Lezione 10: Introduzione a Python per la Programmazione Web

Python è un linguaggio di alto livello, leggibile e con sintassi chiara, creato da Guido van Rossum negli anni '90. È molto popolare globalmente.

Caratteristiche principali:

1. **Semplicità:** Sintassi intuitiva, simile a pseudo-codice, ideale per principianti.
2. **Interpretato:** Eseguito riga per riga (tramite bytecode intermedio), facilitando sviluppo e debug.
3. **Tipizzazione Dinamica:** Tipi controllati a runtime; non serve dichiarazione esplicita e possono cambiare.
4. **Orientato agli Oggetti:** Supporto completo per classi, ereditarietà, polimorfismo. Quasi tutto è un oggetto.
5. **Polivalenza:** Usato in web backend, scripting, AI, data science, automazione, etc.
6. **Vasto Ecosistema:** Ampia disponibilità di librerie/framework (NumPy, Pandas, Django, Flask) che ne estendono le capacità.

10.1 - Installazione e Ambiente di Sviluppo (IDE)

Per usare Python, installa l'interprete dal sito ufficiale: https://www.python.org/.

- **Installazione:** Su Windows, è **consigliato** selezionare "Add Python X.X to PATH" per eseguire Python da qualsiasi terminale. Altrimenti, va configurato il PATH manualmente.

Dopo l'installazione, si può usare l'interprete interattivo (digitando `python` o `python3` nel terminale, prompt `>>>`). Per programmi complessi, è meglio un **IDE** (Ambiente di Sviluppo Integrato) con editor avanzato, debugging, etc.

IDE comuni per Python:

- **IDLE:** Base, incluso con Python.
- **Spyder:** Potente, popolare in ambito scientifico/dati (usato negli esempi).
- **Jupyter (Notebook/Lab):** Ecosistema web per calcolo interattivo, notebook (codice, testo, visualizzazioni).
- **Altri:** Visual Studio Code, PyCharm, Atom, Sublime Text.

10.2 - Esecuzione Interattiva e Debugging in Spyder

Spyder permette diverse modalità di esecuzione:

- **Esecuzione Completa:** Esegue l'intero script nella console IPython. Variabili e output aggiornati.
- **Esecuzione a Pezzi/Selezione:** Utile per eseguire solo parti di codice, magari usando variabili già definite nella console corrente.
  - **Esecuzione di Celle:** Blocchi di codice delimitati da `# %%`, eseguibili singolarmente o in sequenza, utile per analisi passo-passo.
- **Debugging:**
  - **Breakpoint:** Punti di interruzione impostabili cliccando sul margine.
  - **Modalità Debug:** Esecuzione che si ferma ai breakpoint.
  - **Ispezione e Controllo:** Permette di vedere i valori delle variabili, eseguire passo-passo (step), e continuare fino al breakpoint successivo.

10.3 - Caratteristiche Fondamentali del Linguaggio

Riepilogo e approfondimenti:

- **Interpretato:** Sviluppo rapido, senza compilazione esplicita.
- **Type Checking Dinamico:** Tipi legati ai valori, verificati a runtime. Flessibile ma richiede attenzione.

```
x = 10      # int
x = "hello" # str
# x = x + 5 # TypeError a runtime
```

- **Sintassi Snella:** Minimale, leggibile. **Indentazione** obbligatoria per definire blocchi (no `{}`).
- **Nessuna Dichiarazione Variabili:** Create per assegnamento.
- **Variabili come Riferimenti:** Puntano a oggetti in memoria (simile a Java, vedi 10.5).
- **Programmazione a Oggetti:** Tipi built-in sono classi, valori sono oggetti.

```
a = 10
print(a.__class__) # <class 'int'>
s = "ciao"
print(s.upper())   # Metodo su oggetto stringa
```

10.4 - Tipi di Dati Primitivi e Strutturati

Tipi built-in principali:

1. Tipi Numerici:

- `int` : Interi a precisione arbitraria.
- `float` : Virgola mobile doppia precisione.
- `complex` : Numeri complessi (es. `3+5j`).
- *Casting:* `int()` , `float()` , `complex()` per conversioni.
- *Operatori:* `+` , `-` , `*` , `/` (divisione float), `//` (divisione intera), `%` (modulo), `**` (potenza). Operatori incrementali (`+=` , `-=` , ...).

2. Sequenze (Ordinate, Indicizzabili):

- `str` : Stringhe (sequenze immutabili di caratteri Unicode).
  - *Creazione:* `'...'` , `"..."` , `'''...'''` (multi-linea).
  - *Formattazione:* Metodo `format()` o f-string (Python 3.6+): `eta = 30; print(f"L'età è {eta}")` .
  - *Metodi:* Molti metodi per manipolazione (`upper` , `split` , `join` , `replace` ...). Ref
  - *Immutabilità:* Operazioni creano nuove stringhe.
- `list` : Liste (sequenze mutabili, eterogenee). Simili ad array dinamici.
  - *Creazione:* `[...]` .
  - *Metodi:* Modificabili (`append` , `insert` , `remove` , `sort` ...). Ref
  - *Mutabilità:* Contenuto modificabile dopo creazione.

```
l = [1, "b", True]
l.append(3)   # l ora è [1, 'b', True, 3]
l[1] = 'beta' # l ora è [1, 'beta', True, 3]
```

- `tuple` : Tuple (sequenze immutabili, eterogenee).
  - *Creazione:* `(...)` . Singolo elemento: `(1,)` .
  - *Immutabilità:* Non modificabili. Utili per dati fissi o chiavi dict. Concatenazione crea nuova tupla. Ref
- *Operazioni Comuni (str, list, tuple):* `len()` , `seq[i]` , `seq[-i]` , slicing `seq[i:j]` , `seq[i:j:k]` , concatenazione `+` , ripetizione `*` .

3. Tipi Set (Non ordinati, Unici):

- `set` : Set mutabili. Utili per operazioni insiemistiche e rimozione duplicati. Creazione: `set([...])` o `{1, 2, 3}` (ma `{}` è dict vuoto).

- `frozenset` : Set immutabili. Utilizzabili come chiavi dict.

4. Tipi Mapping:

- `dict` : Dizionari (collezioni chiave-valore, ordinate per inserimento da Python 3.7+). Chiavi uniche e immutabili. Valori qualsiasi.
  - *Creazione:* `{chiave: valore, ...}` o `dict()` .
  - *Accesso:* `d[chiave]` .
  - *Mutabilità:* Modificabili. Molto simili a JSON.

```
d1 = {"nome": "Pippo", "eta": 25}
print(d1["nome"]) # Output: Pippo
d1["lavoro"] = "Studente" # Aggiunge/Modifica
```

Mutable vs Immutable:

- **Immutabili:** Stato non cambia dopo creazione (`int` , `float` , `complex` , `str` , `tuple` , `frozenset`). Operazioni restituiscono nuovi oggetti.
- **Mutabili:** Stato modificabile (`list` , `dict` , `set`). Modifiche "in place".

10.5 - Variabili come Riferimenti

Le variabili Python contengono riferimenti a oggetti.

- **Assegnamento (`y = x`):** Entrambe puntano allo *stesso* oggetto.
- **Modifica Oggetti Mutabili:** Modificare un oggetto mutabile (es. `lista.append()`) tramite una variabile è visibile tramite altre variabili che puntano allo stesso oggetto.

```
l1 = [1, 2]; l2 = l1
l1.append(3)
print(l2) # Output: [1, 2, 3] (l2 vede la modifica)
```

- **Operazioni che Creano Nuovi Oggetti:** Concatenazione (`+`) o riassegnamento (`x = ...`) creano nuovi oggetti. La variabile riassegnata punta al nuovo oggetto, le altre rimangono sull'originale.

```
l1 = [1, 2]; l2 = l1
l1 = l1 + [3] # l1 punta a una NUOVA lista
print(l1)     # Output: [1, 2, 3]
print(l2)     # Output: [1, 2] (l2 punta all'originale)
```

- **Oggetti Immutabili:** La distinzione è meno evidente, dato che non si può modificare l'oggetto "in place". Operazioni come `x = x + 1` creano sempre nuovi oggetti.

10.6 - Strutture di Controllo del Flusso

Usano indentazione per definire i blocchi. Istruzioni come `if`, `while`, `for` terminano con `:`.

1. Condizionale (if/elif/else)

- Esecuzione basata su condizioni. `elif` multipli opzionali, `else` opzionale.

```
a = 2
if a == 1: print("A")
elif a == 2: print("Vale") # Eseguito
else: print("Nessuno")
```

2. Ciclo `while`

- Esegue blocco finché condizione è vera (controllo pre-ciclo).

```
somma = 0; n = 1
while n <= 10:
    somma += n; n += 1
print(f"Somma: {somma}") # Output: Somma: 55
```

3. Ciclo `for`

- Itera sugli elementi di una sequenza (o iterabile). È un "for-each".

```
lista = ["a", "aa", "aaa"]; somma_len = 0
for s in lista: somma_len += len(s)
print(f"Media: {somma_len / len(lista)}") # Output: Media: 2.0
```

- **Cicli Contatore:** Si usa `range()`:
  - `range(stop)`: Da 0 a `stop-1`.
  - `range(start, stop)`: Da `start` a `stop-1`.
  - `range(start, stop, step)`: Con passo `step`.

```
for i in range(3): print(i, end=" ") # 0 1 2
```

4. Controllo Ciclo (`break` e `continue`)

- `break` : Esce dal ciclo più interno.
- `continue` : Salta al prossimo giro del ciclo.

5. Clausola `else` nei Cicli

- Eseguita se il ciclo termina normalmente (non con `break`). Utile per eseguire codice solo se non si è usciti anticipatamente.

```
numeri = [1, 5, 7]
for num in numeri:
    if num % 2 == 0: print("Pari trovato"); break
else: print("Nessun pari trovato.") # Eseguito se il break non avviene
```

10.7 - Operatori

Oltre a quelli aritmetici:

- **Confronto:** `==` (valore), `!=` , `>` , `<` , `>=` , `<=` , `is` (identità oggetto), `is not` .

```
a = [1]; b = [1]; c = a
print(a == b) # True (valore)
print(a is b) # False (oggetti diversi)
print(a is c) # True (stesso oggetto)
```

- **Logici:** `and` , `or` , `not` .

10.8 - Input Utente

Funzione `input()` per leggere da tastiera.

- **Funzionamento:** Attende input e Invio. Restituisce sempre una **stringa** (`str`).
- **Prompt:** Può ricevere una stringa opzionale da mostrare all'utente.
- **Conversione:** Necessario convertire (es. `int()` , `float()`) se serve un tipo numerico. È buona norma gestire `ValueError` con `try/except` .

```
nome = input("Come ti chiami? ")
print("Ciao, " + nome)
eta_str = input("Età: ")
try: eta_int = int(eta_str)
except ValueError: print("Inserisci un numero.")
```

10.9 - Funzioni

Raggruppano codice riutilizzabile.

- **Definizione:** `def nome_funzione(parametri):` . Corpo indentato.
- **`return` :** Restituisce un valore (o `None` se omesso).

```
def media(x, y): return (x + y) / 2
print(media(3, 6)) # Output: 4.5
```

- **Funzioni Annidate:** Definibili dentro altre funzioni (scope locale).
- **Scope Variabili:**
  - Leggono variabili esterne.
  - Assegnamento crea variabile locale (shadowing).
  - Possono modificare oggetti *mutabili* esterni (es. `lista.append()`).
  - **`global` :** Per modificare (riassegnare) variabili globali.

```
count = 0
def incrementa():
    global count
    count += 1
incrementa(); print(count) # Output: 1
```

- **Parametri:**
  - **Valori di Default:** `def func(a, b=10): ...` (parametri opzionali in coda).
  - **Argomenti Keyword:** Chiamata con `func(a=1, b=2)` , ordine non conta.
  - **`*args` :** Raccoglie argomenti posizionali extra in una tupla.
  - **`**kwargs` :** Raccoglie argomenti keyword extra in un dizionario.

```
def saluta(nome, eta=30): print(f"{nome}, {eta} anni")
saluta("Mario") # Usa default
saluta("Luisa", 25)
saluta(eta=40, nome="Carla") # Keyword args
```

10.10 - Programmazione a Oggetti (OOP)

Supporto completo.

- **Classi:** `class NomeClasse:` (Convenzione CamelCase). Corpo indentato.

- **Metodi:** Funzioni nella classe. Primo parametro è `self` (istanza corrente), passato implicitamente. Usato per accedere ad attributi/metodi (`self.attributo`).
- **Costruttore (`__init__`)**: Metodo speciale (`def __init__(self, ...):`), chiamato alla creazione dell'istanza (`NomeClasse(...)`). Inizializza attributi (`self.nome = valore`).
- **Attributi:** Variabili dell'oggetto, tipicamente definite in `__init__` .

```
class Archivio:
    def __init__(self): self.elenco = []
    def add_nome(self, nome, eta): self.elenco.append({"nome": nome, "eta": eta})
    def show(self): print(self.elenco)

nomi = Archivio() # Chiama __init__
nomi.add_nome("Pippo", 25)
nomi.show() # Output: [{'nome': 'Pippo', 'eta': 25}]
```

- **Ereditarietà:** Creare sotto-classi basate su super-classi.
  - **Sintassi:** `class SottoClasse(SuperClasse):` .
  - Eredita attributi/metodi.
  - **Overriding:** Ridefinizione di metodi ereditati.
  - **`super()` :** Per chiamare metodi della super-classe (es. `super().__init__(...)`).

```
class Figura:
    def __init__(self, id): self.id = id
    def get_area(self): return 0

class Rettangolo(Figura):
    def __init__(self, id, b, h):
        super().__init__(id)
        self.area = b * h
    def get_area(self): return self.area # Override

r = Rettangolo(1, 2, 3); print(r.get_area()) # Output: 6
```

- **Polimorfismo:** Oggetti di sotto-classi diverse possono essere trattati uniformemente (es. in un ciclo), ma invocando i loro metodi specifici (override).

10.11 - Gestione delle Eccezioni

Meccanismo `try/except` per gestire errori a runtime.

- **Sintassi:**

```
try:
    # Codice a rischio
except TipoEccezione1 as e:
    # Gestione errore specifico (e contiene l'eccezione)
except (TipoEccezione2, TipoEccezione3):
    # Gestione di più tipi
except Exception as e:
    # Gestione generica (usare con cautela)
else:
    # Eseguito se NESSUNA eccezione nel try
finally:
    # Eseguito SEMPRE (pulizia)
```

- **Esempio:** Gestire `ZeroDivisionError` o `ValueError` da `int()`.
- **Generare Eccezioni:** `raise TipoEccezione("Messaggio")`.
- **Eccezioni Custom:** Si possono definire ereditando da `Exception`.
- **`pass`:** Istruzione nulla, usata come placeholder dove la sintassi richiede un blocco.

10.12 - Gestione della Memoria e Conclusioni

- **Garbage Collector (GC):** Python gestisce la memoria automaticamente, liberando oggetti non più referenziati. Non c'è deallocazione manuale. Il modulo `gc` permette interazione avanzata, ma raramente serve.
- **Conclusioni:** Visti i fondamenti di Python (sintassi, tipi, controllo, funzioni, OOP, eccezioni). Manca il concetto di **moduli** (per organizzare codice e usare librerie), argomento della prossima lezione. Per chi conosce OOP/tipizzazione dinamica, Python è spesso veloce da apprendere.