Object Oriented Programming

- Object Oriented Programming

Object Oriented Programming is a programming paradigm that uses objects and classes. It is useful for creating reusable code, and it can also be used to create complex programs. Object Oriented Programming is done using the `class` keyword. Classes are used to create objects, which are instances of a class. Objects can have attributes and methods. Attributes are variables that belong to an object, and methods are functions that belong to an object.

```
class Asset:
    pass
```

Constructor

A constructor is a special method that is used to initialize an object. It is useful for creating objects with default values. Constructors are done using the `__init__` method. The `__init__` method has two arguments: `self` and `args`. The `self` argument is used to refer to the object itself, and the `args` argument is used to pass arguments to the constructor. The `__init__` method is called when an object is created.

```
class Asset:
    def __init__(self, name, price):
        self.name = name
        self.price = price

asset = Asset('Bitcoin', 50000)
```

Attributes

Attributes are variables that belong to an object. They are useful for storing information about an object. Attributes can be accessed using the `.` operator. Attributes can also be accessed using the `getattr` function. Attributes can be set using the `=` operator. Attributes can also be set using the `setattr` function. Attributes can be deleted using the `del` operator. Attributes can also be deleted using the `delattr` function.

```
asset.name # Get attribute
asset.price = 60000 # Set attribute
asset.type = 'Cryptocurrency' # Set attribute not defined in constructor
```

Methods

Methods are functions that belong to an object. Methods can be called using the `()` operator. Since they are functions they are defined using the `def` keyword and always contain the `self` argument first.

```
class Asset:
    ...

    def double_price(self):
        return self.price*2
```

Dunders (Magic methods)

Dunders are special methods that are used to avoid operator overloading. They are useful for creating objects that behave like built-in objects. Dunders are done using the `__` keyword. For example, the `+` operator can be used to add two numbers, but it can also be used to add two strings.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

v1 = Vector2D(1, 2)
v2 = Vector2D(3, 4)
v3 = v1 + v2
```

- Dunders (Magic methods)

Non-exhaustive list of dunders

```
__init__ # Constructor
__str__ # String representation
__add__ # Addition +
__sub__ # Subtraction -
__mul__ # Multiplication *
__truediv__ # Division /
__floordiv__ # Floor division //
__mod__ # Modulo %
__pow__ # Exponentiation **
__lt__ # Less than <
__le__ # Less than or equal to <=
__eq__ # Equal to ==
__ne__ # Not equal to !=
__gt__ # Greater than >
__ge__ # Greater than or equal to >=
```

Example: Portfolio Class

A portfolio consists of a list of assets. Each asset has a name (identifier) as well as a history of prices.

```
class Asset:

    self.mu = np.nan # Expected return
    self.sigma = np.nan # Volatility

    def __init__(self, name: str, price_history: pd.DataFrame):
        self.name = name
        self.price_history = price_history
        self.compute_mu()
        self.compute_sigma()

    def compute_mu(self):
        self.mu = self.price_history.pct_change().mean()

    def compute_sigma(self):
        self.sigma = self.price_history.pct_change().std()
```

Example: Portfolio Class (2)

```
class Portfolio:

    self.mu = np.nan # Expected return
    self.sigma = np.nan # Volatility

    def __init__(self, assets: List[Asset], weights: List[float]):
        self.assets = assets
        self.weights = weights
        self.compute_mu()
        self.compute_sigma()

    def compute_mu(self):
        self.mu = np.sum([asset.mu * weight for asset, weight in zip(self.assets, self.weights)])

    def compute_sigma(self):
        # Covariance matrix
        cov = np.cov([asset.price_history.pct_change().dropna() for asset in self.assets])
        # Weighted covariance matrix
        cov = np.diag(self.weights) @ cov @ np.diag(self.weights)
        # Portfolio volatility
        self.sigma = np.sqrt(np.diag(cov).sum())
```

Python OOP Exercise: Library Management System

Objective

- Practice fundamental OOP concepts:
  - Classes
  - Objects
  - Inheritance
  - Methods

Problem Statement

Design a basic **Library Management System** that allows a user to:

- Add books to the library
- View available books
- Borrow a book
- Return a book

Requirements: `Book` Class

`Book` Class

- **Attributes:**
  - `title` : The title of the book
  - `author` : The author of the book
  - `available` : Whether the book is available for borrowing
- **Methods:**
  - `__init__(self, title, author)` : Initializes the book with title and author, available by default
  - `__str__(self)` : Returns a string representation of the book

Requirements: `Library` Class

`Library` Class

- **Attributes:**
  - `books`: A list of `Book` objects
- **Methods:**
  - `__init__(self)`: Initializes the library with an empty list of books
  - `add_book(self, book)`: Adds a `Book` to the library's list
  - `display_books(self)`: Prints details of all books
  - `borrow_book(self, title)`: Borrows a book by title if available
  - `return_book(self, title)`: Returns a borrowed book, making it available again

Main Program

Interaction Steps

1. Create a `Library` object
2. Add at least three `Book` objects to the library
3. Display the list of available books
4. Allow the user to borrow and return books by title

Example Interaction

```
# Create the library and add books
my_library = Library()
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
book2 = Book("1984", "George Orwell")
book3 = Book("To Kill a Mockingbird", "Harper Lee")

my_library.add_book(book1)
my_library.add_book(book2)
my_library.add_book(book3)

# Display all available books
print("Available books:")
my_library.display_books()

# Borrow a book
print("\nBorrowing '1984'...")
my_library.borrow_book("1984")

# Try to borrow the same book again
print("\nAttempting to borrow '1984' again...")
my_library.borrow_book("1984")

# Return the book
print("\nReturning '1984'...")
my_library.return_book("1984")

# Display all available books after returning
print("\nAvailable books after returning:")
my_library.display_books()
```

Data Classes

Data classes are a way to create classes whose main purpose is to store data. Data classes are done using the `dataclass` decorator. When you work with dataclasses, the constructor, `__init__`, `__repr__`, `__eq__`, and `__hash__` methods are automatically generated for you. Data classes are useful for creating simple classes that only store data.

```
from dataclasses import dataclass

@dataclass
class Asset:
    name: str
    price: float

asset = Asset('Bitcoin', 50000)
```