

# Python for Finance: Object-Oriented Programming

Amedeo Andriollo

Finance Department (DRM)  
Université Paris Dauphine - PSL

**Why classes?**

# Why might a dictionary be inconvenient?

An investment **position** needs three pieces of information: symbol, quantity, and price. MKT value is:  $q \times p$ .

```
acme_data = {
    "symbol": "ACME",
    "quantity": 10,
    "price": 12.5,
}

beta_data = {
    "symbol": "BETA",
    "quantity": 4,
} # price is missing

def market_value(position):
    return position["quantity"] * position["price"]

print(market_value(acme_data))

try:
    print(market_value(beta_data))
except KeyError as error:
    print("Missing field:", error)
```

```
125.0
Missing field: 'price'
```

# What are a class and an instance?

- The calculation requires both quantity and price, but the dictionary itself does not require those keys to be present.
- We could add checks around every operation, but then the rules describing a position would be spread across different parts of the program.
- One definition that says what information a position should contain and which operations belong to it.
- A **class** is a single definition that describes a type of object (info that objects of that type should store and the operations that they can perform).
- An **instance** is one particular object created from that class.
- We will define a class called `Position` that describes the information and operations for an investment position.

| Instance      | Symbol | Quantity | Unit price |
|---------------|--------|----------|------------|
| ACME position | ACME   | 10       | 12.5       |
| BETA position | BETA   | 4        | 30.0       |

- Both objects will be created from the same `Position` class.
- The class definition is shared; the values stored by each instance can differ.

# How do we create an instance from a class?

We have defined the idea of a class and an instance. The smallest possible `Position` class is:

```
class Position:
    pass

first = Position()
second = Position()

print(isinstance(first, Position))
print(first is second)
```

```
True
False
```

- `class Position:` starts the definition of a new class called `Position`.
- `pass` does nothing. It simply allows us to leave the class empty for the moment.
- `Position()` creates an instance of that class. Calling it twice creates two different objects.
- `isinstance(first, Position)` confirms that `first` was created from `Position`. The expression `first is second` is `False` because they are not the same object (but belong to the same class).
- These objects are still empty: they do not yet contain a symbol, quantity or price. We therefore need a way to give each new instance its starting data.

# How does a new instance receive its data?

Every position should start with three values: a symbol, a quantity and a price.

`__init__` is a special method that Python calls automatically when a new instance is being initialised.

```
class Position:
    def __init__(self, symbol, quantity, price):
        self.symbol = symbol
        self.quantity = quantity
        self.price = price
position = Position("ACME", 10, 12.5)
print(position.symbol)
print(position.quantity)
print(position.price)
```

```
ACME
10
12.5
```

- `Position("ACME", 10, 12.5)` supplies the three values needed to initialise this particular position.
- In `self.symbol = symbol`, the value on the right, `symbol`, is the argument we supplied. The expression on the left stores that value on the new object. The same happens for `quantity` and `price`.
- After the object has been created, we can access the stored values through `position.symbol`, `position.quantity` and `position.price`.

# Why do methods need self?

The same class can create many different instances. When Python is running a method, it therefore needs to know **which instance** the method should read from or write to.

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)

print(acme.symbol, acme.quantity, acme.price)
print(beta.symbol, beta.quantity, beta.price)
```

```
ACME 10 12.5
BETA 4 30.0
```

- When `acme` (`beta`) is initialised, `self` refers to that new `ACME` (`BETA`) instance.
- Python supplies the instance automatically as the first argument. We supply the remaining values: `symbol`, `quantity` and `price`.
- This first parameter `self` means: *the particular instance this method is currently working with*.
- `self` is not a Python keyword. It is the standard name used for the first parameter of an **instance method**: a method that works with one particular instance.

# Each instance stores its own attributes

The values created by `__init__` are stored on each individual instance.  
We access them with the dot notation `object.attribute`.

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)

print(acme.price, beta.price)

acme.price = 13.0

print(acme.price, beta.price)
```

```
12.5 30.0
13.0 30.0
```

- `symbol`, `quantity` and `price` are **instance attributes**: values stored on one particular instance.
- ACME and BETA were created from the same class, but each instance stores its own attribute values.
- Changing `acme.price` changes only the ACME instance. The price stored by beta remains 30.0.
- The collection of values currently stored on an object is called its **state**.  
    ↪ Changing ACME's price changes the state of `acme`, not `beta`'s.

# Some information belongs to the class

So far, every attribute describes one particular position, however, we may want to store information about the **class as a whole**. For example, we may want to count how many `Position` objects have been created.

A **class attribute** is stored on the class itself and is shared across its instances. It is useful for information that belongs to the class as a whole rather than to one particular object.

*Adding an instance counter to `Position`:*

```
class Position:
    instances_created = 0
    def __init__(self, symbol, quantity, price):
        self.symbol = symbol
        self.quantity = quantity
        self.price = price

    Position.instances_created += 1
```

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)

print(acme.symbol, beta.symbol)
print(Position.instances_created)
```

```
ACME BETA
2
```

# Instance attributes and class attributes

We now have two different kinds of information:

- some values describe **one particular position**;
- other values describe the **Position class as a whole**.

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)

print(acme.symbol, beta.symbol)
print(Position.instances_created)
```

```
ACME BETA
2
```

- `acme.symbol` and `beta.symbol` can be different because `symbol` belongs to each individual object. It is an **instance attribute**.
  - The number of positions created does not describe ACME or BETA separately. It describes all objects created from `Position`.
  - We therefore store this counter once on the class: `Position.instances_created`. This is a **class attribute**.
- An **instance attribute** stores information about one particular object. A **class attribute** stores information associated with the class as a whole.

# What if every position needs its own price history?

Suppose we now want each Position to keep a list of its past prices. The history should belong to **one particular position**: ACME's history should be independent of BETA's. *Tempted to put the list on the class: (appending)*

```
class Position:
    price_history = []
    def __init__(self, symbol, quantity, price):
        self.symbol = symbol
        self.quantity = quantity
        self.price = price
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)
acme.price_history.append(12.5)
print(acme.price_history)
print(beta.price_history)
```

```
[12.5]
[12.5]
```

- ACME and BETA should have different histories, but both objects are using the same list.
  - The reason is that `price_history` was created as a **class attribute**.  
There is one list stored on Position, and both instances refer to that same list.
  - `append` changes the existing list. The change is therefore visible through both `acme` and `beta` (a **shared state**).
- Create the list as an **instance attribute** inside `__init__`: `self.price_history = []`

# A method can use an instance's attributes

Each `Position` already stores the information needed to calculate its market value: quantity and price. Rather than writing the calculation separately each time, we can define it as an operation that every `Position` object knows how to perform. *Adding the method inside* `Position`:

```
def market_value(self):  
    return self.quantity * self.price
```

```
acme = Position("ACME", 10, 12.5)  
beta = Position("BETA", 4, 30.0)  
  
print(acme.market_value())  
print(beta.market_value())
```

```
125.0  
120.0
```

- `market_value` is an **instance method**: a method that works with one particular instance.
- In `acme.market_value()`, `self` refers to `acme`, therefore using ACME's.
- The attributes store the information about a position.  
The methods define operations that use that information.
- This method only reads the object's attributes and returns a result. It does not change the object.

# A method can also change an object's state

Suppose that, in our example, a valid price must always be positive.

Instead of changing the price directly, we can define a method that first checks the proposed value and only then applies the change. Add inside `Position`:

```
def update_price(self, new_price):  
    if new_price <= 0:  
        raise ValueError("price must be positive")  
    self.price = new_price
```

```
acme = Position("ACME", 10, 12.5)  
print(acme.price)  
acme.update_price(13.0)  
print(acme.price)  
print(acme.market_value())
```

```
12.5  
13.0  
130.0.
```

- `update_price` first checks the value: it is assigned only if the new price is positive.
- Changing the values stored by an existing object changes its **state**.
- This method centralises the price check, but Python still allows direct assignment such as `acme.price = -1`.

# A position should also start in a valid state

The previous method checks a price *when it is updated*. But the same rule must already hold when the position is first created. For this example, we want every valid Position to satisfy:  $\text{quantity} \geq 0$  and  $\text{price} > 0$ .

```
def __init__(self, symbol, quantity, price):
    if quantity < 0:
        raise ValueError("quantity cannot be negative")

    if price <= 0:
        raise ValueError("price must be positive")

    self.symbol = symbol
    self.quantity = quantity
    self.price = price

    Position.instances_created += 1
```

- The checks happen **before** the attributes are stored. If a condition is violated, raise stops the initialization and Python reports a ValueError.
- A condition that is intended to remain true for every valid state of an object is called an **invariant**.  
↪ Here are  $\text{quantity} \geq 0$  and  $\text{price} > 0$ .
- `__init__` checks at creation. `update_price` checks the price rule when we later change the price.
- These checks describe how we intend the class to be used. As we saw earlier, direct assignment such as `acme.price = -1` can still bypass `update_price`.

# What happens when a validity check fails?

Two different checks: an invalid value when an object is created, or later when updated.

```
try:
    invalid = Position("ACME", -2, 12.5)
except ValueError as error:
    print("Creation rejected:", error)
acme = Position("ACME", 10, 12.5)
try:
    acme.update_price(0)
except ValueError as error:
    print("Update rejected:", error)
print(acme.price)
print(acme.market_value())
```

```
Creation rejected: quantity cannot be negative
Update rejected: price must be positive
12.5
125.0
```

- In the first case, the quantity check fails during `__init__`. The attempted position is therefore not successfully initialized. In the second case, `acme` already exists. The proposed price is checked before assignment, so `update_price(0)` is rejected.  $\hookrightarrow$  ACME's price is still 12.5 and its market value is still 125.
- A rejected operation should not leave the object partly changed. The check is performed first; the state changes only after the new value has been accepted.

# What should belong to an instance, to the class, or outside it?

---

Does it concern one particular object, the class as a whole, or neither?

| What do we need?                        | Where does it belong? | Why?                                                                                           |
|-----------------------------------------|-----------------------|------------------------------------------------------------------------------------------------|
| Symbol, quantity and price              | Instance attributes   | They describe one particular position.                                                         |
| Market value                            | Instance method       | Calculation on position's quantity and price.                                                  |
| Number of <code>Position</code> objects | Class attribute       | The count describes the class as a whole, not one particular position.                         |
| Format an unrelated date                | Plain function        | The operation does not need a <code>Position</code> object or the <code>Position</code> class. |

- Instance attributes and instance methods concern one particular object.
- A class attribute stores information associated with the class as a whole.
- Not every operation belongs inside a class. If it does not use the object's data or the class itself, a separate function may be clearer.
- Our `Position` now stores data, calculates market value and checks price changes.

**Displaying and comparing objects**

# How can an object have a useful representation?

Inspecting `Position` without defining how it should be represented: a default description. This tells us that the object is a `Position`, but not which position it is or which values it stores.

`__repr__` defines the string returned by `repr(object)`. Add inside `Position`:

```
def __repr__(self):  
    return (f"Position(symbol='{self.symbol}', "  
            f"quantity={self.quantity}, price={self.price})")
```

```
acme = Position("ACME", 10, 12.5)
```

```
print(repr(acme))  
print([acme])
```

```
Position(symbol='ACME', quantity=10, price=12.5)  
[Position(symbol='ACME', quantity=10, price=12.5)]
```

- `repr(acme)` asks the object for its representation, so Python calls `acme.__repr__()`.
- Our method returns a string containing the class name and the attributes that identify the position.
- Lists also use the representation of the objects they contain. Defining `__repr__` therefore makes `[acme]` informative as well.

# How can we give an object a readable description?

The previous slide defined a representation that is useful when inspecting the object *while programming*. `__str__` defines the string returned by `str(object)`. *Add inside* Position:

```
def __str__(self):  
    return (f"{self.symbol}: {self.quantity} units "  
           f"at {self.price:.2f} per unit")
```

```
acme = Position("ACME", 10, 12.5)  
print(str(acme))  
print(acme)  
print([acme])
```

```
ACME: 10 units at 12.50 per unit  
ACME: 10 units at 12.50 per unit  
[Position(symbol='ACME', quantity=10, price=12.5)]
```

- `str(acme)` calls `acme.__str__()` and returns the readable description we defined.
- `print(acme)` also uses that string, so the first two lines produce the same output.
- A list uses the representation of the objects it contains, so `[acme]` still uses `__repr__`.
- If `__str__` is not defined, Python falls back to `__repr__` when a string description is requested.

# What should `<` mean for two positions?

Python already knows how to compare numbers (`print(125 < 160)`). But Python does not know automatically what it should mean for one `Position` object to be “smaller than” another.

We must first decide: For two positions, one position is smaller than another when its **market value is lower**. Python uses the special method `__lt__` for the `<` operation.

```
def __lt__(self, other):  
    if not isinstance(other, Position):  
        return NotImplemented  
  
    return self.market_value() < other.market_value()
```

```
acme = Position("ACME", 10, 12.5)  
beta = Position("BETA", 4, 40.0)  
print(acme.market_value())  
print(beta.market_value())  
print(acme < beta)
```

```
125.0  
160.0  
True
```

- The method compares their market values:  $125 < 160$ , so the result is `True`.

# The comparison and how positions are sorted

We have defined what `<` means for two `Position` objects. Python can now use that comparison when sorting a list of positions.

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 40.0)
positions = [beta, acme]
ranked = sorted(positions)
print([position.symbol for position in positions])
print([position.symbol for position in ranked])
print([position.market_value() for position in ranked])
```

```
['BETA', 'ACME']
['ACME', 'BETA']
[125.0, 160.0]
```

- The original list starts with BETA, whose market value is 160.
  - `sorted(positions)` needs to decide which object should come first. Because `Position` defines `__lt__`, Python can compare the objects using the meaning we chose.
- `sorted` returns a **new list**. The original `positions` list is unchanged.

# What if we compare a Position with something else?

Our `__lt__` method compares two `Position` objects. What if the other object is not a `Position`?

```
acme = Position("ACME", 10, 12.5)
try:
    print(acme < 100)
except TypeError as error:
    print(type(error).__name__)
```

TypeError

```
def __lt__(self, other):
    if not isinstance(other, Position):
        return NotImplemented

    return self.market_value() < other.market_value()
```

- In `acme < 100`, `self` is a `Position`, but `other` is an integer.
- Our method only defines how to compare one `Position` with another `Position`. It therefore returns `NotImplemented` for this pair of object types.
- Returning `False` would say that the comparison is meaningful and that `ACME` is not smaller than 100.

**Objects working together**

# From one position to a portfolio

So far, the Position class represents **one position**.

## Information stored by one Position

- symbol: which asset the position refers to;
- quantity: how many units are held;
- price: the current price per unit;
- price\_history: the history of a position

## What one Position knows how to do

- check its starting values;
- calculate its own market value;
- check and apply a price update;
- compare its market value with another Position.

## A new problem

A real portfolio usually contains **several positions**.

If we want the total portfolio value, we need to:

1. keep the positions together;
2. ask each position for its market value;
3. add those values.

# How can a Portfolio contain several Positions?

One position is a Position object. A portfolio needs to keep **several of those objects together**. We can therefore give each Portfolio instance a list of positions.

```
class Portfolio:
    def __init__(self, name):
        self.name = name
        self.positions = []
    def add_position(self, position):
        if not isinstance(position, Position):
            raise TypeError("portfolio accepts Position objects")
        self.positions.append(position)
```

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)
portfolio = Portfolio("Core")
portfolio.add_position(acme)
portfolio.add_position(beta)
print(portfolio.name)
print([position.symbol for position in portfolio.positions])
```

```
Core
['ACME', 'BETA']
```

- `self.positions = []` creates an empty list for this particular portfolio. A different Portfolio instance would get its own list.

# How should a Portfolio calculate its total value?

The portfolio knows **which positions it contains**, but it does not need to know how the value of each position is calculated. *Add inside Portfolio:*

```
def total_value(self):  
    total = 0.0  
    for position in self.positions:  
        total += position.market_value()  
    return total
```

```
acme = Position("ACME", 10, 12.5)  
beta = Position("BETA", 4, 30.0)  
portfolio = Portfolio("Core")  
portfolio.add_position(acme)  
portfolio.add_position(beta)  
print(acme.market_value())  
print(beta.market_value())  
print(portfolio.total_value())
```

```
125.0  
120.0  
245.0
```

- The loop visits each Position stored in self.positions.
- For each one, the portfolio calls position.market\_value(). ACME returns 125 and BETA returns

# Does a Portfolio contain copies of its Positions?

When we add acme to a portfolio, Python does not automatically create a second object: the portfolio's list contains the **same object**

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)
portfolio = Portfolio("Core")
portfolio.add_position(acme)
portfolio.add_position(beta)
print(portfolio.positions[0] is acme)
print(portfolio.total_value())
acme.update_price(15.0)
print(acme.price)
print(portfolio.total_value())
print(beta.price)
```

```
True
245.0
15.0
270.0
30.0
```

- `portfolio.positions[0] is acme` is True. Both expressions refer to the **same Position object**.
- We then change that object through `acme.update_price(15.0)`.
- BETA is a different Position object, so its price remains 30.0.

# If we copy the list, do we also copy the Positions?

Suppose we make a copy of the list itself. This creates a new list, but it does **not** automatically create new copies.

```
acme = Position("ACME", 10, 12.5)
beta = Position("BETA", 4, 30.0)
positions = [acme, beta]
copied = positions.copy()
print(copied is positions)
print(copied[0] is positions[0])
```

```
False
True
```

- `copied is positions` is False: these are two different list objects. But `copied[0] is positions[0]` is True: the first element of both lists refers to the same ACME Position.

```
copied.pop()           # remove BETA from copied only
acme.update_price(15.0) # change the shared ACME object

print(len(positions), len(copied))
print(positions[0].price, copied[0].price)
```

```
2 1
15.0 15.0
```

- Removing BETA from `copied` changes only that list. The original `positions` list still contains two elements.

# Which object has the info needed for each task?

The `Position` and `Portfolio` classes do different jobs because they have access to different information.

| Task                                | Object that should do it | Why?                                                     |
|-------------------------------------|--------------------------|----------------------------------------------------------|
| Calculate the value of one position | <code>Position</code>    | It knows its own quantity and price.                     |
| Calculate the total portfolio value | <code>Portfolio</code>   | It knows which positions belong to the portfolio.        |
| Check whether a new price is valid  | <code>Position</code>    | The rule concerns the state of that particular position. |
| Display the final result            | Code using the objects   | It decides what information should be shown to the user. |

- A useful way to decide where a method belongs is to ask: **which object already has the information needed to perform this task?**
- For example, `Portfolio.total_value()` does not calculate `quantity * price`. It asks each `Position` for its `market_value()` and combines the results.
- This keeps the rule for valuing one position in one place: the `Position` class.

# Specialising an account

# A basic Account

We now consider a new example.

An **account** has an owner and a balance. In this simplified example:

- the `balance` is the amount currently held in the account;
- a `deposit` adds a positive amount to that balance;
- `monthly_credit()` returns the amount credited to the account for the month.

We first define a basic Account that does not earn interest.

```
class Account:
    def __init__(self, owner, balance):
        if balance < 0:
            raise ValueError("balance cannot be negative")

        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("deposit must be positive")

        self.balance += amount

    def monthly_credit(self):
        return 0.0
```

# What behaviour should a Savings Account share?

```
account = Account("Leo", 800.0)
account.deposit(50.0)
print(account.balance)
print(account.monthly_credit())
```

```
850.0
0.0
```

- The account starts with a balance of 800. After depositing 50, its balance becomes 850.
- For this basic account, `monthly_credit()` returns 0 because it does not earn monthly interest.

Now suppose we also want a `SavingsAccount`.

- It still needs an owner and a balance.
- It should use the same rule for deposits. But it also needs an interest rate.
- Its monthly credit should therefore be calculated differently.

## What is common, and what is different?

**Common:** owner, balance, balance check, deposit rule.

**Different:** a savings account also stores an interest rate and uses it to calculate the monthly credit.

# A SavingsAccount is a specialised Account

A savings account therefore does not represent a completely different type of object.

↪ It is a **specialised kind of Account**.

## Inheritance

**Inheritance** allows us to define a new class starting from an existing class.

The new class receives the attributes and methods of the existing class, and can then add or change what it needs.

In Python, we express this relationship as:

```
class SavingsAccount(Account):  
    ...
```

- Account is the **base class**: it contains the behaviour common to our account types.
- SavingsAccount is a **subclass**: it inherits that common behaviour from Account.
- A SavingsAccount needs one additional piece of information: its monthly interest rate.
- We therefore want to keep the existing Account setup and add only what is specific to a savings account.

# How can SavingsAccount reuse Account?

The initializer of `Account` already performs the common setup. A `SavingsAccount` needs the same setup, plus a monthly interest rate.

```
class SavingsAccount(Account):
    def __init__(self, owner, balance, monthly_interest_rate):
        super().__init__(owner, balance)

        if monthly_interest_rate < 0:
            raise ValueError("monthly interest rate cannot be negative")

        self.monthly_interest_rate = monthly_interest_rate
```

In this example, `super().__init__(owner, balance)` runs the initializer defined by `Account` for the same `SavingsAccount` object. After that common setup is complete, we add the information that belongs specifically to the savings account.

```
savings = SavingsAccount("Amina", 1000.0, 0.02)

print(savings.owner)
print(savings.balance)
print(savings.monthly_interest_rate)
```

```
Amina
1000.0
0.02
```

# A subclass can change one inherited method

A `SavingsAccount` can use most of the behaviour already defined by `Account` (e.g., the deposit rule). Monthly credit, however, is different:

- a basic `Account` returns a monthly credit of 0;
- a `SavingsAccount` returns  $\text{balance} \times \text{monthly interest rate}$ .

Add inside `SavingsAccount`:

```
def monthly_credit(self):  
    return self.balance * self.monthly_interest_rate
```

```
account = Account("Leo", 800.0)  
savings = SavingsAccount("Amina", 1000.0, 0.02)  
  
print(account.monthly_credit())  
print(savings.monthly_credit())  
  
savings.deposit(50.0)  
print(savings.balance)
```

```
0.0  
20.0  
1050.0
```

# Composition or inheritance?

We have now seen two different ways in which classes can be related.

| Question                                                        | Relationship                | Example                                    |
|-----------------------------------------------------------------|-----------------------------|--------------------------------------------|
| Does one object <b>contain or use</b> objects of another class? | <b>Composition:</b> “has-a” | A Portfolio has Position objects.          |
| Is one class a <b>specialised version</b> of another class?     | <b>Inheritance:</b> “is-a”  | A SavingsAccount is a specialised Account. |

- With **composition**, the objects remain different kinds of things. A Portfolio is not a Position; it contains and uses Position objects.
- With **inheritance**, the subclass represents a more specific version of the base class. A SavingsAccount is still an Account, but it adds an interest rate and changes the monthly credit calculation.
- Inheritance should therefore not be chosen simply because two classes happen to share some code. The important question is whether the “is-a” relationship makes sense.