# AI AND DATA SCIENCE

## Session 2

**Date:** 3-12-2025

**Instructor:** Ahmed Diab

# Outlines

- Python Virtual environment (venv)
- Python oop
- Models / package

## Recap

- In the previous week we take overview about ml, ai, data science
- python basics (input/output, data types/variables, if-else conditions, for/while loop, files/folder, error handling, list, dict, set, tuple, function).

# Virtual Environment?

- A virtual environment (venv) is an isolated workspace where you can install and manage Python packages for a specific project without affecting the global Python installation or other projects.
- It’s like having a separate mini-Python world for each project

# Virtual Environment?

- Check Python installation
- Create a virtual environment

check version

`python --version` or `python3 --version`

Create a virtual environment `python -m venv myenv`

# Virtual Environment?

- Activating the Environment

On Windows: `myenv\Scripts\activate`

On Linux / macOS: `source myenv/bin/activate`

After activation, your terminal will show: `(myenv) C:\Users\Ahmed\project>`

# Virtual Environment?

- To check installed packages:
- saving the Environment (requirements.txt)

Installing Packages `pip install pandas numpy matplotlib`

To check installed packages: `pip list`

Saving at requirements.txt `pip freeze > requirements.txt`

# Virtual Environment?

- Recreate Environment on Another Computer

How to Clonw project with venv

```
python -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

# Practical Time

ML INTRO

# Activity Time

# OOP

- OOP stands for Object-Oriented Programming.
- Python is an object-oriented language, allowing you to structure your code using classes and objects for better organization and reusability.

# OOP

Advantages of OOP

- Provides a clear structure to programs
- Makes code easier to maintain, reuse, and debug
- Helps keep your code DRY (Don't Repeat Yourself)
- Allows you to build reusable applications with less code

# Class in Python

- A class is a blueprint or template for creating objects (real-world entities).
- Objects represent specific examples of the class.
- Think of a class as a design of a car, and each car you make from it (with specific color, speed, etc.) is an object.

# Class in Python

- Creating a Class
- Create Opject

```
class Car:
    # class body
    pass


my_car = Car()
```

# Class in Python

- The __init__() Constructor Method
- The __init__() method automatically runs when a new object is created.
- It’s used to initialize the object’s data (called attributes).

```python
class Car:
    def __init__(self, brand, color, speed):
        self.brand = brand
        self.color = color
        self.speed = speed

my_car = Car("BMW", "Black", 200)
print(my_car.brand)
```

```
BMW
```

# Instance Attributes vs Class Attributes

- Instance attributes belong to each object.
- Class attributes are shared by all objects.

```python
class Car:
    wheels = 4

    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

c1 = Car("BMW", "Red")
c2 = Car("Audi", "Black")

print(c1.wheels)
print(c2.wheels)
```

```
4
4
```

# Instance Attributes vs Class Attributes

- Functions Inside Classes
- A method is a function defined inside a class that operates on the object’s data.

```python
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def accelerate(self):
        self.speed += 10
        print(f"{self.brand} is now going {self.speed} km/h")

    def brake(self):
        self.speed -= 10
        print(f"{self.brand} slowed to {self.speed} km/h")

c = Car("Tesla", 100)
c.accelerate()
c.brake()
```

```
Tesla is now going 110 km/h
Tesla slowed to 100 km/h
```

# Inheritance

- Inheritance allows one class (child) to use the attributes and methods of another (parent).

Why Use Inheritance?

- Reuse code instead of writing it again
- Organize code better (hierarchical structure)
- Extend or customize existing behavior

```python
class Parent:
    def speak(self):
        print("I am the parent.")

class Child(Parent):
    pass

c = Child()
c.speak()
```

```
I am the parent.
```

```python
class Parent:
    def speak(self):
        print("I am the parent.")

class Child(Parent):
    def play(self):
        print("I love playing!")

c = Child()
c.speak()
c.play()
```

```
I am the parent.
I love playing!
```

# Overriding & super()

- A child can override (replace) a parent method with its own version.

```python
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):
    def sound(self):
        print("Woof! Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

dog = Dog()
cat = Cat()
dog.sound()
cat.sound()
```

```
Woof! Woof!
Meow!
```

# Overriding & super()

- Sometimes you want to extend the parent method, not replace it entirely.
- You can use super() to call the parent version.

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

    def info(self):
        print(f"Name: {self.name}")

class Student(Person):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

    def info(self):
        super().info()
        print(f"Grade: {self.grade}")

s = Student("Ahmed", "A+")
s.info()
```

```
1]

Name: Ahmed
Grade: A+
```

# Single & Multilevel & Multiple Inheritance Inheritance

Single Inheritance

```
class Parent:
    def talk(self):
        print("Parent talking")

class Child(Parent):
    def walk(self):
        print("Child walking")

c = Child()
c.talk()
c.walk()
```

```
Parent talking
Child walking
```

Multilevel Inheritance

```
class GrandParent:
    def greet(self):
        print("Hello from GrandParent")

class Parent(GrandParent):
    def talk(self):
        print("Hello from Parent")

class Child(Parent):
    def play(self):
        print("Hello from Child")

c = Child()
c.greet()
c.talk()
c.play()
```

```
Hello from GrandParent
Hello from Parent
Hello from Child
```

```
class Father:
    def skill(self):
        print("Can drive")

class Mother:
    def hobby(self):
        print("Can cook")

class Child(Father, Mother):
    def talent(self):
        print("Can code")

c = Child()
c.skill()
c.hobby()
c.talent()
```

```
Can drive
Can cook
Can code
```

# Method Resolution Order (MRO)

- When multiple parents exist, Python follows an MRO (Method Resolution Order) to decide which method to call first.

Method Resolution Order (MRO)

```
print(Child.__mro__)
```

```
(<class '__main__.Child'>, <class '__main__.Father'>, <class '__main__.Mother'>, <class 'object'>)
```

```
class A:
    def show(self): print("A")

class B(A):
    def show(self): print("B")

class C(A):
    def show(self): print("C")

class D(B, C):
    pass

d = D()
d.show()
print(D.__mro__)
```

```
B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

# Polymorphism

- In Python OOP, it allows the same function or method name to behave differently depending on the object or data type it’s used with.

Why Polymorphism?

- Makes code more flexible and extendable
- Reduces repetition
- Allows different objects to be treated uniformly

# Built-in Functions & (Method Overriding

- The same function len() works for different data types. Each type defines its own behavior for len() internally.
- We define the same method name in different classes, but each class has its own version.

```
print(len("Zag-Eng"))
print(len([1, 2, 3]))
print(len({1: "a", 2: "b"}))
```

```
7
3
2
```

```
class Bird:
    def make_sound(self):
        print("Some generic bird sound")

class Sparrow(Bird):
    def make_sound(self):
        print("Chirp Chirp")

class Eagle(Bird):
    def make_sound(self):
        print("Screech!")

for bird in [Sparrow(), Eagle()]:
    bird.make_sound()
```

```
Chirp Chirp
Screech!
```

# Method Overloading

- In some languages (like Java or C++), you can create multiple methods with the same name, but different parameter counts or types.
- Python doesn’t support method overloading directly, but you can simulate it using default or variable arguments.

```python
class MathOperation:
    def add(self, a=0, b=0, c=0):
        return a + b + c

m = MathOperation()
print(m.add(2, 3))
print(m.add(2, 3, 4))
```

```
5
9
```

# Operator Overloading

- Polymorphism also appears in operators like +, >, etc.
- Python lets you redefine what these operators mean for your objects.

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2

print(v3)
```

```
(6, 8)
```

# Abstract Classes and Polymorphism

- To force all subclasses to implement certain methods, Python provides abstract classes using abc module.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r * self.r

class Square(Shape):
    def __init__(self, s):
        self.s = s

    def area(self):
        return self.s * self.s

shapes = [Circle(5), Square(4)]
for shape in shapes:
    print(shape.area())
```

```
78.5
16
```

# Python Encapsulation

- Encapsulation is one of the four pillars of Object-Oriented Programming (OOP)
- (along with Inheritance, Polymorphism, and Abstraction).
- It means:
- Bundling data (attributes) and methods (functions) that work on that data into a single unit the class
- and restricting direct access to the internal details of that object.

## Real-Life Example

- Think of a car
- You can start(), stop(), accelerate() — public actions.
- But you can’t directly control the engine spark plugs or fuel injectors those are internal (private).
- Same idea in programming:
- You hide internal data and expose only what’s necessary.

# How Encapsulation Protects Data

- We don’t expose variables directly.
- We create getter and setter methods to control access.
- The private variable __balance can’t be changed directly

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Invalid amount!")

account = BankAccount(1000)
print(account.get_balance())
account.deposit(500)
print(account.get_balance())
# print(account.__balance)
```

```
1000
1500
```

# Private Methods

- You can also hide methods (not just variables) by using double underscores

Private Methods

```
class Car:
    def __init__(self):
        self.__engine_status = "OFF"

    def start(self):
        self.__check_engine()
        self.__engine_status = "ON"
        print("Car started")

    def __check_engine(self):
        print("Engine check complete ")

c = Car()
c.start()
# c.__check_engine()
```

```
Engine check complete
Car started
```

# Protected Members

- Protected members (with one _) can be accessed by child classes,
- but it’s a convention that they’re for internal use only.
- It still works, but developers understand _money shouldn’t be used outside the family of classes.

Protected Members

```
[31]
class Parent:
    def __init__(self):
        self._money = 1000

class Child(Parent):
    def spend(self):
        print(f"Spending {self._money}")

c = Child()
c.spend()
```

```
Spending 1000
```

# Practical Time

## models

A module in Python is simply a file containing Python code (with a .py extension).

It can include:

- Variables - Functions - Classes - Executable code
- Modules help organize and reuse code instead of writing everything in one file.

# models

Let’s create a module named math_tools.py

math_tools.py

E: > Main_Folder > Zag_Eng > AI_25-26 > Content > Week2_python_C

```python
PI = 3.14159

def area_circle(radius):
    return PI * radius * radius

def perimeter_circle(radius):
    return 2 * PI * radius
```

## models

Then, use it in another file

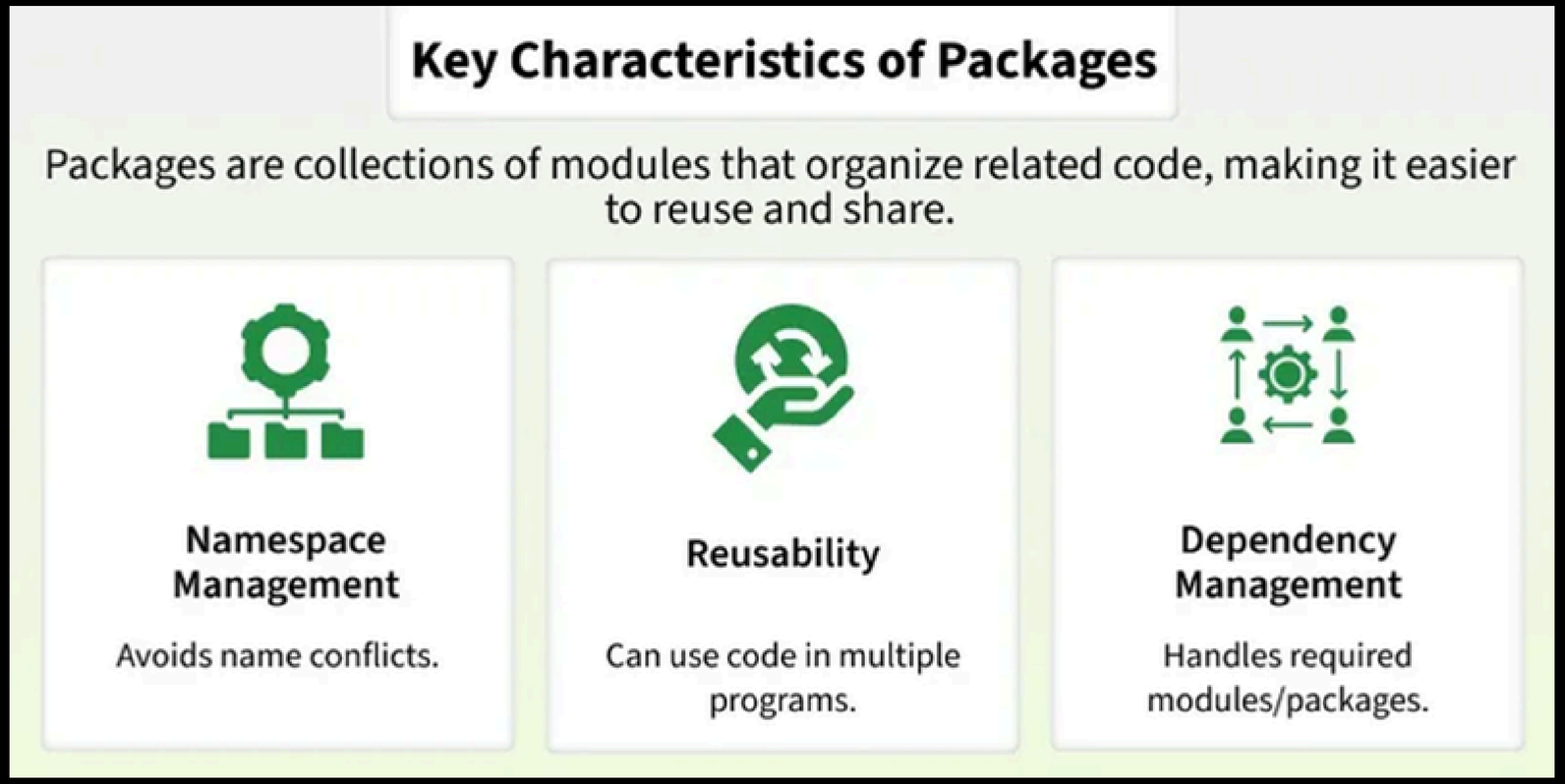
```
import math_tools

print(math_tools.area_circle(5))
print(math_tools.perimeter_circle(5))
```

Every .py file can act as a module.
You can import specific items too:

```
from math_tools import area_circle
print(area_circle(3))
```

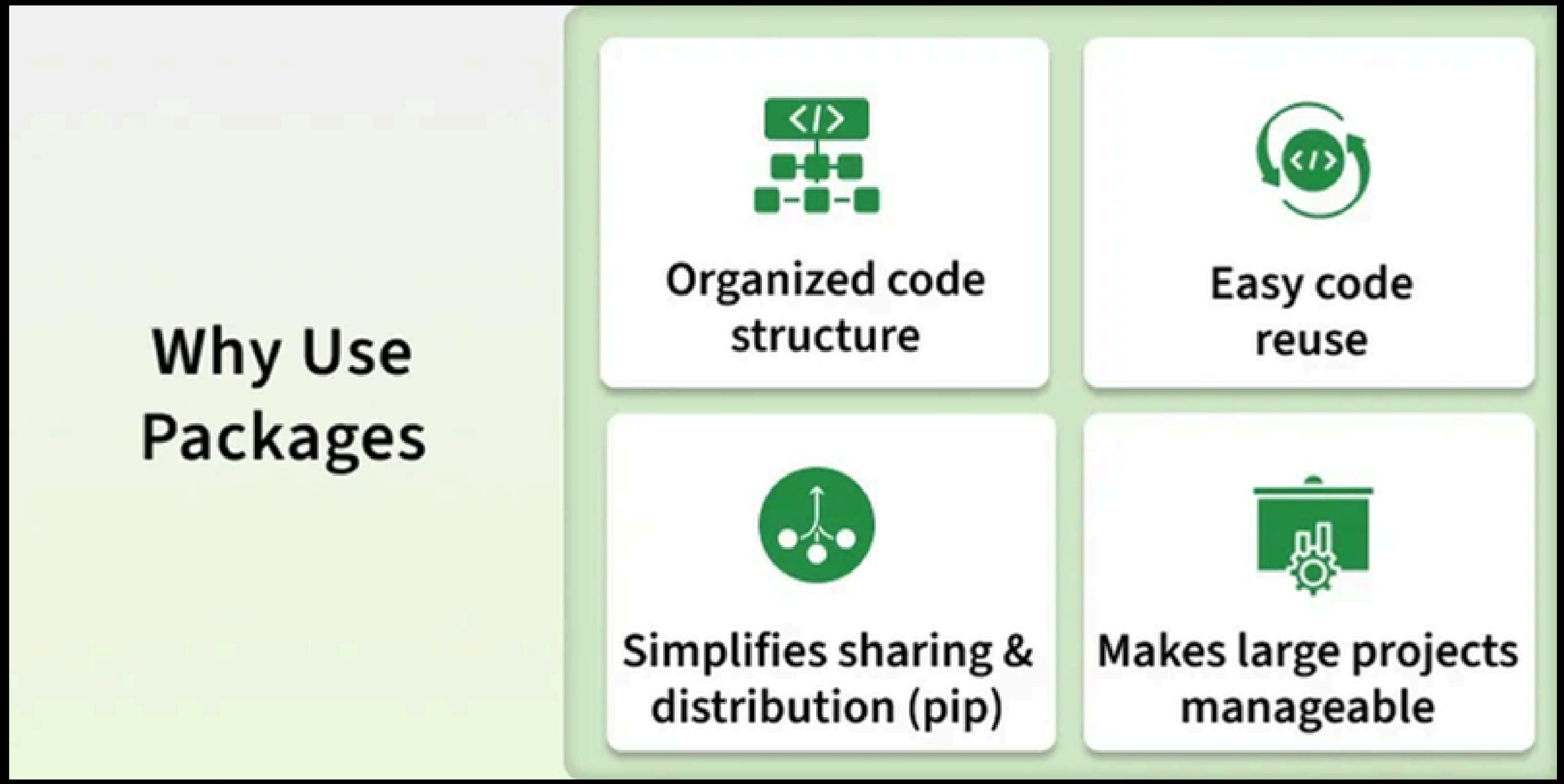
# Packages

A package is a directory (folder) that contains multiple modules and an optional special file called __init__.py



# Packages

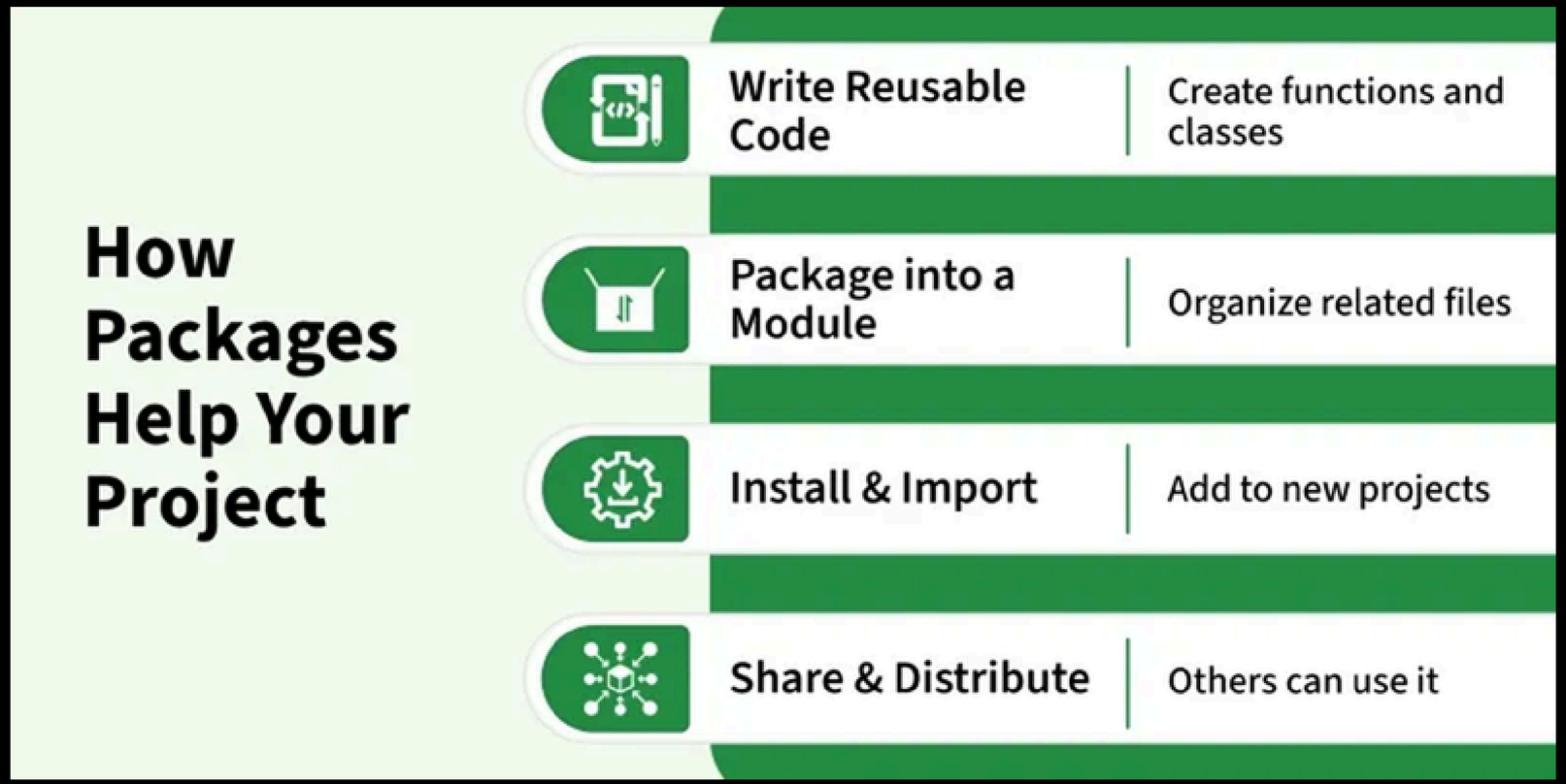
This file tells Python that this folder should be treated as a package



# Packages

Purpose:

To organize large projects into structured, hierarchical code groups.



# MODELS, PACKAGES, AND SUB-PACKAGES

## Packages

example

```
myproject/
|
├── geometry/
|   ├── __init__.py
|   ├── circle.py
|   └── rectangle.py
|
└── main.py
```

circle.py

```python
def area(radius):
    return 3.14 * radius * radius
```

rectangle.py

```python
def area(length, width):
    return length * width
```

# Packages

main.py

```
from geometry import circle, rectangle


print(circle.area(5))
print(rectangle.area(4, 6))
```

- The folder geometry is now a package.
- The modules inside (circle.py, rectangle.py) are imported through the package.

## Sub-Package

- A sub-package is a package inside another package.
- It helps organize complex projects into smaller logical components.
- You'll usually see sub-packages in machine learning, web apps, or engineering toolkits where each package handles a separate function.

## Sub-Package

- Every folder that acts as a package or sub-package must have a file named __init__.py.
- Even if it’s empty, Python needs it to know “this is a package”.

example

```
zageng_project/
|
├── models/
|   ├── __init__.py
|   ├── regression/
|   |   ├── __init__.py
|   |   ├── linear_regression.py
|   |   └── ridge_regression.py
|   |
|   └── classification/
|       ├── __init__.py
|       ├── svm.py
|       └── decision_tree.py
|
└── main.py
```

# Sub-Package

- models/regression/linear_regression.py

```python
def train_linear(X, y):
    print("Training Linear Regression model...")
    slope = sum(X) / len(X)
    intercept = 0.5
    return {"model": "LinearRegression", "slope": slope, "intercept": intercept}
```

- models/regression/ridge_regression.py

```python
def train_ridge(X, y, alpha=0.1):
    print(f"Training Ridge Regression model with alpha={alpha}")
    weight = sum(X) / (len(X) + alpha)
    return {"model": "RidgeRegression", "weight": weight, "alpha": alpha}
```

## Sub-Package

- models/classification/svm.py

```python
def train_svm(X, y):
    print("Training SVM classifier...")
    return {"model": "SVM", "support_vectors": len(X) // 2}
```

- models/classification/decision_tree.py

```python
def train_tree(X, y):
    print("Training Decision Tree classifier...")
    return {"model": "DecisionTree", "depth": 3}
```

# MODELS, PACKAGES, AND SUB-PACKAGES

## Sub-Package

- main.py

```python
from models.regression.linear_regression import train_linear
from models.regression.ridge_regression import train_ridge
from models.classification.svm import train_svm
from models.classification.decision_tree import train_tree

X = [1, 2, 3, 4]
y = [2, 4, 6, 8]

linear_model = train_linear(X, y)
ridge_model = train_ridge(X, y, alpha=0.5)

svm_model = train_svm(X, y)
tree_model = train_tree(X, y)

print("\n--- Model Summary ---")
print(linear_model)
print(ridge_model)
print(svm_model)
print(tree_model)
```

# Sub-Package

- output

```
PS C:\Users\pc> & C:/Users/pc/AppData/Local/Programs/Python/Python311/python
Main package 'models' initialized
Sub-package 'regression' initialized
Training Linear Regression model...
Training Ridge Regression model with alpha=0.5
Training SVM classifier...
Training Decision Tree classifier...

--- Model Summary ---
{'model': 'LinearRegression', 'slope': 2.5, 'intercept': 0.5}
{'model': 'RidgeRegression', 'weight': 2.222222222222223, 'alpha': 0.5}
{'model': 'SVM', 'support_vectors': 2}
{'model': 'DecisionTree', 'depth': 3}
```

# Problems

https://www.hackerrank.com/challenges/py-if-else/problem?isFullScreen=true

https://www.hackerrank.com/challenges/python-arithmetic-operators/problem?isFullScreen=true

https://www.hackerrank.com/challenges/python-division/problem?isFullScreen=true

https://www.hackerrank.com/challenges/find-a-string/problem?isFullScreen=true

https://www.hackerrank.com/challenges/text-alignment/problem?isFullScreen=true

# *references*


- https://www.w3schools.com/python/python_oop.asp.
- https://www.geeksforgeeks.org/python/python-oops-concepts/.
- https://www.youtube.com/watch?v=mlbe7Vxr7yA (from 6:36) .
- https://www.youtube.com/watch?v=RSZEZ3WJn18 part 1/2
- https://www.youtube.com/watch?v=z7g9gCYYLiU part 1/2/3

# INTRO TO PYTHON

دمتم سالمين