python classes: the power of OOP

instantiation: creating an object from a class.

reminder → the class constructor accepts the same arguments as init ( ) (Except for self)

Naming Conventions

Public vs Non-Public Members

✓ in python, all attributes are accessible in one way or another.

✓ However, python has a naming convention that you should use to say that an attribute or method is not intended for use from outside its containing class or object.

When to avoid using classes

1. storing data : use tuples & ...

2. Providing a single method, use functions.

two types of attributes :

1. instance attr : most common / a variable that we define inside an instance method, using the (self).

2. Class attr : A variable that we define in the class body directly but outside of any method.
   ↓ all the objects that we create from a particular class share the same class attr with the same original values.

reminder → class constructor : استفاده از class رو با هر باری که یادمون باشه init method

میسازیم ، ← object از اول ساخته میشه و اتفاق رخ بده.

نکته we try to avoid hard coding and rely on felxible codes.

You can access class attr using either the class or one of its instances.

You can modify a class attr, you must use the class not one of its attributes.

✓ type(self).num_instances ±1 = classname.num_instances ±1

✗ self.num_instances ±1 نشه

نکته • type( ) function returns the class or type of (self) برای type(self)

instance methods : methods that receive (self) as their first argument.

نکته • it's best practice to define all of the instance attr (→ characteristics) inside .__init__( )

you must access all instance attr through the (self) argument → holds reference to the current instance.

you can't access instance attr through the class (car.make) X

you can access it through its containing instance (ford_mustang.make) ✓

the .__dict__ Attribute.

✓ both classes and instances have .__dict__

✓ each key in dict represents an attr name.

key : value بم اين صورت

dunder methods: Allow instance of a class to interact with the built-in functions and operators. __init__

__str__

__dict__

Dynamic class and instance attributes

in python, we can add new attr to your classes and instances dynamically

john_record = Record()

setattr() : sequentially add each feild as an attribute to your instance.

you can also use dot-notation and an assignment

reminder => instance attr vs class attr

self.value = value shared_value = 0

Property and Descriptor_Based Attributes

* add function_like behavior on top of existing instance attr

and turn them into [managed attr]

* you have function_like behavior and attribute_access at the same time.

→ you can use either a property or a descriptor

* Setter: is a method, to set or update the value of a private or protected attribute in a class.

it works with a getter to control how a specific attr is modified, allowing for validation or preprocessing of the input value.

@property: you typically use the @property to write the getter method.

it must return the value of the attribute.

setter

مثلا : to define the setter method of a property_based attribute, you to use the decorator @attr_name.setter

نکته : the property setters need to take an argument providing the value that you want to store in the underlying attribute.

python descriptors : for function like behavior on top of your instance attr.

```
import math

class PositiveNumber:
    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError("positive number expected")
        instance.__dict__[self._name] = value
```

`__set_name__` → called automatically when the descriptor is assigned as a class attr.

owner → descriptor : اون class که داریم ازش استفاده میکنیم

name → class تو اون attribute name

`self._name = name` → Stores the attribute name for later → radius, side

`__get__` → attribute's value رو میده(?)

`[self._name]` → fetches the actual value

`__set__` → sets the attribute's value and enforces constraints.

checks if value is a positive number

stores the validated value in the instance's __dict__

Lightweight classes with .__slots__

1- it reduce the memory footprint of the corresponding instances.

2- instances of the class don't have a .__dict__ ⇒ for making them memory_efficient.

3- it accepts a tuple of allowed attributes.

Providing Behavior with Methods.

reminder → class { attribute : data / method : behavior _ function inside a class

function : take arguments / return values

این بخش عینا بالایی مثال توضیح داده شده(?) ✓

3 types of methods :

1- instance methods → takes (self) as its first argument

2- class methods → takes the current class as its first argument

3- static methods → takes neither the instance nor the class as an argument.

مثال : car.start( ) → car: class, start: استارت

ford_mustang.start( ) → ford_mustang: instance

درسته ✓

special methods and protocols

special methods : dunder methods

* they are typically instance methods

for string representation ⟹ .__str__() and .__repr__()

str : informal string representation — str(instance) — print(instance)

repr : formal string representation — instance — repr(instance) → also used for debugging and logging

python protocols ⇒ related to special methods

Protocol	Provided Feature	Special Methods
Iterator	Allows you to create iterator objects	`.__iter__()` and `.__next__()`
Iterable	Makes your objects iterable	`.__iter__()`
Descriptor	Lets you write managed attributes	`.__get__()` and optionally `.__set__()`, `.__delete__()`, and `.__set_name__()`
Context manager	Enables an object to work on `with` statements	`.__enter__()` and `.__exit__()`

* yeild keyword : is used in a function to make it a generator, instead of returning a single value and exiting, a generator yeilds multiple values, one at a time. as the function is iterated over.

example of iteration context. for loops, list(), tuple(), dict(), these contexts implicitally call an object's __iter__() method

while loop is not an iteration context.

type(self).__name__ : Retrieves the name of the class of the object dynamically.

it means class name is determined at runtime, based on the type of the current object (self)

useful in inheritence scenarios : will return the actual name of the class of the specific object being used, not just the name of the calss where the method is defined.

Static Methods with @staticmethod

→ they don't take self or cls as an argument

→ regular functions defined within a class

→ when do we use them? when the function is closely related to the class, so we do this instead of a regular function outside the class

→ when calling a method, specify the class or object that provides that method.

→ they don't affect the class or its instances

Getter and setter methods vs properties

in python. using property method and @attribute.setter is more common than using something like .get_name() and .set_name()

→ for working with a sensetive data for example

→ گفتن به کاربر خواستیم روی یکی از instance attribute ها بزنیم

Standard library

Data classes ⇒ it provides init, repr, eq, and hash, iter and..

گفتن: وقتی از @dataclass استفاده بشه کار رو راحت‌تر می‌کنه یک‌سری چیزهای از پیش تعیین
شده داره.

Enumerations ⇒ like fixed attributes.

using class diagrams to showcase different types of relationship

Composition : a strong "has_a" ⇒ a robot has an arm, if the robot stops existing, the arm will stops existing too.

Aggregation : a softer "has_a" ⇒ a university has an instructor, if the university stops existing, the instructor does not stop existing.

Association : "uses_a" ⇒ a student is associated with a course, they will use the course.

common in one-to-one, many-to-many, one-to-many

Extended VS Overriden Methods

when using inheritence, there are two strategies to deal with methods.

1. extending ⇒ reuse the functionality provided and add new functionality on top
2. overriding ⇒ provide a completely new functionality

Method Resolution Order (MRO)

python searches for methods and attributes in the following order:

1- the current class D

2- the leftmost superclasses B

3- the superclass listed next, from left to right, up to the last superclass. C

4- the superclasses of inherited classes A

5- the object class object

Python mro.py

```
class A:
    def method(self):
        print("A.method()")

class B(A):
    def method(self):
        print("B.method()")

class C(A):
    def method(self):
        print("C.method()")

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

D.__MRO__ → it is an special attribute, that's why it doesn't come with ( ).

Mixin classes

→ provides methods that you can reuse in many other classes.

→ they don't define new types so, they're not intended to be instantiated but only inherited.

→ use their functionality to attach extra features to other classes quickly.

→ One of the ways to access the functionality of a mixin class is inheritence.

→ place mixins before the base class (Because of mro)

reminder → attributes: hold data ( ) ✗
method: perform a function or behavior ( ) ✓

Using Alternatives to Inheritance

→ because Multipule inheritans is complex, we use alternatives as well.

1) Composition: "has-a" relationship.

→ you create complex objects by combining objects that will work as components

→ these components may not make sense as stand alone objects.

```python
class IndustrialRobot:
    def __init__(self):
        self.body = Body()
        self.arm = Arm()

    def rotate_body_left(self, degrees=10):
        self.body.rotate_left(degrees)

    def rotate_body_right(self, degrees=10):
        self.body.rotate_right(degrees)

    def move_arm_up(self, distance=10):
        self.arm.move_up(distance)

    def move_arm_down(self, distance=10):
        self.arm.move_down(distance)

    def weld(self):
        self.arm.weld()

class Body:
    def __init__(self):
        self.rotation = 0

    def rotate_left(self, degrees=10):
        self.rotation -= degrees
        print(f"Rotating body {degrees} degrees to the left...")

    def rotate_right(self, degrees=10):
        self.rotation += degrees
        print(f"Rotating body {degrees} degrees to the right...")

class Arm:
    def __init__(self):
        self.position = 0

    def move_up(self, distance=1):
        self.position += distance
        print(f"Moving arm {distance} cm up...")

    def move_down(self, distance=1):
        self.position -= distance
        print(f"Moving arm {distance} cm down...")

    def weld(self):
        print("Welding...")
```

you build IndustrialRobot out of its components

Arm and Body. has-a relationship

Delegation

→ "can-do" relationship

→ an object hands a task over to another object, which takes care of executing the task

→ the delegated object can exist independently from the delegator.

Dependency Injection

→ is a design pattern that you can use to achieve loose coupling.

→ you provide an object's dependencies from the outside, rather than inheriting or implementing them in the object itself.

→ to create flexible classes that are able to change their behavior dynamically.

self.body : body()

self.arm: arm()

از این استفاده میشه

```
        self.body = body
        self.arm = arm
```

به اینکلس بعدش پاس میدیم

```
robot = IndustrialRobot(Body(), Arm())
```

اینجا اون کلاس ها رو بعدش پاس میدیم.