

A class is a plan for a kind of thing: what data it holds and what it can do. An object is one real thing made from that plan. Most examples use a small bank `Account` class.

A CLASS AND AN OBJECT

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

__init__ runs once, when you make an object: it sets the
object's starting data

acc = Account("Ada", 100)
acc.owner
→ 'Ada'
call the class like a function to make an object

acc.deposit(50)
→ 150
a method is a function that belongs to the class;
balance is an attribute (data on the object)

Account("Grace").balance
→ 0
each object has its own data; Grace gets the default
balance, 0
```

SELF: THE OBJECT ITSELF

Inside a method, `self` is the object the method was called on. `self.balance` means "this object's balance". Every method takes `self` first, but you never pass it yourself.

```
Account.deposit(acc, 50)
→ 150
this is what acc.deposit(50) really runs: acc is passed
as self
```

PRINTING AN OBJECT

```
def __repr__(self):
    return f"Account({self.owner!r})"

def __str__(self):
    return f"Owner: {self.owner}"

put these two methods inside the Account class

acc
→ Account('Ada')
__repr__ is for programmers: the >>> prompt and lists
use it

str(acc)
→ 'Owner: Ada'
__str__ is for people: print() and f-strings use it
```

CLASS AND INSTANCE ATTRIBUTES

```
class Account:
    bank = "Northwind"    # shared by all
    def __init__(self, owner):
        self.owner = owner # one per object
a, b = Account("Ada"), Account("Linus")

a.bank, b.bank
→ ('Northwind', 'Northwind')
bank is set in the class body, so every object shares it

a.owner, b.owner
→ ('Ada', 'Linus')
owner is set on self, so each object has its own

Account.bank = "Eastgate"
b.bank
→ 'Eastgate'
change bank on the class: every object sees the new
value
```

@PROPERTY

```
class Temp:
    def __init__(self, celsius):
        self.celsius = celsius
    @property
    def fahrenheit(self):
        return self.celsius * 9 / 5 + 32
t = Temp(20)

t.fahrenheit
→ 68.0
a property is a method that you read like an attribute,
without ()

t.celsius = 30
t.fahrenheit
→ 86.0
the value is calculated again each time you read it

t.fahrenheit = 50
AttributeError: a property is read-only unless you also
write a @fahrenheit.setter
```

@CLASSMETHOD AND @STATICMETHOD

```
@classmethod
def from_text(cls, text):
    owner, balance = text.split(",")
    return cls(owner, int(balance))

@staticmethod
def is_valid(amount):
    return amount > 0

two more methods inside Account: the line above each
def says which kind it is

Account.from_text("Hedy, 250").balance
→ 250
classmethod: gets the class as cls, not an object. Use it
for another way to make an object

Account.is_valid(-5)
→ False
staticmethod: gets no self and no cls. It is a plain
function that you keep in the class
```

INHERITANCE AND SUPER()

```
class Savings(Account):
    def __init__(self, owner, rate):
        super().__init__(owner)
        self.rate = rate

    def add_interest(self):
        gain = self.balance * self.rate
        return self.deposit(gain)

Savings is a child class of Account, so it gets all of
Account's methods. super().__init__ runs Account's
__init__

s = Savings("Margaret", 0.02)
s.deposit(1000)
s.add_interest()
→ 1020.0
deposit() comes from Account

isinstance(s, Account)
→ True
a Savings object is also an Account
```

@DATACLASS: THE SHORT WAY

```
from dataclasses import dataclass

@dataclass
class Order:
    item: str
    qty: int
    price: float = 0.0

Order("tea", 2, 1.5)
→ Order(item='tea', qty=2, price=1.5)
@dataclass writes __init__ and __repr__ from the fields

Order("tea", 2) == Order("tea", 2) → True
objects with the same field values are equal

Order("pie", 1).price
→ 0.0
fields with a default go after fields without one
```

SPECIAL METHODS

Names with two underscores on each side let your objects work with Python's own operators and functions.

```
class Playlist:
    def __init__(self, name, songs):
        self.name, self.songs = name, songs
    def __len__(self):
        return self.songs
    def __eq__(self, other):
        return self.songs == other.songs
    def __lt__(self, other):
        return self.songs < other.songs

rock = Playlist("rock", 3)
jazz = Playlist("jazz", 2)

len(rock)
→ 3
__len__ sets the number that len() gives

rock == Playlist("x", 3)
→ True
__eq__ decides what == compares

[p.name for p in sorted([rock, jazz])]
→ ['jazz', 'rock']
sorted() needs only __lt__ (less than)
```

WATCH OUT

```
def deposit(amount):
    self.balance += amount
self is missing, so acc.deposit(5) raises TypeError

class Team:
    members = []
a, b = Team(), Team()
a.members.append("Ada")
b.members
→ ['Ada']
members is one list shared by every object: set
self.members = [] in __init__

acc.deposit
without (), you get the method itself and nothing runs

class Box:
    def __init__(self):
        return 5

Box()
→ raises TypeError
__init__ must not return a value: it only sets up the
object

class Point:
    def __init__(self, x):
        self.x = x
Point(1) == Point(1)
→ False
without __eq__, == asks "is this the same object?".
@dataclass adds __eq__
```