OOP & paradigms

Python Reference

Classes & objects

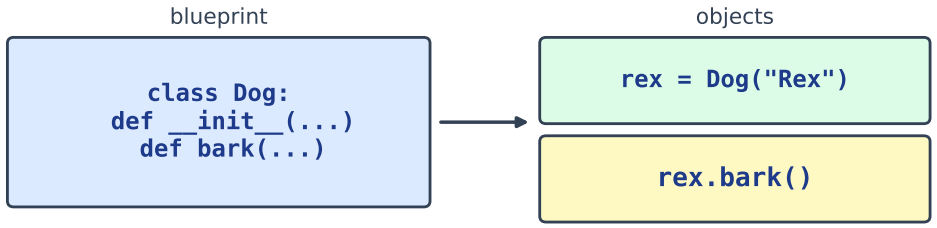
A class is a blueprint. An object is one thing built from it (an instance). `__init__` is the constructor that sets up each object; `self` is the object itself.

```
class Dog:
    def __init__(self, name):
        self.name = name            # an attribute
    def speak(self):
        return self.name + " says woof"

d = Dog("Rex")
print(d.speak())     # Rex says woof
```

- `name` is an attribute (data on the object); `speak` is a method (an action).

Add `__str__` to control what `print(obj)` shows:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"Dog named {self.name}"

print(Dog("Rex"))    # Dog named Rex
```



Exam tip: class defines the type; calling ClassName(...) creates an object (instance)

A class is a blueprint; calling it creates an object

Inheritance, encapsulation & polymorphism

Inheritance lets a subclass reuse a superclass. Use `super()` to call the parent; override a method to change it.

```
class Animal:
    def speak(self):
        return "some sound"

class Cat(Animal):
    def speak(self):                # override
        return "meow"

print(Cat().speak())   # meow
```

Encapsulation groups data with methods and exposes an intended interface. In Python, a leading underscore marks an intended non-public (often called private) name by convention; it does not prevent outside code from accessing it. Double-leading-underscore name mangling helps avoid accidental subclass name clashes, not enforce security.

```
class Account:
    def __init__(self):
        self._balance = 0           # non-public by convention
    def deposit(self, n):
        self._balance += n
    def balance(self):
        return self._balance

a = Account()
a.deposit(50)
print(a.balance())     # 50
```

Polymorphism means one name, many behaviours – the right `speak` runs for each object.

```
class Cat:
    def speak(self):
        return "meow"

class Cow:
    def speak(self):
        return "moo"

for animal in [Cat(), Cow()]:
    print(animal.speak())   # meow, then moo
```

Programming paradigms

A paradigm is a style of writing programs. Procedural code is a sequence of steps and functions. Object-oriented code groups data and methods into objects. Declarative code says **what** you want, not how (a list comprehension or SQL).

```
def total(nums):           # procedural
    t = 0
    for n in nums:
        t += n
    return t
print(total([1, 2, 3]))    # 6

print(sum([1, 2, 3]))      # 6  (declarative: same result)
```

Common mistakes

- Every method needs `self` as its first parameter.
- `__init__` sets up a new object and runs automatically when you create one.
- Two objects of the same class have separate attributes; changing one does not change the other.