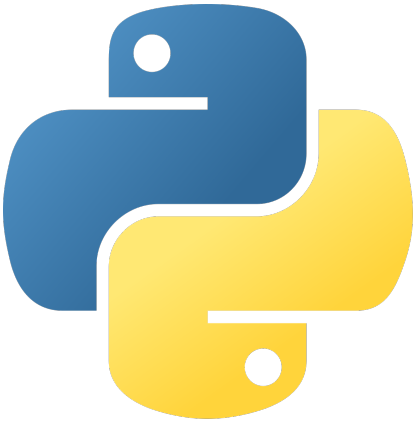
Andrea Augello andrea.augello01@unipa.it
Department of Engineering, University of Palermo, Italy

# Introduzione al Python

Università degli Studi di Palermo

# Perché il Python

# Perché il Python



- **Librerie & Frameworks**
- **Comunità fiorente**
- **Leggibilità & Apprendimento**
- **Interoperabilità**
- **Strumenti di visualizzazione**
- **Ecosistema data science**
- **Ricerca & Istruzione**
- **Modelli precostituiti**

# Introduzione

- Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti.
- Creato a inizio degli anni novanta da Guido van Rossum.
- Il nome deriva dalla passione del suo creatore per i Monty Python.
- Esistono due versioni principali del linguaggio, Python 2 e Python 3, incompatibili tra loro.

# Introduzione

- Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti.
- Creato a inizio degli anni novanta da Guido van Rossum.
- Il nome deriva dalla passione del suo creatore per i Monty Python.
- Esistono due versioni principali del linguaggio, Python 2 e Python 3, incompatibili tra loro.
- Guido van Rossum si è auto-proclamato “Benevolent Dictator for Life” del linguaggio.
- Si è dimesso da BDFL nel 2018.

# Python 2 o Python 3?

- Python 2 e 3 sono volutamente incompatibili.
- Dall'1 Gennaio 2020, Python 2 non è più supportato.
- Noi utilizzeremo Python 3
- Esistono ancora molte librerie che supportano solo Python 2, ma la maggior parte di esse sono state portate su Python 3.

# Configurazione dell’ambiente di sviluppo

# Anaconda Python

Per usare Python 3, si consiglia di installare Anaconda: *https://www.anaconda.com/distribution*. Molte librerie utili sono già incluse e l'installazione è semplice, su Windows e MacOS è disponibile un installer grafico.

Per Linux:

- Una volta scaricato lo script di installazione, è necessario rendere eseguibile lo script di installazione (il nome può ovviamente cambiare) e farlo partire:

```
chmod +x Anaconda3-2024.06-1-Linux-x86_64.sh
./Anaconda3-2019.10-Linux-x86_64.sh
```

- Eseguire l'installer con il proprio utente, senza utilizzare `sudo`.
- Al termine dell'installazione, rispondere `yes` quando viene chiesto se si vuole che Anaconda sia inizializzato automaticamente (Do you wish the installer to initialize Anaconda3 by running conda init? -> yes).
- Esiste anche una versione minimale chiamata Miniconda, per chi ha problemi di spazio.

# Installazione manuale

Python potrebbe essere già installato sul vostro sistema. Per verificarlo, aprite un terminale e digitate: `python -V` o `python3 -V`.

Se non è installato, potete scaricare Python dal sito ufficiale: *https://www.python.org/downloads/*.

Per installare le librerie necessarie, potete usare il package manager `pip` (Python Package Installer). Per installare una libreria, digitate `pip install nome_libreria`.

In questo caso, conviene creare un ambiente virtuale per ogni progetto, in modo da evitare conflitti tra le versioni delle librerie. Per farlo, digitate:

- `python -m venv nome_ambiente` per creare un ambiente virtuale.
- `source nome_ambiente/bin/activate` per attivare l'ambiente virtuale.
- `deactivate` per disattivare l'ambiente virtuale.

# Editor di testo

Si può scrivere codice Python con qualsiasi editor di testo, dai più semplici ai più sofisticati. Si consiglia di utilizzare Visual Studio Code, essendo l'editor che troverete installato sui PC sui quali svolgerete la prova pratica.
VScode rende anche più semplice la gestione di ambienti virtuali.

Qualunque editor usiate, evitate estensioni troppo "intelligenti" che forniscono suggerimenti automatici: all'esame non saranno disponibili.

# Repository con il codice del corso

Il codice del corso è disponibile su GitHub:
*https://github.com/Intelligenza-Artificiale-1/Codice-e-slide*

Clonando il repository avrete tutte le slide e il codice Python utilizzato durante le esercitazioni.

# Usare Python

# Python - Hello world

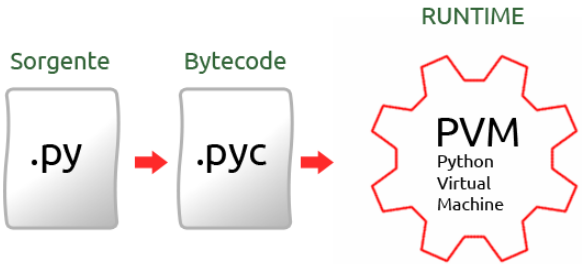
```
print("Hello world!")
```

Come si esegue un programma Python? Due possibilità:

- Scrivendo il codice in un file (`hello_world.py`) ed eseguendo il comando `python hello_world.py`
- Lanciando l'interprete, con il comando `python`, e inserendo lì il codice.

Nella maggior parte dei casi, useremo la seconda opzione.

# Cosa succede quando viene eseguito un programma Python?



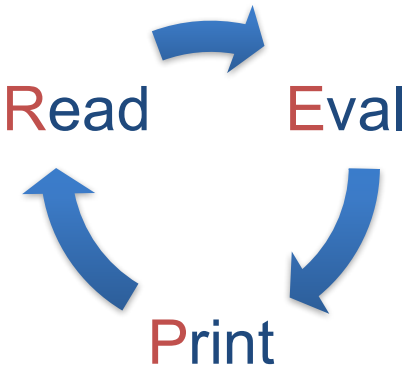

- Il codice sorgente viene convertito in Bytecode (un po' come in Java).
- Il Bytecode viene eseguito dalla Python Virtual Machine.
- Queste operazioni possono variare in base all'implementazione di Python utilizzata (Cython, Jython…).

Cython

# Interprete Python

**Read – Eval – Print Loop**

- Ambiente di programmazione interattivo.
- Esegue sempre le stesse operazioni in loop:
  - accetta input da parte dell'utente (read);
  - elabora un'espressione (eval);
  - stampa il risultato (print).



# Sintassi

# Sintassi

La peculiarità più evidente del Python è la sua sintassi, che si distingue dalla maggior parte dei linguaggi C-like per l'assenza di parentesi graffe e punti e virgola.
Gli scope sono definiti dall'indentazione, che deve essere consistente all'interno di un blocco di codice.

```python
for x in range(-3,3):
    if x > 0:
        print(x," is positive")
    elif x < 0:
        print(x," is negative")
    else:
        print(x," is zero")
```

```c
for (int x = -3; x < 3; x++) {
    if (x > 0) {
        printf("%d is positive", x);
    } else if (x < 0) {
        printf("%d is negative", x);
    } else {
        printf("%d is zero", x);
    }
}
```

# Caratteristiche di Python

- Non si usa il punto e virgola alla fine delle istruzioni.
- I blocchi di codice non sono indicati con le parentesi graffe… Tutto si basa sull'indentazione!

- Per indentare un blocco di codice si utilizzano gli spazi (normalmente 4), o il carattere di tabulazione.
- È buona norma non mischiare indentazione con spazi e indentazione con tab.

- Il codice non deve trovarsi necessariamente all'interno di una funzione.
- Possiamo scrivere un programma senza main (ma è buona norma utilizzare comunque un main).

# Perché forzare l'utilizzo dell'indentazione?

- Programmi più chiari.
- La struttura del programma coincide sempre con quella dell'indentazione.
- Lo stile di indentazione è necessariamente uniforme.
- Modificando l'indentazione cambia il comportamento dei programmi:

```
# Non stampa niente
x = 2
if x > 5:
    print('Hello')
    print('world')
```

```
# Stampa world
x = 2
if x > 5:
    print('Hello')
print('world')
```

# Variabili e tipi di dato

# Variabili

- Esempi di variabili:

```
x = 5
stringa = "Hello"
lista = [1, 2, 3, 4, 5]
```

- Non è necessario definire le variabili prima di utilizzarle.
- Non è necessario specificare il tipo delle variabili.
- I nomi delle variabili seguono regole standard:
  - possono contenere una combinazione di lettere minuscole, maiuscole, cifre o underscore;
  - non possono iniziare con una cifra;

# Variabili

- Esempi di variabili:

```
x = 5
stringa = "Hello"
lista = [1, 2, 3, 4, 5]
```

- Non è necessario definire le variabili prima di utilizzarle.
- Non è necessario specificare il tipo delle variabili.
- I nomi delle variabili seguono regole standard:
  - possono contenere una combinazione di lettere minuscole, maiuscole, cifre o underscore;
  - non possono iniziare con una cifra;
- Le variabili sono puntatori a oggetti, e possono riferirsi a dati di qualsiasi tipo.
- Una variabile potrebbe puntare prima a un intero, poi a una stringa:

```
x = 26
x = "Hello"
```

# Tipi di dato

| Nome | Descrizione | Esempi |
|---|---|---|
| `int` | Numero intero di lunghezza arbitraria | `54`, `1234567890` |
| `float` | Numero a virgola mobile | `23.16`, `4.1e-3` |
| `bool` | Booleano | `True`, `False` |
| `complex` | Numero complesso | `5 + 3j` |
| `str` | Stringa | `'Abcdef'` |
| `list` | Lista (sequenza mutabile) | `[1, 2, 3, 4]` |
| `tuple` | Tupla (sequenza immutabile) | `(4, 'Prova', 9)` |
| `set` | Insieme di oggetti unici | `{4, 9, 3}` |
| `dict` | Dizionario (coppie chiave-valore) | `{'x': 2, 'y': 5}` |

# Tipi di dato

- Esistono particolari funzioni che convertono valori da un tipo ad un altro.
- Queste funzioni hanno lo stesso nome dei tipi. Esempi:

```
int(x)     # Conversione in intero
float(x)   # Conversione in float
str(x)     # Conversione in stringa
bool(x)    # Conversione in booleano
```

- Esempi di conversione:

```
x1 = '12345'
print(type(x1))     # <class 'str'>

x2 = int(x1)
print(type(x2))     # <class 'int'>
```

# Assegnamento multiplo ed espressione di assegnamento

**Assegnamento multiplo**

- È possibile effettuare più assegnamenti in contemporanea:

```
x, y, z = 4, 9, 12
```

- Gli assegnamenti avvengono in contemporanea.
- Esempio - Swap di due variabili:

```
a, b = b, a
```

# Assegnamento multiplo ed espressione di assegnamento

**Assegnamento multiplo**

- È possibile effettuare più assegnamenti in contemporanea:

```
x, y, z = 4, 9, 12
```

- Gli assegnamenti avvengono in contemporanea.
- Esempio - Swap di due variabili:

```
a, b = b, a
```

**Espressione di assegnamento**

- L'operatore tricheco := (detto walrus operator) permette di assegnare un valore a una variabile e contemporaneamente utilizzarlo in un'espressione.
- Esempio - Catturare variabili di condizione:

```
x = 5
if (y := x ) > 5:
    print(y)
```

```
x = 5
y = x
if y > 5:
    print(y)
```

# Commenti

- I commenti sono indicati dal simbolo #

```
# Ogni riga di commento deve essere preceduta
# dal carattere cancelletto

x = 3   # commento
```

- Per commentare più righe, si può utilizzare il triplo apice:

```
'''
Questo commento
occupa multiple righe
'''
```

Questo tipo di commento è spesso utilizzato per scrivere la docstring di una funzione.

# Operatori aritmetici e booleani

**Operatori aritmetici**

| Operatore | Descrizione | Esempi |
|---|---|---|
| `+` | Somma | `3 + 2` → `5` |
| `-` | Sottrazione | `7 - 4` → `3` |
| `*` | Moltiplicazione | `3 * 2` → `6` |
| `/` | Divisione[1] | `5 / 2` → `2.5` |
| `//` | Divisione intera | `5 // 2` → `2` |
| `%` | Modulo | `7 % 4` → `3` |

**Operatori booleani**

| Operatore | Descrizione | Esempi |
|---|---|---|
| `and` | And logico | `x and y` |
| `or` | Or logico | `b or c` |
| `not` | Negazione | `not z` |

[1] In Python 2 il risultato di `5/2` sarebbe `2`

# Operatori di confronto

**Operatori di confronto**

| Operatore | Descrizione | Esempi |
|---|---|---|
| `==` | Uguale | `4 == 2` → `False` |
| `!=` | Diverso | `5 != 3` → `True` |
| `<` | Minore | `2 < 6` → `True` |
| `<=` | Minore o uguale | `5 <= 1` → `False` |
| `>` | Maggiore | `7 > 4` → `True` |
| `>=` | Maggiore o uguale | `2 >= 2` → `True` |

# Stringhe

- Le stringhe possono essere racchiuse tra apici singoli o doppi.
- Per accedere a singoli caratteri della stringa si utilizzano gli indici.
- È possibile utilizzare anche indici negativi (partendo dalla fine).

```
string = 'abcdef'

print(string[0])    # 'a'
print(string[-1])   # 'f'

print(string[20])   # IndexError: string index out of range
```

- Le stringhe sono immutabili.

```
string[-3] = 'z'    # TypeError: 'str' object does not support
    item assignment
```

# Stringhe - Slicing

- È possibile selezionare una sotto-stringa utilizzando lo slicing.
- `string[start:end]` permette di ottenere una nuova stringa con tutti i caratteri compresi tra gli indici `start` (incluso) e `end` (escluso).
- È possibile omettere l'indice iniziale o finale (o entrambi).

```
string = 'abcdef'

print(string[0:2])      # 'ab'
print(string[:3])       # 'abc'
print(string[1:])       # 'bcdef'
print(string[2:-1])     # 'cde'
print(string[:])        # 'abcdef'
```

- È anche possibile indicare uno step (positivo o negativo).

```
string = 'abcdef'

print(string[1:-1:2])   # 'bd'
```

# Stringhe - Altre operazioni

**Verificare se una sotto-stringa è contenuta in un'altra stringa**

- Per verificare se una stringa è contenuta in un'altra stringa si usa `in`.

```
string = 'Hello world'

'world' in string        # True
'world' not in string    # False
```

**Concatenamento, ripetizione**

- Gli operatori aritmetici `+` e `*` assumono un significato particolare, se usati con le stringhe (o altri tipi di sequenza).
- L'operatore della somma, `+`, concatena due stringhe.
- L'operatore del prodotto, `*`, ripete una stringa un certo numero di volte.

```
s1 = 'Hello'
s2 = 'world'

print(s1 + ' ' + s2)    # 'Hello world'
print(s2 * 3)           # 'worldworldworld' (stringa 'ripetuta'
    3 volte)
```

# Stringhe - Funzioni e metodi

**Lunghezza di una stringa - Funzione `len`**

```
string = 'Hello world'
print(len(string))      # 11
print(len(''))          # 0
print(len('abc' * 3))   # 9
```

**Dividere una stringa - Metodo `split`**

```
string = 'Prova uno,due tre'
list1 = string.split(' ')        # ['Prova', 'uno,due', 'tre']
list2 = string.split(',')        # ['Prova uno', 'due tre']
```

**Altri metodi applicabili sulle stringhe**

```
string = 'Hello world'
print(string.upper())           # 'HELLO WORLD'
print('  prova '.strip())       # 'prova'
print(string.find('wor'))       # 6 (indice della prima occorrenza)
print('123'.isdigit())          # True (tutti i caratteri sono cifre)
print(string.startswith('He')) # True
```

# Formattazione di stringhe - Metodo format

- È possibile inserire dei placeholder nelle stringhe, che poi saranno sostituiti con valori presenti in variabili.

```
nome = 'Francesca'
print('Ciao {}. Tutto bene?'.format(nome))
```

- È possibile specificare la posizione o il nome dei parametri (obbligatorio in Python 2.6 o versioni precedenti!)

```
x, y = 12, 6
print('x = {0}, y = {1}'.format(x, y))

nome, cognome = 'Mario', 'Rossi'
print('Ciao {nome} {cognome}!'.format(nome=nome,
    cognome=cognome))
```

- È anche possibile specificare la larghezza del campo, l'allineamento, il numero di cifre decimali…

# Formattazione di stringhe - f-strings

- Python 3.6 introduce un nuovo metodo per formattare le stringhe: le f-strings.
- La sintassi è simile a quella del metodo `format`, ma meno prolissa.
- Le f-string si indicano con una `f` prima degli apici, e possono contenere variabili racchiuse tra parentesi graffe:

```
name, surname = 'Mario', 'Rossi'
age = 25
print(f'Ciao {name} {surname}. Hai {age} anni.')
```

- Tra le parentesi graffe si può inserire una qualsiasi espressione, che sarà valutata a runtime.
- Per scrivere il carattere `{` si usano due `{{`, per scrivere `}` si usano due `}}`.

```
x = 5
print(f'3 + 2 = {3 + 2}')                  #3 + 2 = 5
print(f'{{x}} is the value of x:{x}.')     #{x} is the value of x:5
```

# Formattazione di stringhe - f-strings

**Formattazione con le f-strings**

- Utilizzando le f-string, il codice è chiaro anche se ci sono molte variabili.

```
first_name = 'Eric'
last_name = 'Idle'
age = 74
profession = 'comedian'
affiliation = 'Monty Python'

print(f'Hello, {first_name} {last_name}. You are {age}. You are a
    {profession}. You were a member of {affiliation}.')

# 'Hello, Eric Idle. You are 74. You are a comedian. You were a
    member of Monty Python.'
```

# Stringhe multi-linea

- Le stringhe multi-linea iniziano e terminano con tre apici consecutivi (singoli o doppi).
- Possono essere formattate sia con il metodo `format` sia con le f-strings.

```
string = f'''Prova stringa multi-linea.
È possibile formattare la stringa normalmente.
Prova espressione: 5 + 3 = {5 + 3}'''

print(string)

# Prova stringa multi-linea.
# È possibile formattare la stringa normalmente.
# Prova espressione: 5 + 3 = 8
```

# Tuple

- Le tuple sono sequenze immutabili di oggetti (anche eterogenei).
- Per separare gli elementi di una tupla si usa l'operatore , (virgola).
- Per evitare ambiguità, spesso gli elementi delle tuple sono racchiusi tra parentesi.

- Come le stringhe, anche le tuple sono sequenze, e consentono di utilizzare le operazioni di indexing, slicing, concatenamento e ripetizione viste in precedenza.

```
x = 24, 'abc', 7
x[1]                    # 'abc'
x[-1]                   # 7
x * 2                   # (24, 'abc', 7, 24, 'abc', 7)
x + (1, 2, 3)           # (24, 'abc', 7, 1, 2, 3)
```

- È possibile estrarre i singoli valori di una tupla (unpacking).

```
x = 24, 'abc', 7
a, b, c = x
```

# Liste

- Le liste sono sequenze mutabili di oggetti.
- Si può creare una lista elencando elementi racchiusi tra parentesi quadre.

```
list1 = [1, 2, 3, 4, 5]
list2 = []
```

- Anche le liste sono sequenze e supportano le operazioni di indexing, slicing, concatenazione e ripetizione.

```
list1 = [1, 2, 3]
list1 + [6, 7, 8]          # [1, 2, 3, 6, 7, 8]
list1 * 2                  # [1, 2, 3, 1, 2, 3]
list1[:2]                  # [1, 2]
2 in list1                 # True
```

- Altre operazioni comuni alle sequenze sono `len`, `min`, `max`, `index`…

```
len([1, 2, 3, 4])          # 4
min([4, 2, 6, 1, 8])       # 1
max([3, 2, 5, 4])          # 5
[1, 2, 3].index(2)         # 1
```

# Liste

- Per aggiungere elementi alla fine della lista si usa `append`.

```
lettere = ['a', 'b', 'c']
lettere.append('d')
print(lettere)                  # ['a', 'b', 'c', 'd']
```

- Altre operazioni sulle liste:

```
lettere = ['a', 'b', 'c']
lettere[1] = 'z'                # ['a', 'z', 'c']
lettere.reverse()               # Inverte l'ordine degli elementi
lettere.insert(2, 'm')          # Inserisce 'm' in posizione 2

del lettere[1]                  # Rimuove l'elemento in posizione 1
lettere.remove('a')             # Cerca e rimuove l'elemento 'a'
```

- Per ordinare una lista si usa la funzione `sorted` o il metodo `sort`.

```
lista = [4, 2, 1, 5, 8]
sorted(lista)                   # Restituisce [1, 2, 4, 5, 8]
lista.sort()                    # Ordina lista in-place
```

# Dizionari

- Tipo mutabile e non ordinato formato da coppie chiave-valore.
- Inserimento, cancellazione e ricerca in tempo costante, O(1).
- Per definire un dizionario si usa la notazione `{key: value}`.

```
d = {'nome': 'Mario', 'cognome': 'Rossi'}
d['eta'] = 25
print(d['cognome'])
del d['nome']
'cognome' in d     # True
```

- È possibile ottenere la lista delle chiavi, la lista dei valori o la lista di tuple chiave-valore.

```
d = {'nome': 'Mario', 'cognome': 'Rossi', 'eta': 25}
d.keys()           # ['cognome', 'eta', 'nome']
d.values()         # ['Rossi', 25, 'Mario']
d.items()          # [('cognome', 'Rossi'), ('eta', 25),
    ('nome', 'Mario')]
```

# Set

- I set rappresentano insiemi non ordinati di oggetti unici.
- La sintassi per definire un set è `{el1, el2, el3, ...}`.
- I duplicati vengono eliminati automaticamente.

```
numeri = {21, 6, 8, 14, 6, 21}
lettere = set('helloworld')        # set(['e', 'd', 'h', 'l',
    'o', 'r', 'w'])
```

- Altre operazioni:

```
numeri = {21, 6, 8, 14}
numeri.add(12)          # Aggiunge un elemento, se non presente
numeri.remove(8)        # Rimuove un elemento
numeri.clear()          # Elimina tutti gli elementi del set
```

- Esistono anche metodi per effettuare le classiche operazioni insiemistiche: unione, intersezione, set difference…

# Istruzioni condizionali e cicli

# If-elif-else

- Il costrutto `if` viene utilizzato per eseguire un blocco di codice solo quando si verificano una o più condizioni.
- Nella forma più semplice si usa la parola chiave `if` seguita dalla condizione, dai due punti e da un blocco di codice.

```
x = input('Inserisci un numero: ')
x = int(x)
if x > 5:
    print('Il numero inserito è maggiore di 5')
```

- Aggiungendo un blocco `else` è possibile specificare cosa fare se la condizione dell'`if` è falsa.

```
x = int(input('Inserisci un numero: '))
if x >= 0:
    print('Il numero inserito è positivo')
else:
    print('Il numero inserito è negativo')
```

# If-elif-else

- È anche possibile aggiungere uno o più blocchi `elif`, con ulteriori condizioni.

```
if (x := int(input('Inserisci un numero: '))) > 0:
    print('Il numero inserito è positivo')
elif x < 0:
    print('Il numero inserito è negativo')
else:
    print('Il numero inserito è 0')
```

- Nel caso in cui più condizioni siano vere, verrà eseguito solo il blocco corrispondente alla prima condizione vera.

# While

- Esegue un blocco di istruzioni finché la condizione è vera.

```
x = 5
while x >= 0:
    print('{} - Hello world'.format(x))
    x -= 1
```

- Non esiste un equivalente del do-while.
- Esistono anche i costrutti `break` e `continue`, simili alle loro controparti C e Java.

```
x = 1
while x <= 10:
    if x % 4 == 0:
        print('{} è divisibile per 4'.format(x))
        break
    else:
        print('{} non è divisibile per 4'.format(x))
    x += 1
```

# For

- Il `for` in Python itera su tutti gli elementi di un iterabile ed esegue un blocco di codice.
- È più simile al `foreach` presente in altri linguaggi, rispetto al classico `for`. Non ci sono indici incrementati automaticamente o da gestire manualmente.
- In questo caso `x` assumerà, a turno, i valori `1`, `4`, `2`, `6` e `8`:

```
for x in [1, 4, 2, 6, 8]:
    print(x)
```

- Quando il blocco di codice è stato eseguito per tutti i valori, il ciclo termina.
- Esempio con un dizionario:

```
dizionario = {'nome': 'Mario', 'cognome': 'Rossi', 'eta': 25}

for (key, value) in dizionario.items():
    print('Key: {}, Value: {}'.format(key, value))
```

# For

- Se interessa solo leggere gli elementi di una lista, senza modificarli, i cicli `for` consentono di farlo senza preoccuparsi della sua lunghezza e senza dover indicare manualmente gli indici.

- Esempio - Sommare tutti gli elementi di una lista:

```
lista = [3, 2, 8, 5]
somma = 0

for elemento in lista:
    somma = somma + elemento

print(somma)
```

- Esempio di `if` all'interno di un `for`:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for x in numbers:
    if x % 2 == 0:
        print('{} is even'.format(x))
```

# For - Esempio

Esempio - Calcolo del massimo in una lista (`while` vs. `for`):

**Versione con while:**

```
lista = [6, 3, 8, 14, 9]
massimo = float('-inf')

num_elementi = len(lista)
indice = 0
while indice <
    num_elementi:
    if lista[indice] >
        massimo:
        massimo =
            lista[indice]
    indice += 1
print(massimo)
```

**Versione con for:**

```
lista = [6, 3, 8, 14, 9]
massimo = float('-inf')

for elemento in lista:
    if elemento > massimo:
        massimo = elemento

print(massimo)
```

# For e range

- Quando si vuole lavorare su sequenze di numeri, è possibile utilizzare `range`.
- `range(end)` restituisce una lista[2] di interi compresi tra `0` (incluso) ed `end` (escluso).
- `range(start, stop, [step])` restituisce una lista di interi compresi tra `start` (incluso) ed `end` (escluso), con passo `step` (di default `1`).

- Esempi di `range`:

```
l1 = range(5)          # [0, 1, 2, 3, 4]
l2 = range(1, 7)       # [1, 2, 3, 4, 5, 6]
l3 = range(0, 11, 2)   # [0, 2, 4, 6, 8, 10]
```

[2] In realtà, in Python 3, `range` non restituisce una lista, ma un iteratore. Per ottenere una lista vera e propria, si può utilizzare la funzione `list`. Esempio: `list(range(3))` → `[0, 1, 2]`

# For e range - Esempi

- Esempi di cicli `for` che utilizzano `range`:

```
# Stampa i numeri pari compresi tra 1 e 10
for x in range(1, 11):
    if x % 2 == 0:
        print('{} is even'.format(x))

# Stampa 5 volte la stringa 'Hello world'
for x in range(5):
    print('Hello world')

# Stampa i numeri compresi tra 0 e 100, con step 10
for x in range(0, 101, 10):
    print(x)
```

# Funzioni

# Funzioni

Per definire una funzione si usa la parola chiave `def`, seguita dal nome della funzione, dalla lista dei parametri (racchiusi tra parentesi tonde) e dai due punti.

```
def say_hello():
    print('Hello world')

def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False

def factorial(number):
    if number <= 1:
        return 1
    return number * factorial(number - 1)
```

# Funzioni - Docstring

- La prima riga dopo la definizione di una funzione può essere una docstring per documentare lo scopo della funzione.
- Le docstring sono racchiuse fra triplici apici doppi.

```
def is_even(number: int) -> bool:
    """Restituisce True se number è pari, False altrimenti"""
    if number % 2 == 0:
        return True
    else:
        return False
```

- È possibile leggere la documentazione di una funzione utilizzando `help`.

```
help(is_even)
```

```
Help on function is_even in module __main__:

is_even(number: int) -> bool
    Restituisce True se number è pari, False altrimenti
```

# Funzioni

- Non è necessario indicare il tipo dei parametri, né il tipo del valore restituito.
- È possibile restituire più di un valore in modo semplice, utilizzando le tuple.

```
def divisione(dividendo, divisore):
    quoziente = dividendo / divisore
    resto = dividendo % divisore
    return quoziente, resto

a, b = 14, 9
quoziente, resto = divisione(a, b)
```

- Il passaggio di argomenti può avvenire per posizione o per nome.

```
quoziente, resto = divisione(dividendo=a, divisore=b)
```

- Alcuni parametri possono avere valori di default.

```
def hello(name='Marco'):
    print('Hello {}'.format(name))
```

# Moduli

# Moduli

- I moduli Python (librerie in altri linguaggi) contengono costanti, funzioni e classi.
- Python include un vasto numero di moduli standard, ma è possibile definirne o scaricarne altri.
- Anaconda contiene centinaia di altri moduli, particolarmente indicati per il calcolo scientifico e statistico, la creazione di grafici, il machine learning…

# Moduli

- I moduli Python (librerie in altri linguaggi) contengono costanti, funzioni e classi.
- Python include un vasto numero di moduli standard, ma è possibile definirne o scaricarne altri.
- Anaconda contiene centinaia di altri moduli, particolarmente indicati per il calcolo scientifico e statistico, la creazione di grafici, il machine learning…

- Il modo più semplice per importare un modulo è usare la sintassi
  `import nome_modulo`
- Una volta importato un modulo, sarà possibile utilizzare le sue funzioni come `nome_modulo.nome_funzione`. La stessa sintassi vale anche per costanti e classi.

```
import math

x = math.factorial(125)
pi = math.pi
```

# Importazione di moduli

- È anche possibile importare solo alcune funzioni / classi / costanti, utilizzando la sintassi

```
from nome_modulo import funzione, classe, costante
```

- In questo caso non sarà necessario scrivere sempre il nome del modulo per utilizzare la funzione / classe / costante.

```
from math import pi, sqrt

print('Valore di pi: {}'.format(pi))

x = sqrt(25)
print('La radice quadrata di 25 è {}'.format(x))
```

- È anche possibile rinominare un modulo o una funzione / classe / costante, per evitare ambiguità o nomi troppo lunghi.

```
from math import sqrt as radice_quadrata
import sys as sistema
```

# Importazione di moduli - Riepilogo

- `import modulo`
  importa tutto il modulo, e per accedere alle funzioni si usa la sintassi `modulo.funzione`.
- `from modulo import funzione`
  importa solo `funzione`, e non è necessario utilizzare il nome del modulo per utilizzare la funzione.
- `import modulo as nuovo_nome`
  e
  `from modulo import funzione as nuovo_nome`
  consentono di definire degli alias per moduli/funzioni.
- `from modulo import *`
  importa tutto il modulo, e non è necessario utilizzare il nome del modulo per utilizzare le funzioni (sconsigliato, potrebbe creare conflitti nei nomi).

# A che serve il main?

- In Python non esiste una distinzione netta tra i moduli importati e lo script principale che viene eseguito.
- Qualunque file .py può essere sia eseguito che importato come modulo.

# A che serve il main?

- In Python non esiste una distinzione netta tra i moduli importati e lo script principale che viene eseguito.
- Qualunque file .py può essere sia eseguito che importato come modulo.
- Esiste una variabile speciale chiamata `__name__`.
  - Questa variabile assume il valore `'__main__'` se il file viene eseguito direttamente.
  - Se invece il file è stato importato, `__name__` è una stringa che rappresenta il modulo.
- È abbastanza comune, nello script principale, controllare la variabile `__name__` e richiamare una funzione `main()` se il valore è `'__main__'`.

# A che serve il main?

- Esempio:

```
def hello(name='Mario'):
    print('Hello {}'.format(name))


def main():
    name = input('Inserisci un nome: ')
    hello(name)


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

- In questo modo, se il modulo viene importato non sarà chiesto all'utente di inserire un nome.
- Viceversa, se lo script viene eseguito direttamente, la funzione `main()` sarà chiamata in automatico.
- È buona norma dichiarare sempre un `main` in questo modo.

# Varie ed eventuali

# Rendere eseguibile uno script Python

- Per eseguire uno script Python è necessario richiamare l'interprete:

```
$ python nome_script.py
```

- In alternativa, è possibile rendere uno script Python eseguibile e lanciarlo come qualsiasi altro programma:

```
$ ./nome_script.py
```

- Per fare questo è necessario compiere due operazioni:
  1. Rendere eseguibile il file (`chmod +x nome_script.py`).
  2. Aggiungere una riga in cima allo script, che indichi alla shell quale interprete dovrà essere chiamato per eseguire lo script.

     Per utilizzare la versione di Python di default:

```
#!/usr/bin/env python
```

# Espressioni lambda

Per funzioni piccole e semplici, python consente di essere pigri e di non definire una funzione con `def`.

- Le espressioni lambda sono funzioni anonime, definite con la parola chiave `lambda`.
- Sintassi: `lambda arg1, arg2, ... : espressione`
- Esempio: `f = lambda x, y: x + y`
- Le funzioni lambda possono avere un numero qualsiasi di argomenti, ma possono contenere solo un'espressione, non un blocco di istruzioni.
- Una lambda può essere utilizzata ovunque sia richiesta una funzione.
- Una lambda può essere utilizzata immediatamente, senza essere assegnata a una variabile: `(lambda x: x + 1)(2)`.

# List/set/dict comprehension

- Le comprehension consentono di creare in modo chiaro e conciso nuove liste, set e dizionari, a partire da sequenze esistenti.
- Se necessario, permettono anche di filtrare e trasformare gli elementi.

- La sintassi per utilizzarle è la seguente:
  - `[expr for elem in seq]` per le list comprehension.
  - `{expr for elem in seq}` per le set comprehension.
  - `{key_expr: value_expr for elem in seq}` per le dict comprehension.

# List/set/dict comprehension

- Le comprehension consentono di creare in modo chiaro e conciso nuove liste, set e dizionari, a partire da sequenze esistenti.
- Se necessario, permettono anche di filtrare e trasformare gli elementi.

- La sintassi per utilizzarle è la seguente:
  - `[expr for elem in seq]` per le list comprehension.
  - `{expr for elem in seq}` per le set comprehension.
  - `{key_expr: value_expr for elem in seq}` per le dict comprehension.

- In modo opzionale, è possibile anche indicare una condizione:

```
[expr for elem in seq if condition]
{expr for elem in seq if condition}
{key_expr: value_expr for elem in seq if condition}
```

# List comprehension

- Per ogni elemento della sequenza, l'espressione viene valutata e il risultato viene aggiunto alla lista, set o dizionario.
- Quando tutti gli elementi sono stati creati, una nuova lista, set o dizionario viene restituito.
- Esempi di list comprehension:

```
# quadrati dei numeri tra 1 e 10
lista1 = [x ** 2 for x in range(1, 11)]

# elementi di lista1 moltiplicati per 3
lista2 = [elem * 3 for elem in lista1]

# elementi di lista2 che sono pari
lista3 = [elem for elem in lista2 if elem % 2 == 0]
```

# List comprehension

- Le comprehension consentono anche di indicare `for` aggiuntivi.
- Il secondo `for` si comporta come se fosse annidato dentro il primo.

```
# Restituisce ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1','c2','c3']
list1 = [char + digit for char in 'abc' for digit in '123']
```

- Il codice precedente è equivalente a:

```
list1 = []
for char in 'abc':
    for digit in '123':
        list1.append(char + digit)
```

# List/set/dict comprehension

- Esempi di list comprehension:

```
# Crea una lista che contiene solo i numeri
string = 'Hello 123 world 456'
only_digits = [x for x in string if x.isdigit()]
```

- Esempi di dict comprehension:

```
# Mappa i numeri con i loro quadrati
squares = {x : x ** 2 for x in range(1, 11)}
```

- `zip(list1, list2)` restituisce una lista di tuple:

```
l1 = ['Mario', 'Paolo', 'Chiara']
l2 = ['Rossi', 'Verdi', 'Conti']
# zip(l1, l2) -> [('Mario', 'Rossi'), ('Paolo', 'Verdi'),
    ('Chiara', 'Conti')]
d = {cognome: nome for (cognome, nome) in zip(l1, l2)}
# {'Mario': 'Rossi', 'Paolo': 'Verdi', 'Chiara': 'Conti'}
```

# Generatori

Se il contenuto della lista non è necessario tutto in una volta, è possibile utilizzare un generatore.

- I generatori sono simili alle liste, ma non vengono salvati in memoria.
- Gli elementi vengono generati al volo, uno alla volta, quando necessario.
- Per creare un generatore si utilizza la stessa sintassi delle list comprehension, ma con le parentesi tonde.
- Esempio:

```
gen = (x ** 2 for x in range(1, 11))
print(next(gen))  # Stampa 1
print(next(gen))  # Stampa 4
print(next(gen))  # Stampa 9
```

- Un generatore può essere utilizzato in un ciclo `for` o convertito in una lista con `list(gen)`.

# Yield

Per operazioni meno semplici, è possibile utilizzare la parola chiave `yield` per creare un generatore a partire da una funzione.

- La funzione restituisce un valore con `yield`, e si mette in pausa.
- Quando il generatore viene chiamato di nuovo, la funzione riprende da dove si era interrotta.
- Esempio:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

L'uso di un generatore permette di iniziare ad elaborare i valori nella sequenza prima che la sequenza sia stata completamente generata.

# Yield from

Se il generatore deve restituire dei valori presi da un altro iterabile (per esempio per una chiamata ricorsiva alla funzione stessa), si può usare `yield from`.

```
def gen():
    yield from 'Hello'
    yield from range(5)
    yield from gen()

for x in gen():
    print(x, end=' ')
```

Questo codice stamperà:

```
H e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4 H e l l o
   0 1 2 3 4 H e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4 H e l l o 0 1 2
   3 4 H e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4 H
   e l l o 0 1 2 3 4 H e l l o 0 1 2 3 4

RecursionError: maximum recursion depth exceeded
```

fino ad esaurimento dello stack (o a un'interruzione manuale).

# Eccezioni

- Quando un programma Python esegue un'operazione non valida, viene generata un'eccezione.
- Le eccezioni più comuni sono `SyntaxError`, `NameError`, `ValueError`, `TypeError`...
- Esempi:

```
print 'Hello'
# SyntaxError: Missing parentheses in call to 'print'. Did you
    mean print('Hello')?

int('prova')
# ValueError: invalid literal for int() with base 10: 'prova'

x = '12'
y = x + 2
# TypeError: must be str, not int

printz('Hello world')
# NameError: name 'printz' is not defined
```

# Gestione delle eccezioni

- Per catturare le eccezioni e gestirle si usano i costrutti `try` ed `except`.

```
try:
  int('prova')
except ValueError:
  print('Il valore inserito non è un numero intero')
```

- È possibile utilizzare più blocchi `except` per gestire più tipi di eccezioni.
- Se non si indica un tipo di eccezione da gestire con `except`, verranno catturate tutte le eccezioni.
- Per lanciare un'eccezione si usa il costrutto `raise NomeEccezione`.
- Se `raise` viene usato all'interno di un `except`, viene lanciata nuovamente l'eccezione che è gestita in quel momento.
- È anche possibile ottenere informazioni sull'eccezione:

```
try:
  int('prova')
except ValueError as err:
  print(f'Messaggio di errore: {err}')
```

# Gestione delle eccezioni

- Un blocco `else` dopo l'`except` verrà eseguito solo se non si verificano eccezioni eseguendo il blocco `try`.
- Un blocco `finally` verrà eseguito in ogni caso.
- Esempio:

```
try:
  x = int(input('Inserisci un numero intero: '))
except ValueError:
  print('Il valore inserito non è un numero intero')
else:
  print('Tutto OK')
finally:
  print('Blocco di codice eseguito in ogni caso')
```

# Gestione delle eccezioni - Esempio

**Esempio - Controllare se una stringa rappresenta un numero decimale**

- Per verificare se una stringa rappresenta un numero intero, si può usare il metodo `isdigit`.
- Non esiste un metodo (o una funzione) `isfloat`: un possibile approccio consiste nel provare a convertire il valore in `float`, e verificare se viene lanciata un'eccezione di tipo `ValueError`.

```
def isfloat(x):
    try:
        float(x)
    except ValueError:
        return False

    return True


valore = input('Inserisci un numero decimale: ')
print(is_float(valore))
```

# Gestione delle eccezioni - Esempio

**Esempio - Cercare un valore in una lista**

- Il metodo `index` della classe `list` restituisce l'indice della prima occorrenza di un elemento nella lista, oppure lancia un `ValueError` se l'elemento non è presente.

```
lista = [2, 4, 3, 5, 7]

print(lista)
elem = int(input('Inserisci un valore da cercare: '))

try:
    indice = lista.index(elem)
    print(f'Indice: {indice}')
except ValueError:
    print(f'Il valore {elem} non è presente in {lista}')
```

# Programmazione orientata agli oggetti

# Programmazione orientata agli oggetti

- Python è un linguaggio di programmazione **orientato agli oggetti**.
- Significa che in Python tutto è un oggetto.
- Un oggetto è un'entità che raggruppa **dati** e **funzionalità**.
- I dati sono memorizzati in **attributi**, mentre le funzionalità sono definite da **metodi**.
- Gli oggetti sono istanze di **classi**.
- Una classe è un modello che definisce la struttura e il comportamento di un oggetto.

# Definizione di Classe

- Utilizzare la parola chiave `class` per definire una classe.
- Specificare attributi e metodi all'interno della classe.

```
class Automobile:
    # Attributi
    marca = ""
    modello = ""
    anno = 0

    # Metodo
    def accelerare(self):
        print("Sto accelerando!")
```

# Creazione di Oggetti

- Per accedere agli attributi e ai metodi dell'oggetto si utilizza la notazione punto.

```
# Creazione di un oggetto
auto = Automobile()

# Accesso agli attributi
auto.marca = "Toyota"
auto.modello = "Camry"
auto.anno = 2022

# Chiamata a un metodo
auto.accelerare()
```

# Ereditarietà

- Per creare una nuova classe basata su una classe esistente, utilizzare l'ereditarietà.
- La nuova classe eredita attributi e metodi dalla classe genitore.
- Il costruttore `__init__` per inizializzare gli oggetti.
- Con la funzione `super()` è possibile accedere ai metodi della classe genitore.

```
class AutoElettrica(Automobile):
    def __init__(self, marca, modello, anno, batteria=100):
        super().__init__()
        self.marca = marca
        self.modello = modello
        self.anno = anno
        self.batteria = batteria

    def caricare_batteria(self, percentuale):
        print("Sto caricando la batteria!")
        batteria = min(self.batteria + percentuale, 100)
```

# Polimorfismo

- Gli oggetti di classi diverse possono rispondere allo stesso metodo.
- Questo consente la scrittura di codice più flessibile.

```
def esegui_accelerazione(auto):
    auto.accelerare()

auto1 = Automobile()
auto2 = AutoElettrica()

esegui_accelerazione(auto1)
esegui_accelerazione(auto2)
```

# Incapsulamento

- Incapsulare gli attributi e i metodi all'interno di una classe per nasconderne l'implementazione.
- Utilizzare il doppio trattino basso (__) per rendere gli attributi privati alla sola classe.
- Utilizzare il singolo trattino basso (_) per rendere gli attributi protetti (accessibili solo dalla classe e dalle sottoclassi).

```
class Studente:
    def __init__(self, nome, cognome):
        self.__nome = nome
        self.__cognome = cognome

    def get_nome(self):
        return self.__nome

    def set_nome(self, nuovo_nome):
        self.__nome = nuovo_nome
```

# Proprietà

- In Python è possibile utilizzare le proprietà per definire metodi `get` e `set` come attributi con un comportamento connesso.
- Sintassi: `@property` per il metodo `get`, `@nome_attributo.setter` per il metodo `set`. In assenza del metodo `set`, l'attributo è di sola lettura.
- implementazione alternativa della classe `Studente`:

```
class Studente():
    def __init__(self, nome, cognome):
        self.__nome = nome
        self.cognome = cognome

    @property
    def nome(self):
        print("Sto accedendo al nome")
        return self.__nome

    @nome.setter
    def nome(self, nuovo_nome):
        self.__nome = nuovo_nome.upper()
```

# Getattr e setattr

Un altro modo per sostituire getter e setter tradizionali è attraverso i metodi `.__getattr__` e `.__setattr__`.

```
class Studente():
    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def __getattr__(self, property: str):
      print(f"Accesso a {property}")
      return self.__dict__[f"__{property}"]

    def __setattr__(self, property: str, value):
      print(f"Modifica di {property} inserendo {value}")
      self.__dict__[f"__{property}"] = str(value)
```

getattr e setattr vengono chiamati automaticamente quando si cerca di accedere a un attributo o di modificarlo.

# Fine

# Fine

- Numerosi altri argomenti non sono stati trattati in questa breve introduzione.
- Li andremo a vedere sul momento, quando ne avremo bisogno.
- Per maggiori approfondimenti sul python: