**LECTURE-13 (Saturday 26-July-2025)**

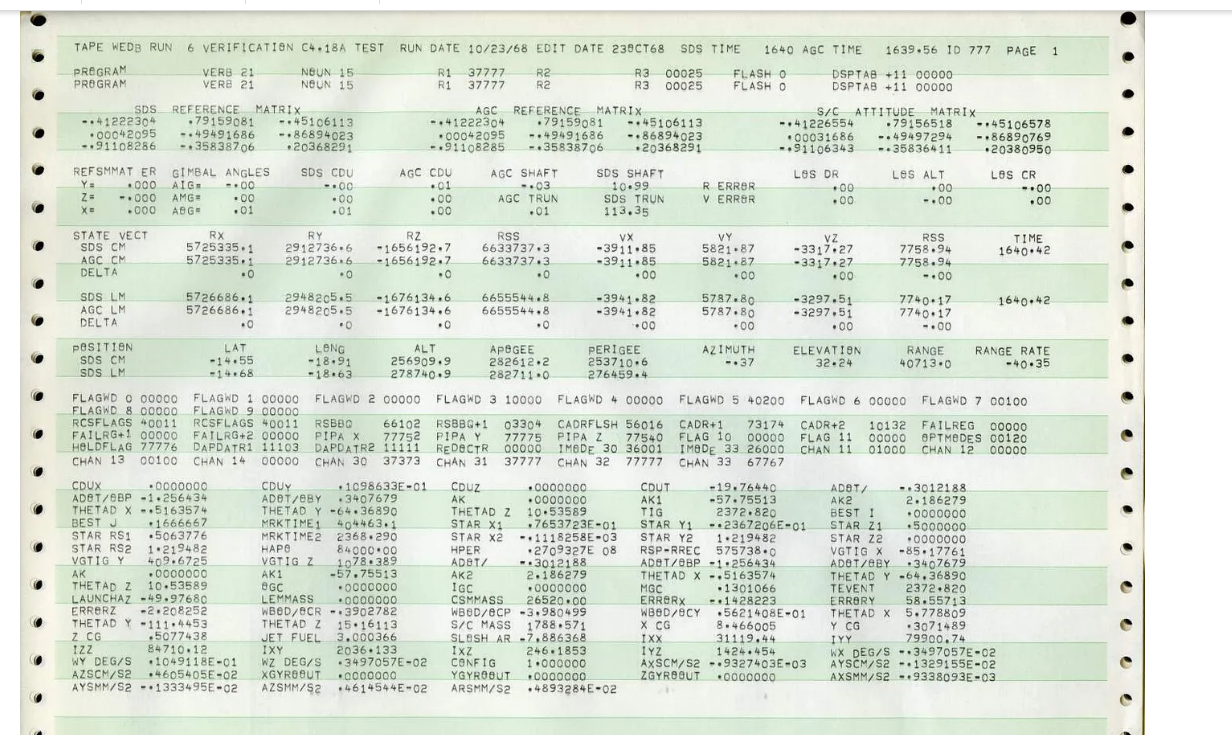
* TEST
* We have to review step 12 and 13 ourselves from <https://github.com/panaversity/learn-modern-ai-python/tree/main/00_python_colab>
* Step 12: <https://github.com/panaversity/learn-modern-ai-python/tree/main/00_python_colab/12_traditional_oop_part_1>



* <https://colab.research.google.com/drive/10kCAqWJarynBmsvZwD9OAo9SYDq6Bwvr?usp=sharing>
* Below picture have computer scientist “Margaret Hamilton” with her NASA’s Apollo guidance software, which she and her team developed at MIT.



* Below is the picture of “Special Assembly Language” code, which MIT Programmers created. This complete code consists of 145,000 lines that successfully landed “Apollo 11” on moon.



* Key Principles of OOP

1. Encapsulation: Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit (a class). It also restricts direct access to some of an object’s components/members by using access modifiers, which is a way of preventing unintended interference.

Example:

class Car:

    def \_\_init\_\_(self, color, speed):

        self.color = color

        self.\_\_speed = speed

    def accelerate(self):

        self.\_\_speed += 10

    def get\_speed(self):

        return self.\_\_speed

car = Car("red", 0)

print("Current speed: ", car.get\_speed())

for i in range(10):

    car.accelerate()

print("Speed after acceleration: ", car.get\_speed())

Output:

Current speed: 0

Speed after acceleration: 100

Here,

“\_\_init\_\_” method is the constructor of class “Car”.

“color” is a public member/field/variable of class “Car”.

“\_\_speed” is a private member/field/variable of class “Car”.

“accelerate” method: It increases the value of private member “speed” by 10.

“get\_speed” method provides a safe way to read value of private member “speed”.

car = Car("red", 0): creates “car” object/instance of class “Car” with

color = "red"

speed = 0

Therefore, when we called “get\_speed” method

print("Current speed: ", car.get\_speed())

then output is:

Current speed: 0

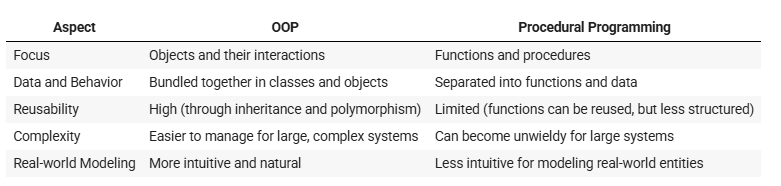
Then we run “for” loop 10 times and called “accelerate” method each time which increases value of private member “speed” by 10.

Therefore, when we again called “get\_speed” method then output is:

Speed after acceleration: 100

1. Abstraction: It means hiding complex implementation details and exposing only the necessary features of an object. Explained below.
2. Inheritance: Explained below.
3. Polymorphism: Explained below.

* OOP vs Procedural Programming:



* Default Constructors: If we do not define an “\_\_init\_\_” method in your class, Python provides a default constructor that does nothing. However, you can define an “\_\_init\_\_” method without parameters to act as a default constructor.

Example:

class Dog:

    def \_\_init\_\_(self):

        self.name = "Unknown"

        self.age = 0

dog1 = Dog()

print(dog1.name)

print(dog1.age)

Output:

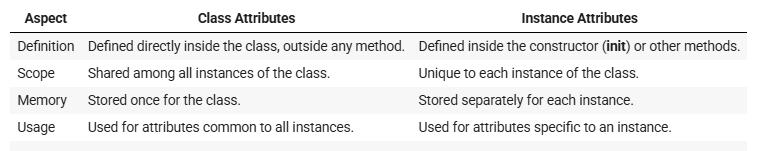
Unknown

0

Here,

dog1 = Dog(): “dog1” is instance/object of class “Dog” created with a default constructor “Dog()” means without parameter.

* Class Attributes and Instance Attributes:



* Accessing and Modifying Attributes

1. Class attributes can be accessed using the class name or an instance of the class.
2. Instance attributes can only be accessed using an instance of the class.
3. Modifying a class attribute through an instance creates a new instance attribute with the same name, shadowing the class attribute.

* “\_\_dict\_\_” attribute: Every class and instance in Python has a “\_\_dict\_\_” attribute, which is a dictionary that stores the attributes and their values.

Example:

class Dog:

    species = "Canis familiaris"

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

    def display(self):

        print(f"{self.name} is {self.age} years old and is a {self.species}.")

dog1 = Dog("Buddy", 5)

dog2 = Dog("Max", 3)

print(f"Dog species: {Dog.species}")

print(f"Buddy's species: {dog1.species}")

print(f"Max's species: {dog2.species}")

print(f"Buddy's name: {dog1.name}")

print(f"Max's age: {dog2.age}")

Dog.species = "Canis lupus"

print(f"Buddy's species after modification: {dog1.species}")

print(f"Max's species after modification: {dog2.species}")

dog1.species = "Canis aureus"

print(f"Buddy's species after shadowing: {dog1.species}")

print(f"Max's species remains: {dog2.species}")

print(f"Dog class attributes: {Dog.\_\_dict\_\_}")

print(f"dog1 instance attributes: {dog1.\_\_dict\_\_}")

print(f"dog2 instance attributes: {dog2.\_\_dict\_\_}")

Output:

Dog species: Canis familiaris

Buddy's species: Canis familiaris

Max's species: Canis familiaris

Buddy's name: Buddy

Max's age: 3

Buddy's species after modification: Canis lupus

Max's species after modification: Canis lupus

Buddy's species after shadowing: Canis aureus

Max's species remains: Canis lupus

Dog class attributes: {'\_\_module\_\_': '\_\_main\_\_', 'species': 'Canis lupus', '\_\_init\_\_': <function Dog.\_\_init\_\_ at 0x78e6d0eb6660>, 'display': <function Dog.display at 0x78e6d0eb7b00>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Dog' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Dog' objects>, '\_\_doc\_\_': None}

dog1 instance attributes: {'name': 'Buddy', 'age': 5, 'species': 'Canis aureus'}

dog2 instance attributes: {'name': 'Max', 'age': 3}

Here,

“species” is a class attribute (shared by all instances) of class “Dog”.

“name” and “age” are instance attributes of class “Dog”.

Value “Canis familiaris” for “Dog.species”, “dog1.species” and “dog2.species” is same because “species” is a class attribute.

While value of instance attributes are as:

dog1.name = Buddy

dog1.age = 5

dog2.name = Max

dog2.age = 3

When we changed value of “species” to "Canis lupus" then value of “Dog.species”, “dog1.species” and “dog2.species” is changed to "Canis lupus" because “species” is a class attribute.

“Dog.\_\_dict\_\_” returns attributes of “Dog” class.

“dog1.\_\_dict\_\_” returns attributes of “dog1” instance/object.

“dog2.\_\_dict\_\_” returns attributes of “dog2” instance/object.

We found that “dog2.\_\_dict\_\_” does not show “species” attribute while “dog1.\_\_dict\_\_” does because “Dog.species” is a class attribute, when “dog1.species = "Canis aureus"” is executed, a new “species” attribute is created specifically for the “dog1” instance/object, shadowing the class attribute. This is why “'species': 'Canis aureus'” appears in “dog1.\_\_dict\_\_”.

* Getter and Setter Methods: To access or modify private or protected attributes, “getter” and “setter methods are used. These methods provide controlled access to the data, ensuring that any changes validated or processed appropriately.

Example:

class BankAccount:

    def \_\_init\_\_(self, account\_holder, balance):

        self.account\_holder = account\_holder

        self.\_balance = balance

        self.\_\_pin = "1234"

    def get\_balance(self):

        return self.\_balance

    def set\_balance(self, amount):

        if amount >= 0:

            self.\_balance = amount

            print(f"Balance updated to {self.\_balance}.")

        else:

            print("Invalid amount. Balance cannot be negative.")

    def get\_pin(self):

        return "Access denied. PIN is private."

    def display(self):

        print(f"Account Holder: {self.account\_holder}")

        print(f"Balance: {self.\_balance}")

account = BankAccount("Alice", 1000)

print(f"Account Holder: {account.account\_holder}")

print(f"Balance: {account.\_balance}")

print(account.get\_pin())

print(f"Initial Balance: {account.get\_balance()}")

account.set\_balance(1500)

account.set\_balance(-500)

account.display()

Output:

Account Holder: Alice

Balance: 1000

Access denied. PIN is private.

Initial Balance: 1000

Balance updated to 1500.

Invalid amount. Balance cannot be negative.

Account Holder: Alice

Balance: 1500

Here,

“account\_holder” is public attribute of class “BankAccount”.

“\_balance” is protected attribute of class “BankAccount”.

“\_\_pin” is private attribute of class “BankAccount”.

“get\_balance” is getter method for “\_balance”.

“set\_balance” is setter method for “\_balance”.

account.account\_holder: accessed the public attribute “account\_holder” of class “BankAccount” from object “account”.

account.\_balance: accessed the protected attribute “\_balance” of class “BankAccount” from object “account”. It is not recommended, but possible.

If we update code and try to use code:

print(account.\_\_pin)

Output:

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

[/tmp/ipython-input-3456966630.py](https://localhost:8080/) in <cell line: 0>()

**28** print(f"Balance: {account.\_balance}")

**29**

---> 30 print(account.\_\_pin)

**31**

**32** print(account.get\_pin())

AttributeError: 'BankAccount' object has no attribute '\_\_pin'

Here,

We are trying to access private attribute from object due to which getting error but it is possible to access private attribute from object in Python through “mangling” approach but not recommended.

* “Object” class: In Python, “object” class is the base class of all data types. Every class in Python, whether built-in or user-defined, directly or indirectly inherits from “object” class. This means that all classes in Python are part of a single inheritance tree with object at the root.
* Key characteristics of the “object” class:

1. Base Class: “object” class is the root of the class hierarchy. If you define a class without specifying a base class, it implicitly inherits from object.
2. Default Methods: The “object” class provides default implementations for several important methods that are inherited by all other classes.
3. Universal Behavior: Since all classes inherit from “object” class, methods and attributes defined in “object” class are available to all objects in Python.

* Key Methods of the “object” Class: The “object” class defines several special methods (also called "dunder" methods because they are surrounded by double underscores, e.g., \_\_init\_\_). These methods are inherited by all other classes unless overridden.

1. Initialization and Representation
2. “\_\_init\_\_” (self): The default constructor. It does nothing but can be overridden in derived classes.
3. “\_\_new\_\_” (cls): The method responsible for creating a new instance of a class. It is rarely overridden unless you need custom instance creation behavior.
4. “\_\_repr\_\_” (self): Returns a string representation of the object, typically used for debugging. The default implementation returns something like <ClassName object at 0x...>.
5. “\_\_str\_\_” (self): Returns a string representation of the object, typically used for user-friendly output. The default implementation calls \_\_repr\_\_.
6. Comparison and Hashing
7. “\_\_eq\_\_” (self, other): Defines behavior for the == operator. The default implementation checks if two objects are the same instance (self is other).
8. “\_\_hash\_\_” (self): Returns a hash value for the object. The default implementation uses the object's memory address.
9. “\_\_ne\_\_” (self, other): Defines behavior for the != operator. The default implementation is the negation of \_\_eq\_\_.
10. Attribute Access
11. “\_\_getattribute\_\_” (self, name): Handles attribute access. It is called whenever an attribute is accessed (e.g., obj.attr).
12. “\_\_setattr\_\_” (self, name, value): Handles attribute assignment. It is called whenever an attribute is assigned (e.g., obj.attr = value).
13. “\_\_delattr\_\_” (self, name): Handles attribute deletion. It is called whenever an attribute is deleted (e.g., del obj.attr).
14. Class and Type Information
15. “\_\_class\_\_”: A special attribute that refers to the class of the object.
16. “\_\_dict\_\_”: A dictionary containing the object's attributes.
17. Other Methods
18. “\_\_dir\_\_” (self): Returns a list of valid attributes for the object. The default implementation includes all attributes in “\_\_dict\_\_” and those defined in the class.

* Exploring the object Class: We can use “dir” and “help” methods to explore the methods and attributes of the object class.

Example (dir):

print(dir(object))

Output:

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_']

Example (help):

print(help(object))

Output:

Help on class object in module builtins:

class object

| The base class of the class hierarchy.

|

| When called, it accepts no arguments and returns a new featureless

| instance that has no instance attributes and cannot be given any.

|

| Built-in subclasses:

| anext\_awaitable

| async\_generator

| async\_generator\_asend

| async\_generator\_athrow

| ... and 94 other subclasses

|

| Methods defined here:

|

| \_\_delattr\_\_(self, name, /)

| Implement delattr(self, name).

|

| \_\_dir\_\_(self, /)

| Default dir() implementation.

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_format\_\_(self, format\_spec, /)

| Default object formatter.

|

| Return str(self) if format\_spec is empty. Raise TypeError otherwise.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getstate\_\_(self, /)

| Helper for pickle.

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_hash\_\_(self, /)

| Return hash(self).

|

| \_\_init\_\_(self, /, \*args, \*\*kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_reduce\_\_(self, /)

| Helper for pickle.

|

| \_\_reduce\_ex\_\_(self, protocol, /)

| Helper for pickle.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_setattr\_\_(self, name, value, /)

| Implement setattr(self, name, value).

|

| \_\_sizeof\_\_(self, /)

| Size of object in memory, in bytes.

|

| \_\_str\_\_(self, /)

| Return str(self).

|

| ----------------------------------------------------------------------

| Class methods defined here:

|

| \_\_init\_subclass\_\_(...)

| This method is called when a class is subclassed.

|

| The default implementation does nothing. It may be

| overridden to extend subclasses.

|

| \_\_subclasshook\_\_(...)

| Abstract classes can override this to customize issubclass().

|

| This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_().

| It should return True, False or NotImplemented. If it returns

| NotImplemented, the normal algorithm is used. Otherwise, it

| overrides the normal algorithm (and the outcome is cached).

|

| ----------------------------------------------------------------------

| Static methods defined here:

|

| \_\_new\_\_(\*args, \*\*kwargs)

| Create and return a new object. See help(type) for accurate signature.

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_class\_\_ = <class 'type'>

| type(object) -> the object's type

| type(name, bases, dict, \*\*kwds) -> a new type

None

* Python's Unified Type System: It is one of its core design principles. It means that everything in Python is an object and all objects ultimately inherit from the object class. This includes:

1. Primitive types (e.g., integers, strings, floats)
2. Data structures (e.g., lists, dictionaries, sets)
3. Functions and classes
4. Modules and types themselves

* Key Aspects of the Unified Type System:

1. Everything is an Object: In Python, even basic types like integers (int), strings (str) and booleans (bool) are instances of classes. For example, the integer “5” is an instance of the “int” class and the string "hello" is an instance of the “str” class.

Example:

print(type(5))

print(type("hello"))

print(type(3.14))

Output:

<class 'int'>

<class 'str'>

<class 'float'>

Here,

“int”, “str” and “float” are classes but all classes inherit from object, these types are also objects.

1. 2. Consistent Behavior: Because all objects inherit from object, they share common methods and behaviors. For example, all objects can be converted to a string using the “str” method, and their type can be checked using the “type” function.

Example:

print(str(42))

print(str([1, 2, 3]))

Output:

42

[1, 2, 3]

1. 3. Dynamic Typing: Python's unified type system allows for dynamic typing means that the type of a variable is determined at runtime and can change. For example, a variable can hold an integer at one point and a string at another.

Example:

x = 42

print(x)

x = "hello"

print(x)

Output:

42

hello

1. Uniform Access to Attributes and Methods: All objects have attributes and methods that can be accessed in the same way, regardless of their type. For example, we can use the “dir” function to list all attributes and methods of an object.

Example:

print(dir(5))

print("\n")

print(dir("hello"))

Output:

['\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_bool\_\_', '\_\_ceil\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_float\_\_', '\_\_floor\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getnewargs\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_index\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_le\_\_', '\_\_lshift\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_round\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_', '\_\_xor\_\_', 'as\_integer\_ratio', 'bit\_count', 'bit\_length', 'conjugate', 'denominator', 'from\_bytes', 'imag', 'is\_integer', 'numerator', 'real', 'to\_bytes']

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

1. Inheritance from object: All classes whether built-in or user-defined, inherit from object directly or indirectly. This ensures that even custom classes have access to default methods like str, repr and eq.

Example:

class MyClass:

    pass

obj = MyClass()

print(isinstance(obj, object))

Output:

True

* Benefits of the Unified Type System:

1. Simplicity: The unified type system makes Python intuitive and easy to learn. We do not need to worry about different rules for different types.
2. Consistency: Since all objects behave similarly, you can apply the same concepts (e.g., methods, attributes, inheritance) across all types.
3. Extensibility: You can create custom classes that behave like built-in types, thanks to the unified object model.
4. Polymorphism: The unified type system enables polymorphism, where objects of different types can be used interchangeably if they share the same interface.

* Special (Magic/Dunder) Methods: Special methods are Python's secret commands that make objects work with regular operations (like “+” or “print”) using a Toy Store analogy.
* Essential Special Methods:

1. “\_\_init\_\_“ - The Setup Crew: Prepares new objects like unboxing toys:

Example:

class ToyShelf:

    def \_\_init\_\_(self):

        self.toys = []

shelf = ToyShelf()

Output:

Empty

Here,

shelf = ToyShelf(): “\_\_init\_\_” (crew) setup empty shelf.

1. “\_\_str\_\_” vs “\_\_repr\_\_” - Toy Labels:

“\_\_str\_\_”: Friendly display (for customers)

“\_\_repr\_\_”: Exact specs (for warehouse staff)

Example:

class Puzzle:

    def \_\_init\_\_(self, pieces):

        self.pieces = pieces

    def \_\_str\_\_(self):

        return f"Puzzle with {self.pieces} fun pieces!"

    def \_\_repr\_\_(self):

        return f"Puzzle({self.pieces})"

puzzle = Puzzle(100)

print(str(puzzle))

print(repr(puzzle))

Output:

Puzzle with 100 fun pieces!

Puzzle(100)

1. “\_\_len\_\_” and “\_\_getitem\_\_” - Inventory Control: Make objects work with len() and [].

Example:

class ToyBox:

    def \_\_init\_\_(self):

        self.toys = ["teddy", "car", "blocks"]

    def \_\_len\_\_(self):

        return len(self.toys)

    def \_\_getitem\_\_(self, index):

        return self.toys[index]

box = ToyBox()

print(len(box))

print(box[1])

Output:

3

car

1. Comparison Methods (“\_\_eq\_\_”, “\_\_lt\_\_”) - Toy Sorting: enable “==” and “<” comparisons:

Example:

class Doll:

    def \_\_init\_\_(self, height):

        self.height = height

    def \_\_eq\_\_(self, other):

        return self.height == other.height

    def \_\_lt\_\_(self, other):

        return self.height < other.height

doll\_A = Doll(30)

doll\_B = Doll(25)

print("doll\_A > doll\_B  =",doll\_A > doll\_B)

doll\_C = Doll(30)

print("doll\_A == doll\_C = ",doll\_A == doll\_C)

Output:

doll\_A > doll\_B = True

doll\_A == doll\_C = True

* Complete Example: Magic Toy Catalog

class ToyCatalog:

    def \_\_init\_\_(self):

        self.toys = {}

    def \_\_setitem\_\_(self, name, price):

        self.toys[name] = price

    def \_\_getitem\_\_(self, name):

        return self.toys.get(name, "Not in stock")

    def \_\_contains\_\_(self, name):

        return name in self.toys

    def \_\_len\_\_(self):

        return len(self.toys)

store = ToyCatalog()

store["RoboDog"] = 49.99

print(store["RoboDog"])

print("Train" in store)

print(len(store))

Output:

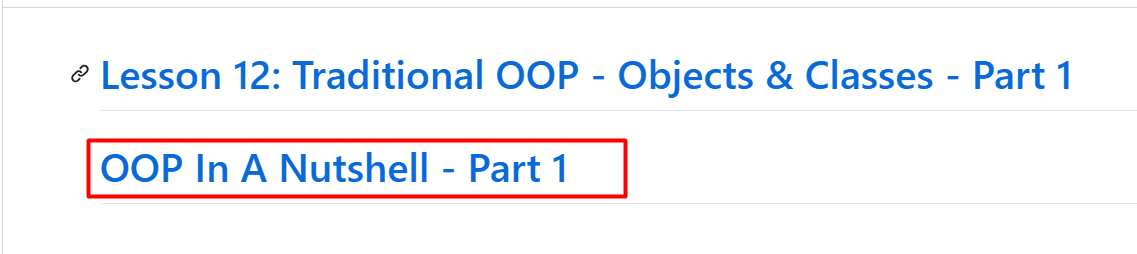
49.99

False

1

* Why Special Methods Matter:

1. Natural integration: Your objects work with Python's built-in functions
2. Readable code: len(box) instead of box.get\_length()
3. Custom behavior: Define how YOUR objects should add/compare/display



* <https://colab.research.google.com/drive/1pBG6oOzuF0Gg4Y-2Qb3BTSi09JSp0h7E?usp=sharing>
* Object-Oriented Programming (OOP) is a programming paradigm (model) that organizes software design around objects and classes rather than functions and logic.
* Class is a blueprint.
* An object is an instance of a class.
* Key pillars/principles of OOP:

1. Encapsulation: It wraps data/fields/attributes and methods/functions into a single unit (class). It restricts direct access to internal data (using private/protected attributes).
2. Abstraction: It hides complex implementation details from the user and shows only the essential features and relevant information.
3. Inheritance: It allows a child class to inherit properties and behaviors from parent class but child class can override or extend parent functionality. It promotes code reusability and hierarchical classification.
4. Polymorphism: It allows same method name to behave differently based on the object.

* Class: It is a blueprint or template for creating objects. It defines the attributes/data/fields and methods/functions that the objects created from the class will have.
* Object: An object is an instance of a class, which contains its own unique data.

Example:

class Person:

    def \_\_init\_\_(self, name):

        self.name = name

        self.state = "Idle"

    def running(self):

        self.state = "Running"

        print(f"{self.name} is now running.")

    def walking(self):

        self.state = "Walking"

        print(f"{self.name} is now walking.")

    def sleeping(self):

        self.state = "Sleeping"

        print(f"{self.name} is now sleeping.")

    def show\_state(self):

        print(f"{self.name} is currently in '{self.state}' state.")

person1 = Person("ABC")

person1.show\_state()

person1.walking()

person1.show\_state()

person1.sleeping()

person1.show\_state()

Output:

ABC is currently in 'Idle' state.

ABC is now walking.

ABC is currently in 'Walking' state.

ABC is now sleeping.

ABC is currently in 'Sleeping' state.

Here,

“Person” is a class.

“\_\_init\_\_” is a parameterized constructor of class “Person”.

“name” and “state” are attributes of class “Person”.

“running”, “walking”, “sleeping” and “show\_state” are functions of class “Person”.

“self” refers to the current instance of the class. “self” is used to access attributes and methods of the object within the class. It will always the first parameter of any method in a class but when we call the method, we do not pass self manually because Python does that for us.

self.state = "Idle": It sets default value “Idle” for “state” attribute.

person1 = Person("ABC"): “person1” is the instance/object of “Person” class.

* Attributes are variables that belong to an object. They represent the object’s state or properties.
* Methods are functions that belong to an object. They define the object’s behavior.
* Instantiating/Initializing Objects : To create/initialize an object/instance of a class, you call the class name like a function. This invokes the “\_\_init\_\_” method (the constructor) to initialize the object. The process of creating object from class is called instantiation/initialization
* In Python, the constructor method named as “\_\_init\_\_”.

Example of Parameterized Constructor:

class Person:

    def \_\_init\_\_(self, name, age, address):

        self.name = name

        self.age = age

        self.address = address

person1 = Person("ABC", 30, "DEF")

print("Name:    ", person1.name)

print("Age:     ", person1.age)

print("Address: ", person1.address)

Output:

Name: ABC

Age: 30

Address: DEF

* In Python, the destructor method named as “\_\_del\_\_”.

Example:

class Car:

    def \_\_init\_\_(self, brand, model):

        self.brand = brand

        self.model = model

    def \_\_del\_\_(self):

        print(f"The {self.brand} {self.model} has been destroyed.")

my\_car = Car("Toyota", "Corolla")

del my\_car

Output:

The Toyota Corolla has been destroyed.

* Access Modifiers: In Python, access control implemented using naming conventions rather than strict access modifiers like in other languages (e.g., Java or C#). In Python, there are three access modifiers:

1. Public: Attributes and methods are accessible from anywhere. No special syntax is used.

Example: age

1. Private: Attributes and methods are intended for internal use within the class and its subclasses/child class. A double underscore “\_\_” is used as a prefix. Example: \_\_age

Example:

class Employee:

    def \_\_init\_\_(self, name, salary):

        self.name = name

        self.\_\_salary = salary

    def show\_salary(self):

        print(f"{self.name}'s salary is {self.\_\_salary}")

emp = Employee("PQR", 50000)

emp.show\_salary()

print(emp.\_Employee\_\_salary)

emp.\_\_salary

Output:

PQR's salary is 50000

50000

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

[/tmp/ipython-input-746968800.py](https://localhost:8080/) in <cell line: 0>()

**11**

**12** print(emp.\_Employee\_\_salary)

---> 13 emp.\_\_salary

AttributeError: 'Employee' object has no attribute '**\_\_**salary'

Here,

Value of private attribute “\_\_salary” of class “Employee” is set to “50000” at the time of creating object “emp” and its value retrieved from public method “show\_salary” of class “Employee”.

Private attribute “\_\_salary” of class “Employee” cannot be directly accessed by object “emp” as shown in above example that when we tried to access private attribute “\_\_salary” of class “Employee” from object “emp” (emp.\_\_salary) then error raised : AttributeError: 'Employee' object has no attribute '\_\_salary'

But there is one approach named “mangling” by which we can access private member but it is not recommended approach. Python "name-mangles" private attributes/methods of class. Mangling is used when you are trying to avoid name conflicts in child classes.

Mangling approach: objectname.\_ClassFullName\_\_privatemembername

emp.\_Employee\_\_salary (present in above example)

1. Protected: Attributes and methods are accessible only within the class itself. A single underscore “\_” is used as a prefix. Example: \_age

Example 1:

class Father:

    def \_\_init\_\_(self):

        self.\_secret\_location = "Under the old oak tree"

    def \_where\_i\_hide\_the\_gold(self):

        return f"The gold is hidden at: {self.\_secret\_location}"

class Son(Father):

  pass

son = Son()

son.\_where\_i\_hide\_the\_gold()

Output:

The gold is hidden at: Under the old oak tree

Here,

“where\_i\_hide\_the\_gold” method is a protected member of class “Father” and is accessed by object “son” of child class “Son”.

“Son(Father)”: class “Son” is inherited by “Father” class

Example 2 (Not recommended approach):

class Father:

    def \_\_init\_\_(self):

        self.\_secret\_location = "Under the old oak tree"

    def \_where\_i\_hide\_the\_gold(self):

        return f"The gold is hidden at: {self.\_secret\_location}"

class Son(Father):

  pass

class Relatives:

    def \_\_init\_\_(self):

        self.father = Father()

    def try\_access\_secret(self):

            print(self.father.\_secret\_location)

            print(self.father.\_where\_i\_hide\_the\_gold())

son = Son()

print(son.\_where\_i\_hide\_the\_gold())

relatives = Relatives()

relatives.try\_access\_secret()

Output:

The gold is hidden at: Under the old oak tree

Under the old oak tree

The gold is hidden at: Under the old oak tree

Here,

Protected attribute “\_secret\_location” and protected method “\_where\_i\_hide\_the\_gold” of class “Father” are accessible in class “Relatives” without inheriting it by class “Father”. This approach not recommended.

* Types of Inheritance:

1. Single Inheritance: A subclass/child class inherits from a single super/parent class.
2. Multiple Inheritance: A subclass/child class inherits from multiple super/parent classes.

Example:

class Bird:

    def fly(self):

        return "Flying high!"

class Fish:

    def swim(self):

        return "Swimming deep!"

class FlyingFish(Bird, Fish):

    pass

flying\_fish = FlyingFish()

print(flying\_fish.fly())

print(flying\_fish.swim())

Output:

Flying high!

Swimming deep!

Here,

Class “FlyingFish” is inherited by two classes “Bird” and “Fish” due to which methods “fly” of “Bird” class and “swim” of “Fish” class is called from object “flying\_fish” of class “FlyingFish”.

In case of multiple inheritance, if same method is present in both classes then method of class which is inherited first, will be called.

Example:

class Bird:

    def fly(self):

        return "Flying Bird high!"

class Fish:

    def fly(self):

        return "Flying Fish high!"

class FlyingFish(Fish, Bird):

    pass

flying\_fish = FlyingFish()

print(flying\_fish.fly())

Output:

Flying Fish high!

Here,

“fly” method of parent class “Fish” is called from object “flying\_fish” of class “FlyingFish” because “Fish” is inherited first.

Similarly

Example:

class Bird:

    def fly(self):

        return "Flying Bird high!"

class Fish:

    def fly(self):

        return "Flying Fish high!"

class FlyingFish(Bird, Fish):

    pass

flying\_fish = FlyingFish()

print(flying\_fish.fly())

Output:

Flying Bird high!

Here,

“fly” method of parent class “Bird” is called from object “flying\_fish” of class “FlyingFish” because “Bird” is inherited first.

* “super” Function: It is used to call methods from the parent class inside child class. It is particularly useful in the constructor (\_\_init\_\_) to initialize attributes from the parent class.
* Polymorphism:

1. Method Overriding: It occurs when a subclass/child class provides a specific implementation of a method that is already defined in its superclass/parent class. The subclass/child class method overrides/changes implementation the superclass/parent class method.

Example:

class Animal:

    def \_\_init\_\_(self, name):

        self.name = name

    def speak(self):

        return "Animal sound"

class Dog(Animal):

    def \_\_init\_\_(self, name):

      super().\_\_init\_\_(name)

    def speak(self):

        return "Woof!"

class Cat(Animal):

    def \_\_init\_\_(self, name):

      super().\_\_init\_\_(name)

dog = Dog("Tommy")

print(f"{dog.name} says {dog.speak()}")

cat = Cat("Muffin")

print(f"{cat.name} says {cat.speak()}")

Output:

Tommy says Woof!

Muffin says Animal sound

Here,

“Dog” class overrides “speak” method of its parent class “Animal” due to which when we called “speak” method from object “dog” of class “Dog” then overridden method inside “Dog” class executed.

“Cat” class does not overrides “speak” method of its parent class “Animal” due to which when we called “speak” method from object “cat” of class “Cat” then method inside “Animal” parent class executed.

1. Method Overloading

* Operator Overloading: It is customizing the behavior of operators (e.g., +, -, \* etc.) for user-defined classes using special methods like \_\_add\_\_, \_\_sub\_\_ etc.

Example:

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_add\_\_(self, other):

        return Point(self.x + other.x, self.y + other.y)

    def \_\_str\_\_(self):

        return f"Point({self.x}, {self.y})"

point\_1 = Point(1, 2)

point\_2 = Point(3, 4)

point\_3 = point\_1 + point\_2

print(point\_3)

Output:

Point(4, 6)

Here,

Operator “+” overloaded by function “\_\_add\_\_”.

Similarly, “\_\_str\_\_” is also overloaded based on custom definition.

point\_1.x = 1

point\_1.y= 2

point\_2.x = 3

point\_2.x = 4

Due to which when

“point\_3 = point\_1 + point\_2” is executed then function “\_\_add\_\_” of object “point\_3” is called where parameter of “\_\_add\_\_” method are “self” is “point\_1” object while “other” is “point\_2” object and “\_\_add\_\_” return a new object of class “Point”, so new object is returned as “Point(1+3, 2+4)” means “Point(4,6)”.

When we print object “point\_3” then “\_\_str\_\_” method of “point\_3” is executed which return string value “Point(4,6)”.

* Duck Typing: It is a fundamental concept in Python programming that enables developers to write flexible and dynamic code. It emphasizes "we are all consenting adults here".

Example:

class Duck:

    def speak(self):

        return self.\_\_repr\_\_() + "   : Quack!"

class Person:

    def speak(self):

        return self.\_\_repr\_\_() + " : I can also quack like a duck!"

def make\_it\_quack(duck):

    print(duck.speak())

duck = Duck()

person = Person()

make\_it\_quack(duck)

make\_it\_quack(person)

Output:

<\_\_main\_\_.Duck object at 0x7c35149d41d0> : Quack!

<\_\_main\_\_.Person object at 0x7c35149d42c0> : I can also quack like a duck!

Here,

“make\_it\_quack” method shows duck typing which means that when we call method “make\_it\_quack” then we just pass any object and in method “make\_it\_quack”, “speak” method of passed object will be executed as there is no datatype mentioned in “duck” parameter of method “make\_it\_quack” so we can pass any object in argument of method “make\_it\_quack”.

We found that when we pass “duck” object in method argument of “make\_it\_quack” then “speak” method of class “Duck” is called.

Similarly, when we pass “person” object in method argument of “make\_it\_quack” then “speak” method of class “Person” is called.

When “duck.speak()” is called then “\_\_repr\_\_” method returns a string like “<\_\_main\_\_.Duck object at 0x...>” which indicates that object is a “Duck” class object and its memory location.

Similarly, when “person.speak()” is called then “\_\_repr\_\_” method returns a string like <\_\_main\_\_.Person object at 0x...> indicating a Person class object and its memory location.

* “\_\_repr\_\_” method is a special method in Python that used to return a string representation of an object. This representation primarily intended for developers and should be unambiguous and informative. It often used for debugging and logging purposes.

Essentially, method “\_\_repr\_\_” helps to identify which object is called in the output by showing its type and memory address.

* Abstraction: It simplifies complex systems by breaking them into manageable parts.

For example, all vehicles (cars, bikes) must have a “start” method but each vehicle starts differently. Abstraction lets us enforce this rule without worrying about implementation details.

* Abstract Classes and Methods are "incomplete" templates and you cannot use them directly. Abstract Classes and Methods:

1. define required methods for child classes
2. ensure consistency across related classes
3. prevent creating objects from the abstract class itself

Example:

from abc import ABC, abstractmethod

class Vehicle(ABC):

    @abstractmethod

    def start(self):

        pass

Output:

Empty

Here,

“abc” stands for “Abstract Base Classes” and “abc” module/library in Python provides tools to define abstract base classes.

“ABC” is a helper class from “abc” module that you inherit from when creating your own abstract base class like “class Vehicle(ABC)”, “ABC” makes the class abstract.

“@abstractmethod” is a decorator that marks a method as abstract and it means that subclasses/child classes must implement this abstract method like “start” method, otherwise they cannot be instantiated/initialized.

“@abstractmethod” marked method as abstract.

Child classes must implement all abstract methods like “start” method.

Now, update code as:

from abc import ABC, abstractmethod

class Vehicle(ABC):

    @abstractmethod

    def start(self):

        pass

class Truck(Vehicle):

    pass

truck: Truck = Truck()

Output:

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

[/tmp/ipython-input-158688706.py](https://localhost:8080/) in <cell line: 0>()

**9** pass

**10**

---> 11 truck: Truck = Truck()

TypeError: Can't instantiate abstract class Truck without an implementation for abstract method 'start'

Here,

Error raised because abstract method “start” of abstract class “Vehicle” not defined in child class “Truck” due to which we are not able to create object “truck” of class “Truck”.

Now, update code as:

from abc import ABC, abstractmethod

class Vehicle(ABC):

    @abstractmethod

    def start(self):

        pass

class Truck(Vehicle):

    def start(self):

        return "Truck started"

truck: Truck = Truck()

print(truck.start())

Output:

Truck started

Here,

“truck” object is created because we have defined abstract method “start” of abstract class “Vehicle” inside childe class “Truck”.

When we call “start” method from object “truck” then method defined inside “Truck” class executed.

* Class variable (shared by all instances):

Example:

class City:

    name = "Unknown City"

    def \_\_init\_\_(self, name):

       City.name = name

    def show\_info(self):

        print(f"City Name: {City.name} ")

city1 = City("Kolachi1")

city1.show\_info()

print("\n")

city2 = City("Kolachi2")

city1.show\_info()

city2.show\_info()

City.name = "Karachi"

print("\n")

city1.show\_info()

city2.show\_info()

Output:

City Name: Kolachi1

City Name: Kolachi2

City Name: Kolachi2

City Name: Karachi

City Name: Karachi

Here,

Variable “name” is class variable of class “City”.

Any change in class variable at any place, will affect value for all instances/objects.

* Methods in Python Classes:

1. Instance methods: They operate on an instance of the class and can access and modify instance attributes. Their first parameter is always self, which refers to the instance of the class. These methods are used to access or modify instance-specific data.
2. Class methods: These methods operate on the class itself rather than on instances. They can access and modify class attributes. Their first parameter is always “cls”, which refers to the class. These methods defined using the “@classmethod” decorator. They used to work with class-level data or perform operations related to the class.

Example:

class Human:

  specie = "Homo sapiens"

  @classmethod

  def get\_specie(cls):

    return cls.specie

Human.get\_specie()

Output:

Homo sapiens

Here,

“specie” is a class variable.

Python environment will automatically pass the “cls” as a parameter.

1. Static methods: These methods are independent of both the class and its instances. They do not have access to “self” or “cls”. These methods defined using “@staticmethod” decorator. These methods used for utility functions that do not depend on class or instance data.

“@staticmethod” makes a method independent of the class instance/object.

A static method does not need “self” (no access to instance/object variables) and does not need cls (no access to class variables). A static method belongs to the class namespace, but behaves like a normal function inside the class.

A static member can be called directly with the name of class.

Example:

class Human:

    @staticmethod

    def breathe():

        print("Breathing...")

Human.breathe()

human = Human()

human.breathe()

Output:

Breathing...

Breathing...

Here,

Human.breathe(): “breathe” is a static function due to which it is directly called with the name of class “Human”.

human.breathe(): In other languages like C#, static member cannot be called from instance/object of class but in Python, we can still call static member “breathe” from “human” instance/object of class “Human” lets you call it using an instance because at backend, Python just ignores the “human” object/instance of class “Human” and runs directly “Human.breathe()”.

Now, updated code as:

class Human:

    testing = "Testing 1"

    @staticmethod

    def breathe():

        print("Breathing...")

    def eat(self):

        print("Eating...")

        self.breathe()

    @classmethod

    def sleep(cls):

      print("Sleeping...")

      cls.breathe()

Human.breathe()

human = Human()

human.breathe()

human.eat()

human.sleep()

Output:

Breathing...

Breathing...

Eating...

Breathing...

Sleeping...

Breathing...

Here,

As “breathe” is a static method, “breathe” does not take “self” or “cls” automatically but “breathe” can still be called through either an instance/object (self) or the class (cls) because Python just looks it up in the class namespace.

self.breathe(): call breathe from class “Human” namespace.

cls.breathe(): works the same way as self.breathe() as Python resolves “breathe” from the class “Human” namespace and runs it.

We can call a static member through “self” or “cls”, but they are not passed to it. Python just ignores them and calls the function as-is.

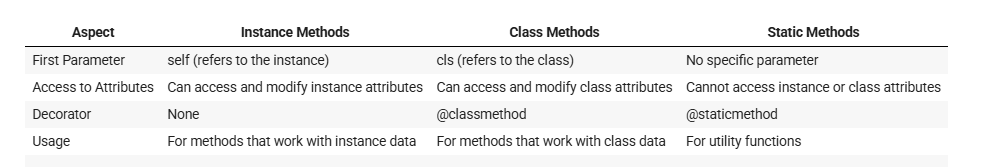
“cls.breathe()”: Here, “cls” would truly be the class itself, not the instance/object.

Difference between methods:

Instance method “eat” needs an object/instance “self”.

Class method “sleep” gets the class itself “cls” (Human) automatically, whether you call it via class or instance/object.

Static method “breathe” gets nothing automatically as it is just a normal function sitting inside the class namespace.



* If we can access class members using the class name, why we need to pass “cls” in “@classmethod”?

Example:

class Animal:

    species = "Unknown"

    @classmethod

    def make(cls):

        return cls()

class Dog(Animal):

    species = "Dog"

animal = Animal.make()

print(type(animal))

dog = Dog.make()

print(type(dog))

Output:

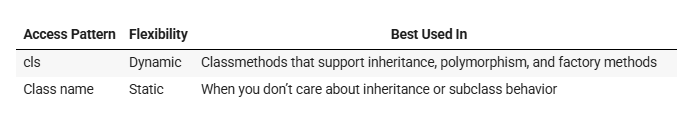
<class '\_\_main\_\_.Animal'>

<class '\_\_main\_\_.Dog'>

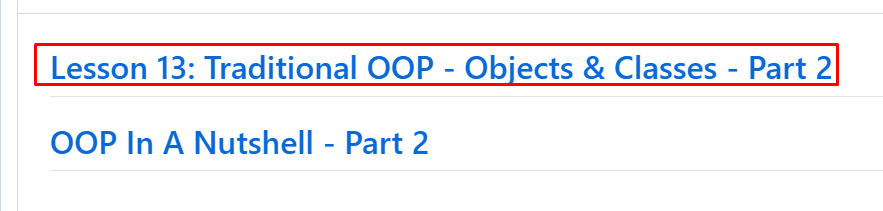
Here,

Animal.make(): returned class “Animal” instance/object

Dog.make(): returned class “Dog” instance/object although “make” method is defined in parent class “Animal”.



* Step 13: <https://github.com/panaversity/learn-modern-ai-python/tree/main/00_python_colab/13_traditional_oop_part_2>



* <https://colab.research.google.com/drive/13i4LN9HrdFbp_GK-IEoKU8OCZh6F4Ilp?usp=sharing>
* Class and Static Variables: They are used to store data that is related to the class itself, rather than to instances of the class. They offer a way to share data and behavior across all instances.
* Difference between Class and Static Variables: While the terms are often used “interchangeably” in Python, class variables are the preferred terminology. These variables are defined within the class but outside of any method.

Class variables: These are associated with the class itself and are shared among all instances of the class. They are defined within the class but outside of any method.

Instance variables: These are unique to each object or instance of the class.

* Accessing and Modifying Class Variables: Class variables can be accessed and modified using the class name or an instance of the class.

Accessing: Class variables are accessed using the class name directly (e.g., ClassName.variable\_name) or through an instance of the class (e.g., instance\_name.variable\_name).

Modifying: When modifying a class variable through an instance, you are actually creating a new instance variable that shadows the class variable for that particular instance. To modify the class variable itself, you should use the class name.

Example:

class Bakery:

    type = "cake"

    def \_\_init\_\_(self, flavor, price):

        self.flavor = flavor

        self.price = price

    def update\_cake\_count(cls, count):

        cls.cake\_count = count

print(Bakery.type)

cake1 = Bakery("Chocolate", 25.00)

cake2 = Bakery("Vanilla", 22.00)

print(cake1.flavor)

print(cake2.price)

Bakery.type = "pastry"

print(cake1.type)

cake1.type = "cookie"

print(cake1.type)

print(cake2.type)

Output:

cake

Chocolate

22.0

pastry

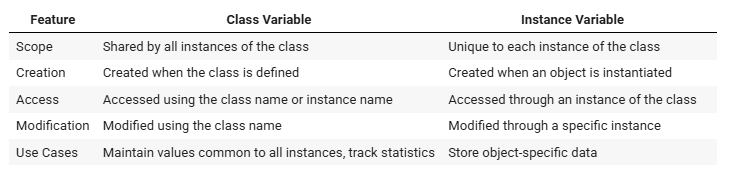
cookie

pastry

Here,

“type” is a class variable.

“flavor” and “price” are instance variables.



* Composition: They are both ways to relate classes in object-oriented programming, focusing on how objects contain or interact with each other.

Composition (Strong Relationship): It is a "has-a" relationship where one object contains another, and the contained object's lifecycle is dependent on the container object. If the container object is destroyed, the contained object is also destroyed. It is a strong relationship like a car and its engine; the car cannot function without the engine, and the engine is intrinsically part of the car. Composition involves building complex objects by combining simpler ones.

Example:

class Engine:

    def start(self):

        return "Engine starting"

class Car:

    def \_\_init\_\_(self):

        self.engine = Engine()

    def start(self):

        return f"Car starting: {self.engine.start()}"

Output:

Empty

Here,

Car and Engine demonstrate composition. The Car class creates and owns an instance of the Engine class. If the Car object is destroyed, the Engine object is also destroyed.

* Aggregation (Weak Relationship): Aggregation is also a "has-a" relationship, but it represents a weaker association than composition. In aggregation, the contained object can exist independently of the container object. It is like a university and its departments; the university contains departments, but the departments can exist even if the university ceases to exist.

Example:

class Department:

    def \_\_init\_\_(self, name):

        self.name = name

class University:

    def \_\_init\_\_(self, name):

        self.name = name

        self.departments = []

    def add\_department(self, department):

        self.departments.append(department)

Output:

Empty

Here,

University and Department demonstrate aggregation. The University class contains a list of Department objects, but the Department objects can exist independently of the University object. The university does not exclusively own the Department.

* Difference Between Composition and Inheritance: Composition and inheritance are both mechanisms for code reuse in OOP but they differ significantly:

1. Relationship: Composition is a "has-a" relationship, while inheritance is an "is-a" relationship.
2. Flexibility: Composition offers more flexibility than inheritance. You can easily change or replace components without affecting the entire system.
3. Coupling: Composition reduces coupling between classes, making the code easier to maintain and understand. Inheritance can lead to tight coupling, code duplication, and other problems.
4. Reuse: Composition reuses code by combining objects while inheritance reuses code by inheriting properties and methods from parent classes.
5. Composition allows for independent modification and replacement of components, promoting cleaner code design. Inheritance has its place, but composition should be preferred.

* Method Resolution Order (MRO): Method Resolution Order (MRO) is the order in which Python searches for methods and attributes in a class hierarchy, especially in cases of multiple inheritance. It ensures that the correct method or attribute is found and called when there are overlapping names in the inheritance tree.

“mro” Method: It is a built-in method in Python that returns a list of classes in the order they will be searched for attributes and methods. This list represents the linearization of the class hierarchy.

* How Python Resolves Method Calls in Multiple Inheritance: Python uses the C3 Linearization algorithm to determine the MRO. This algorithm ensures that:

1. Subclasses come before their parent classes.
2. The order of inheritance is preserved.
3. No class is visited more than once.

Example:

class A:

    def greet(self):

        return "Hello from A"

class B(A):

    def greet(self):

        return "Hello from B"

class C(A):

    def greet(self):

        return "Hello from C"

class D(B, C):

    pass

d = D()

print(D.mro())

print(d.greet())

Output:

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

Hello from B

Here,

Class Hierarchy: A is the parent class of both B and C while D inherits from both B and C.

MRO of Class D: The mro() method returns the order in which Python will search for methods and attributes in class D: [D, B, C, A, object]

Method Call Resolution: When d.greet() is called, Python searches for the greet method in the following order:

a) First in D: Not found.

b) Then in B: Found! The greet method of B is called, and the search stops.

Visualizing MRO: The MRO for class D can be visualized as a linear sequence: D → B → C → A → object

Example 2 (Diamond Inheritance):

class X:

    def greet(self):

        return "Hello from X"

class Y(X):

    def greet(self):

        return "Hello from Y"

class Z(X):

    def greet(self):

        return "Hello from Z"

class W(Y, Z):

    pass

w = W()

print(W.mro())  # Output: [<class '\_\_main\_\_.W'>, <class '\_\_main\_\_.Y'>, <class '\_\_main\_\_.Z'>, <class '\_\_main\_\_.X'>, <class 'object'>]

print(w.greet())  # Output: Hello from Y

Output:

[<class '\_\_main\_\_.W'>, <class '\_\_main\_\_.Y'>, <class '\_\_main\_\_.Z'>, <class '\_\_main\_\_.X'>, <class 'object'>]

Hello from Y

Here,

Class Hierarchy: X is the parent class of both Y and Z while W inherits from both Y and Z.

MRO of Class W:The mro() method returns the order in which Python will search for methods and attributes in class W: [W, Y, Z, X, object]

Method Call Resolution: When w.greet() is called, Python searches for the greet method in the following order:

a) First in W: Not found.

b) Then in Y: Found! The greet method of Y is called, and the search stops.

* Decorators in Classes: Decorators in Python are a powerful feature that allows you to modify or extend the behavior of functions or methods. When applied to classes, decorators can enhance or alter the behavior of the class or its methods. Additionally, Python provides specific property decorators (@property, @setter, and @deleter) to manage attribute access in a controlled way.

Example:

class CountCalls:

    def \_\_init\_\_(self, func):

        self.func = func

        self.call\_count = 0

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        self.call\_count += 1

        print(f"Decorator: Call {self.call\_count} of {self.func.\_\_name\_\_}")

        return self.func(\*args, \*\*kwargs)

@CountCalls

def say\_hello(name):

    print(f"Hello, {name}!")

say\_hello("Alice")

say\_hello("Bob")

Output:

Decorator: Call 1 of say\_hello

Hello, Alice!

Decorator: Call 2 of say\_hello

Hello, Bob!

Here,

CountCalls is a class decorator that counts the number of times a function is called. The @CountCalls “say\_hello” function applies the CountCalls decorator to it. Each time “say\_hello” is called, the “\_\_call\_\_” method of CountCalls is executed, incrementing the “call\_count” and printing a message.

* Property Decorators: The “@property” decorator is used to define a method as a getter for a class attribute. It allows you to access the attribute like a property rather than a method. You can also define setter and deleter methods using the @setter and @deleter decorators, respectively.

The “@property” decorator lets you control how an attribute is accessed, set, or deleted in a class. It allows you to add logic (like validation or calculations) while still making it look like a simple attribute.

1. Getter (Read-Only Property): It turns a method into a "getter" (like reading an attribute) and no parentheses needed for it.

Example:

class Person:

    def \_\_init\_\_(self, name):

        self.\_name = name

    @property

    def name(self):

        """Getter for name"""

        return self.\_name

p = Person("Alice")

print(p.name)

Output:

Alice

Here,

“name” acts like an attribute but it is a method.

1. Setter (Change a Value with Validation): “@prop\_name.setter” lets you modify the attribute and add checks for validation before setting.

Example:

class Person:

    def \_\_init\_\_(self, name):

        self.\_name = name

    @property

    def name(self):

        return self.\_name

    @name.setter

    def name(self, new\_name):

        if not isinstance(new\_name, str):

            raise ValueError("Name must be a string!")

        self.\_name = new\_name

p = Person("Bob")

p.name = "Charlie"

print(p.name)

Output:

Charlie

If we add code:

p.name = 123

Output:

Charlie

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

[/tmp/ipython-input-4183801898.py](https://localhost:8080/) in <cell line: 0>()

**17** print(p.name)

**18**

---> 19 p.name = 123

[/tmp/ipython-input-4183801898.py](https://localhost:8080/) in name(self, new\_name)

**10** def name(self, new\_name):

**11** if not isinstance(new\_name, str):

---> 12 raise ValueError("Name must be a string!")

**13** self.\_name = new\_name

**14**

ValueError: Name must be a string!

Here,

Our added validation is invoked and error message that we have mentioned is raised.

1. Deleter (Remove an Attribute): “@prop\_name.deleter” runs when we delete property of an object/instance (obj.prop). It is used for cleanup.

Example:

class Person:

    def \_\_init\_\_(self, name):

        self.\_name = name

    @property

    def name(self):

        return self.\_name

    @name.deleter

    def name(self):

        print("Deleting name!")

        del self.\_name

# Usage

p = Person("Dave")

print(p.name)

del p.name

Output:

Dave

Deleting name!

Here,

del p.name: removed property “name” from object “p”.

Therefore, when we add code:

print(p.name)

Output:

Dave

Deleting name!

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

[/tmp/ipython-input-2072947445.py](https://localhost:8080/) in <cell line: 0>()

**17**

**18** del p.name

---> 19 print(p.name) # ❌ Error! (AttributeError: 'Person' has no attribute '\_name') # uncomment to see error

[/tmp/ipython-input-2072947445.py](https://localhost:8080/) in name(self)

**5** @property

**6** def name(self):

----> 7 return self.\_name

**8**

**9** @name.deleter

AttributeError: 'Person' object has no attribute '\_name'

1. Computed Property (Dynamic Value): Calculate a value at run time or dynamically.

Example:

class Person:

    def \_\_init\_\_(self, weight\_kg, height\_m):

        self.weight = weight\_kg

        self.height = height\_m

    @property

    def bmi(self):

        """Body Mass Index (weight / height²)"""

        return self.weight / (self.height \*\* 2)

p = Person(70, 1.75)

print(p.bmi)

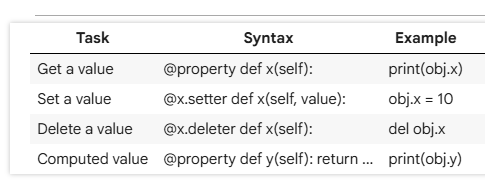
Output:

22.857142857142858

Here,

bmi.setter : we cannot change “bmi” directly.

* Summary of Properties:



* When to use “@property”:

1. Add validation (e.g., check if age ≥ 0)
2. Make read-only attributes (like bmi)
3. Hide internal variables (use “\_name” instead of “name”)
4. Change how attributes work without breaking existing code

* Callable: In Python, a “callable” is an object that can be called like a function. In other words, it is an object that can be invoked with parentheses “()” to execute some code.

Examples:

1. Functions: We can define a function using the “def” keyword and call it by its name followed by parenthesis.
2. Lambda functions: These are small, anonymous functions that can be defined inline.
3. Classes: weou can call a class like a function! When you do, it creates a new instance of the class.
4. Methods: These are functions that are part of a class.
5. Instances of classes that implement “\_\_call\_\_”: If a class defines a special method called “\_\_call\_\_”, instances of that class become callable.

* Making an object callable: An object is “callable” if it has a “\_\_call\_\_” method. This method is a special method that is invoked when you call the object like a function.

Example:

class MyClass:

    def \_\_call\_\_(self):

        print("I'm callable!")

obj = MyClass()

obj()

Output:

I'm callable!

Here,

“MyClass” defines a “\_\_call\_\_” method so instances of “MyClass” become callable.

* Checking if an object is callable: We can use the “callable()” function to check if an object is callable.

Example:

def my\_function():

    pass

print(callable(my\_function))

class MyClass:

    def \_\_call\_\_(self):

        pass

obj = MyClass()

print(callable(obj))

print(callable("hello"))

Output:

True

True

False

Here,

We define a function “my\_function” and a class “MyClass” with a “\_\_call\_\_” method. We then create an instance of “MyClass” and check if each object is callable using “callable()”. The string "hello" is not callable, so “callable("hello")” returns False.

* Use cases for callables:

1. Creating higher-order functions that take other functions as arguments
2. Implementing callbacks or event handlers
3. Creating factories or constructors that return instances of classes
4. Creating domain-specific languages (DSLs) or mini-languages

* Modules and Packages in OOP: The organizing code into “modules” and “packages” is essential for maintaining clean, scalable, and maintainable code, especially in large projects. In Python, a “module” is a single file containing Python code (e.g., classes, functions, variables), and a “package” is a directory containing multiple modules and an “\\_\\_init\_\_.py” file.

Organizing Classes in Modules:

1. Group related classes into a single module.
2. Use meaningful names for modules and classes.
3. Avoid putting all classes in a single module (unless the project is very small)

Importing Classes from Modules:

1. Use the import statement to import the entire module.
2. Use from module import “ClassName” to import specific classes.
3. Use from module import “\*” to import all classes (not recommended for large projects).

Example “test1” project.

Output:

Woof!

Meow!

Squawk!

Chirp!

Car: Toyota Corolla

Bike: Harley Davidson

Here,

Module Organization: related classes grouped into modules (mammals.py, birds.py, cars.py, bikes.py) and “\_\_init\_\_.py” files in the “animals” and “vehicles” directories/folders make them “packages” and allow importing classes directly from the package.

Importing Classes: In “main.py”, classes are imported from their respective modules as:

from animals import Dog, Cat, Parrot, Sparrow

from vehicles import Car, Bike

Using Classes: instances of the classes created and their methods called in main.py.

* Advanced Object-Oriented Programming (OOP) Concepts:

1. Metaclasses: A metaclass is the class of a class. It defines how a class behaves. In Python, the default metaclass is type. We can create custom metaclasses to control class creation and behavior.

Example:

class Meta(type):

    def \_\_new\_\_(cls, name, bases, dct):

        print(f"Creating class: {name}")

        return super().\_\_new\_\_(cls, name, bases, dct)

class MyClass(metaclass=Meta):

    pass

Output:

Creating class: MyClass

Here,

“Meta” is a custom metaclass.

“\_\_new\_\_” is the method that actually creates a new class object.

“cls” is the metaclass itself.

“name” is the name of the class being defined (in this case "MyClass").

“bases” is the base classes of the new class (a tuple).

“dct” is the class body dictionary, containing methods/attributes defined in the class.

“class MyClass(metaclass=Meta)”: Python instead of calling the default type to create “MyClass”, it calls “\_\_new\_\_” method of custom class “Meta”. Execution steps:

1. Python parses the class “MyClass” body. As “pass” so no attributes.
2. Python calls:

Meta.\_\_new\_\_(Meta, "MyClass", (object,), {"\_\_module\_\_": "\_\_main\_\_", "\_\_qualname\_\_": "MyClass"})

1. Executes “Meta.\_\_new\_\_”
2. “super().\_\_new\_\_” creates the actual class “MyClass” object.
3. Singleton Design Pattern: It ensures that a class has only one instance and provides a global point of access to it. This is useful when you need to manage shared resources, such as a database connection or configuration settings. Every time we try to create a new object, it will give you back the same instance/object.

Example:

class Singleton:

    \_instance = None

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

        if cls.\_instance is None:

            cls.\_instance = super().\_\_new\_\_(cls)

        return cls.\_instance

singleton1 = Singleton()

singleton2 = Singleton()

print(singleton1 is singleton2)

print(id(singleton1) == id(singleton2))

Output:

True

True

Here,

Class attribute “\_instance” is set to “None” to store the single instance.

“\_\_new\_\_” is a special method in Python that actually creates an object (before “\_\_init\_\_” initializes it). If we want to control object creation, we need to override “\_\_new\_\_” method.

singleton1 = Singleton(): It runs “\_\_new\_\_” first as “\_instance” is “None”, it creates a new object using “super().\_\_new\_\_(cls)” and stores it in “\_instance”.

singleton2 = Singleton(): As “\_instance” already exists because created at the time of “singleton1” so it just returns the same stored object instead of making a new one.

Therefore, both conditions “singleton1 is singleton2” (both variables point to same object) and “id(singleton1) == id(singleton2))” (same memory address) returns “True”.

1. Factory Design Pattern: It is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It promotes loose coupling and flexibility.

Example:

class Animal:

    def speak(self):

        pass

class Dog(Animal):

    def speak(self):

        return "Woof!"

class Cat(Animal):

    def speak(self):

        return "Meow!"

class AnimalFactory:

    @staticmethod

    def create\_animal(animal\_type):

        if animal\_type == "dog":

            return Dog()

        elif animal\_type == "cat":

            return Cat()

        else:

            raise ValueError("Invalid animal type")

dog = AnimalFactory.create\_animal("dog")

cat = AnimalFactory.create\_animal("cat")

print(dog.speak())

print(cat.speak())

Output:

Woof!

Meow!

Here,

“Animal” is a base class (or abstract-like class) which contains a method “speak” but it does not do anything (pass). The idea is that all animals should have a “speak” method but each animal will implement it differently.

Classes “Dog” and “Cat” both inherited from class “Animal” and they override “speak” method of base class “Animal”.

“AnimalFactory” is a Factory Class which contains a static method “create\_animal” (doesn’t need self, so you can call it without creating an object).

Based on animal\_type, it returns an instance of Dog or Cat.

If an invalid type is passed, it raises an error.

This pattern is called the Factory Design Pattern, where object creation logic is centralized in one place.

* To know more about Design Pattern please visit:

1. The Catalog of Python Design Pattern Examples: <https://refactoring.guru/design-patterns/python>
2. Gang of Four design patterns in Python: <https://github.com/tuvo1106/python_design_patterns>
3. Gang of Four (GOF) Design Patterns: <https://www.geeksforgeeks.org/system-design/gang-of-four-gof-design-patterns/>

* Error Handling in OOP: Error handling is a critical aspect of writing robust and reliable code. In Object-Oriented Programming (OOP), you can handle errors by raising exceptions in methods and creating custom exceptions to represent specific error conditions in your application.

Raising Exceptions in Methods: In Python, you can raise exceptions using the “raise” keyword. This is useful for signaling that an error has occurred in a method.

Custom Exceptions in Classes: Custom exceptions are user-defined exceptions that inherit from Python’s built-in Exception class. They allow you to define specific error types for your application.

Example:

class InsufficientFundsError(Exception):

    def \_\_init\_\_(self, balance, amount):

        super().\_\_init\_\_(f"Insufficient funds: balance is {balance}, but {amount} was requested.")

        self.balance = balance

        self.amount = amount

class BankAccount:

    def \_\_init\_\_(self, account\_holder, balance=0):

        self.account\_holder = account\_holder

        self.balance = balance

    def deposit(self, amount):

        if amount <= 0:

            raise ValueError("Deposit amount must be positive.")

        self.balance += amount

        print(f"Deposited {amount}. New balance: {self.balance}")

    def withdraw(self, amount):

        if amount <= 0:

            raise ValueError("Withdrawal amount must be positive.")

        if self.balance < amount:

            raise InsufficientFundsError(self.balance, amount)

        self.balance -= amount

        print(f"Withdrew {amount}. New balance: {self.balance}")

    def display(self):

        print(f"Account Holder: {self.account\_holder}, Balance: {self.balance}")

account = BankAccount("Alice", 1000)

try:

    account.deposit(500)

except ValueError as e:

    print(e)

try:

    account.withdraw(2000)

except InsufficientFundsError as e:

    print(e)

except ValueError as e:

    print(e)

try:

    account.deposit(0)

except ValueError as e:

    print(e)

account.display()

Output:

Deposited 500. New balance: 1500

Insufficient funds: balance is 1500, but 2000 was requested.

Deposit amount must be positive.

Account Holder: Alice, Balance: 1500

Here,

“InsufficientFundsError” is a custom exception class, extended/inherited by Python’s built-in class “Exception”. “InsufficientFundsError” exception raised when a withdrawal request is more than the available balance. The error message of exception is automatically formatted like "Insufficient funds: balance is 1000, but 2000 was requested."

Python’s “ValueError” exception raised when value of parameter “amount” of “deposit” method is less than or equal to 0 (zero).

Custom exception error “InsufficientFundsError” exception raised when value of parameter “amount” is greater than class attribute “balance”.

account = BankAccount("Alice", 1000): creates an account for “Alice” with “1000” initial balance.

account.deposit(500) : valid deposit “500” as greater than zero so added to existing balance “1000”. Now, new balance = 1500.

account.withdraw(2000): Balance is “1500” but requested withdraw amount is “2000” so raises exception “InsufficientFundsError” with exception message "Insufficient funds: balance is 1500, but 2000 was requested."

account.deposit(0): invalid deposit “0” so raises exception “ValueError” with exception message "Deposit amount must be positive."

account.display(): printed account summary:"Account Holder: Alice, Balance: 1500"

* Testing OOP Code: Testing is a critical part of software development, especially when working with Object-Oriented Programming (OOP). It ensures that your classes and methods behave as expected. Python provides libraries like “unittest” and “pytest” for writing and running tests.

Unit Testing Classes and Methods: Unit testing involves testing individual components (e.g., classes and methods) in isolation to ensure they work correctly.

“unittest” library: Python’s built-in testing framework, inspired by Java’s JUnit.

“pytest” library: A popular third-party testing framework that is more flexible and easier to use than “unittest”.

Example project “test2”:

“calculator.py” contains “Calculator” class

“unittest\_calculator.py” contains unit tests related to class “Calculator” using “unittest” library, we execute unit tests using command: python -m unittest unittest\_calculator.py

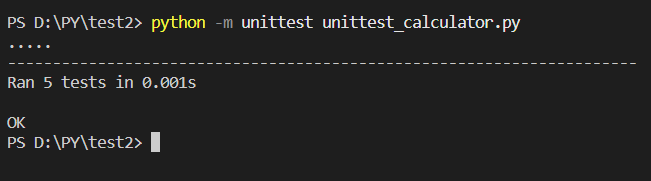
Output:

.....

----------------------------------------------------------------------

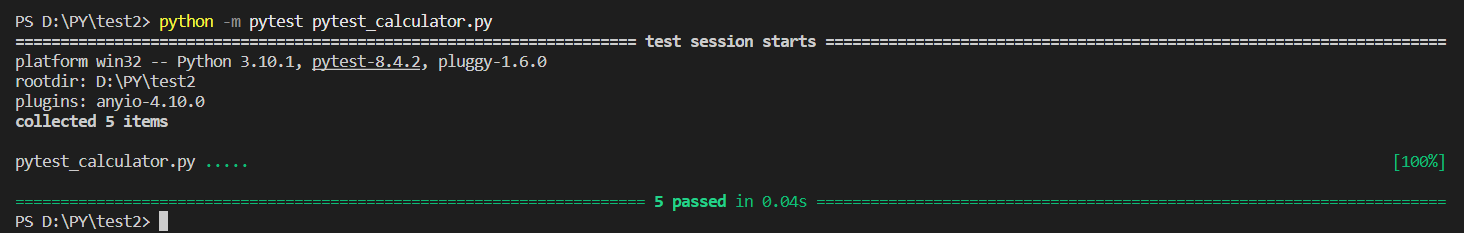
Ran 5 tests in 0.001s

OK



“pytest\_calculator.py” unit tests related to class “Calculator” using “pytest” library, we execute unit tests using command: python -m pytest pytest\_calculator.py

Output:



Here,

Calculator Class:

1. Implements basic arithmetic operations (add, subtract, multiply, divide).
2. Raises “ValueError” exception if division by zero is attempted.

Testing with “unittest” library: “TestCalculator” class inherits from class “unittest.TestCase”. The “setup” method initializes a “Calculator” instance “calc” for each test. Test methods use “assertEqual” to check if the actual output matches the expected output. The “assertRaises” method is used to test for exceptions.

Testing with “pytest” library: The “@pytest.fixture” decorator used to create a “Calculator” class instance for each test. Test functions use the “assert” keyword to check if the actual output matches the expected output. The “pytest.raises” context manager used to test for exceptions.

* SOLID Principles in Python:

1. Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility.

Example:

class Report:

    def generate\_report(self, data):

        pass

    def save\_report(self, file\_path):

        pass

class ReportGenerator:

    def generate\_report(self, data):

        pass

class ReportSaver:

    def save\_report(self, report, file\_path):

        pass

Here,

“Report” is a bad class with multiple responsibilities like generating report and saving report while “ReportGenerator” and “ReportSaver” are good classes having single responsibility as responsibilities are separated into different classes.

1. Open/Closed Principle (OCP): A class should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.

Example:

class AreaCalculator:

    def calculate\_area(self, shape):

        if shape.type == "circle":

            return 3.14 \* shape.radius \*\* 2

        elif shape.type == "rectangle":

            return shape.length \* shape.width

Here,

“AreaCalculator” is a bad class which is modifying existing code to add new functionality while:

from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod

    def area(self):

        pass

class Circle(Shape):

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def area(self):

        return 3.14 \* self.radius \*\* 2

class Rectangle(Shape):

    def \_\_init\_\_(self, length, width):

        self.length = length

        self.width = width

    def area(self):

        return self.length \* self.width

class AreaCalculator:

    def calculate\_area(self, shape):

        return shape.area()

Here,

Good implementation as it extend functionality without modifying existing code.

1. Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Example:

class Bird:

    def fly(self):

        pass

class Ostrich(Bird):

    def fly(self):

        raise NotImplementedError("Ostriches can't fly")

Here,

Bad example of subclass changes the behavior of the parent class while:

class Bird:

    def move(self):

        pass

class Sparrow(Bird):

    def move(self):

        print("Flying")

class Ostrich(Bird):

    def move(self):

        print("Running")

Here,

Good example of subclass adheres to the behavior of the parent class.

1. Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. Instead of one large interface, create smaller, specific ones.

Example:

class Printer:

    def print\_document(self):

        pass

    def scan\_document(self):

        pass

    def fax\_document(self):

        pass

Here,

Bad example of one large interface while:

class Printer:

    def print\_document(self):

        pass

class Scanner:

    def scan\_document(self):

        pass

class FaxMachine:

    def fax\_document(self):

        pass

Here,

Good example of smaller and specific interfaces

1. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Example:

class LightBulb:

    def turn\_on(self):

        pass

    def turn\_off(self):

        pass

class Switch:

    def \_\_init\_\_(self):

        self.bulb = LightBulb()

    def operate(self):

        if condition:

            self.bulb.turn\_on()

        else:

            self.bulb.turn\_off()

Here,

Bad example of High-level module depends on low-level module while:

from abc import ABC, abstractmethod

class Switchable(ABC):

    @abstractmethod

    def turn\_on(self):

        pass

    @abstractmethod

    def turn\_off(self):

        pass

class LightBulb(Switchable):

    def turn\_on(self):

        pass

    def turn\_off(self):

        pass

class Switch:

    def \_\_init\_\_(self, device: Switchable):

        self.device = device

    def operate(self):

        if condition:

            self.device.turn\_on()

        else:

            self.device.turn\_off()

Here,

Good example of both depend on abstractions.

* Iterable: In Python, it is not a parent class but rather an abstract base class (ABC) defined in the “collections.abc” module. It serves as a protocol or interface that other classes can implement to indicate that they are iterable (i.e., they can be looped over using a “for” loop or other iteration constructs).

An iterable is any object that can return an iterator when the “iter” function is called on it. The iterator is used to traverse through the elements of the iterable.

Parent Class Relationship: The Iterable abstract base class is not a parent class in the traditional sense (like inheritance in object-oriented programming). Instead, it is used to define a protocol that other classes can adhere to by implementing the “iter” method.

Example of Iterable Classes:

1. Lists: list
2. Tuples: tuple
3. Strings: str
4. Dictionaries: dict
5. Sets: set
6. Ranges: range
7. Generators: generator

* We can use “isinstance” function with “collections.abc.Iterable” to check if an object is iterable:

Example:

from collections.abc import Iterable

print("isinstance([1, 2, 3], Iterable) = ",isinstance([1, 2, 3], Iterable))

print('isinstance("hello", Iterable)   = ', isinstance("hello", Iterable))

print("isinstance(123, Iterable)       = ", isinstance(123, Iterable))

Output:

isinstance([1, 2, 3], Iterable) = True

isinstance("hello", Iterable) = True

isinstance(123, Iterable) = False

* How to Make a Custom Class Iterable: To make a custom class iterable, you need to implement “iter” method and “iter” method should return an iterator object. The iterator object must implement the “next” method.

Example:

from collections.abc import Iterable, Iterator

class MyIterable(Iterable):

    def \_\_init\_\_(self, data):

        self.data = data

    def \_\_iter\_\_(self):

        return MyIterator(self.data)

class MyIterator(Iterator):

    def \_\_init\_\_(self, data):

        self.data = data

        self.index = 0

    def \_\_next\_\_(self):

        if self.index >= len(self.data):

            raise StopIteration

        value = self.data[self.index]

        self.index += 1

        print("Called: MyIterator.\_\_next\_\_")

        return value

my\_iterable = MyIterable([1, 2, 3])

for item in my\_iterable:

    print("item : ", item)

Output:

Called: MyIterator.\_\_next\_\_

item : 1

Called: MyIterator.\_\_next\_\_

item : 2

Called: MyIterator.\_\_next\_\_

item : 3

Here,

“raise StopIteration” is used to tell Python to stop looping. It is used inside custom iterators.

* Object-Based Language vs Object-Oriented Language

1. Object-Based Language: A language that supports objects (data structures with attributes and methods) and encapsulation (data hiding), but lacks key OOP features like inheritance and polymorphism. Features:
2. Objects as instances with properties and methods.
3. Encapsulation (e.g., public/private access modifiers).
4. May include basic polymorphism (e.g., operator overloading).

Examples: JavaScript (prototype-based, but lacks classical inheritance), classic Visual Basic, Ada.

1. Object-Oriented Language: A language that implements the four pillars of OOP:
2. Encapsulation: Hiding internal state and requiring interaction via methods.
3. Inheritance: Creating hierarchical relationships between classes (e.g., subclasses reusing parent class code).
4. Polymorphism: Allowing objects of different classes to respond to the same method (via inheritance or interfaces).
5. Abstraction: Simplifying complexity through abstract classes/interfaces.

Examples: Python, Java, C++, C#.

* Python's Object-Centric Nature: In Python, absolutely everything is an object. This is a fundamental characteristic of the language and a core design principle.

1. Numbers: Integers, floats, complex numbers are objects.
2. Strings: Textual data is represented as string objects.
3. Lists, Tuples, Dictionaries, Sets: These are built-in container types and are all objects.
4. Functions: Functions are first-class objects in Python. You can assign them to variables, pass them as arguments to other functions, and even return them from functions.
5. Classes and Modules: Classes themselves are objects (instances of metaclasses), and modules are also objects.
6. Even None: None, which represents the absence of a value, is an object of the NoneType class.
7. Types/Classes: In Python, types (like int, str, list) are also objects (they are instances of the metaclass type).

We can use “type” function in Python to check the type of any entity. It will always return a class(which is itself an object).

* Important in Python:

1. Consistency: It creates a consistent and unified way to work with data and code. Everything behaves like an object, leading to a more predictable programming model.
2. Flexibility: Because functions and classes are objects, Python is highly dynamic and allows for powerful meta-programming techniques. You can inspect, modify, and create objects dynamically at runtime.
3. Object-Oriented Programming: This "everything is an object" nature is foundational to Python's object-oriented features. It makes it natural to work with classes, inheritance, and polymorphism, as all entities are treated as objects.

* <https://github.com/panaversity/learn-modern-ai-python/tree/main/00_python_colab/13_traditional_oop_part_2>
* Both contains same content

