Fondamenti di Machine Learning

Laurea Triennale in Ingegneria delle Comunicazioni

E1 - Python (introduzione)

Docente: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

- Cosa è Python.
- Comprendere come viene eseguito il codice (REPL, Script, Notebook).
- Saper organizzare un progetto minimo con moduli e import.

- Linguaggio interpretato ad alto livello, con tipizzazione dinamica.
- **Filosofia**: "Readability counts" (The Zen of Python).
- Vasto ecosistema di librerie scientifiche.

Perché serve in ML?

Velocità di prototipazione + supporto di librerie "heavy-lifting" scritte in C++/CUDA (come PyTorch o NumPy) che garantiscono performance elevate.

- **Linguaggi Compilati** (es. C, C++): Il codice viene tradotto interamente in linguaggio macchina prima dell'esecuzione.
- **Linguaggi Interpretati** (es. Python): Il codice viene letto ed eseguito riga per riga da un **interprete**.

Vantaggi di Python:

- Portabilità (gira ovunque ci sia l'interprete).
- Sviluppo rapido (no tempi di compilazione).
- Facilità di debugging.

- **REPL (Interattivo)**: Terminale per test rapidi. Input diretto, output immediato.
- **Script (.py)**: Codice salvato in file di testo ed eseguito dall'interprete. Fondamentale per pipeline e produzione.
- **Notebook (.ipynb)**: Celle interattive con codice, testo e grafici. Standard per esperimenti e analisi dati.

Il REPL permette di eseguire codice riga per riga:

```
>>> a = 10
>>> b = 20
>>> print(a + b)
30
>>> type(a)
<class 'int'>
```

- Utile per testare piccole espressioni o ispezionare oggetti.
- Si avvia semplicemente digitando `python` nel terminale.

In Python, un **modulo** è semplicemente un file con estensione `.py` che contiene codice.

- Permettono di organizzare il codice in modo logico e riutilizzabile.
- Possiamo usare funzioni definite in altri file tramite il comando `import`.
- Es: `import math` ci permette di usare `math.sqrt(4)`.

Perché serve in ML?

Tutte le principali librerie (NumPy, PyTorch, Scikit-Learn) sono organizzate in moduli e pacchetti che dobbiamo importare per usarne le funzionalità.

```python
# utils.py
def greet(name: str) -> str:
return f"Ciao, {name}!"

# main.py
from utils import greet

def main():
print(greet("Python"))

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

- **Virtual Environments** (`venv`, `conda`): Isolano le librerie di un progetto per evitare conflitti tra versioni diverse.
- **Gestione pacchetti**: Uso di `pip` / `uv` e file di configurazione (es: `requirements.txt`).

Perché serve in ML?

La riproducibilità è chiave: un esperimento deve poter girare mesi dopo con le stesse identiche versioni delle librerie.

- **Riproducibilità**: Nei notebook, le celle eseguite fuori ordine possono creare bug difficili da replicare.
- **Working directory**: Lanciare script da cartelle diverse può rompere i path di caricamento dei dati.
- **Install vs Import**: Confondere l'installazione (`pip install`) con l'uso (`import`).

Prova a replicare la struttura vista prima:

1. Crea un file `calc.py` con una funzione `add(a, b)`.
2. Crea un file `run.py`, importa `add` e stampa il risultato di un'operazione.
3. Prova a eseguire la stessa funzione definendola direttamente nel REPL (terminale Python).

- Comprendere la distinzione tra **nomi** e **valori**.
- Capire la differenza tra tipi **mutabili** e **immutabili**.
- Imparare a gestire l'**aliasing** e le copie.

In Python, le variabili sono *etichette* (nomi) applicate a oggetti (valori) in memoria.

```
a = [1, 2, 3] # 'a' punta a una lista
b = a # 'b' punta allo STESSO oggetto
b.append(4)
print(a) # Output: [1, 2, 3, 4] (Aliasing!)
```

Perché serve in ML?

Molte librerie (come NumPy o PyTorch) restituiscono ”viste” dei dati invece di copie per efficienza. Modificare una vista modifica l’originale!

- **Strong typing**: I tipi vanno dichiarati ovunque e sono *vincolanti*.
- **Weak typing** (Python): Non ci sono controlli sui tipi di dato passati alle varie funzioni, si possono creare situazioni molto particolari (es., *duck typing*).

Type Hints: Suggerimenti non vincolanti a runtime.

```
1    def process_data(value: float, name: str) -> str:
2    return f"{name}: {value}"
```

- **Immutabili**: int, float, str, tuple, bool. (Non possono cambiare ”in-place”)
- **Mutabili**: list, dict, set. (Possono cambiare stato)

```
s = "ciao"
# s[0] = "C" # Errore!

l = [1, 2]
l[0] = 10 # OK!
```

Per evitare l'aliasing, dobbiamo creare una copia esplicita:

```
a = [1, 2, 3]
b = a.copy() # Copia shallow
b.append(4)
print(a) # [1, 2, 3] - a è preservato!
```

Identity vs Equality:

- `a == b`: Controlla se i *valori* sono uguali.
- `a is b`: Controlla se sono lo *stesso oggetto* (stesso indirizzo).

- Ripassare i tipi numerici e booleani.
- Padroneggiare i costrutti `if`, `for` e `while`.
- Comprendere il concetto di *truthy*/*falsy*.

```python
a = 5 # Integer
b = 4.2 # Float
c = 'cat' # String
d = (4 == 3) # Boolean (False)

print(type(a)) # <class 'int'>
```

Perché serve in ML?

Diversi tipi di dato richiedono tipi di variabili diversi. Inoltre, la precisione numerica (float32 vs float64) è fondamentale per le performance dei modelli.

Python offre diversi modi per stampare variabili:

- **Standard**: `print("Il valore è", x)`
- **f-strings** (Python 3.6+): `print(f"Il valore è x")`

```
acc = 0.95
epoch = 10

# f-strings permettono espressioni e formattazione
print(f"Epoch {epoch}: accuracy = {acc:.2%}")
# Output: Epoch 10: accuracy = 95.00%
```

```
x = 10
if x > 5:
print("Grande")
elif x == 5:
print("Medio")
else:
print("Piccolo")
```

Truthy/Falsy: Valori ”vuoti” (`0`, `[]`, `""`, `None`) sono valutati come `False`.

```python
# range(start, stop, step)
for i in range(0, 5, 2):
print(i) # 0, 2, 4

# Iterare su elementi
names = ["ML", "AI", "DL"]
for name in names:
if name == "AI":
continue # Salta
print(name)
```

Scrivi un ciclo che stampa tutti i numeri da 1 a 20, ma:

- Se il numero è divisibile per 3, stampa ”Machine”.
- Se è divisibile per 5, stampa ”Learning”.
- Se è divisibile per entrambi, stampa ”Machine Learning”.

- **Indentazione**: Python usa gli spazi, non le graffe!
- **Off-by-one**: `range(5)` arriva a 4, non 5.
- **Modifica durante l'iterazione**: Mai rimuovere elementi da una lista mentre ci stai iterando sopra.

- Padroneggiare le liste (indexing e slicing).
- Usare list comprehensions per codice sintetico.
- Gestire dizionari per configurazioni e parametri.
- Sfruttare `enumerate` e `zip`.

```
l = [10, 20, 30, 40, 50]
print(l[0]) # 10
print(l[-1]) # 50 (ultimo)

# Slicing: [start:stop:step]
print(l[1:3]) # [20, 30]
print(l[::-1]) # [50, 40, 30, 20, 10] (reverse)
```

Perché serve in ML?

Lo slicing è il modo standard per fare "mini-batching" o selezionare porzioni di un dataset.

Modo rapido per creare liste:

```
[expression for item in iterable if condition]
```

```
1    data = [1, 2, 3, 4, 5, 6]
2    # Seleziona solo i pari e fanne il quadrato
3    squares = [x**2 for x in data if x % 2 == 0]
4    # squares: [4, 16, 36]
```

Le **tuple** sono simili alle liste, ma sono **immutabili**.

- **Sintassi**: `t = (1, 2, 3)` (parentesi tonde).
- Una volta creata, non puoi aggiungere o modificare elementi.

```
l = [1, 2]
l[0] = 10 # OK

t = (1, 2)
# t[0] = 10 # Errore!
```

Perché serve in ML?

Le tuple vengono spesso usate per rappresentare le **dimensioni** (shape) di un'immagine o di un dataset: `shape = (224, 224, 3)`.

In Python, l'uso di più parentesi indica spesso **strutture annidate**.

```
# Una lista di tuple
points = [(0, 0), (1, 1), (2, 2)]

# Passare una tupla come singolo argomento
# random.sample vuole (popolazione, k)
import random
res = random.sample((10, 20, 30), 2)
```

- `func((1, 2))`: Stai passando **un solo** argomento (una tupla).
- `func(1, 2)`: Stai passando **due** argomenti distinti.

Coppie chiave-valore. Utili come contenitori di iper-parametri.

```
config = {
'lr': 0.01,
'epochs': 10,
'optimizer': 'Adam'
}
print(config['lr'])
config['batch_size'] = 32
```

Iterazione:

```
for k, v in config.items():
print(f"{k} -> {v}")
```

Per accedere ai contenuti di un dizionario possiamo usare:

- `.keys()`: Restituisce le chiavi.
- `.values()`: Restituisce i valori.
- `.items()`: Restituisce coppie (chiave, valore).

```
d = {"lr": 0.01, "optimizer": "Adam"}

print(d.keys()) # dict_keys(['lr', 'optimizer'])
print(d.values()) # dict_values([0.01, 'Adam'])
print(d.items()) # dict_items([('lr', 0.01), ...])
```

- **Enumerate**: Indice + Valore.
- **Zip**: Mette insieme più sequenze.

```
inputs = [0.1, 0.2, 0.3]
labels = [0, 1, 0]

for i, (x, y) in enumerate(zip(inputs, labels)):
print(f"Esempio {i}: Input {x}, Label {y}")
```

Data una lista di stringhe `["train", "test", "val"]`, usa una list comprehension per creare una lista di tuple `(indice, stringa_maiuscola)`.
Risultato atteso: [(0, 'TRAIN'), (1, 'TEST'), ...]

- **KeyError**: Cercare una chiave inesistente in un dict (usa `.get()`).
- **Performance**: Cercare in una lista è $O(n)$, in un set/dict è $O(1)$.
- **Mutabilità**: Ricordare che una lista come chiave di un dict **non è ammessa** (usa le tuple!).

- Definire funzioni con parametri opzionali.
- Distinguere tra parametri posizionali e keyword.
- Capire il ritorno di valori multipli.

```python
def train_model(data, lr=0.01, model_type="CNN"):
"""Docstring: addestra un modello."""
print(f"Training {model_type} with lr={lr}")
return "Success", 0.95 # Ritorna una tupla
```

Chiamate flessibili:

```python
res, acc = train_model(my_data, model_type="Transformer")
```

In Python le funzioni sono ”first-class citizens”.

```
    def double(x): return x * 2
    f = double
    print(f(5)) # 10

    # Lambda (anonime)
    f_add = lambda x, y: x + y
    print(f_add(2, 3)) # 5
```

- **Default mutabili**: Mai usare `def f(l=[])`. La lista `l` viene creata una volta sola e riutilizzata tra le chiamate! Usa `None` come default.
- **Scope**: Variabili definite dentro la funzione non sono visibili fuori.
- **Shadowing**: Non chiamare variabili con i nomi di funzioni builtin (es. `list = [1, 2]`).

```python
import math
print(math.sqrt(16))

from random import shuffle, seed
seed(42) # Riproducibilità!

import os
print(os.getcwd()) # Directory corrente
```

Perché serve in ML?

Il modulo `random` è vitale per lo split dei dataset e l'inizializzazione dei pesi. La riproducibilità (`seed`) è fondamentale.

Non aver paura degli errori! L'ultima riga dice **cosa**, le righe sopra dicono **dove**.

```
l = [1, 2]
# l[5] -> IndexError: list index out of range

# d = {"a":1}
# d["b"] -> KeyError: 'b'
```

Asserzioni: Utili per sanity check nei training loop.

```
assert x > 0, "Il learning rate deve essere positivo!"
```

```
try:
val = int(input("Inserisci un numero: "))
except ValueError:
print("Non era un numero!")
```