`
    - `remove()`

Tuple

Vediamo un'altra *Data Structure*: la **tupla**.

Le tuple sono utili per raggruppare insieme informazioni in qualche modo correlate tra loro. Non c'è nessuna descrizione di cosa indichi il singolo elemento, ma possiamo dedurlo. In sunto, la tupla ci consente di raggruppare informazioni correlate tra loro ed utilizzarle come un unico **oggetto**.

Un **oggetto** di tipo `tuple` è una sequenza di valori separati da virgola `,` (*comma-separated*). Sebbene non sia necessario, per convenzione si racchiude la sequenza tra parentesi tonde `( )`.

Costruiamo una tupla:

Tuple

Vediamo un'altra *Data Structure*: la **tupla**.

Le tuple sono utili per raggruppare insieme informazioni in qualche modo correlate tra loro. Non c'è nessuna descrizione di cosa indichi il singolo elemento, ma possiamo dedurlo. In sunto, la tupla ci consente di raggruppare informazioni correlate tra loro ed utilizzarle come un unico **oggetto**.

Un **oggetto** di tipo `tuple` è una sequenza di valori separati da virgola `,` (*comma-separated*). Sebbene non sia necessario, per convenzione si racchiude la sequenza tra parentesi tonde `( )`.

Costruiamo una tupla:

In [11]:
```
ibra = ("Zlatan", "Ibrahimovic", 11, "Milan", 1.95, 95, "Attaccante")
```

Il tipo `tuple` è **immutabile**, quindi, come per `str`, non possiamo modificare un'istanza di tipo `tuple` una volta che è stata creata, a differenza di quanto facevamo con le `list`.

Possiamo quindi accedere ai singoli elementi della tupla, con il solito operatore `[]`:

Il tipo `tuple` è **immutabile**, quindi, come per `str`, non possiamo modificare un'istanza di tipo `tuple` una volta che è stata creata, a differenza di quanto facevamo con le `list`.

Possiamo quindi accedere ai singoli elementi della tupla, con il solito operatore `[]`:

In [12]:
```
numero = ibra[2]
numero
```

Out[12]:
```
11
```

Il tipo `tuple` è **immutabile**, quindi, come per `str`, non possiamo modificare un'istanza di tipo `tuple` una volta che è stata creata, a differenza di quanto facevamo con le `list`.

Possiamo quindi accedere ai singoli elementi della tupla, con il solito operatore `[]`:

In [12]:
```
numero = ibra[2]
numero
```

Out[12]:
```
11
```

Ma non possiamo fare un **item assignment**:

Il tipo `tuple` è **immutabile**, quindi, come per `str`, non possiamo modificare un'istanza di tipo `tuple` una volta che è stata creata, a differenza di quanto facevamo con le `list`.

Possiamo quindi accedere ai singoli elementi della tupla, con il solito operatore `[]`:

In [12]:
```
numero = ibra[2]
numero
```

Out[12]:
```
11
```

Ma non possiamo fare un **item assignment**:

In [13]:
```
ibra[2] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-dff14e41454f> in <module>
----> 1 ibra[2] = 10

TypeError: 'tuple' object does not support item assignment
```

Esercizi

1. Create una tupla e provate a fare un'operazione di *slicing*
2. Provate a chiamare la funzione `type()` sui seguenti due oggetti:

```
tupla_test = (11)
tupla_test = (11,)
```

Tuple assignment

Una funzionalità molto comoda che Python mette a disposizione è l'assegnazione di variabili tramite tuple, o **tuple assignment**.

Il **tuple assignment** ci consente di assegnare un **tupla di variabili** (*lvalue*) a una **tupla di valori** (*rvalue*):

Tuple assignment

Una funzionalità molto comoda che Python mette a disposizione è l'assegnazione di variabili tramite tuple, o **tuple assignment**.

Il **tuple assignment** ci consente di assegnare un **tupla di variabili** (*lvalue*) a una **tupla di valori** (*rvalue*):

In [14]:
```
(nome, cognome, numero, squadra, altezza, peso, ruolo) = ibra
```

Tuple assignment

Una funzionalità molto comoda che Python mette a disposizione è l'assegnazione di variabili tramite tuple, o **tuple assignment**.

Il **tuple assignment** ci consente di assegnare un **tupla di variabili** (*lvalue*) a una **tupla di valori** (*rvalue*):

In [14]:
```
(nome, cognome, numero, squadra, altezza, peso, ruolo) = ibra
```

Nell'esempio abbiamo assegnato 7 variabili ad altrettanti valori in una sola riga:

Tuple assignment

Una funzionalità molto comoda che Python mette a disposizione è l'assegnazione di variabili tramite tuple, o **tuple assignment**.

Il **tuple assignment** ci consente di assegnare un **tupla di variabili** (*lvalue*) a una **tupla di valori** (*rvalue*):

In [14]:
```
(nome, cognome, numero, squadra, altezza, peso, ruolo) = ibra
```

Nell'esempio abbiamo assegnato 7 variabili ad altrettanti valori in una sola riga:

In [15]:
```
print(nome, cognome, numero, squadra, altezza, peso, ruolo)
```

```
Zlatan Ibrahimovic 11 Milan 1.95 95 Attaccante
```

Dobbiamo solo fare attenzione alla lunghezza delle tuple a sinistra e a destra: la **tupla di variabili** deve avere lo stesso numero di elementi della **tupla di valori**:

Dobbiamo solo fare attenzione alla lunghezza delle tuple a sinistra e a destra: la **tupla di variabili** deve avere lo stesso numero di elementi della **tupla di valori**:

In [16]:
```
(nome, cognome, numero, squadra, altezza, peso) = ibra
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-983c09b70e94> in <module>
----> 1 (nome, cognome, numero, squadra, altezza, peso) = ibra

ValueError: too many values to unpack (expected 6)
```

Dobbiamo solo fare attenzione alla lunghezza delle tuple a sinistra e a destra: la **tupla di variabili** deve avere lo stesso numero di elementi della **tupla di valori**:

In [16]:
```
(nome, cognome, numero, squadra, altezza, peso) = ibra
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-983c09b70e94> in <module>
----> 1 (nome, cognome, numero, squadra, altezza, peso) = ibra

ValueError: too many values to unpack (expected 6)
```

In [17]:
```
(nome, cognome, numero, squadra, altezza, peso, ruolo, gol_segnati) = ibra
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-0e696d597ff0> in <module>
----> 1 (nome, cognome, numero, squadra, altezza, peso, ruolo, gol_segnati) = ibr
a

ValueError: not enough values to unpack (expected 8, got 7)
```

`tuple` e `return`

Abbiamo visto come le funzioni possano restituire un certo valore, grazie alla keyword `return`.

Le funzioni possono restituire un solo valore e in molti casi questo non è un grosso limite:

`tuple` e `return`

Abbiamo visto come le funzioni possano restituire un certo valore, grazie alla keyword `return`.

Le funzioni possono restituire un solo valore e in molti casi questo non è un grosso limite:

In [18]:
```
def add(a, b):
    result = a + b
    return result
```

`tuple` e `return`

Abbiamo visto come le funzioni possano restituire un certo valore, grazie alla keyword `return`.

Le funzioni possono restituire un solo valore e in molti casi questo non è un grosso limite:

In [18]:
```
def add(a, b):
    result = a + b
    return result
```

In [23]:
```
somma = add(6, 4)
print(somma)
```

```
10
```

Potremmo però scrivere una funzione che restituisca due valori: la somma e la differenza di due numeri. La **tupla** ci consente di raggruppare insieme quanti più valori vogliamo e di restituire un solo **oggetto** di tipo `tuple` tramite `return`:

Potremmo però scrivere una funzione che restituisca due valori: la somma e la differenza di due numeri. La **tupla** ci consente di raggruppare insieme quanti più valori vogliamo e di restituire un solo **oggetto** di tipo `tuple` tramite `return`:

In [21]:
```
def add_and_diff(a, b):
    sum_result = a + b
    diff_result = a - b
    tuple_result = (sum_result, diff_result)
    return tuple_result
```

Potremmo però scrivere una funzione che restituisca due valori: la somma e la differenza di due numeri. La **tupla** ci consente di raggruppare insieme quanti più valori vogliamo e di restituire un solo **oggetto** di tipo `tuple` tramite `return`:

In [21]:
```
def add_and_diff(a, b):
    sum_result = a + b
    diff_result = a - b
    tuple_result = (sum_result, diff_result)
    return tuple_result
```

In [24]:
```
(somma, differenza) = add_and_diff(6, 4)
print(somma)
print(differenza)
```

```
10
2
```

Esercizi

1. Scrivete una funzione che ritorni 3 diversi valori utilizzando la sintassi:

   ```
   return valore_1, valore_2, valore_3
   ```

   e chiamatela per inizializzare tre diverse variabili

Set

Vediamo un terzo tipo di *Data Structure*: `set`

Un **set** è una collezione di elementi **non ordinata**. A differenza di liste e tuple, ogni elemento del set deve essere **unico** e **non modificabile**.

Tuttavia l'**oggetto** di tipo `set` è modificabile.

Un **oggetto** di tipo `set` si definisce racchiudendo gli elementi che lo compongono tra parentesi graffe `{ }`, separati da `,`.

Un altro modo per costruire un set è quello di usare la funzione *built-in* `set()`.

Vediamo un po' nella pratica quanto abbiamo detto:

Creiamo un set:

Vediamo un po' nella pratica quanto abbiamo detto:

Creiamo un set:

In [27]:
```
set_numeri = {7, 11, 9, 17}
```

Vediamo un po' nella pratica quanto abbiamo detto:

Creiamo un set:

In [27]:
```
set_numeri = {7, 11, 9, 17}
```

Il **set** è una collezioni **non ordinata**: significa che non possimo fare affidamento sull'indicizzazione degli elementi:

Vediamo un po' nella pratica quanto abbiamo detto:

Creiamo un set:

In [27]:
```
set_numeri = {7, 11, 9, 17}
```

Il **set** è una collezioni **non ordinata**: significa che non possimo fare affidamento sull'indicizzazione degli elementi:

In [26]:
```
set_numeri[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-9bd0d40e9c43> in <module>
----> 1 set_numeri[1]

TypeError: 'set' object is not subscriptable
```

Inoltre, ogni elemento del set deve essere **unico**:

Inoltre, ogni elemento del set deve essere **unico**:

In [31]:
```
set_numeri = {7, 11, 9, 17, 9}
set_numeri
```

Out[31]:
```
{7, 9, 11, 17}
```

Inoltre, ogni elemento del set deve essere **unico**:

In [31]:
```
set_numeri = {7, 11, 9, 17, 9}
set_numeri
```

Out[31]:
```
{7, 9, 11, 17}
```

E **non modificabile**:

Inoltre, ogni elemento del set deve essere **unico**:

In [31]:
```
set_numeri = {7, 11, 9, 17, 9}
set_numeri
```

Out[31]:
```
{7, 9, 11, 17}
```

E **non modificabile**:

In [29]:
```
set_numeri = {7, 11, 9, [17, 9]}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-35cf0909e3fb> in <module>
----> 1 set_numeri = {7, 11, 9, [17, 9]}

TypeError: unhashable type: 'list'
```

Esercizi

1. Create un set con un elemento di tipo `set` contenente un elemento di tipo `str`
2. Create un set con un elemento di tipo `set` contenente un elemento di tipo `tuple`
3. Create un oggetto `{}` e passatelo alla funzione `type()`

Modificare un set

I **set** sono collezioni **modificabili**. Il tipo `set` mette a disposizione i metodi `add()`, `update()`, `discard()` e `remove()` per aggungere e rimuovere elementi.

Il metodo `add()` aggiunge **un solo elemento** mentre `update()` ci permette di aggiungere **più elementi** al set:

Modificare un set

I **set** sono collezioni **modificabili**. Il tipo `set` mette a disposizione i metodi `add()`, `update()`, `discard()` e `remove()` per aggungere e rimuovere elementi.

Il metodo `add()` aggiunge **un solo elemento** mentre `update()` ci permette di aggiungere **più elementi** al set:

In [36]:
```
set_numeri = {7, 11, 9}
print(set_numeri)
set_numeri.add(17)
print(set_numeri)
```

```
{9, 11, 7}
{9, 11, 17, 7}
```

Modificare un set

I **set** sono collezioni **modificabili**. Il tipo `set` mette a disposizione i metodi `add()`, `update()`, `discard()` e `remove()` per aggungere e rimuovere elementi.

Il metodo `add()` aggiunge **un solo elemento** mentre `update()` ci permette di aggiungere **più elementi** al set:

In [36]:
```
set_numeri = {7, 11, 9}
print(set_numeri)
set_numeri.add(17)
print(set_numeri)
```

```
{9, 11, 7}
{9, 11, 17, 7}
```

In [37]:
```
set_numeri.update([90, 99, 99])
print(set_numeri)
```

```
{99, 7, 9, 11, 17, 90}
```

Il metodo `discard()` e il metodo `remove()` permettono di rimuovere un elemento dal set:

Il metodo `discard()` e il metodo `remove()` permettono di rimuovere un elemento dal set:

In [39]:
```
set_numeri.discard(90)
print(set_numeri)
```

```
{99, 7, 9, 11, 17}
```

Il metodo `discard()` e il metodo `remove()` permettono di rimuovere un elemento dal set:

In [39]:
```
set_numeri.discard(90)
print(set_numeri)
```

```
{99, 7, 9, 11, 17}
```

In [40]:
```
set_numeri.remove(99)
print(set_numeri)
```

```
{7, 9, 11, 17}
```

La differenza sta nel comportamento delle due funzioni se l'elemento non esiste:

La differenza sta nel comportamento delle due funzioni se l'elemento non esiste:

In [41]:
```
set_numeri.discard(100)
print(set_numeri)
```

```
{7, 9, 11, 17}
```

La differenza sta nel comportamento delle due funzioni se l'elemento non esiste:

In [41]:
```
set_numeri.discard(100)
print(set_numeri)
```

```
{7, 9, 11, 17}
```

In [42]:
```
set_numeri.remove(100)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-42-b10e02c65f33> in <module>
----> 1 set_numeri.remove(100)

KeyError: 100
```

Esercizi

1. Create un set e utilizzate i metodi `pop()` e `clear()`

Operazioni con i set

I set si rivelano incredibilmente utili nell'effettuare operazioni matematiche su insiemi: **unione**, **intersezione**, **differenza** e **differenza simmetrica**.

Prendiamo due set (insiemi) A e B:

Operazioni con i set

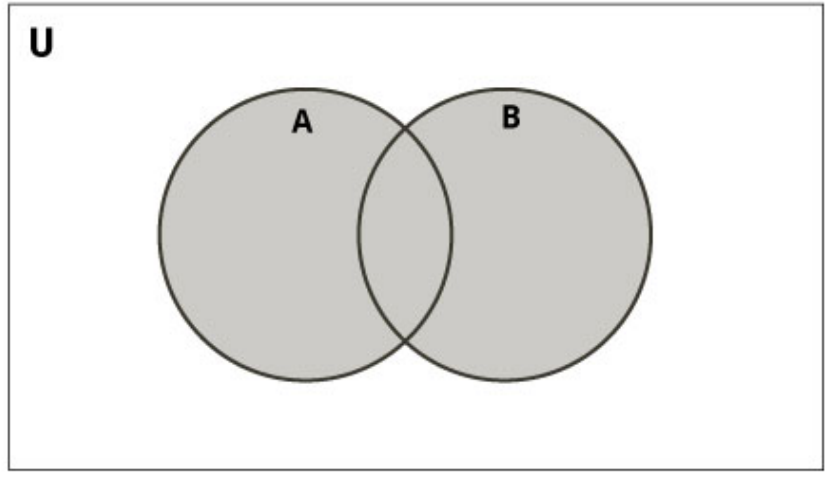
I set si rivelano incredibilmente utili nell'effettuare operazioni matematiche su insiemi: **unione**, **intersezione**, **differenza** e **differenza simmetrica**.

Prendiamo due set (insiemi) A e B:

In [44]:
```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

Unione

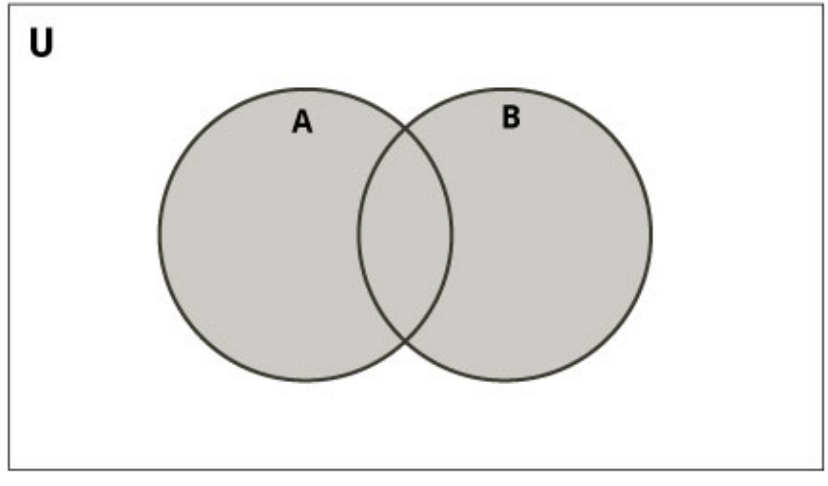
L'unione dei due set A e B è un set che contiene tutti gli elementi di A e tutti gli elementi di B:



L'unione tra due set si fa tramite l'operatore `|`:

Unione

L'unione dei due set A e B è un set che contiene tutti gli elementi di A e tutti gli elementi di B:



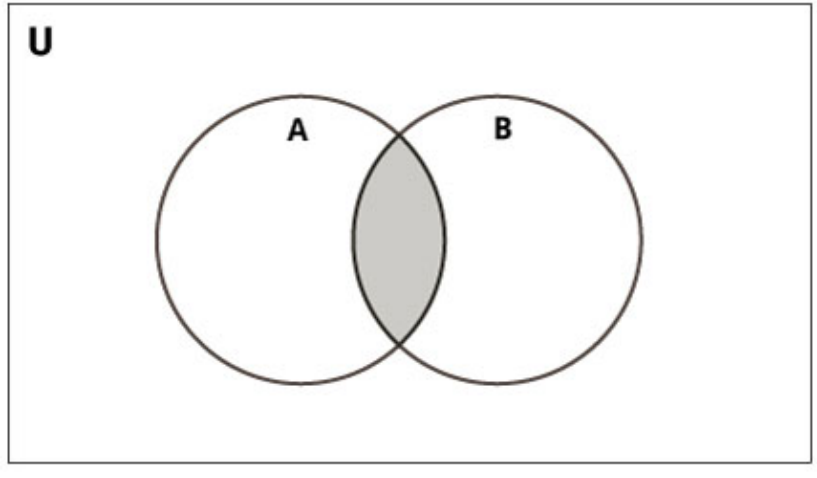
L'unione tra due set si fa tramite l'operatore `|`:

In [45]:
```
A | B
```

Out[45]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Intersezione

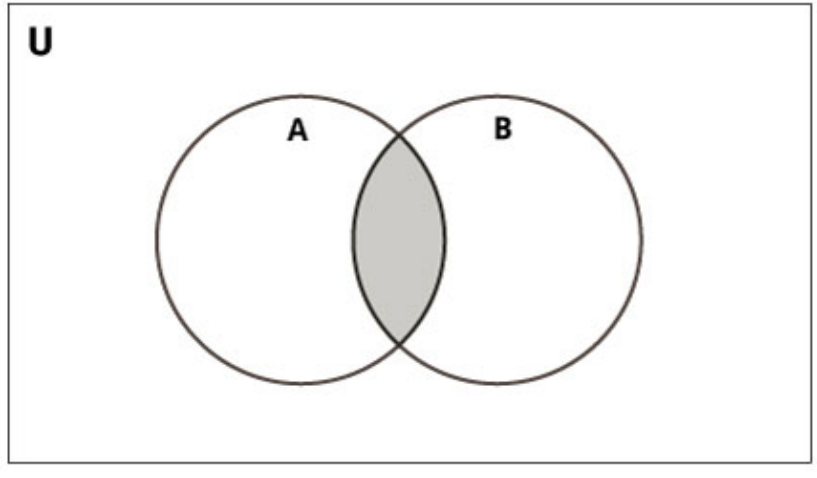
L'intersezione dei due set A e B è un set che contiene tutti gli elementi comuni ad A e B:



L'intersezione tra set si fa con l'operatore `&`:

Intersezione

L'intersezione dei due set A e B è un set che contiene tutti gli elementi comuni ad A e B:



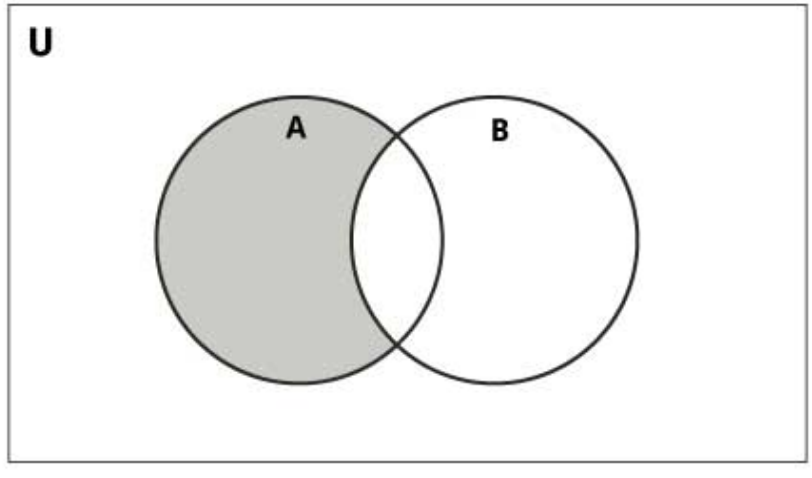
L'intersezione tra set si fa con l'operatore `&`:

In [47]:
```
A & B
```

Out[47]:
```
{4, 5}
```

Differenza

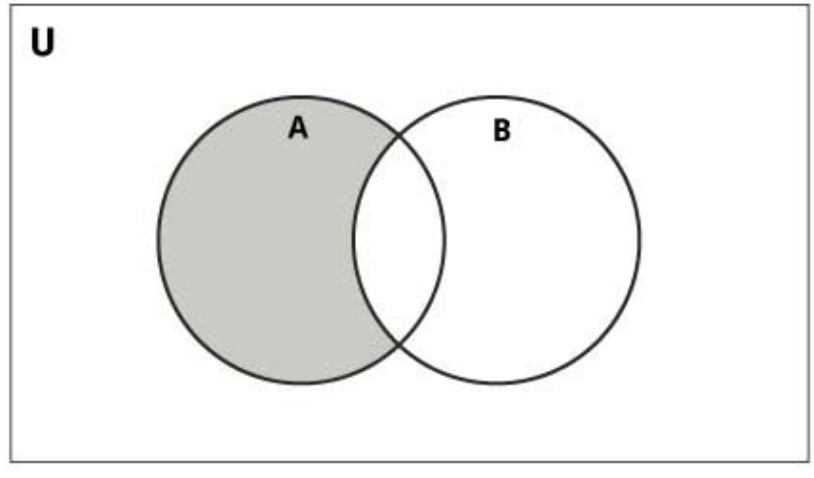
La differenza tra due set A e B è un set che contiene elementi di A ma non di B:



La differenza tra set si fa con l'operatore `-`:

Differenza

La differenza tra due set A e B è un set che contiene elementi di A ma non di B:



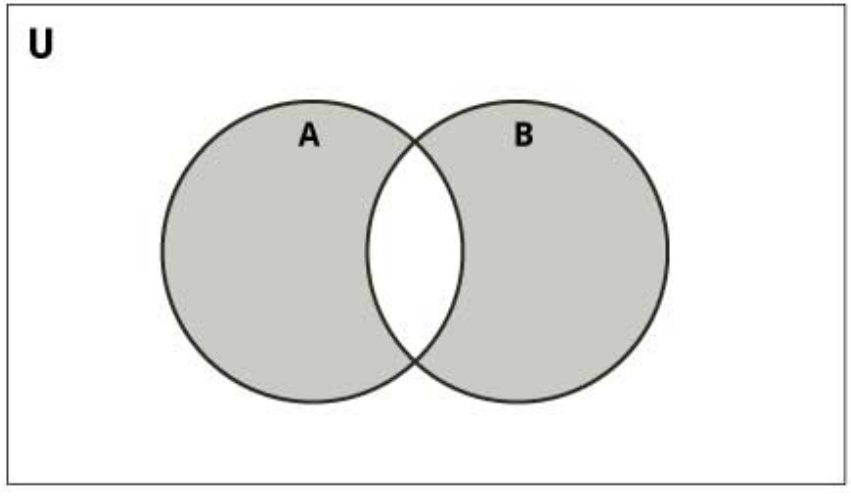
La differenza tra set si fa con l'operatore `-`:

In [48]:
```
A - B
```

Out[48]:
```
{1, 2, 3}
```

Differenza simmetrica

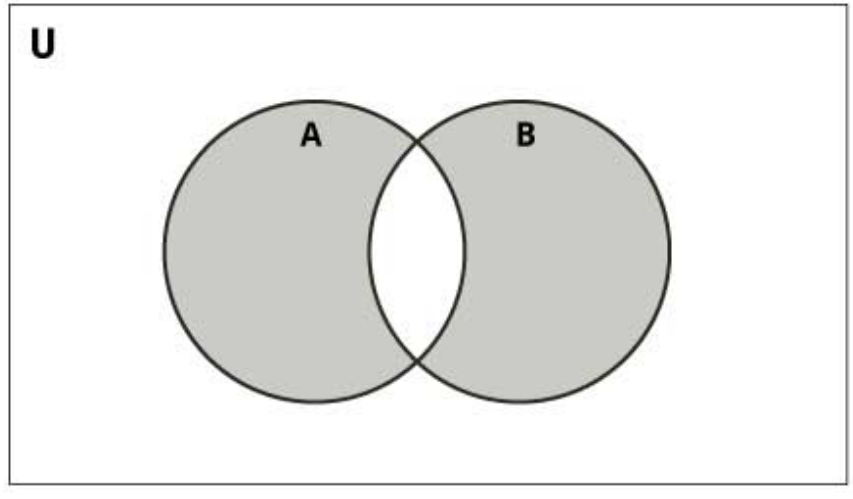
La differenza simmetrica fra sue set A e B è un set che contiene elementi di A e di B ma esclude l'intersezione:



La differenza simmetrica tra set si fa con l'operatore `^`:

Differenza simmetrica

La differenza simmetrica fra sue set A e B è un set che contiene elementi di A e di B ma esclude l'intersezione:



La differenza simmetrica tra set si fa con l'operatore `^`:

In [49]:
```
A ^ B
```

Out[49]:
```
{1, 2, 3, 6, 7, 8}
```

Esercizi

1. Dati i due set:

   ```
   A = {1,2,3,4,5}
   B = {4,5,6,7,8}
   ```

   svolgete le quattro operazioni tra set utilizzando i metodi `union()`, `intersection()`, `difference()` e `symmetric_difference()`

Dizionari

Finora, escludendo i set, abbiamo sempre parlato di **sequenze**: tipi composti che permettono di accedere ai singoli elementi che li compongono tramite indici interi.

I **dizionari** rappresentano un **tipo composto** (*compund data type*) diverso. Il tipo `dict` rappresenta quello che viene definito *built-in mapping type* di Python.

Un dizionario **mappa** una serie di **chiavi** (*keys*) ai relativi **valori** (*values*). La **chiave** può essere di qualunque **tipo non modificabile**, solitamente si usa il tipo `str`, mentre il **valore** può assumere **qualunque tipo**.

Un **oggetto** di tipo `dict` si definisce racchiudendo gli elementi che lo compongono tra parentesi graffe `{ }`, separati da `,`. Ogni elemento che compone il dizionario è una **coppia chiave-valore**, definita dalla sintassi `chiave:valore`:

Dizionari

Finora, escludendo i set, abbiamo sempre parlato di **sequenze**: tipi composti che permettono di accedere ai singoli elementi che li compongono tramite indici interi.

I **dizionari** rappresentano un **tipo composto** (*compund data type*) diverso. Il tipo `dict` rappresenta quello che viene definito *built-in mapping type* di Python.

Un dizionario **mappa** una serie di **chiavi** (*keys*) ai relativi **valori** (*values*). La **chiave** può essere di qualunque **tipo non modificabile**, solitamente si usa il tipo `str`, mentre il **valore** può assumere **qualunque tipo**.

Un **oggetto** di tipo `dict` si definisce racchiudendo gli elementi che lo compongono tra parentesi graffe `{ }`, separati da `,`. Ogni elemento che compone il dizionario è una **coppia chiave-valore**, definita dalla sintassi `chiave:valore`:

In [3]:
```
inventario = {"banane": 7, "lamponi": 11}
```

Di fatto un **dizionario** non è una struttura ordinata, per cui L'ordine delle coppie chiave-valore non è del tutto sotto controllo. Non è un problema, perchè accediamo ai valori salvati nel dizionario **per chiavi** e **non per indice**:

Di fatto un **dizionario** non è una struttura ordinata, per cui L'ordine delle coppie chiave-valore non è del tutto sotto controllo. Non è un problema, perchè accediamo ai valori salvati nel dizionario **per chiavi** e **non per indice**:

In [4]:
```
inventario['banane']
```

Out[4]:
```
7
```

Modificare i dizionari

I dizionari sono una *Data Structure* **modificabile**.

Per assegnare una nuova coppia chiave-valore a un dizionario, usiamo la seguente sintassi:

Modificare i dizionari

I dizionari sono una *Data Structure* **modificabile**.

Per assegnare una nuova coppia chiave-valore a un dizionario, usiamo la seguente sintassi:

In [5]:
```
inventario['mele'] = 9
print(inventario)
```

```
{'banane': 7, 'lamponi': 11, 'mele': 9}
```

Modificare i dizionari

I dizionari sono una *Data Structure* **modificabile**.

Per assegnare una nuova coppia chiave-valore a un dizionario, usiamo la seguente sintassi:

In [5]:
```
inventario['mele'] = 9
print(inventario)
```

```
{'banane': 7, 'lamponi': 11, 'mele': 9}
```

Possiamo anche (ri)assegnare un nuovo valore a una chiave già esistente:

Modificare i dizionari

I dizionari sono una *Data Structure* **modificabile**.

Per assegnare una nuova coppia chiave-valore a un dizionario, usiamo la seguente sintassi:

In [5]:
```
inventario['mele'] = 9
print(inventario)
```

```
{'banane': 7, 'lamponi': 11, 'mele': 9}
```

Possiamo anche (ri)assegnare un nuovo valore a una chiave già esistente:

In [6]:
```
inventario['banane'] = 17
inventario['lamponi'] = 'esaurito'
print(inventario)
```

```
{'banane': 17, 'lamponi': 'esaurito', 'mele': 9}
```

Esercizi

1. Create un dizionario e passatelo come argomento alla funzione `len()`
2. Create un oggetto `{}` e riempitelo con due coppie chiave-valore

Metodi e attributi

Come ormai abbiamo imparato, ogni **tipo** ha i suoi **metodi** e **attributi**. Anche il tipo `dict` ha i suoi!

Un metodo indispensabile per lavoare con i dizionari è `keys()`:

Metodi e attributi

Come ormai abbiamo imparato, ogni **tipo** ha i suoi **metodi** e **attributi**. Anche il tipo `dict` ha i suoi!

Un metodo indispensabile per lavoare con i dizionari è `keys()`:

In [7]:
```
inventario.keys()
```

Out[7]:
```
dict_keys(['banane', 'lamponi', 'mele'])
```

Metodi e attributi

Come ormai abbiamo imparato, ogni **tipo** ha i suoi **metodi** e **attributi**. Anche il tipo `dict` ha i suoi!

Un metodo indispensabile per lavoare con i dizionari è `keys()`:

In [7]:
```
inventario.keys()
```

Out[7]:
```
dict_keys(['banane', 'lamponi', 'mele'])
```

L'oggetto che otteniamo è di un tipo particolare: è un **oggetto** di tipo `view`. Non approfondiremo questo tipo, per i nostri scopi, ci basta sapere che si tratta di un oggetto **iterabile**. Possiamo costruirci sopra un loop:

Metodi e attributi

Come ormai abbiamo imparato, ogni **tipo** ha i suoi **metodi** e **attributi**. Anche il tipo `dict` ha i suoi!

Un metodo indispensabile per lavoare con i dizionari è `keys()`:

In [7]:
```
inventario.keys()
```

Out[7]:
```
dict_keys(['banane', 'lamponi', 'mele'])
```

L'oggetto che otteniamo è di un tipo particolare: è un **oggetto** di tipo `view`. Non approfondiremo questo tipo, per i nostri scopi, ci basta sapere che si tratta di un oggetto **iterabile**. Possiamo costruirci sopra un loop:

In [9]:
```
for key in inventario.keys():
    testo = "Numero di " + key + " nell'inventario:"
    valore = inventario[key]
    print(testo, valore)
```

```
Numero di banane nell'inventario: 17
Numero di lamponi nell'inventario: esaurito
Numero di mele nell'inventario: 9
```

Esercizi

1. Create un dizionario e provate a utilizzare i metodi
    - `values()`
    - `items()`
    - `get()`

List comprehension, *set comprehension* e *dict comprehension*

Python fornisce una sintassi compatta per generare liste, set e dizionari a partire sequenze di valori.

Ad esempio, se volessimo costruire la lista:

```
lista_numeri = [0,1,2,3,4,5,6,7,8,9]
```

potremmo utilizzare un loop `for`:

List comprehension, *set comprehension* e *dict comprehension*

Python fornisce una sintassi compatta per generare liste, set e dizionari a partire sequenze di valori.

Ad esempio, se volessimo costruire la lista:

```
lista_numeri = [0,1,2,3,4,5,6,7,8,9]
```

potremmo utilizzare un loop `for`:

In [3]:
```
lista_numeri = []
for i in range(10):
    lista_numeri.append(i)
print(lista_numeri)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension, *set comprehension* e *dict comprehension*

Python fornisce una sintassi compatta per generare liste, set e dizionari a partire sequenze di valori.

Ad esempio, se volessimo costruire la lista:

```
lista_numeri = [0,1,2,3,4,5,6,7,8,9]
```

potremmo utilizzare un loop `for`:

In [3]:
```
lista_numeri = []
for i in range(10):
    lista_numeri.append(i)
print(lista_numeri)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Oppure utilizzare quella che si chiama *list comprehension*:

List comprehension, *set comprehension* e *dict comprehension*

Python fornisce una sintassi compatta per generare liste, set e dizionari a partire sequenze di valori.

Ad esempio, se volessimo costruire la lista:

```
lista_numeri = [0,1,2,3,4,5,6,7,8,9]
```

potremmo utilizzare un loop `for`:

In [3]:
```
lista_numeri = []
for i in range(10):
    lista_numeri.append(i)
print(lista_numeri)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Oppure utilizzare quella che si chiama *list comprehension*:

In [4]:
```
lista_nuemeri = [i for i in range(10)]
print(lista_numeri)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Questo particolare tipo di sintassi può essere utilizzato anche per costruire gli altri tipi di *collections*:

list comprehension

```
lista_nuemeri = [i for i in range(10)]
```

---

set comprehension

```
set_numeri = {i for i in range(10)}
```

---

dict comprehension

```
dict_numeri = {i: i*i for i in range(10)}
```

Questo particolare tipo di sintassi può essere utilizzato anche per costruire gli altri tipi di *collections*:

list comprehension

```python
lista_nuemeri = [i for i in range(10)]
```

---

set comprehension

```python
set_numeri = {i for i in range(10)}
```

---

dict comprehension

```python
dict_numeri = {i: i*i for i in range(10)}
```

> ***ATTENZIONE**: la tuple comprehension **NON** esiste!*
>
> *Un oggetto del tipo: `(x for x in range(10))` si chiama **generatore** e vedremo più avanti di che si tratta*

Esercizi

1. Create una lista e vuota e riempitela con un loop `for`
2. Create una lista con una *list comprehension*
3. Usando una *dict comprehension*, create un dizionario per mappare la stringa "x" al quadrato di x dove x è un numero che va da 2 a 22