Introduction to Python

Workshop 2

Thomas Zilliox

Agenda

- Object-Oriented Programming (classes, inheritance,...)
- Handling JSON, CSV, and SQLite
- Basics of NumPy, Pandas, and Matplotlib
- Virtual environments and project layout (src/, tests/, setup.py or pyproject.toml)
- Working with APIs and REST endpoints

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that organizes software around **objects** and **classes**.

OOP is built on four key principles:

1. **Encapsulation**: Grouping data and methods together to protect internal state and provide a clean interface.

2. **Abstraction**: Hiding complex implementation details behind simple, high-level operations.

3. **Inheritance**: Creating new classes from existing ones to promote reuse and reduce duplication.

4. **Polymorphism**: Allowing different objects to respond to the same operation in their own way.

Object-Oriented Programming - Class

A class is a **blueprint** that defines the **structure** and **behavior** of objects in object-oriented programming. It specifies:

1. **Attributes**: the data or properties an object will hold.

2. **Methods**: the actions or functions the object can perform.

Object-Oriented Programming - Object

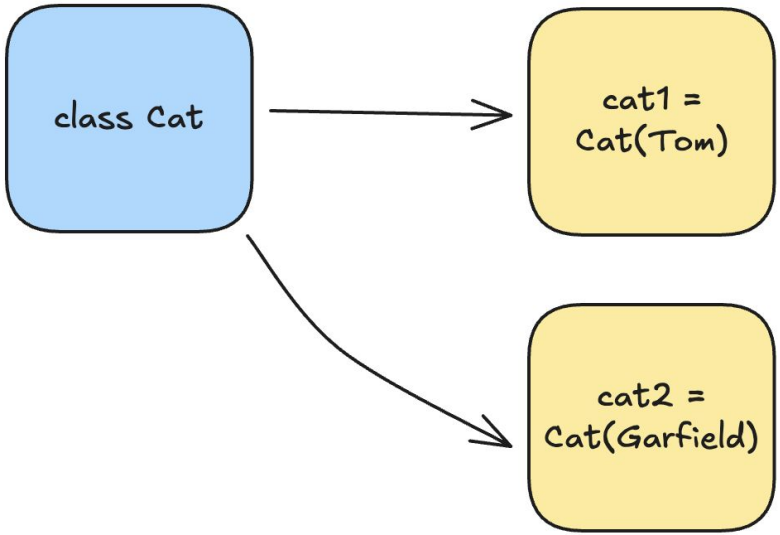
An object is an **instance** of a class.

It contains **actual data** and can perform the **behaviors** defined by its class.

1. It has its own **state** (specific values stored in its attributes).

2. It provides **behavior** through methods defined in the class.

3. It represents a concrete, usable version of the class blueprint.



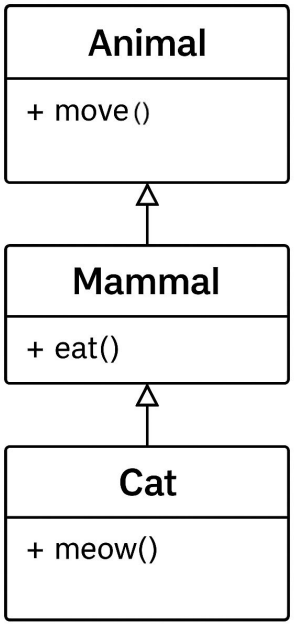
Object-Oriented Programming - Inheritance

Inheritance is an OOP mechanism that allows one class to **reuse**, **extend**, or **modify** the behavior and attributes of another class.
It helps reduce repetition and creates natural hierarchies.

1. The existing class is the **parent** (or base/super) class.

2. The new class is the **child** (or derived/sub) class.

3. The child inherits attributes and methods, and can add or override behavior.

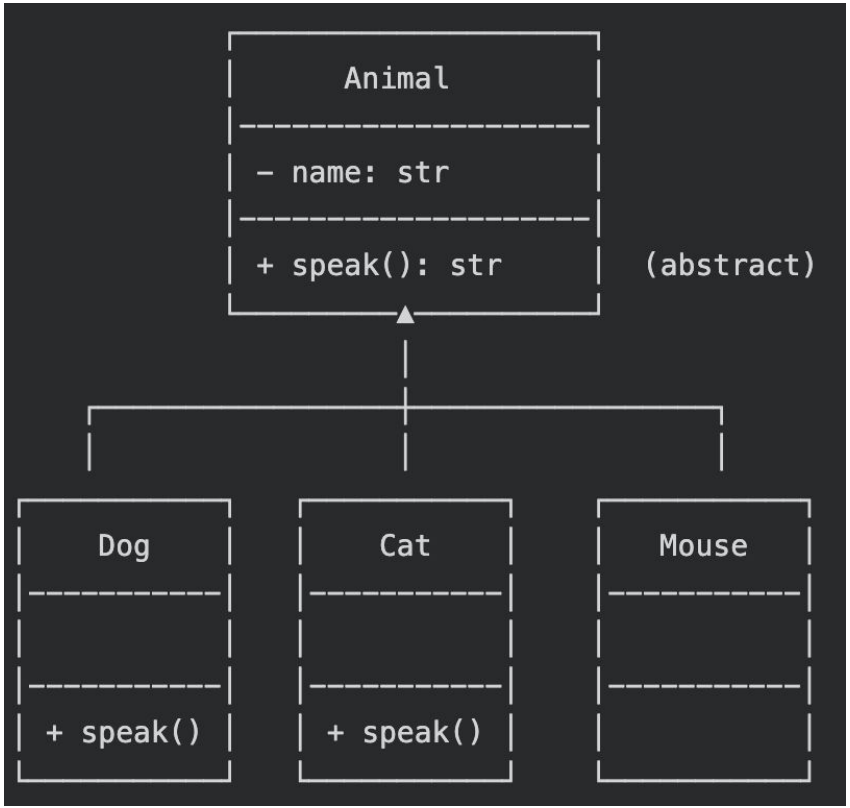


Object-Oriented Programming - example

```
class Animal:

    def __init__(self, name):

        self.name = name

    def speak(self):

        return f"{self.name} says: ???"


class Dog(Animal):

    def speak(self):
          return f"{self.name} says: Woof!"

class Cat(Animal):

    def speak(self):
          return f"{self.name} says: Meow!"
```

```
class Mouse(Animal):
    pass



animals = [
     Dog("Dug"),
     Cat("Garfield"),
     Mouse("Mickey")
]

for animal in animals:

    print(animal.speak())
```

Object-Oriented Programming - example



Object-Oriented Programming - example

```
class Animal:

    def __init__(self, name):

        self.name = name

    def speak(self):

        return f"{self.name} says: ???"


class Dog(Animal):

    def speak(self):
          return f"{self.name} says: Woof!"

class Cat(Animal):

    def speak(self):
          return f"{self.name} says: Meow!"
```

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self,sound="???"):
        return f"{self.name} says: {sound}"


class Dog(Animal):
    def speak(self):
          return super().speak("Woof!")


class Cat(Animal):
    def speak(self):
          return super().speak("Meow!")
```

Object-Oriented Programming - super()

When you call:

super().method()

Python looks at the next class in the **Method Resolution Order** (MRO) and calls that version of the method.

The MRO is the order in which Python looks up methods and attributes in a class's inheritance hierarchy.

More info: https://docs.python.org/3/howto/mro.html

Object-Oriented Programming - example

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self,sound="???"):
        return f"{self.name} says: {sound}"

class Dog(Animal):
    def speak(self):
          return super().speak("Woof!")

class Cat(Animal):
    def speak(self):
          return super().speak("Meow!")
```

```
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def speak(self):
        """Every animal must implement this."""
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} says: Woof!"
```

Object-Oriented Programming - ABC

ABC stands for Abstract Base Class.

An **Abstract Base Class (ABC)** is a special kind of class in object-oriented programming that defines a common interface for its subclasses but **cannot be instantiated on its own**. It often contains one or more **abstract methods**,forcing child classes to provide their own implementation.

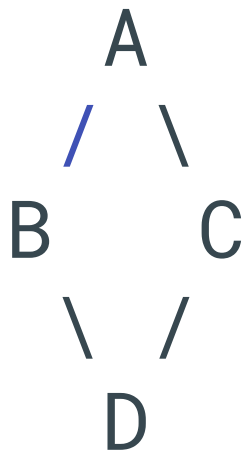
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-2900861230.py in <cell line: 0>()
     27     Dog("Dug"),
     28     Cat("Garfield"),
---> 29     Mouse("Mickey")
     30 ]
     31

TypeError: Can't instantiate abstract class Mouse without an implementation for abstract method 'speak'
```

Object-Oriented Programming - Multiple inheritance

```
class FlyerMixin:
    def fly(self):
        return f"{self.name} is flying!"

class SwimmerMixin:
    def swim(self):
        return f"{self.name} is swimming!"

class Animal:
    def __init__(self, name):
        self.name = name

class Duck(Animal, FlyerMixin, SwimmerMixin):
    pass
```

```
donald = Duck("Donald")

# Donald is flying!
print(donald.fly())

# Donald is swimming!
print(donald.swim())
```

Object-Oriented Programming - Diamond inheritance




```
class A:
    def say(self):
        print("A says hello")

class B(A):
    def say(self):
        print("B says hello")
        super().say()

class C(A):
    def say(self):
        print("C says hello")
        super().say()

class D(B, C):
    pass
```

What happened ?

```
d = D()
d.say()
```

The following line will let us know:

```
print(D.__mro__)
```

Object-Oriented Programming - Dunder

Dunder methods (short for **“double-underscore methods”**) are special methods in Python whose names start and end with two underscores.

https://docs.python.org/3/reference/datamodel.html#basic-customization

Object-Oriented Programming - Dunder (Life cycle)

```
class A:
    def __init__(self, name):
        self.name = name
        print(f"[INIT] of {self.name}.")

    def __del__(self):
        # Not reliable, but good to illustrate object lifetime
        print(f"[DEL] {self.name} is being cleaned up.")
```

Object-Oriented Programming - Dunder (Life cycle)

```
class FakeServer:
    def __init__(self, name):
        ...

    def __enter__(self):
        ...

    def __exit__(self, ...):
        ...

    def handle_request(self, request):
     ...
```

```
with FakeServer("DemoServer") as server:
    server.handle_request("GET /hello")
    server.handle_request("POST /data")
```

Object-Oriented Programming - Dataclass

```
class User:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"Hello, I'm
{self.name}"
```

```
from dataclasses import dataclass

@dataclass
class UserDC:
    name: str

    def __str__(self):
        return f"Hello, I'm {self.name}"
```

```
u = User("Alice")
print(str(u))

u_dc = UserDC("Bob")
print(str(u_dc))
```

Agenda

- Object-Oriented Programming (classes, inheritance,...)
- Handling JSON, CSV, and SQLite
- Basics of NumPy, Pandas, and Matplotlib
- Virtual environments and project layout (src/, tests/, setup.py or pyproject.toml)
- Working with APIs and REST endpoints

Handling JSON, CSV and SQLite - JSON

A JSON file (JavaScript Object Notation) is a lightweight, text-based format used to store and exchange structured data. It's easy for humans to read and write, and easy for machines to parse and generate.

JSON represents data using a small set of simple structures:

- Objects → key–value pairs (like Python dictionaries)
- Arrays → ordered lists of values
- Primitive values → strings, numbers, booleans, and null

More info on www.json.org

💡 In practice a json will behave like a python dictionary.

Handling JSON, CSV and SQLite - JSON

```
import json

data = {

     "name": "Alice",
     "age": 30,
     "languages": ["Python", "JavaScript"]

}


# Write JSON to a file

with open("data.json", "w") as f:

    json.dump(data, f, indent=4)  # indent for readability
```

Handling JSON, CSV and SQLite - JSON

```
import json


# Read JSON from a file
with open("data.json", "r") as f:
    data = json.load(f)


print(data)
print(f"Name is {data["name"]}")
```

Handling JSON, CSV and SQLite - CSV

A CSV file (Comma-Separated Values) is a simple, text-based format used to store tabular data. Each line represents a row, and each value in the row is separated by a delimiter, typically a comma.

Key points:

- Easy to read and edit
- Supported by almost all data tools (Excel, Python, databases, etc.)
- Lightweight and human-readable

They’re commonly used for exporting, sharing, or storing structured data in a simple table-like form.

Handling JSON, CSV and SQLite - CSV

```
import pandas as pd


# Create a simple DataFrame

df = pd.DataFrame({

    "name": ["Alice", "Bob", "Charlie"],
    "age": [30, 25, 35],
    "city": ["Paris", "Berlin", "London"]

})

# Write the DataFrame to a CSV file

df.to_csv("data.csv", index=False)
```

Handling JSON, CSV and SQLite - CSV

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv("data.csv")

print(df)
mean_age = df["age"].mean()

print("Mean age:", mean_age)
```

Handling JSON, CSV and SQLite - SQLite

An SQLite .db file is a single, self-contained database file used by the SQLite database engine. It stores structured data in tables, just like a traditional SQL database, but without requiring a server.
Key points:

- Self-contained: all data (tables, indexes, schemas) lives inside one file.
- Serverless: no database server is needed; applications read/write directly to the file. -> local
- Lightweight: small footprint, ideal for local storage or embedded applications.
- Uses SQL: you interact with it using standard SQL queries (SELECT, INSERT, etc.).
- Cross-platform: the same .db file works on Windows, macOS, Linux, Android, etc.
- Easy to migrate

Handling JSON, CSV and SQLite - SQLite

```
import sqlite3

# Connect to (or create) the database file
conn = sqlite3.connect("example.db")
cursor = conn.cursor()

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 30))
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Bob", 25))

# Save (commit) changes
conn.commit()

# Close connection
conn.close()
```

Agenda

- Object-Oriented Programming (classes, inheritance,...)
- Handling JSON, CSV, and SQLite
- Basics of NumPy, Pandas, and Matplotlib
- Virtual environments and project layout (src/, tests/, setup.py or pyproject.toml)
- Working with APIs and REST endpoints

Basics of NumPy, Pandas, and Matplotlib - Numpy

→ Numerical Computing Foundation

NumPy is the fundamental package for scientific computing in Python. It provides fast, memory-efficient arrays and operations optimized in C.

Key points:

- Enables vectorized operations (much faster than Python loops).
- Provides ndarray, a powerful n-dimensional array structure.
- Offers mathematical functions for linear algebra, statistics, and more.

Basics of NumPy, Pandas, and Matplotlib - Numpy

```
import numpy as np


# Create arrays

a = np.array([1, 2, 3])
b = np.zeros((2, 2))
c = np.random.rand(3, 3)

# Vectorized operations

d = a * 10           # [10, 20, 30]
e = a + np.array([4, 5, 6])  # [5, 7, 9]

# Useful methods

mean_val = a.mean()
```

Basics of NumPy, Pandas, and Matplotlib - Pandas

→ Data Manipulation and Analysis

Pandas is built on top of NumPy and provides two key structures: Series (1D) and DataFrame (2D table).

Key points:

- Makes working with tabular data intuitive.
- Provides powerful tools for filtering, grouping, merging, cleaning, and transforming data.
- Integrates well with NumPy and Matplotlib.

Basics of NumPy, Pandas, and Matplotlib - Pandas

```python
import pandas as pd

# Create DataFrame
data = {
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 32, 18]
}
df = pd.DataFrame(data)

# Inspect data
df.head()
df.describe()
```

```python
# Select and filter
adults = df[df["age"] > 20]

# Add a new column
df["age_plus_10"] = df["age"] + 10

# Sort by a column
df.sort_values("age", ascending=False)

# Select multiple columns
df[["name", "age"]]
# Select by position
df.iloc[0:2]  # first 2 rows
df.loc[0, "name"]  # by label
```

Basics of NumPy, Pandas, and Matplotlib - Pandas

```
import pandas as pd

# Sample data
data = {
    "name": ["Alice", "Bob", "Charlie",
"David", "Eve"],
    "category": ["A", "B", "A", "B",
"A"],
    "age": [25, 32, 18, 45, 29],
    "salary": [50000, 60000, 45000,
70000, 55000]
}
```

```
# Group by category and get mean age
df.groupby("category")["age"].mean()
# category
# A    24.0
# B    38.5

# Multiple statistics on the same column
df.groupby("category")["age"].agg(["mean"
, "min", "max", "count"])
#           mean  min  max  count
# category
# A         24.0   18   29      3
# B         38.5   32   45      2
```

Basics of NumPy, Pandas, and Matplotlib - Matplotlib

→ Visualization Basics

Matplotlib is the standard plotting library in Python, enabling line plots, scatter plots, histograms, etc.

Key points:

- Highly flexible; foundation for other plotting libraries.
- Great for fast exploratory data visualization.
- Integrates seamlessly with NumPy and Pandas.

Basics of NumPy, Pandas, and Matplotlib - Matplotlib

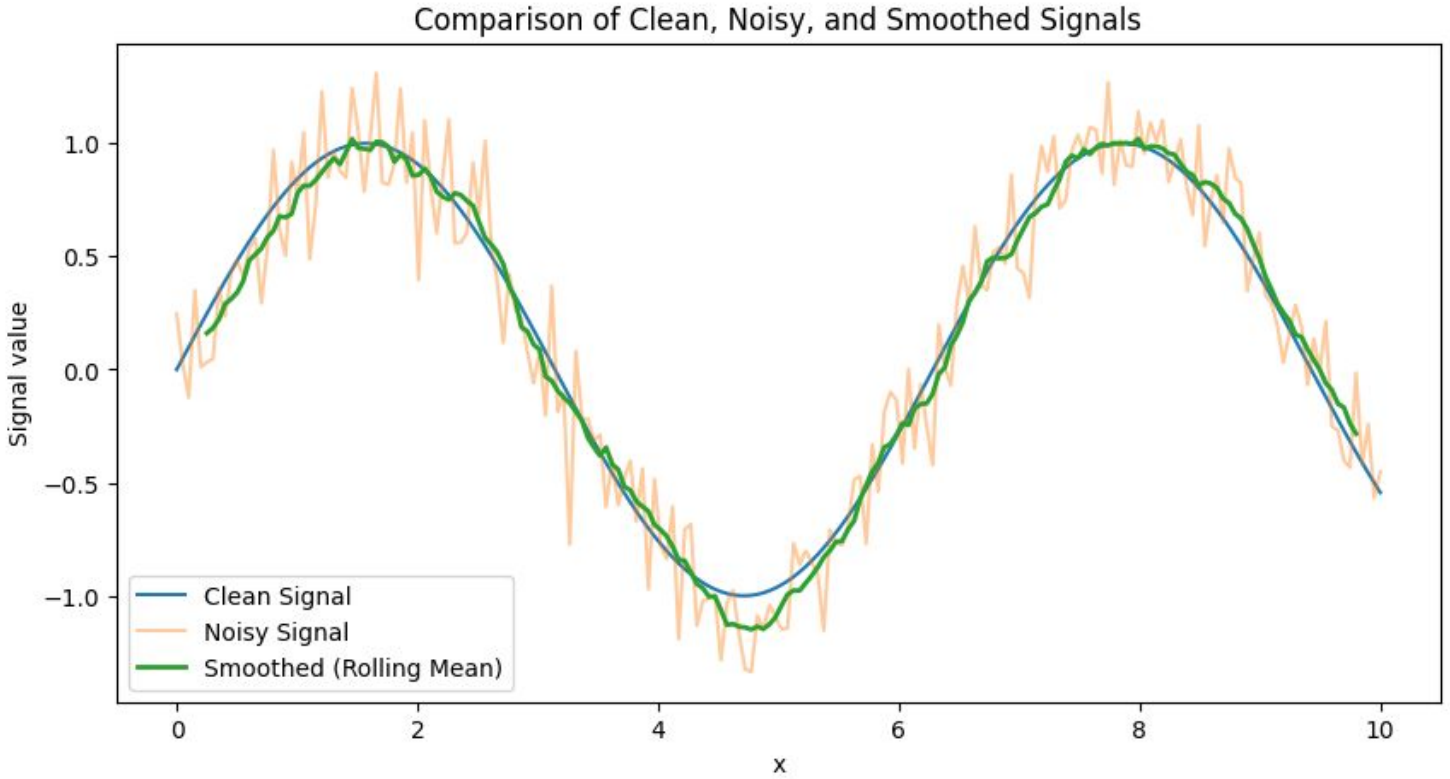
```
import matplotlib.pyplot as plt
import numpy as np

# Example data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Line plot
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.title("Sine Wave")
plt.show()
```

```
# Scatter plot
plt.scatter(x, y)
plt.show()

# Histogram
plt.hist(np.random.randn(1000))
plt.show()
```

Basics of NumPy, Pandas, and Matplotlib - Example

```
# Build DataFrame
df = pd.DataFrame({
      "x": x,
      "clean_signal": y1,
      "noisy_signal": y2
})

# Smooth noisy signal using rolling mean
df["smoothed_signal"] = df["noisy_signal"].rolling(window=10, center=True).mean()

plt.figure(figsize=(10, 5))

plt.plot(df["x"], df["clean_signal"], label="Clean Signal")
plt.plot(df["x"], df["noisy_signal"], alpha=0.4, label="Noisy Signal")
plt.plot(df["x"], df["smoothed_signal"], linewidth=2, label="Smoothed (Rolling Mean)")
```

Basics of NumPy, Pandas, and Matplotlib - Example



Agenda

- Object-Oriented Programming (classes, inheritance,...)
- Handling JSON, CSV, and SQLite
- Basics of NumPy, Pandas, and Matplotlib
- Virtual environments and project layout (src/, tests/, setup.py or pyproject.toml)
- Working with APIs and REST endpoints

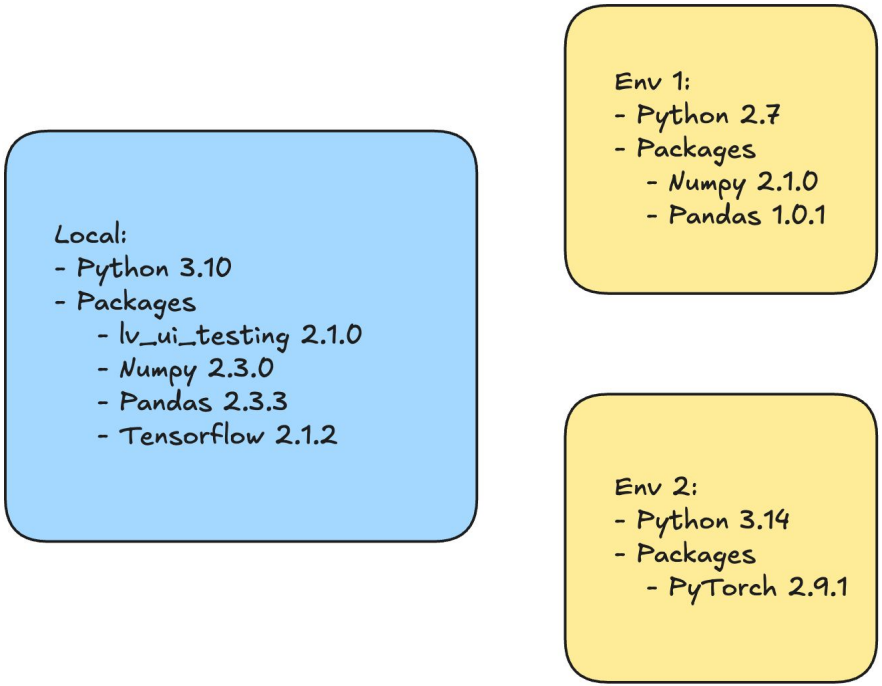
Virtual Environments

→ A virtual env is a self-contained Python setup for one project.

✅ It's a good practice to create one venv per project.

A virtual environment is a self-contained Python workspace that includes its own interpreter, libraries, and dependencies, kept completely separate from the system-wide Python installation.

more info:
https://docs.python.org/3/library/venv.html



Project layout - Application

As mentionned on [wiki.python.org:]

The topic of how to structure your project invokes **lots of opinions**.

```
my_repository/
├── README.md
├── requirements.txt
├── .gitignore
├── src/
│   └── my_project/
│       ├── __init__.py.    # Required to make a directory a package so that your scripts can import it.
│       ├── core.py
│       ├── _utils.py
│       ├── config/         # For YAML/JSON/TOML configs
│       │   └── default.yaml
│       └── features/       # Submodules
│           ├── __init__.py
│           ├── processor.py
│           └── sub_feature/
│               ├── __init__.py
│               └── helper.py
├── tests/
│   ├── test_core.py
│   ├── test_features.py
│   └── conftest.py
├── scripts/                # Executable scripts or utilities
│   ├── train_model.py
│   └── run_server.py
├── data/                   # Local data, dumped models, static files
│   └── example.csv
├── notebooks/              # Jupyter notebooks (if used)
│   └── exploration.ipynb
└── docs/                   # Documentation
    └── architecture.md
```

Project layout - Package

This section is based on [Packaging Python Projects tutorial]

```
my_repository/
├── README.md
├── requirements.txt
├── setup.py
├── src/
│   └── my_package/
│       ├── pyproject.toml    # Only if you want to share it on PyPI
│       ├── __init__.py
│       ├── core.py
│       ├── _utils.py
│       └── my_sub_package/
│           ├── __init__.py
│           ├── core.py
│           └── my_sub_sub_package/
│               ├── __init__.py
│               └── core.py
└── tests/
    ├── tests.py
    └── conftest.py
```

Agenda

- Object-Oriented Programming (classes, inheritance,...)
- Handling JSON, CSV, and SQLite
- Basics of NumPy, Pandas, and Matplotlib
- Virtual environments and project layout (src/, tests/, setup.py or pyproject.toml)
- Working with APIs and REST endpoints

Working with APIs and REST endpoints

API means **Application Programming Interface**. It's an "interface" that allows applications or servers to **communicate** between each others.

REST means **Representational State Transfer**. REST has been employed throughout the software industry to create **stateless, reliable, web-based applications**. REST APIs are one of the common API that are available, as many web app/services rely on them.

Working with APIs and REST endpoints

```
import requests

response = requests.get("https://api.example.com/data")

if response.status_code == 200:
    print("Success:", response.json())
else:
    print("Error:", response.status_code)
```