lezione4

April 16, 2025

1 Lezione 4

1.1 Ambienti Virtuali

Gli ambienti virtuali Python sono spazi isolati dove è possibile installare pacchetti e dipendenze specifiche per un determinato progetto senza interferire con altri progetti o con l'installazione di sistema.

1.1.1 Perché usare ambienti virtuali?

In ambito biomedico, spesso si lavora con librerie specializzate (come scikit-learn, biopython, pydicom) che potrebbero richiedere versioni specifiche o avere dipendenze in conflitto tra progetti diversi.

1.1.2 Strumenti principali per ambienti virtuali

1. **venv** - Modulo integrato nella libreria standard Python
2. **conda** - Gestore pacchetti e ambiente per data science (incluso in Anaconda)
3. **virtualenv** - Strumento esterno simile a venv, più vecchio ma con più funzionalità
4. **pipenv** - Combina pip e virtualenv in un unico strumento

1.1.3 venv: l'opzione integrata

`venv` è il modulo standard Python per creare ambienti virtuali ed è disponibile a partire da Python 3.3.

Creazione di un ambiente virtuale

```
# Creare un ambiente virtuale chiamato 'myenv'
python -m venv myenv
```

Attivazione dell'ambiente Su Windows:

[ ]:
```
myenv\Scripts\activate
```

Su macOS/Linux:

[ ]:
```
source myenv/bin/activate
```

Utilizzo dell'ambiente Una volta attivato, vedrete il nome dell'ambiente nel prompt dei comandi. Ora potete installare pacchetti specifici:

```
[ ]: pip install numpy pandas matplotlib scikit-learn
```

Salvare e riprodurre l'ambiente

```
[ ]: # Salvare i pacchetti installati in requirements.txt
pip freeze > requirements.txt

# Reinstallare gli stessi pacchetti in un altro ambiente
pip install -r requirements.txt
```

Disattivazione dell'ambiente

```
[ ]: deactivate
```

1.1.4 conda: soluzione per data science

Particolarmente utile per applicazioni biomediche con dipendenze complesse:

```
[ ]: # Creare ambiente conda
conda create --name biomed_env python=3.9

# Attivare
conda activate biomed_env

# Installare pacchetti scientifici
conda install numpy pandas scikit-learn matplotlib seaborn
conda install -c conda-forge biopython pydicom

# Disattivare
conda deactivate
```

1.1.5 Scenario biomedico

Immaginate di sviluppare un algoritmo per l'analisi di immagini MRI che richiede TensorFlow 2.0, mentre un altro progetto per l'analisi di segnali ECG necessita di una versione specifica di scikit-learn. Gli ambienti virtuali vi permettono di mantenere queste configurazioni separate e riproducibili.

1.2 Quick Guide: Installing Jupyter Notebook

Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and text.

1.2.1 Ubuntu Installation

Using pip “‘bash # Install pip if not already installed sudo apt update sudo apt install python3-pip

2 Install Jupyter Notebook

pip3 install notebook

3 Launch Jupyter Notebook

jupyter notebook

3.0.1 macOS Installation

Using pip

[ ]:
```
# Install pip if not already installed (most macOS comes with pip)
python3 -m ensurepip --upgrade

# Install Jupyter Notebook
pip3 install notebook

# Launch Jupyter Notebook
jupyter notebook
```

Using conda

[ ]:
```
# Install Miniconda (if not already installed)
# Download from https://docs.conda.io/en/latest/miniconda.html
# Or use Homebrew:
brew install --cask miniconda

# Install Jupyter Notebook
conda install -c conda-forge notebook

# Launch Jupyter Notebook
jupyter notebook
```

3.0.2 Using Jupyter in VS Code

1. **Install Jupyter Extension**:
   - Search for “Jupyter” in the Extensions marketplace
   - Install the Jupyter extension
2. **Create/Open Notebooks**:
   - Create a new file with `.ipynb` extension
   - Or open an existing notebook
   - VS Code will automatically use the Jupyter environment
3. **Select Python Interpreter**:
   - Click on “Select Kernel” in the top-right of the notebook

- Choose your Python environment

3.0.3 Basic Usage

After installation on any platform: - Jupyter Notebook will open in your default web browser - Navigate to your project folder - Click "New" → "Python 3" to create a new notebook - Write code in cells and execute with Shift+Enter

In VS Code: - Execute cells with the "Run Cell" button or Shift+Enter - Manage kernels and variables through the VS Code interface

---

3.1 Funzioni

3.1.1 Definizione di Base delle Funzioni

Le funzioni permettono di organizzare il codice in blocchi riutilizzabili. In Python, le funzioni sono definite usando la parola chiave `def`.

[ ]:
```python
def greet_patient():
    """Una semplice funzione che saluta un paziente"""
    print("Welcome to the cardiology department!")

# Chiamata della funzione
greet_patient()
```

3.1.2 Funzioni con Parametri

Le funzioni diventano più utili quando possono accettare dati in input.

[5]:
```python
def calculate_bmi(weight_kg, height_m):
    """Calcola l'Indice di Massa Corporea"""
    bmi = weight_kg / (height_m ** 2)
    print(f"BMI: {bmi:.1f}")

# Calcola BMI per un paziente
calculate_bmi(70, 1.75)  # Circa 22.9
```

```
BMI: 22.9
```

3.1.3 Valori di Ritorno

Le funzioni possono elaborare dati e restituire risultati.

[ ]:
```python
# Stessa funzione di prima solo che ora restitusice un valore
# che può essere salvato
def calculate_bmi(weight_kg, height_m):
    """Calcola l'Indice di Massa Corporea"""
    bmi = weight_kg / (height_m ** 2)
    print(f"BMI: {bmi:.1f}")
```

```
    return bmi

# Calcola BMI per un paziente
patient_bmi = calculate_bmi(70, 1.75)  # Circa 22.9
```

```
BMI: 22.9
```

[6]:
```
def is_fever(temperature):
    """Determina se il paziente ha febbre in base alla temperatura"""
    return temperature > 37.5


# Controlla se i pazienti hanno febbre
has_fever = is_fever(38.2)
print(f"Patient has fever: {has_fever}")  # True
```

```
Patient has fever: True
```

[ ]:
```

```

3.1.4 Parametri con Valori Predefiniti

Le funzioni possono avere parametri con valori predefiniti.

[ ]:
```
# stessa funzione di prima solo che ora permette di controllare
# anche in Fahrenheit
def is_fever(temperature, celsius=True):
    """Determina se il paziente ha febbre in base alla temperatura"""
    if celsius:
        return temperature > 37.5
    else:
        return temperature > 99.5  # Fahrenheit

# Controlla se i pazienti hanno febbre
has_fever = is_fever(38.2)
print(f"Patient has fever: {has_fever}")  # True

# Usando Fahrenheit
has_fever = is_fever(98.6, celsius=False)
print(f"Patient has fever: {has_fever}")  # False
```

```
Patient has fever: True
Patient has fever: False
```

[8]:
```
def prescribe_antibiotic(infection, dosage_mg=500):
    """Prescrive un antibiotico con un dosaggio predefinito"""
    return f"For {infection}, take {dosage_mg}mg of antibiotics twice daily."

# Usando il dosaggio predefinito
```

```
print(prescribe_antibiotic("strep throat"))

# Specificando un dosaggio diverso
print(prescribe_antibiotic("severe pneumonia", 750))
```

```
For strep throat, take 500mg of antibiotics twice daily.
For severe pneumonia, take 750mg of antibiotics twice daily.
```

3.1.5 *Scope* (ambito) delle Variabili

Le variabili definite all'interno delle funzioni sono locali a quelle funzioni.

[14]:
```
def analyze_blood_sample():
    wbc_count = 7500  # Conta dei globuli bianchi (variabile locale)
    print(f"WBC count: {wbc_count} cells/ L")
    return wbc_count > 10000  # Restituisce True se elevata
```

[15]:
```
result = analyze_blood_sample()
print(f"WBC elevated: {result}")
```

```
WBC count: 7500 cells/ L
WBC elevated: False
```

[16]:
```
# Questo causerebbe un errore perché wbc_count non è definita al
# di fuori della funzione
print(wbc_count)  # NameError: name 'wbc_count' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[16], line 3
      1 # Questo causerebbe un errore perché wbc_count non è definita al
      2 # di fuori della funzione
----> 3 print(wbc_count)  # NameError: name 'wbc_count' is not defined

NameError: name 'wbc_count' is not defined
```

3.1.6 Valori di Ritorno Multipli

Le funzioni possono restituire più valori sottoforma di tupla.

[ ]:
```
def blood_pressure_status(systolic, diastolic):
    """Categorizza la pressione sanguigna e restituisce la categoria con i␣
 ↪valori"""
    if systolic < 120 and diastolic < 80:
        category = "Normal"
    elif systolic < 130 and diastolic < 80:
        category = "Elevated"
    elif 130 <= systolic < 140 or 80 <= diastolic < 90:
```

```
        category = "Stage 1 Hypertension"
    else:
        category = "Stage 2 Hypertension"

    return category, systolic, diastolic

# Ottieni lo stato della pressione sanguigna
status, sys_bp, dia_bp = blood_pressure_status(142, 88)
print(f"Patient has {status} with BP {sys_bp}/{dia_bp}")
```

3.1.7 *args e **kwargs

Questi permettono alle funzioni di accettare un numero variabile di argomenti.

[ ]:
```
def calculate_average_glucose(*readings):
    """Calcola la glicemia media da più misurazioni"""
    return sum(readings) / len(readings)

# Media della glicemia da mattina, mezzogiorno, sera
avg = calculate_average_glucose(95, 110, 87)
print(f"Average glucose: {avg} mg/dL")

def patient_info(name, **data):
    """Memorizza informazioni del paziente con campi variabili"""
    print(f"Patient: {name}")
    for key, value in data.items():
        print(f"  {key}: {value}")

# Crea una cartella clinica con dati variabili
patient_info("Jane Doe",
             age=42,
             blood_type="O+",
             allergies=["penicillin", "latex"],
             heart_rate=72)
```

3.1.8 Funzioni Lambda

Le funzioni Lambda sono piccole funzioni anonime definite con la parola chiave `lambda`.

[ ]:
```
# Converti Celsius in Fahrenheit usando una funzione lambda
c_to_f = lambda celsius: (celsius * 9/5) + 32

# Converti la temperatura corporea
body_temp_c = 37.0
body_temp_f = c_to_f(body_temp_c)
print(f"{body_temp_c}°C = {body_temp_f}°F")
```

```
# Utilizzo di lambda in una funzione filter per trovare frequenze cardiache␣
 ↪anomale
heart_rates = [62, 70, 95, 110, 65, 72]
abnormal_rates = list(filter(lambda hr: hr < 60 or hr > 100, heart_rates))
print(f"Abnormal heart rates: {abnormal_rates}")
```

3.1.9 Documentazione delle Funzioni

Documenta le funzioni con docstring per spiegarne lo scopo e i parametri.

[ ]:
```
def estimate_drug_dosage(weight_kg, age, kidney_function=1.0):
    """
    Calcola il dosaggio raccomandato di farmaco in base ai parametri del␣
 ↪paziente.

    Args:
        weight_kg: Peso del paziente in chilogrammi
        age: Età del paziente in anni
        kidney_function: Punteggio della funzione renale (0.0-1.0,␣
 ↪predefinito=1.0)

    Returns:
        Dosaggio raccomandato del farmaco in mg
    """
    base_dose = weight_kg * 2
    age_factor = 1.0 if age < 65 else 0.85
    return base_dose * age_factor * kidney_function

# Ottieni il dosaggio per diversi pazienti
young_patient_dose = estimate_drug_dosage(70, 30)
elderly_patient_dose = estimate_drug_dosage(65, 70, kidney_function=0.7)

print(f"Young patient dosage: {young_patient_dose}mg")
print(f"Elderly patient dosage: {elderly_patient_dose}mg")

# Accedi alla documentazione della funzione
print(estimate_drug_dosage.__doc__)
```

3.1.10 Esercizio: Creare un Calcolatore di Variabilità della Frequenza Cardiaca (HRV)

Prova a creare la tua funzione che analizza la variabilità della frequenza cardiaca da una lista di intervalli tra battiti:

[ ]:
```
# Soluzione di esempio
def calculate_hrv(beat_intervals):
    """
```

```
    Calcola metriche di variabilità della frequenza cardiaca da una lista di␣
↪intervalli RR

    Args:
        beat_intervals: Lista di intervalli tra battiti cardiaci (in ms)

    Returns:
        Dizionario con metriche HRV
    """
    # Calcola SDNN (Deviazione Standard degli intervalli NN)
    mean = sum(beat_intervals) / len(beat_intervals)
    variance = sum((x - mean) ** 2 for x in beat_intervals) /␣
↪len(beat_intervals)
    sdnn = variance ** 0.5

    # Calcola RMSSD (Radice Quadrata della Media delle Differenze Successive al␣
↪Quadrato)
    successive_diffs = [abs(beat_intervals[i+1] - beat_intervals[i])
                        for i in range(len(beat_intervals)-1)]
    rmssd = (sum(x ** 2 for x in successive_diffs) / len(successive_diffs)) **␣
↪0.5

    # Frequenza cardiaca media
    average_hr = 60000 / mean  # 60000ms = 1 minuto

    return {
        "sdnn": sdnn,
        "rmssd": rmssd,
        "average_hr": average_hr
    }

# Test con dati di un paziente sano (intervalli in ms)
healthy_intervals = [825, 832, 820, 834, 828, 830, 824, 828, 832, 830]
hrv_results = calculate_hrv(healthy_intervals)
print(f"HRV Analysis Results:")
for metric, value in hrv_results.items():
    print(f"  {metric}: {value:.2f}")
```

3.2 Classi in Python: Concetti Introduttivi

Le classi in Python permettono di implementare la programmazione orientata agli oggetti (OOP), particolarmente utile per modellare concetti del mondo reale ed organizzare codice complesso.

3.2.1 Definizione di Base

Una classe è un modello per creare oggetti con attributi (dati) e metodi (funzioni) che operano su quei dati.

[ ]:
```python
class Patient:
    """Classe rappresentante un paziente."""

    def __init__(self, name, age, patient_id):
        """Costruttore: inizializza un nuovo paziente."""
        self.name = name            # Attributo pubblico
        self.age = age              # Attributo pubblico
        self.patient_id = patient_id # Attributo pubblico
        self._medical_history = []  # Attributo "protetto" (convenzione)

    def add_diagnosis(self, diagnosis):
        """Aggiunge una diagnosi alla storia clinica."""
        self._medical_history.append(diagnosis)

    def get_summary(self):
        """Restituisce un riepilogo del paziente."""
        return f"Patient {self.name} (ID: {self.patient_id}), Age: {self.age}"
```

3.2.2 Creazione di Istanze

Un'istanza è un oggetto specifico creato da una classe.

[ ]:
```python
# Creazione di istanze della classe Patient
patient1 = Patient("Maria Rossi", 45, "P12345")
patient2 = Patient("Carlo Bianchi", 62, "P67890")

# Accesso agli attributi
print(patient1.name)      # Output: "Maria Rossi"

# Chiamata di metodi
patient1.add_diagnosis("Hypertension")
print(patient1.get_summary())
```

3.2.3 Prossimamente...

Ereditarietà... Metodi Speciali... Altro...

3.3 Iteratori in Python

Gli iteratori sono oggetti che permettono di attraversare collezioni di dati in modo sequenziale, un elemento alla volta, senza dover caricare l'intera collezione in memoria.

3.3.1 Concetto di base

In Python, un iteratore è un oggetto che implementa due metodi speciali: - `__iter__()`: ritorna l'oggetto iteratore stesso - `__next__()`: restituisce il prossimo elemento della collezione e solleva `StopIteration` quando non ci sono più elementi

3.3.2 Perché usare gli iteratori?

In applicazioni biomediche, gli iteratori sono particolarmente utili quando si lavora con: - Dataset di grandi dimensioni (come sequenze genomiche) - File di immagini medicali in serie - Flussi continui di dati da dispositivi di monitoraggio

3.3.3 Iteratori integrati

Molte strutture dati Python sono già iterabili:

[ ]:
```python
# Liste, tuple, dizionari, set sono tutti iterabili
patient_readings = [98.6, 99.1, 97.8, 98.2]

for reading in patient_readings:  # Usa implicitamente un iteratore
    print(f"Temperature: {reading}°F")
```

3.3.4 Creare iteratori personalizzati

È possibile definire iteratori personalizzati implementando i metodi richiesti:

[ ]:
```python
class ECGDataIterator:
    def __init__(self, data, window_size=100):
        self.data = data
        self.window_size = window_size
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index + self.window_size >= len(self.data):
            raise StopIteration
        segment = self.data[self.index:self.index+self.window_size]
        self.index += self.window_size
        return segment

# Uso dell'iteratore personalizzato
ecg_data = [random.random() for _ in range(1000)]  # Dati simulati
ecg_iterator = ECGDataIterator(ecg_data)

for segment in ecg_iterator:
    # Processa ogni segmento di 100 punti dati
    peak = max(segment)
    print(f"Segment peak value: {peak:.3f}")
```

3.3.5 Generatori: iteratori semplificati

I generatori sono un modo più semplice per creare iteratori, utilizzando la parola chiave `yield`:

[ ]:
```python
def bp_readings_generator(num_readings):
    """Genera letture simulate della pressione sanguigna."""
    for i in range(num_readings):
        # Simula una lettura della pressione (sistolica/diastolica)
        systolic = random.randint(110, 140)
        diastolic = random.randint(70, 90)
        yield (systolic, diastolic)

# Utilizzo del generatore
for systolic, diastolic in bp_readings_generator(5):
    print(f"BP: {systolic}/{diastolic} mmHg")
```

3.3.6 Funzioni integrate per iteratori

Python offre funzioni utili per lavorare con iteratori: - `map()`: applica una funzione a ogni elemento - `filter()`: seleziona elementi in base a una condizione - `zip()`: combina più iterabili

[ ]:
```python
temperatures = [36.5, 37.2, 38.0, 36.8, 37.5]
patient_ids = ["A001", "A002", "A003", "A004", "A005"]

# Identifica pazienti con febbre
fever_patients = list(filter(lambda x: x[1] > 37.2,
                             zip(patient_ids, temperatures)))
print(f"Patients with fever: {fever_patients}")
```

3.3.7 Vantaggi in ambito biomedico

Gli iteratori consentono di elaborare efficientemente grandi set di dati biologici o medici (come sequenze genomiche o serie di immagini) senza sovraccaricare la memoria, rendendoli strumenti preziosi nell'analisi di dati biomedici.

3.4 NumPy: Introduzione Rapida

NumPy è una libreria Python fondamentale per il calcolo scientifico. Fornisce supporto per array multidimensionali, funzioni matematiche avanzate e strumenti per manipolare dati numerici in modo efficiente.

3.4.1 Concetti Chiave

- **ndarray**: Struttura dati principale di NumPy - array multidimensionale omogeneo
- **Vectorization**: Operazioni eseguite su interi array senza cicli espliciti
- **Broadcasting**: Meccanismo per operazioni tra array di forme diverse

3.4.2 Creazione di Array

[ ]:
```python
import numpy as np

# Array monodimensionale
vital_signs = np.array([98.6, 72, 120, 80])  # temp, heart_rate, BP_sys, BP_dia

# Array bidimensionale (matrice)
patient_data = np.array([
    [98.6, 72, 120, 80],  # Paziente 1
    [97.9, 68, 118, 76],  # Paziente 2
    [99.1, 88, 135, 90]   # Paziente 3
])

# Array con valori specifici
zeros = np.zeros((3, 4))  # Matrice 3x4 di zeri
ones = np.ones((2, 2))    # Matrice 2x2 di uni
identity = np.eye(3)      # Matrice identità 3x3

# Array con sequenze
range_array = np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
linear_space = np.linspace(0, 1, 5)  # 5 punti equidistanti tra 0 e 1
```

Operazioni di Base

[ ]:
```python
# Operazioni elementwise
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
sum_array = a + b  # [5, 7, 9]
product = a * b    # [4, 10, 18]

# Statistiche
data = np.array([70.5, 68.2, 73.1, 69.8, 72.4])
mean = np.mean(data)    # Media
std_dev = np.std(data)  # Deviazione standard
min_val = np.min(data)  # Valore minimo
max_val = np.max(data)  # Valore massimo

# Algebra lineare
matrix_A = np.array([[1, 2], [3, 4]])
matrix_B = np.array([[5, 6], [7, 8]])
matrix_product = np.dot(matrix_A, matrix_B)  # Moltiplicazione matriciale
determinant = np.linalg.det(matrix_A)        # Determinante
inverse = np.linalg.inv(matrix_A)            # Inversa
```

Slicing e Indicizzazione

[ ]:
```
# Dati ECG simulati (una porzione)
ecg = np.random.normal(0, 1, 1000)

# Ottenere i primi 100 punti
first_segment = ecg[:100]

# Matrice di dati paziente (rows=pazienti, cols=misurazioni)
patient_matrix = np.random.randint(60, 150, size=(10, 5))
first_patient = patient_matrix[0, :]   # Tutte le misurazioni del primo paziente
blood_pressure = patient_matrix[:, 2]  # Terza misurazione di tutti i pazienti
```

Applicazione Biomedica

[2]:
```
# Filtro semplice per segnale biomedico (media mobile)
def moving_average(signal, window_size=5):
    return np.convolve(signal, np.ones(window_size)/window_size, mode='valid')

# Calcolo BMI da matrici di dati
heights = np.array([1.75, 1.82, 1.65])  # metri
weights = np.array([75, 85, 60])         # kg
bmi = weights / (heights ** 2)
print(f"BMI values: {bmi}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 6
      3     return np.convolve(signal, np.ones(window_size)/window_size,␣
 ↪mode='valid')
      5 # Calcolo BMI da matrici di dati
----> 6 heights = np.array([1.75, 1.82, 1.65])  # metri
      7 weights = np.array([75, 85, 60])         # kg
      8 bmi = weights / (heights ** 2)

NameError: name 'np' is not defined
```

3.5 Culture Pill / Esterni