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

Introduzione al Python

Università degli Studi di Palermo

Perché il Python

Il Grande Salto: Da C/Java a Python

- Filosofie di Design a Confronto:
  - C: Linguaggio compilato e a tipizzazione statica.
  - Python: Linguaggio interpretato e a tipizzazione dinamica.
- L'approccio alla tipizzazione:
  - Tipizzazione Statica (C):
    - I tipi sono dichiarati esplicitamente (`int`, `char`, `struct`).
    - Controllo dei tipi a **compile time**.
    - Maggiore affidabilità e rilevazione precoce degli errori.
  - Tipizzazione Dinamica (Python):
    - I tipi sono determinati a **runtime**.
    - Nessuna dichiarazione esplicita dei tipi.
    - Vantaggi: **produttività** e **velocità** di prototipazione.

Modelli di Esecuzione e Performance

- C:
  - Linguaggio Compilato.
  - Codice sorgente tradotto direttamente in codice macchina nativo.
  - **Performance eccezionali** per compiti intensivi in CPU.
  - Ideale per: programmazione di sistema, sistemi embedded.
- Python:
  - Linguaggio Interpretato.
  - Richiede uno step di traduzione a **runtime**.
  - Generalmente più lento per il codice "puro" Python.
- Il Modello Ibrido di Python:
  - La vera forza è agire da **orchestratore** di librerie ad alte prestazioni.
  - Offload del "lavoro pesante" a codice ottimizzato scritto in C.
  - Esempi: **NumPy**, **Pandas**.

Gestione della Memoria

- C:
  - **Controllo manuale**.
  - Strumenti: `malloc` e `free`.
  - L'ingegnere ha il controllo totale sulla memoria.
  - Rischio: **memory leaks** e altri errori.
- Python:
  - **Gestione automatica**.
  - Heap privato gestito internamente.
  - Semplifica la codifica, a discapito di un leggero overhead.
  - L'ingegnere può concentrarsi sulla logica dell'applicazione.

Configurazione dell’ambiente di sviluppo

Configurazione dell'ambiente di sviluppo

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

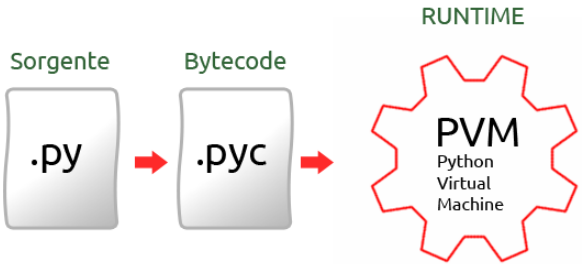
```
print("Hello world!")
```

Come si esegue del codice Python? Due possibilità:

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

Nella maggior parte dei casi, useremo la seconda opzione.

Cosa succede quando viene eseguito un programma Python?



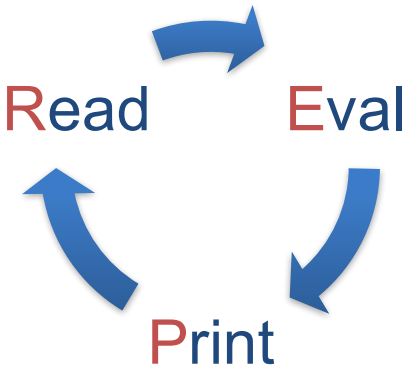
- Il codice sorgente viene convertito in Bytecode (un po' come in Java).
- Il Bytecode viene eseguito dalla Python Virtual Machine.
- Queste operazioni possono variare in base all'implementazione di Python utilizzata (Cython, Jython…).

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).



Il Modello a Oggetti di Python

- In Python, tutto è un oggetto.
  - Non è una metafora, ma una filosofia di design.
  - Dati, dagli interi alle strutture complesse, sono istanze di classi.
  - Questa astrazione fornisce un'interfaccia unificata per la manipolazione dei dati.

Memoria e "Everything is an Object"

- Il modello a oggetti è legato alla gestione della memoria di Python.
- Un **memory manager** interno gestisce un **heap privato**.
- Tutte le operazioni di storage dinamico (segmentazione, preallocazione, caching) sono gestite automaticamente.
- Il programmatore non ha controllo diretto sulla memoria.
- Questo semplifica lo sviluppo, ma nasconde i meccanismi sottostanti.

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}`

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'
```

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
```

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
```

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 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(f'{x} is even')

# 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)
```

L'Eleganza delle Comprehensions

- Le Comprehensions sono un cambiamento stilistico rispetto ai cicli `for` imperativi.
- Sono un modo conciso e potente per creare nuove liste, dizionari o set.
- Applicano un'espressione a ogni elemento di un'iterabile esistente.
- 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.

L'Eleganza delle Comprehensions

- Le Comprehensions sono un cambiamento stilistico rispetto ai cicli `for` imperativi.
- Sono un modo conciso e potente per creare nuove liste, dizionari o set.
- Applicano un'espressione a ogni elemento di un'iterabile esistente.
- 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)
```

Dictionary Comprehensions

- Sintassi: `{chiave: valore for elemento in iterabile}`.
- Utili per creare dizionari da iterabili.
- Particolarmente efficaci quando usate con la funzione `zip()` per accoppiare elementi da liste separate.

Set Comprehensions

- Sintassi: `{espressione for elemento in iterabile}`.
- Simili ai dizionari ma senza coppie chiave-valore.
- Creano una collezione di elementi unici da un iterabile in una sola riga di codice.

Lo Slicing

- Lo slicing è un'astrazione che sostituisce i cicli manuali e l'accesso basato su indici.
- Fornisce un'API intuitiva e di alto livello per manipolare le sequenze.
- La sintassi base è `[start:stop:step]`.
  - `start`: indice iniziale (inclusivo).
  - `stop`: indice finale (esclusivo).
  - `step`: intervallo tra gli elementi.

Sintassi Semplificata e Indicizzazione Negativa

- I parametri di slicing possono essere omessi per usare valori predefiniti.
- Esempio: `mia_lista[7:]` crea una slice dal settimo elemento fino alla fine della lista.
- L'**indicizzazione negativa** consente di contare dalla fine della sequenza.
  - `-1`: ultimo elemento.
  - `-2`: penultimo elemento.
  - Utile quando la lunghezza della sequenza è sconosciuta.

Potenzialità Avanzate dello Slicing

- Il parametro `step` con valore negativo inverte la direzione dello slicing.
- **Inversione di una sequenza**: la sintassi `[::-1]` inverte qualsiasi sequenza in modo semplice ed elegante.
- L'operazione è concisa e leggibile, a differenza dei cicli manuali richiesti in altri linguaggi.

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

- 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))
```

Le funzioni “pigre”

Quando scriviamo una funzione siamo abituati a pensare che, quando viene chiamata, essa esegua tutte le istruzioni al suo interno e restituisca un valore (o più valori) tramite la parola chiave `return`.

Tuttavia, Python offre un modo alternativo di definire le funzioni, consentendo di creare delle funzioni “pigre”, chiamate generatori, che producono un valore alla volta su richiesta, rimanendo in uno stato di pausa tra una produzione e l’altra e consentendo al chiamante di riprendere l’esecuzione da dove era stata interrotta dopo aver fatto qualcosa con il valore prodotto.

Le funzioni “pigre”

- I generatori sono una delle funzionalità più potenti di Python.
- Rappresentano un passaggio dalla valutazione eager (avida) alla valutazione lazy (pigra).
- Una lista è eager: tutti i suoi elementi sono calcolati e salvati in memoria.
- Un generatore è lazy: produce un valore alla volta, su richiesta, senza salvare l’intera sequenza.
- Questo li rende efficienti in termini di memoria, ideali per dati molto grandi o flussi infiniti.

Il Ruolo di `yield`

- La parola chiave `yield` definisce una funzione generatrice.
- Ha uno scopo diverso da `return`:
  - `return` termina la funzione e restituisce un valore singolo.
  - `yield` mette in pausa la funzione, restituisce un valore e salva lo stato interno.
- Quando il generatore viene chiamato di nuovo, la funzione riprende esattamente da dove si era interrotta.
- Questo meccanismo permette di produrre una sequenza di valori nel tempo senza dover ripartire da zero e senza aspettare che tutti i valori siano calcolati.
- Esempio:

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

Efficienza e Pipeline di Dati

- La differenza di memoria tra liste e generatori è notevole.
- Una lista alloca la memoria per tutti gli elementi, mentre un generatore salva solo lo stato dell'iteratore.
- È possibile concatenare più generatori per creare pipeline di dati complesse e efficienti in termini di memoria.
- Ad esempio, un generatore produce numeri, un secondo li filtra e un terzo li eleva al quadrato, senza creare liste intermedie.

Comprehension di generatori

È possibile creare dei generatori anche attraverso delle comprehension.

- 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 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).

Alcune funzioni di utilità

Alcune funzioni di utilità

- Al di là dei suoi paradigmi di base, l'ecosistema di funzioni integrate di Python è progettato per semplificare pattern comuni.
- L'obiettivo è rendere il codice più espressivo, leggibile e meno soggetto a errori.

any() e all(): Semplificare la Logica Condizionale

- Le funzioni integrate `any()` e `all()` eseguono controlli logici su un intero iterabile con una singola chiamata.
- `any()`: Restituisce `True` se almeno un elemento dell'iterabile è vero (agisce come un OR logico).
- `all()`: Restituisce `True` solo se tutti gli elementi dell'iterabile sono veri (agisce come un AND logico).
- Sono molto potenti se combinati con le espressioni generatrici, poiché implementano il short-circuiting.
- Esempio: `any(car.isdigit()for car in mia_stringa)` restituisce `True` non appena trova la prima cifra.

enumerate(): Iterare con un Contatore

- La funzione `enumerate()` è progettata per sostituire l'uso di indici manuali nei cicli `for`.
- Aggiunge un contatore a un iterabile, fornendo un modo pulito per accedere sia all'elemento che al suo indice durante l'iterazione.
- Restituisce un iteratore di tuple, dove ogni tupla contiene l'indice e l'elemento corrispondente.
- Questo rende il codice più leggibile ed evita bug comuni legati al tracciamento manuale dell'indice.

```
for index, value in enumerate(['a', 'b', 'c']):
    print(index, value)
# Output:
# 0 a
# 1 b
# 2 c
```

zip(): Iterazione Parallela e "Unzipping"

- La funzione `zip()` combina più iterabili, elemento per elemento, in un singolo iteratore di tuple.
- Utile per iterare in parallelo su più sequenze, ad esempio quando si elaborano dati correlati da liste diverse.
- Un caso d'uso comune è la creazione di un dizionario da due liste, una per le chiavi e una per i valori.

```
keys = ['nome', 'cognome', 'eta']
values = ['Mario', 'Rossi', 25]
dizionario = {k: v for k, v in zip(keys, values)}
```

- Un altro uso comune è utilizzare `zip` per iterare su coppie di elementi da due liste.

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)
```

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.

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:

Qualche esercizio per riscaldarsi

Esercizi

Si scriva del codice per verificare che una lista `l1`, contenente numeri interi, sia ordinata in modo crescente.

Esercizi

Si scriva del codice per verificare che una lista `l1`, contenente numeri interi, sia ordinata in modo crescente.

```
l1 = [1, 2, 3, 4, 5]
# Soluzione 1
def is_sorted(l):
    for i in range(len(l) - 1):
        if l[i] > l[i + 1]:
            return False
    else:
        return True

# Soluzione 2
all(l[i] <= l[i + 1] for i in range(len(l) - 1))

# Soluzione 3
not any(a > b for a, b in zip(l1, l1[1:]))
```

Esercizi

Si scriva del codice per estrarre da una matrice quadrata (lista di liste) tutti gli elementi della diagonale secondaria (dall'angolo in alto a destra a quello in basso a sinistra).

Esercizi

Si scriva del codice per estrarre da una matrice quadrata (lista di liste) tutti gli elementi della diagonale secondaria (dall'angolo in alto a destra a quello in basso a sinistra).

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

# Soluzione 1
def secondary_diagonal(m):
  diagonal = []
  for i in range(len(m)):
    diagonal.append(m[i][len(m) - 1 - i])
  return diagonal

# Soluzione 2
[matrix[i][- 1 - i] for i in range(len(matrix))]

# Soluzione 3
[row[::-1][i] for i, row in enumerate(matrix)]
```

Esercizi

Si scriva del codice per ottenere una lista senza duplicati, a partire da una lista `l1` che può contenere duplicati, mantenendo l'ordine degli elementi.

Esercizi

Si scriva del codice per ottenere una lista senza duplicati, a partire da una lista `l1` che può contenere duplicati, mantenendo l'ordine degli elementi.

```
l1 = [1, 2, 2, 3, 4, 4, 5]
# Soluzione 1
def remove_duplicates(l):
    l2 = []
    for elem in l:
        if elem not in l2:
            l2.append(elem)
    return l2

# Soluzione 2
[x for x, i in zip(l1, range(len(l1))) if x not in l1[:i]]
```

Esercizi

Si scriva del codice per calcolare la trasposta di una matrice (lista di liste).

Esercizi

Si scriva del codice per calcolare la trasposta di una matrice (lista di liste).

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

# Soluzione 1
def transpose(m):
    transposed = []
    for i in range(len(m)):
        new_row = []
        for j in range(len(m)):
            new_row.append(m[j][i])

# Soluzione 2
[[row[i] for row in matrix] for i in range(len(matrix))]
```