1. Everything in Python is an Object

Python's philosophy: **"Everything is an object"**. This includes:

- Numbers, strings, lists, dicts
- Functions
- Modules
- Classes themselves

```
x = 42
print(type(x))         # <class 'int'>
print(isinstance(x, object))  # True
```

```
<class 'int'>
True
```

```
def hello():
    return "Hi"

print(type(hello))      # <class 'function'>
print(isinstance(hello, object))  # True
```

```
<class 'function'>
True
```

```
class MyClass:
    pass

print(type(MyClass))    # <class 'type'>
print(isinstance(MyClass, object))  # True
```

```
<class 'type'>
True
```

💡 **Tip:** Since classes are objects, you can:

- Pass classes as arguments to functions.
- Return classes from functions.
- Create classes dynamically using `type()`.

```
# Dynamic class creation
DynamicPerson = type("DynamicPerson", (object,), {"greet": lambda self: "Hello"})
p = DynamicPerson()
print(p.greet())  # Hello
```

```
Hello
```

2. Class Anatomy in Python

A class in Python consists of:

- **Attributes (fields)**: Variables associated with the object
  - Instance attributes (`self.name`)
  - Class attributes (`Person.species`)
- **Methods**: Functions inside classes
  - Instance methods (need `self`)
  - Class methods (use `cls`)
  - Static methods (don't use `self` or `cls`)
- **Special Methods**: Dunder methods (`__init__`, `__str__`, `__add__`, etc.)

```
class Car:
    wheels = 4  # class attribute

    def __init__(self, brand, color):
        self.brand = brand  # instance attribute
        self.color = color

    def drive(self):
        print(f"{self.brand} is driving.")

    @classmethod
    def car_info(cls):
        print(f"A car usually has {cls.wheels} wheels.")

    @staticmethod
    def honk():
        print("Beep beep!")
```

3. Object Creation & Initialization

- Python calls `__new__()` to **allocate memory**.
- Then `__init__()` **initializes the instance**.

💡 **Tip:** You rarely need `__new__`, but it's useful for **singleton patterns** or **immutable objects** like `int` or `str`.

```
class Example:
    def __new__(cls, *args, **kwargs):
        print("Creating instance...")
        return super().__new__(cls)

    def __init__(self, value):
        print("Initializing instance...")
        self.value = value

e = Example(10)
```

```
Creating instance...
Initializing instance...
```

4. Instance vs Class Attributes

- **Instance attributes**: Unique to each object
- **Class attributes**: Shared across all instances

```
class Student:
    school = "XYZ School"  # class attribute

    def __init__(self, name):
        self.name = name  # instance attribute

s1 = Student("Alice")
s2 = Student("Bob")

print(s1.school, s1.name)  # XYZ School Alice
print(s2.school, s2.name)  # XYZ School Bob

# Changing class attribute
Student.school = "ABC School"
print(s1.school)  # ABC School
```

```
XYZ School Alice
XYZ School Bob
ABC School
```

💡 **Trick:** Avoid mutable class attributes like lists or dicts unless intentional, because all instances share them.

```
class BadExample:
    items = []  # shared mutable list

a = BadExample()
b = BadExample()
```

```
a.items.append(1)
print(b.items)  # [1] → b shares the same list!
```

```
[1]
```

✅ **Fix**: Use instance attributes for mutable defaults.

```
class GoodExample:
    def __init__(self):
        self.items = []
```

5. Inheritance & Polymorphism

- **Single Inheritance**

```

class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):
    def speak(self):
        print("Bark")

d = Dog()
d.speak()  # Bark
```

```
Bark
```

- **Multiple Inheritance**

```
class Flyer:
    def fly(self):
        print("Flying...")

class Swimmer:
    def swim(self):
        print("Swimming...")

class Duck(Flyer, Swimmer):
    pass

d = Duck()
d.fly()   # Flying...
d.swim()  # Swimming...
```

```
Flying...
Swimming...
```

💡 **Tip:** Python uses **Method Resolution Order (MRO)** to decide which method to call. Check it using:

```
print(Duck.mro())
```

```
[<class '__main__.Duck'>, <class '__main__.Flyer'>, <class '__main__.Swimmer'>, <class 'object'>]
```

6. Encapsulation & Name Mangling

- **Public** → normal attributes
- **Protected** → prefix `_` (convention, not enforced)
- **Private** → prefix `__` (name mangling)

```
class Secret:
    def __init__(self):
        self.public = "visible"
        self._protected = "semi-hidden"
        self.__private = "hidden"
```

```
s = Secret()
print(s.public)       # visible
print(s._protected)   # semi-hidden
# print(s.__private)  # AttributeError
print(s._Secret__private)  # hidden (name mangling)
```

```
visible
semi-hidden
hidden
```

💡 **Trick:** Name mangling avoids accidental overrides in subclasses.

```
Start coding or generate with AI.
```

7. Property Decorators (`@property`)

Pythonic way to create **getters and setters**.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

c = Circle(5)
print(c.radius)  # 5
c.radius = 10    # OK
# c.radius = -3  # ValueError
```

```
5
```

💡 **Tip:** Use `@property` to keep a clean API without exposing internal attributes.

8. Special Methods (Magic / Dunder Methods)

Method	Purpose
`__init__`	Constructor
`__new__`	Memory allocation
`__str__`	Human-readable string
`__repr__`	Official string representation
`__len__`	Support `len(obj)`
`__getitem__`	Indexing support `obj[key]`
`__setitem__`	Setting item `obj[key]=value`
`__add__`	Overload `+` operator
`__call__`	Make object callable like a function

Example: Custom Vector Class

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

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

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

v1 = Vector(1, 2)
```

```
v2 = Vector(3, 4)
print(v1 + v2)  # Vector(4, 6)
```

```
Vector(4, 6)
```

💡 **Trick:** Implementing `__repr__` properly makes debugging easier.

9. Classmethods & Staticmethods

```
class Temperature:
    scale = "Celsius"

    @classmethod
    def set_scale(cls, new_scale):
        cls.scale = new_scale

    @staticmethod
    def c_to_f(c):
        return c * 9/5 + 32

Temperature.set_scale("Fahrenheit")
print(Temperature.scale)  # Fahrenheit
print(Temperature.c_to_f(0))  # 32
```

```
Fahrenheit
32.0
```

Rules:

- `@staticmethod` : No access to `cls` or `self`
- `@classmethod` : Access to class (`cls`) but not instance (`self`)

10. Metaclasses (Advanced)

- **Metaclasses** define **how classes themselves are created**.
- Everything is an object; classes are instances of `type` .

```
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
```

```
Creating class MyClass
```

💡 **Tip:** Metaclasses are powerful for **frameworks, ORM models**, or **automatic registration of classes**.

11. Python Tips & Tricks with Classes

1. **Dynamic attributes**

```
class Person: pass
p = Person()
p.name = "Alice"  # Add attribute at runtime
```

2. **Dynamic methods**

```
def greet(self):
    print("Hello!")
```

```
import types
p.say_hello = types.MethodType(greet, p)
p.say_hello()  # Hello!
```

```
Hello!
```

3. **Using `__slots__`** to save memory for many objects

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

4. **Callable objects**

```
class Adder:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return x + self.n

add5 = Adder(5)
print(add5(10))  # 15
```

```
15
```

5. **Singleton pattern**

```
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance

a = Singleton()
b = Singleton()
print(a is b)  # True
```

```
True
```

12. Summary

- **Classes = Blueprints**, objects = instances.
- **Everything in Python is an object**, even classes.
- **Attributes**: Instance vs Class
- **Methods**: Instance, Class, Static
- **Encapsulation**: Public, Protected, Private
- **Magic Methods**: Overload operators, indexing, calling
- **Inheritance**: Single & Multiple
- **Metaclasses**: Customize class creation
- **Tips/Tricks**: Dynamic attributes, `__slots__`, callable objects, singletons.