Introduzione al linguaggio Python

Roberto Girau

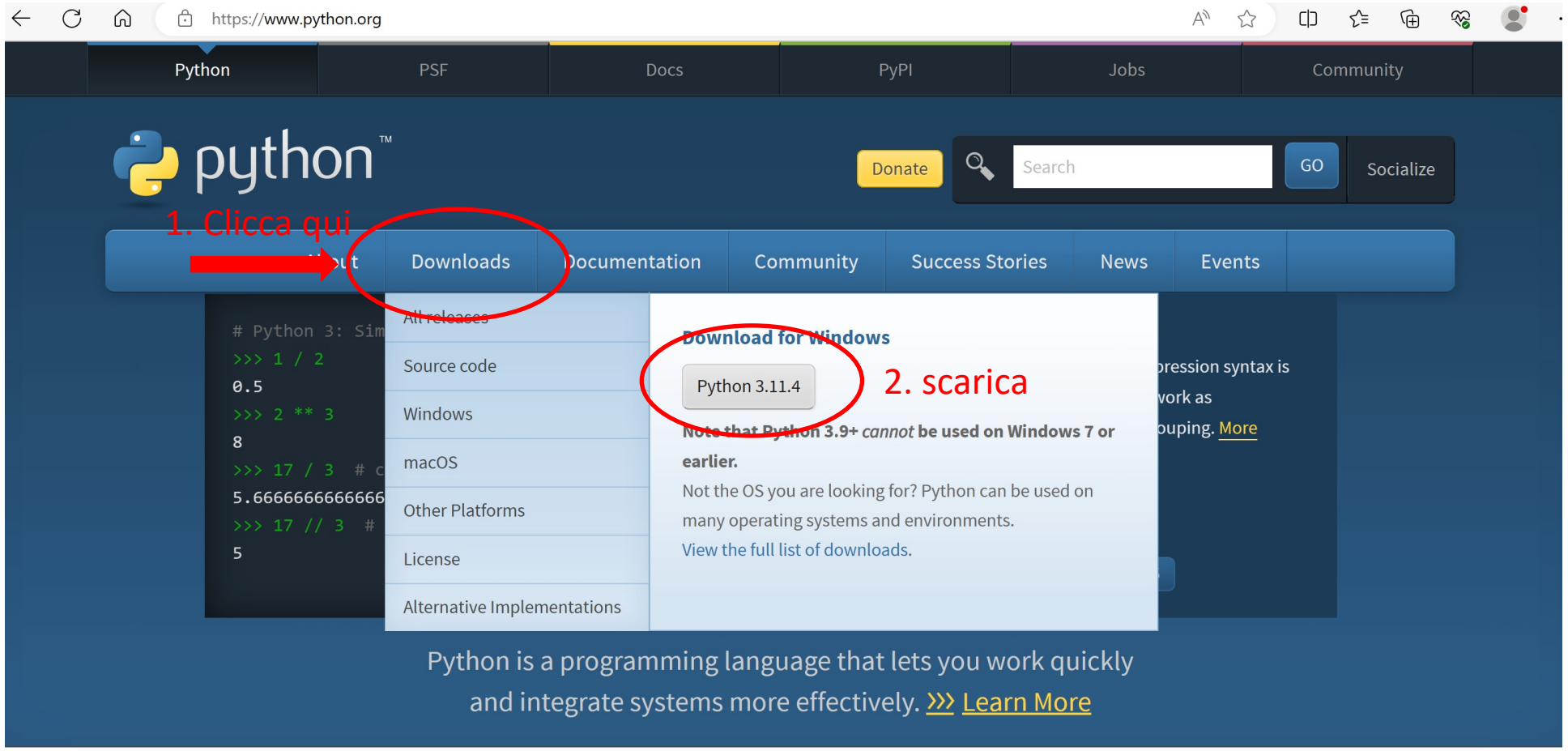
Dipartimento di Informatica – Scienze e Ingegneria

Introduzione a Python

- Python è un linguaggio di programmazione ad alto livello, dinamico, interpretato e versatile.
- Usato comunemente in molti campi, tra cui sviluppo web, data science, intelligenza artificiale, e altro.
- Installazione: sito web ufficiale di Python (https://www.python.org/).
- Introduzione a Python Shell e IDE come PyCharm o Jupyter Notebook.
- Python è un linguaggio di programmazione potente e facile da imparare.
- Dispone di strutture dati ad alto livello efficienti e un approccio semplice ma efficace alla programmazione orientata agli oggetti.
- La sintassi elegante di Python e la tipizzazione dinamica, insieme alla sua natura interpretata, lo rendono il linguaggio ideale per la scrittura di script e lo sviluppo rapido di applicazioni in molte aree su molte piattaforme.

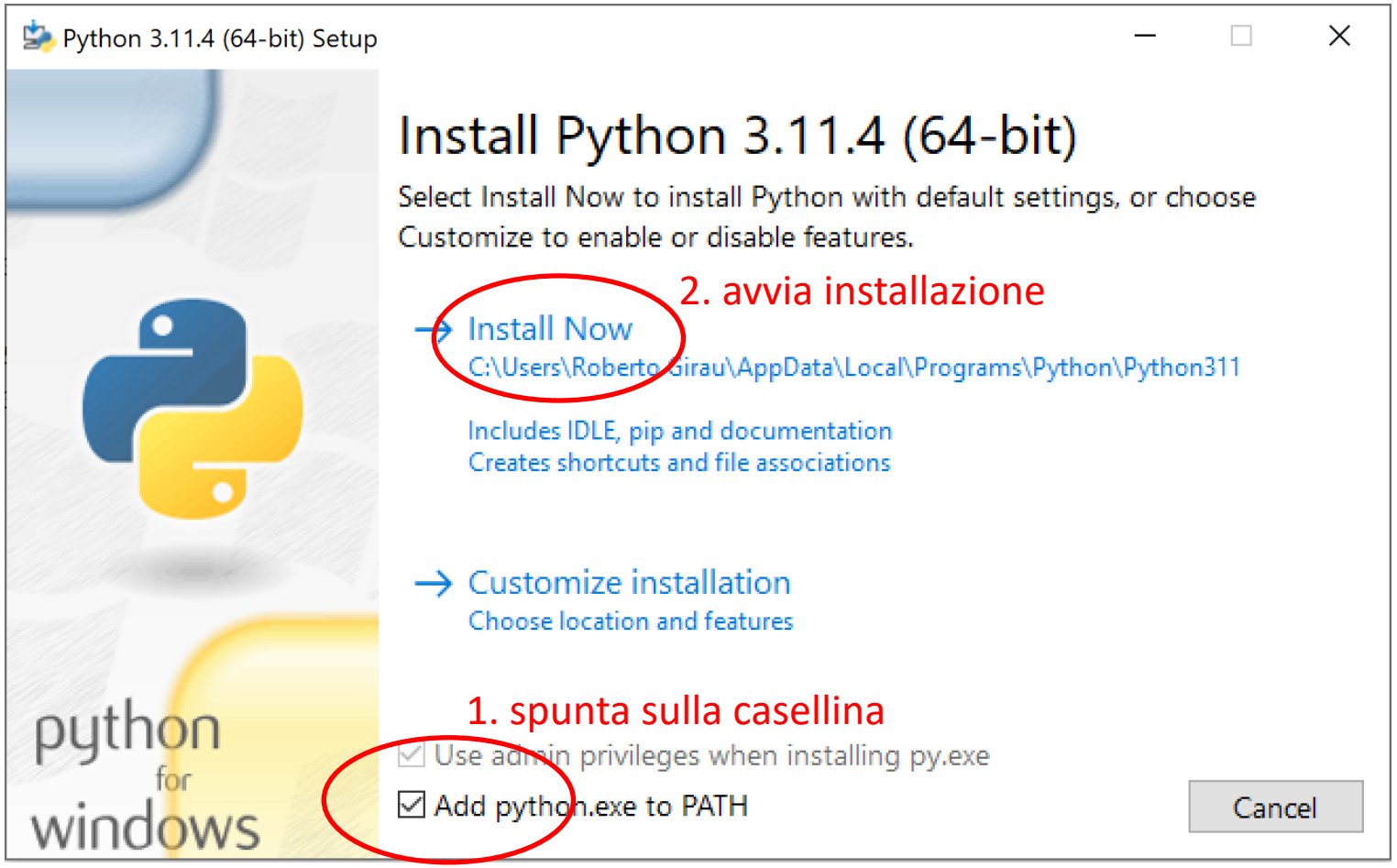
Installazione di Python

- Andare sul sito ufficiale https://www.python.org/ scaricare l'installer:



Installazione di Python

- Eseguire l'installer ricordando di selezionare la casella per inserire il PATH



Python shell

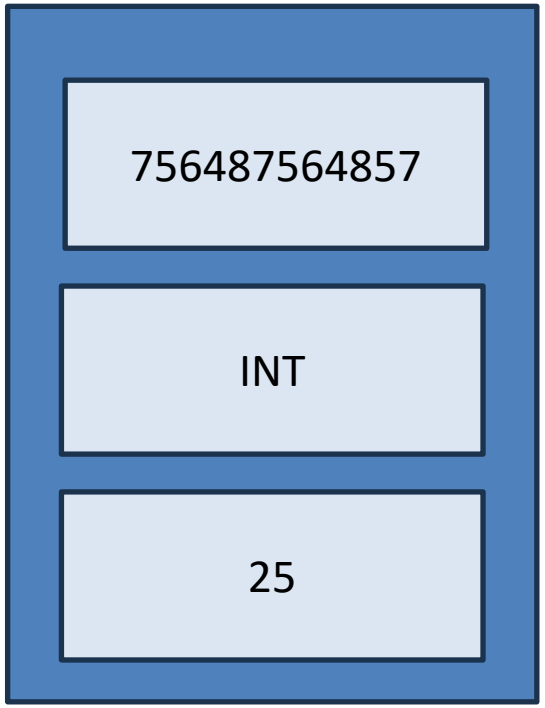
- La Python Shell è un luogo dove possiamo eseguire le nostre istruzioni Python direttamente e ottenere una risposta immediata.
- Apertura della Python Shell:
  - Puoi aprire la Python Shell semplicemente digitando python o python3 (a seconda dell'installazione) nel terminale o nel prompt dei comandi.
- Esecuzione di Istruzioni Python:
  - Puoi eseguire istruzioni Python direttamente nella shell. Dopo aver digitato l'istruzione e premuto invio, la shell eseguirà l'istruzione e visualizzerà il risultato.

Python shell

- Utilità della Python Shell:
  - La shell Python è molto utile per testare rapidamente piccoli frammenti di codice, esplorare nuove librerie o funzioni, o imparare il linguaggio.
- Uscita dalla Python Shell:
  - Puoi uscire dalla shell Python digitando exit() o premendo Ctrl+D.

Oggetti Python

- in Python tutto è rappresentato da oggetti.
- Ogni oggetto contiene delle informazioni:
    - id
    - type
    - value



Literal

- Per definire e inizializzare un dato in python usiamo la forma letterale:

```
>>> 20
20
>>> [1,2,3]
[1,2,3]
```

Le variabili

- Per poterci riferire a un oggetto all'interno di un programma noi normalmente non useremo la sua identità.
- Una variabile è un some che viene associato ad un oggetto
- Per abbinare un nome ad un oggetto si fa un assegnamento con il simbolo '='

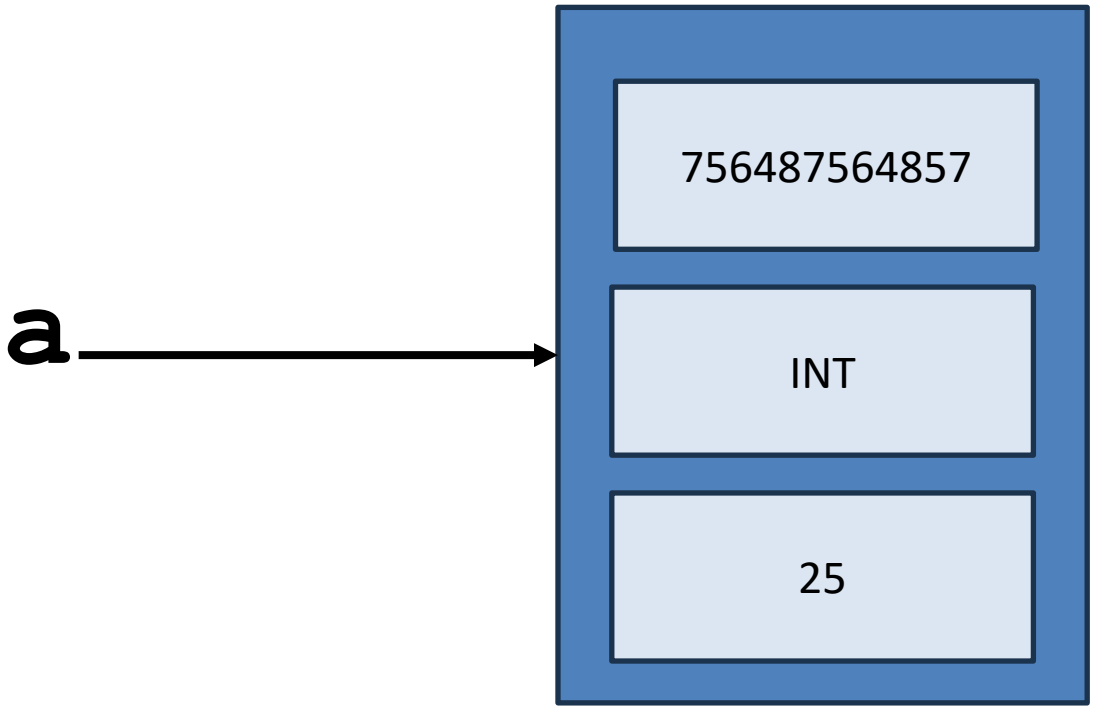
Variabili

- Nomi validi:
- Può contenere lettere, numeri, caratteri UNICODE, o underscore('_')
- Non può iniziare con un numero
- Non può essere una parola riservata.

```
a = 25
testo1 = "ciao"
anni_di_studio = 5
```

Variabili e oggetti

- Quando si fa un assegnamento si fa puntare la variabile ad un oggetto



Collable Objects

- quando parliamo di oggetti chiamabili ci riferiamo ad oggetti che contengono delle istruzioni

```
>>> print('python')
'python'
>>> x = 25
>>> print(x)
25
```

Gli attributi

- Ad un oggetto possono essere associati degli attributi
- Gli attributi definiscono definiscono delle caratteristiche dell'oggetto
- Gli attributi sono degli altri oggetti
    - possono essere dati
    - possono essere oggetti chiamabili
  - per richiamarli si usa la dot notation:
    - a.x
    - a.y()

```
>>> x = 'ciao'
>>> x.upper()
'CIAO'
>>> 'ciao'.upper()
'CIAO'
```

Tipi numerici

- Integer
- Floating-Point
- Boolean

```
0 25 -120 -1 10_000

0. 0.0 4.25 -2.50 1e3 (1000.0) 1e-3 (0.001)

True False
```

Tipi numerici

- Integers (Int)
    - Questi sono numeri interi, positivi o negativi, senza decimali.
    - Esempio:

```
my_int = 7
print(my_int)  # Output: 7
print(type(my_int))  # Output: <class 'int'>
```

Tipi numerici

- Floating Point Numbers (Float)
    - Questi sono numeri reali con una parte decimale.
    - Esempio:

```
my_float = 7.0
print(my_float)  # Output: 7.0
print(type(my_float))  # Output: <class 'float'>
```

Tipi numerici

- Booleano
    - I booleani sono un tipo di dati che possono assumere uno dei due valori: True o False.

```
is_active = True
is_inactive = False
print(type(is_active))  # Output: <class 'bool'>
```

- I booleani sono comunemente utilizzati in istruzioni condizionali e cicli.

Espressioni e operatori

- Addizione:

```
a = 5
b = 3
result = a + b
print(result)  # Output: 8
```

- Sottrazione:

```
a = 5
b = 3
result = a - b
print(result)  # Output: 2
```

Espressioni e operatori

- Moltiplicazione:

```
a = 5
b = 3
result = a * b
print(result)  # Output: 15
```

- Divisione:

```
a = 5
b = 3
result = a / b
print(result)  # Output: 1.6666666666666667
```

Espressioni e operatori

- Modulo (Resto della divisione):

```python
a = 5
b = 3
result = a % b
print(result)  # Output: 2
```

- Esponente:

```python
a = 5
b = 3
result = a ** b
print(result)  # Output: 125
```

Espressioni e operatori

- Divisione intera (Quoziente della divisione):

```
a = 5
b = 3
result = a // b
print(result)  # Output: 1
```

- Operatori di Assegnazione:
  - =: Assegna un valore alla variabile
  - +=, -=, *=, /=, %=, **=, //=: Assegna un valore dopo aver eseguito un'operazione

Operatori di confronto

- Uguale a (==):

```
a = 5
b = 5
print(a == b)  # Output: True
```

- Non uguale a (!=):

```
a = 5
b = 3
print(a != b)  # Output: True
```

Operatori di confronto

- Minore di (<) / Maggiore di (>):

```
a = 5
b = 3
print(a < b)  # Output: False
print(a > b)  # Output: True
```

- Per ‘maggiore o uguale’ e ‘minore o uguale’ aggiungiamo dopo gli operatori sopra l’operatore di assegnazione (=)

stringhe

- una stringa è una sequenza di caratteri Unicode. È uno dei tipi di dati predefiniti di Python e può essere dichiarata utilizzando singoli, doppi o tripli apici.

- esempio di come dichiarare una stringa in Python:

```
my_string = "Ciao, mondo!"
```

Stringhe

- Concatenazione di stringhe:

```python
first_name = "Alice"
last_name = "Smith"
full_name = first_name + " " + last_name
print(full_name)  # Output: Alice Smith
```

Stringhe

- Interpolazione di stringhe:

```
age = 20
print(f"My name is {first_name} and I am {age} years old.")  #
Output: My name is Alice and I am 20 years old.
```

Stringhe

- Alcuni metodi comuni delle stringhe:

```python
text = "Python Programming"

print(text.lower())  # Output: python programming
print(text.upper())  # Output: PYTHON PROGRAMMING
print(text.startswith('Python'))  # Output: True
print(text.endswith('Python'))  # Output: False
print(text.split())  # Output: ['Python', 'Programming']
```

Stringhe

- Accesso a singoli caratteri e slicing di stringhe:

```
text = "Hello, World!"
print(text[0])  # Output: H
print(text[-1])  # Output: !
print(text[7:12])  # Output: World
```

- Lunghezza di una stringa:

```
text = "Hello, World!"
print(len(text))  # Output: 13
```

Stringhe

- Escape characters (caratteri di escape):

```
print("Hello,\nWorld!")  # Output:
                          # Hello,
                          # World!
print("Hello,\tWorld!")  # Output: Hello,   World!
print("Hello,\\World!")  # Output: Hello,\World!
```

Conversioni di tipo

- La conversione di tipo, o casting, è il processo di conversione di un tipo di dato in un altro.
- Da intero a float:
    - Utilizzare la funzione float() per convertire un intero in un float.

```
x = 10
print(float(x))  # Output: 10.0
```

Conversioni di tipo

- Da float a intero:
    - Utilizzare la funzione int() per convertire un float in un intero. Questo troncherà il valore decimale.

```
y = 10.7
print(int(y))  # Output: 10
```

Conversioni di tipo

- Da stringa a intero/float:
  - Utilizzare le funzioni int() o float() per convertire una stringa numerica in un intero o un float. Se la stringa non può essere convertita in un numero, verrà generato un errore.

```
str_num = "123"
print(int(str_num))  # Output: 123
print(float(str_num))  # Output: 123.0
```

Conversioni di tipo

- Da intero/float a stringa:
  - Utilizzare la funzione str() per convertire un numero in una stringa.

```
num = 123
print(str(num))  # Output: "123"
```

Conversioni di tipo

- Da booleano a intero:
  - In Python, True è equivalente a 1 e False è equivalente a 0 quando si esegue il casting a intero.

```
print(int(True))  # Output: 1
print(int(False))  # Output: 0
```

- Da intero a booleano:
  - In Python, tutti i numeri diversi da 0 sono considerati True quando si esegue il castina a booleano, mentre 0 è considerato False.

```
print(bool(10))  # Output: True
print(bool(0))  # Output: False
```

Liste

- Una lista è una collezione ordinata e modificabile di elementi. Gli elementi possono essere di qualsiasi tipo: numeri, stringhe, altre liste, ecc.

```
my_list = [1, 2, 3, 'ciao', [5, 6, 7]]
print(my_list)  # Output: [1, 2, 3, 'ciao', [5, 6, 7]]
```

- Puoi accedere agli elementi di una lista tramite l'indicizzazione, ricordando che gli indici iniziano da 0.

```
print(my_list[0])  # Output: 1
print(my_list[3])  # Output: 'ciao'
print(my_list[-1])  # Output: [5, 6, 7] (l'indice -1 si riferisce
all'ultimo elemento)
```

Liste

- Le liste in Python sono modificabili, il che significa che puoi cambiare i loro elementi.

```
my_list[1] = 'due'
print(my_list)  # Output: [1, 'due', 3, 'ciao', [5, 6, 7]]
```

- Puoi aggiungere elementi alla fine di una lista con il metodo append().

```
my_list.append('fine')
print(my_list)  # Output: [1, 'due', 3, 'ciao', [5, 6, 7], 'fine']
```

Liste

- Puoi rimuovere un elemento da una lista con il metodo remove(), oppure rimuovere un elemento in un indice specifico con il comando del.

```
my_list.remove('ciao')
print(my_list)  # Output: [1, 'due', 3, [5, 6, 7], 'fine']

del my_list[0]
print(my_list)  # Output: ['due', 3, [5, 6, 7], 'fine']
```

- La funzione len() restituisce il numero di elementi in una lista.

```
print(len(my_list))  # Output: 4
```

Liste

- Puoi verificare se un elemento è presente in una lista con l'operatore in.

```
my_list = [1, 2, 3, 'ciao', [5, 6, 7]]
print('ciao' in my_list)  # Output: True
```

Liste

- Sulle liste è possibile fare lo slicing come con le stringhe:

```python
# Esempio di Lista
numeri = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Accedere ai primi tre elementi (indice 0, 1, 2)
primi_tre = numeri[0:3]
print(primi_tre)  # Output: [0, 1, 2]

# Accedere agli elementi dall'indice 3 all'indice 6
mezzo = numeri[3:7]
print(mezzo)  # Output: [3, 4, 5, 6]

# Accedere agli ultimi tre elementi
ultimi_tre = numeri[-3:]
print(ultimi_tre)  # Output: [7, 8, 9]

# Accedere a tutti gli elementi con passo 2 (elementi con indice
pari)
passo = numeri[::2]
print(passo)  # Output: [0, 2, 4, 6, 8]
```

Tuple

- Una tupla è una collezione ordinata e immutabile di elementi. Una volta creata una tupla, non è possibile modificarne gli elementi.
- Creazione di una Tupla:

```
my_tuple = (1, 2, 3, "ciao", (5, 6, 7))
print(my_tuple)  # Output: (1, 2, 3, 'ciao', (5, 6, 7))
```

- Si può accedere agli elementi di una tupla tramite l'indicizzazione, come per le liste.

```
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 'ciao'
print(my_tuple[-1])  # Output: (5, 6, 7)
```

Tuple

- A differenza delle liste, le tuple sono immutabili, quindi non è possibile modificare gli elementi.

```
# my_tuple[1] = 'due'  # Questo genererà un errore
```

- Utilità delle Tuple:
  - Le tuple sono utili quando hai una sequenza di dati che non dovrebbe essere modificata.
  - Le tuple sono in genere più efficienti in termini di prestazioni rispetto alle liste.

Tuple

- Operazioni sulle Tuple:
  - Puoi utilizzare l'operatore + per concatenare tuple e * per ripetere una tupla.

```
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b', 'c')
print(tuple1 + tuple2)  # Output: (1, 2, 3, 'a', 'b', 'c')

tuple3 = ('x',)
print(tuple3 * 5)  # Output: ('x', 'x', 'x', 'x', 'x')
```

- Nota: Ricorda che creare una tupla con un solo elemento richiede una virgola finale, come in tuple3 nell'esempio precedente.

Tuple

- Puoi anche utilizzare la funzione len() per ottenere la lunghezza di una tupla e l'operatore in per verificare la presenza di un elemento.

```
print(len(my_tuple))  # Output: 5
print('ciao' in my_tuple)  # Output: True
```

- Le tuple, pur essendo meno flessibili delle liste, sono un'importante struttura dati in Python per la manipolazione di collezioni immutabili di elementi.

Dizionari

- Un dizionario è una collezione non ordinata, modificabile e indicizzata di elementi. In Python, i dizionari sono scritti con parentesi graffe {} e hanno chiavi e valori.
- Creazione di un Dizionario:

```python
my_dict = {'nome': 'Mario', 'età': 30, 'professione': 'ingegnere'}
print(my_dict)  # Output: {'nome': 'Mario', 'età': 30,
'professione': 'ingegnere'}
```

```python
my_dict = dict([(1, 'uno'), (2, 'due'), (3, 'tre')])
print(my_dict)  # Output: {1: 'uno', 2: 'due', 3: 'tre'}
```

Dizionari

- Puoi accedere al valore di un elemento tramite la sua chiave.

```
print(my_dict['nome'])  # Output: 'Mario'
```

- È possibile modificare il valore di un elemento del dizionario.

```
my_dict['età'] = 31
print(my_dict)  # Output: {'nome': 'Mario', 'età': 31,
'professione': 'ingegnere'}
```

Dizionari

- Operazioni sui Dizionari:
  - Puoi aggiungere un nuovo elemento a un dizionario semplicemente assegnando un valore a una nuova chiave.

```
my_dict['hobby'] = 'ciclismo'
print(my_dict)  # Output: {'nome': 'Mario', 'età': 31,
'professione': 'ingegnere', 'hobby': 'ciclismo'}
```

  - Puoi rimuovere un elemento da un dizionario utilizzando la parola chiave del.

```
del my_dict['hobby']
print(my_dict)  # Output: {'nome': 'Mario', 'età': 31,
'professione': 'ingegnere'}
```

Dizionari

- Puoi utilizzare la funzione len() per ottenere il numero di elementi (coppie chiave-valore) in un dizionario.

```
print(len(my_dict))  # Output: 3
```

- Puoi utilizzare l'operatore in per verificare se una chiave esiste in un dizionario.

```
print('nome' in my_dict)  # Output: True
```

Set

- Un set è una collezione non ordinata e indicizzata di elementi unici. In Python, i set sono scritti con parentesi graffe {}.
- Creazione di un Set:

```python
my_set = {1, 2, 3, "ciao"}
print(my_set)  # Output: {1, 2, 3, 'ciao'}

#oppure

my_set = set([1, 2, 3, 4, 5])
print(my_set)  # Output: {1, 2, 3, 4, 5}
```

- Usando il costruttore set() bisogna passare un iterabile (lista o stringa)

Set

- A differenza dei dizionari, i set non contengono coppie chiave-valore, ma solo valori unici.
- I set non supportano l'indicizzazione o lo slicing perché sono collezioni non ordinate.

```
# print(my_set[0])  # Questo genererà un errore
```

Set

- Operazioni sui Set:
  - Puoi aggiungere un elemento a un set usando il metodo add().

```
my_set.add('nuovo')
print(my_set)  # Output: {1, 2, 3, 'ciao', 'nuovo'}
```

  - Puoi rimuovere un elemento da un set usando il metodo remove().

```
my_set.remove('nuovo')
print(my_set)  # Output: {1, 2, 3, 'ciao'}
```

Set

- Puoi utilizzare la funzione len() per ottenere il numero di elementi in un set.
- Puoi utilizzare l'operatore in per verificare se un elemento esiste in un set.
- Utilità dei Set:
  - I set sono utili quando vuoi tenere traccia di una collezione di elementi, ma ti interessa solo se un elemento è presente o non presente (non quanti volte è presente o in che ordine appaiono gli elementi).
  - I set offrono operazioni di insieme potenti, come unione(|), intersezione(&) e differenza(-).

Input e output di base

- Python fornisce funzioni built-in per l'input da tastiera e l'output su schermo. Queste funzioni sono input() e print().
- La Funzione input():
  - input() legge una linea di testo dall'input standard (in genere la tastiera) e la restituisce come stringa.

```
nome = input("Inserisci il tuo nome: ")
print(f"Ciao, {nome}!")
```

Input e output di base

- La Funzione print():
  - print() stampa i suoi argomenti sullo standard output (in genere lo schermo).

```
print("Ciao, mondo!")
```

  - Puoi stampare più argomenti separandoli con una virgola. Di default, print() separerà gli argomenti con uno spazio.

```
print("Ciao", "mondo!")  # Stampa: Ciao mondo!
```

Linee e blocchi di codice

- In Python, il codice è organizzato in linee e blocchi, che sono fondamentali per la struttura del codice e il controllo del flusso di esecuzione.
- Linee di Codice:
  - Una linea di codice in Python è una unità di testo che l'interprete Python può eseguire come un'istruzione.
  - In generale, Python considera una nuova riga come il termine di un'istruzione.

```
x = 5  # Questa è una linea di codice
print(x)  # Anche questa è una linea di codice
```

Linee e blocchi di codice

- Blocchi di Codice:
  - Un blocco di codice in Python è un gruppo di linee di codice correlate.
  - I blocchi di codice in Python sono definiti dal loro rientro. Tutte le linee di codice con lo stesso livello di rientro fanno parte dello stesso blocco di codice.
  - Ciò è noto come "indentazione significativa" e rende il codice Python molto leggibile.

```
if x > 0:  # Questo è l'inizio di un blocco di codice
    print("x è positivo")  # Questa linea fa parte dello stesso
blocco di codice
    x -= 1  # Questa linea fa parte dello stesso blocco di codice
# Questa linea non fa più parte del blocco di codice
```

Linee e blocchi di codice

- Indentazione:
- L'indentazione deve essere utilizzata in modo coerente in Python. Generalmente si usa un tab o quattro spazi per rientrare.
- Non rientrare correttamente il codice causerà un IndentationError.
- L'organizzazione del codice in linee e blocchi è fondamentale per la leggibilità e la funzionalità del codice Python.

Statement if

- Lo statement if è uno degli strumenti di controllo del flusso più fondamentali in Python. Ci permette di eseguire codice in base a una condizione.
- Sintassi di base:

```
if condition:
    # Blocco di codice da eseguire se la condition è True
```

- La condizione è un'espressione che può essere valutata come vero o falso.
- Se la condizione è vera, il blocco di codice indentato sotto l'if viene eseguito.
- Se la condizione è falsa, il blocco di codice viene saltato.

Statement if

- Esempio di Statement if:

```
x = 10
if x > 0:
    print("x è positivo")
```

- Statement if-else:
  - Possiamo aggiungere un blocco di codice else per eseguire quando la condizione if non è vera.

```
if condition:
    # Blocco di codice da eseguire se la condition è True
else:
    # Blocco di codice da eseguire se la condition è False
```

Statement if

- Esempio di Statement if-else:

```
x = -10
if x > 0:
    print("x è positivo")
else:
    print("x non è positivo")
```

Statement if

- Statement if-elif-else:
  - Possiamo usare elif (contrazione di "else if") per verificare più condizioni.

```
if condition1:
    # Blocco di codice da eseguire se la condition1 è True
elif condition2:
    # Blocco di codice da eseguire se la condition2 è True
else:
    # Blocco di codice da eseguire se nessuna delle condizioni
precedenti è True
```

Statement if

- Esempio di Statement if-elif-else:

```
x = 0
if x > 0:
    print("x è positivo")
elif x < 0:
    print("x è negativo")
else:
    print("x è zero")
```

Statement while

- Lo statement while in Python è usato per eseguire un blocco di istruzioni ripetutamente fintanto che una certa condizione è vera.
- Sintassi Base:

```
while condition:
    # Blocco di codice da eseguire finché la condition è True
```

- La condizione è un'espressione che viene valutata come vero o falso.
- Se la condizione è vera, il blocco di codice indentato sotto il while viene eseguito.
- Questo continua fino a quando la condizione diventa falsa o un'istruzione break interrompe il ciclo.

Statement while

- Esempio di Statement while:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

- Il Ciclo Infinito:
- Un ciclo while con una condizione che rimane sempre vera diventa un ciclo infinito.
- Un ciclo infinito continua per sempre a meno che non sia interrotto da un'istruzione break o da un evento esterno.

Statement while

- Statement while-else:

```
i = 0
while i < 5:
    print(i)
    i += 1
else:
    print("i è ora 5 o più")
```

Statement for

- Lo statement for in Python è usato per eseguire un blocco di codice per ogni elemento in un iterabile, come una lista, una stringa, un dizionario, un set o un generatore.
- Sintassi Base:

```
for variable in iterable:
    # Blocco di codice da eseguire per ogni elemento
```

- L'iterabile è un oggetto che può restituire i suoi elementi uno alla volta.
- La variabile assume il valore di ogni elemento a turno, e il blocco di codice viene eseguito.

Statement for

- Esempio di Statement for:

```
for i in range(5):
    print(i)
```

- Ciclo for con Tuple Unpacking:
- Se l'iterabile restituisce tuple, è possibile scomporle direttamente nel ciclo for.

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three')]
for number, name in pairs:
    print(f"{number} is called {name}")
```

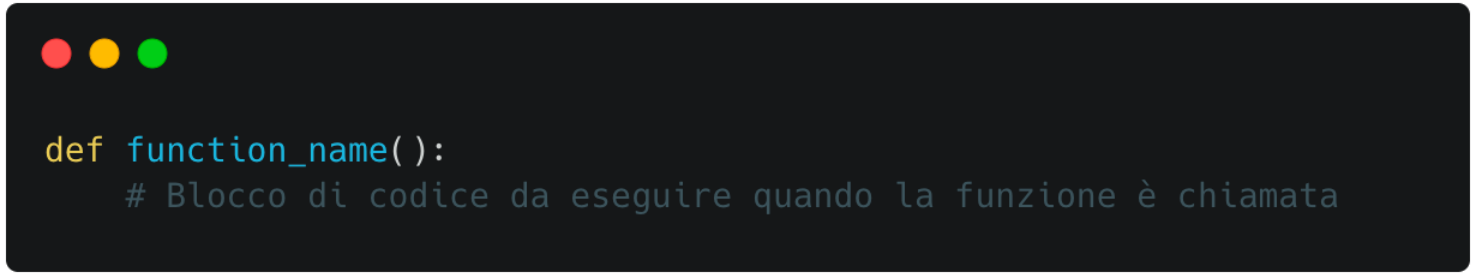
Statement for

- Statement for-else:
- Come con il while, è possibile aggiungere un blocco else al for.
- Il blocco else viene eseguito quando l'iterabile è esaurito.
- Il blocco else non viene eseguito se il ciclo è interrotto da un'istruzione break.

```
for i in range(5):
    print(i)
else:
    print("Ho finito di contare!")
```

Funzioni

- Le funzioni sono blocchi di codice riutilizzabili che eseguono una specifica operazione. In Python, le funzioni sono definite utilizzando la parola chiave def seguita dal nome della funzione e parentesi tonde ().
- Definizione di una Funzione:



- I due punti : indicano l'inizio del blocco di codice che compone la funzione.
- Questo blocco di codice è indentato per indicare che fa parte della funzione.

Parametri di una funzione

- Le funzioni possono accettare parametri, che sono valori passati alla funzione al momento della chiamata.
- I parametri sono specificati tra le parentesi tonde nella definizione della funzione.

```python
def saluta(nome):
    print(f"Ciao, {nome}!")

saluta("Mario")  # Stampa: Ciao, Mario!
```

Statement return

- Valore di Ritorno:
- Le funzioni possono restituire un valore utilizzando l'istruzione return.
- Una volta che l'istruzione return viene eseguita, la funzione termina e il valore viene restituito al chiamante.

```python
def addizione(a, b):
    return a + b

somma = addizione(3, 4)  # somma è ora 7
```

Chiamata di Funzione

- Chiamata di una Funzione:
- Una volta definita, una funzione può essere chiamata utilizzando il suo nome seguito da parentesi tonde.

```
function_name()

#oppure con valore di ritorno

a = function_name()
```

Chiamata di Funzione

- Le funzioni in Python possono avere tre tipi di argomenti: posizionali, nominali (o per parola chiave) e default.
- Argomenti Posizionali:
  - Gli argomenti posizionali devono essere passati nell'ordine in cui sono definiti nella funzione.

```python
def saluta(nome, cognome):
    print(f"Ciao, {nome} {cognome}")

saluta("Mario", "Rossi")  # Output: Ciao, Mario Rossi
```

Chiamata di Funzione

- Argomenti Nominali:
  - Gli argomenti nominali vengono passati specificando il nome del parametro e il valore. L'ordine non è importante.

```python
def saluta(nome, cognome):
    print(f"Ciao, {nome} {cognome}")

saluta(cognome="Rossi", nome="Mario")  # Output: Ciao, Mario Rossi
```

Chiamata di Funzione

- Argomenti Default:
  - Gli argomenti di default sono argomenti che hanno un valore predefinito. Se non viene passato alcun valore per un argomento di default, verrà utilizzato il valore predefinito.

```python
def saluta(nome, cognome="Rossi"):
    print(f"Ciao, {nome} {cognome}")

saluta("Mario")  # Output: Ciao, Mario Rossi
saluta("Mario", "Bianchi")  # Output: Ciao, Mario Bianchi
```

Oggetti Mutabili e Immutabili in Python e loro Utilizzo nelle Funzioni

- Definizione:
  - Mutabili: Gli oggetti di cui è possibile modificare il valore dopo che sono stati creati. Esempi includono liste, dizionari e set.
  - Immutabili: Gli oggetti di cui non è possibile modificare il valore dopo che sono stati creati. Esempi includono interi, float, stringhe e tuple.

Oggetti Mutabili e Immutabili in Python e loro Utilizzo nelle Funzioni

- Oggetti Mutabili nelle Funzioni:
  - Se passiamo un oggetto mutabile a una funzione, le modifiche all'oggetto all'interno della funzione influenzeranno anche l'oggetto al di fuori della funzione.

```python
def aggiungi_elemento(lista):
    lista.append('Nuovo elemento')

mia_lista = ['A', 'B']
aggiungi_elemento(mia_lista)
print(mia_lista)  # Risultato: ['A', 'B', 'Nuovo elemento']
```

Oggetti Mutabili e Immutabili in Python e loro Utilizzo nelle Funzioni

- Oggetti Immutabili nelle Funzioni:
- Se passiamo un oggetto immutabile a una funzione, le modifiche all'oggetto all'interno della funzione non influenzeranno l'oggetto al di fuori della funzione.

```python
def incrementa(numero):
    numero += 1

mio_numero = 1
incrementa(mio_numero)
print(mio_numero)  # Risultato: 1
```

Funzione come oggetto

- In Python, tutto è un oggetto, inclusi i numeri, le stringhe, le liste, e anche le funzioni. Questo significa che le funzioni possono essere assegnate a variabili, immesse in strutture dati e passate come argomenti ad altre funzioni.
- Funzione Assegnata a Variabile:
- Puoi assegnare una funzione a una variabile e poi usare quella variabile come fosse la funzione originale.

```python
def saluta(nome):
    print(f"Ciao, {nome}!")

mia_funzione = saluta
mia_funzione("Mario")  # Stampa: Ciao, Mario!
```

Funzione come oggetto

- Funzioni in Strutture Dati:
- Le funzioni possono essere messe in liste, dizionari, set e altre strutture dati.

```python
def addizione(a, b):
    return a + b

def sottrazione(a, b):
    return a - b

operazioni = [addizione, sottrazione]
risultato = operazioni[0](10, 5)  # risultato è ora 15
```

Funzione come oggetto

- Funzioni come Argomenti:
- Le funzioni possono essere passate come argomenti ad altre funzioni. Queste ultime funzioni sono spesso chiamate funzioni di ordine superiore.

```python
def saluta(nome):
    return f"Ciao, {nome}!"

def esegui_funzione(funzione, argomento):
    return funzione(argomento)

print(esegui_funzione(saluta, "Mario"))  # Stampa: Ciao, Mario!
```

Namespace e Scope

- Un namespace in Python è un sistema per garantire che i nomi delle variabili siano unici in modo da evitare conflitti di nomi. Differenti namespace possono coesistere senza interferire tra di loro poiché sono isolati. I namespace sono implementati come dizionari.
- Namespace Locali e Globali:
  - I namespace locali vengono creati quando una funzione è chiamata. I nomi definiti nella funzione esistono solo in questo namespace.
  - Il namespace globale è creato quando lo script inizia a essere eseguito. Contiene i nomi definiti al livello più alto del programma.

Namespace e Scope

- La Funzione built-in locals() e globals():
  - locals() restituisce il namespace locale corrente.
  - globals() restituisce il namespace globale.

```python
def mia_funzione():
    variabile_locale = "Sono locale"
    print(locals())

mia_funzione()
print(globals())
```

Global e Non Local

- Normalmente, quando assegniamo un valore a una variabile in una funzione, quella variabile è locale. Se vogliamo assegnare un valore a una variabile globale, dobbiamo dichiararlo con global.
- Analogamente, nonlocal permette di assegnare un valore a una variabile nel namespace più vicino che non sia né locale né globale.

```
variabile_globale = "Sono globale"

def mia_funzione():
    global variabile_globale
    variabile_globale = "Sono stata modificata"

mia_funzione()
print(variabile_globale)  # Stampa: Sono stata modificata
```

Gestione degli errori

- La gestione degli errori è un aspetto fondamentale della programmazione. In Python, gli errori vengono gestiti con un costrutto speciale chiamato try/except.
- Blocco Try/Except:

```
try:
    # Codice che potrebbe sollevare un errore
except TipoErrore:
    # Cosa fare se si verifica questo tipo di errore
```

- Il codice nel blocco try viene eseguito.
- Se si verifica un errore del tipo specificato, il controllo passa al blocco except e il codice in esso viene eseguito.

Gestione degli errori

- Esempio di Try/Except:

```python
try:
    print(1/0)  # Questo genera un errore di divisione per zero
except ZeroDivisionError:
    print("Non si può dividere per zero!")
```

Gestione degli errori

- Blocco Finally:
- Il blocco finally viene eseguito indipendentemente dal fatto che si verifichi un errore o meno.

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Non si può dividere per zero!")
finally:
    print("Questo viene stampato indipendentemente da ciò che
accade.")
```

Gestione degli errori

- Rilanciare Errori:
  - Puoi rilanciare errori con l'istruzione raise.

```
if x < 0:
    raise ValueError("x non può essere negativo")
```

File I/O

- Python fornisce funzioni built-in per la lettura e la scrittura di file.
- La funzione open() apre un file e restituisce un oggetto file.

```
f = open("miofile.txt", "r")  # Apre il file in modalità lettura
```

- Il secondo argomento specifica la modalità in cui il file viene aperto:
  - 'r' per la lettura (default)
  - 'w' per la scrittura (sostituisce il file se esiste già)
  - 'a' per aggiungere contenuto alla fine del file
  - 'x' per creare un nuovo file e scriverci dentro (fallisce se il file esiste già)
  - 'b' per aprire il file in modalità binaria

File I/O

- Lettura da un File:
  - Il metodo read() legge tutto il contenuto del file.

```
contenuto = f.read()
print(contenuto)
```

  - Il metodo readline() legge una singola riga del file.

```
riga = f.readline()
print(riga)
```

File I/O

- Scrittura su un File:
  - Il metodo write() scrive una stringa nel file. Ritorna il numero di caratteri scritti.

```
f = open("miofile.txt", "w")
f.write("Ciao, mondo!")
```

- Chiudere un File:
  - Dopo aver finito con un file, dovresti chiuderlo con il metodo close().

```
f.close()
```

Esempio di programma python

- Visual Studio Code (VS Code) è un editor di codice sorgente sviluppato da Microsoft che supporta vari linguaggi di programmazione, inclusa Python.
- Installare l'Estensione Python:
  - Prima di iniziare a scrivere script Python in VS Code, assicurati di aver installato l'estensione Python da Visual Studio Marketplace.
- Creare un Nuovo File Python:
  - Vai su "File" > "Nuovo File" o usa la scorciatoia Ctrl+N (o Cmd+N su macOS).
  - Salva il file con l'estensione .py.

Esempio di programma python

- Scrivere il Codice Python:
  - Ora puoi iniziare a scrivere il tuo codice Python nel file. VS Code offre funzionalità come l'evidenziazione della sintassi e il completamento del codice per facilitare la scrittura del codice.

- Eseguire il Codice Python:
  - Puoi eseguire il tuo codice Python direttamente in VS Code.
  - Clicca destro sul tuo codice e seleziona "Esegui il codice Python nel terminale".

Librerie e moduli

- In Python, le librerie e i moduli sono collezioni di codice riutilizzabili.
- Moduli:
  - Un modulo è un file contenente definizioni di funzioni e istruzioni Python. Puoi utilizzare qualsiasi file Python come modulo importandolo in un altro script.

```python
# my_module.py
def saluta(nome):
    print(f"Ciao, {nome}!")
```

```python
# main.py
import my_module
my_module.saluta("Mario")
```

Librerie e moduli

- Librerie:
  - Una libreria è una collezione di moduli. Python viene fornito con una vasta libreria standard di moduli per vari compiti, come l'accesso ai file, le operazioni di rete, l'elaborazione di testo e altro ancora.
- Importazione di Moduli e Librerie:
  - Usa l'istruzione import per includere un modulo o una libreria nel tuo script.

Librerie e moduli

- Puoi importare solo specifiche funzioni o variabili da un modulo usando la sintassi from ... import ....

```
from math import pi
print(pi)
```

- Librerie di Terze Parti:
- Esistono anche molte librerie di terze parti disponibili che offrono funzionalità avanzate, come il web scraping, l'analisi dei dati, l'apprendimento automatico e altro ancora. Queste librerie possono essere installate utilizzando strumenti come pip.

Librerie e moduli

- Dopo l'installazione, possono essere importate nei tuoi script Python come qualsiasi altra libreria.

```
pip install numpy
```

```
import numpy as np
```

Librerie e moduli

- La libreria Math:
  - math fornisce funzioni matematiche.

```python
import math

# Calcolo del raggio
raggio = 5
area = math.pi * raggio**2
print(f"L'area del cerchio di raggio {raggio} è {area}")

# Calcolo del logaritmo
numero = 100
log = math.log(numero)
print(f"Il logaritmo di {numero} è {log}")
```

Librerie e moduli

- La libreria Datetime:
  - datetime fornisce funzioni per lavorare con date e orari.

```python
import datetime

# Ottenere la data corrente
oggi = datetime.date.today()
print(f"Oggi è {oggi}")

# Ottenere l'ora corrente
ora = datetime.datetime.now().time()
print(f"L'ora attuale è {ora}")

# Creare un oggetto datetime
compleanno = datetime.datetime(2000, 1, 1)
print(f"Il compleanno è {compleanno}")
```

Virtual environment

- Un ambiente virtuale è uno strumento che permette di mantenere le dipendenze richieste da diversi progetti separate, installandole in ambienti isolati.
- Perché Utilizzare un Ambiente Virtuale?:
  - Supponiamo che tu abbia due progetti, 'ProgettoA' e 'ProgettoB', che richiedono versioni diverse della stessa libreria. Un ambiente virtuale può risolvere questo problema creando ambienti isolati per entrambi i progetti, ognuno con la propria versione della libreria.

Virtual environment

- Creazione di un Ambiente Virtuale:
  - Python3 include un modulo per la creazione di ambienti virtuali chiamato venv.

  - Questo comando crea un nuovo ambiente virtuale chiamato 'myenv' nella directory corrente.

Virtual environment

- Attivazione di un Ambiente Virtuale:
  - Dopo aver creato un ambiente virtuale, devi attivarlo.
  - Su Windows:

```
myenv\Scripts\activate
```

  - Su Unix o MacOS:

```
source myenv/bin/activate
```

Virtual environment

- Installazione dei Pacchetti:
  - Una volta attivato l'ambiente virtuale, puoi installare i pacchetti richiesti usando pip. Questi pacchetti saranno isolati nell'ambiente virtuale.
- Disattivazione dell'Ambiente Virtuale:
  - Quando hai finito di lavorare in un ambiente virtuale, puoi disattivarlo con il comando deactivate.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Roberto Girau

Dipartimento di Informatica – Scienze e Ingegneria

roberto.girau@unibo.it