Object Oriented Programming

There are 4 principles when it comes to object oriented programming, Where possible, it's important to try an implement each of the principles.

The ease of applying each of the principles depends on the language you are using, some languages like Java and C# are built around these concepts and make them really easy to implement, where a language like Python or JavaScript would require more thought to implement these techniques.

The most important techniques to keep in mind are `Encapsulation` and `Inheritance`, `Abstraction` and `Polymorphism` are not always easy to implement in languages like Python and JavaScript

Principles

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

+ Code + Text

Encapsulation

This is the underlying principle behind classe, the goal of this is to take related operations and attributes and contain then within a single unit.

Benefits

- Makes it easier to pass data between different functions
    - Instead of passing 5 different variables, we pass a single class object
- We can hide the complexity of our code from the user
    - The user will only need to see what we need them to see, reducing the chances of code misuse
- Allows use to enforce implementation rules
    - We can set parameters that are required so that the object can be created, allowing us to confidently pass the object knowing that we have complete data

1. Examine the List object

- Remember that all data types and variables are objects in python

Built in data types and data structures are a good way to see encapsulation in action as they put related operations and attributes together in order to make working with data easier, they also allow for easier transfer of information.

Methods from data types and data structures give us a good view of how encapsulation works, one example would be the `remove()` method from the `list` class. You might have used the `remove()` method to delete an item from a position, but have you ever thought of what this method actually does in order to be able to delete the value.

Have you ever thought of how you are still able to access index `1` after performing `my_list.remove(1)`. This is the point of encapsulation, it allows the user of the method to just use the method knowing that they will get their required output and not really have to know about the inner workings of the method.

```
# Implementing the List method

# Create list
my_list = [1,2,3]


# Print list
print("Original list:", my_list)

# Remove first item
my_list.remove(1)

# Print final list
print("After removing item 1:", my_list)
```

```
     Original list: [1, 2, 3]
     After removing item 1: [2, 3]
```

Assumptions about `remove()`

Every method has code behind it, in this section, we make some assumptions about how the `remove()` method might be running under the hood in our own custom implementation of the method.

```python
class CustomList():
    __list = []

    def __init__(self, list):
        self.__list = list

    def remove(self, position):

        # The original method throws an error if the position is out of range
        if position < 0 or position >= len(self.__list):
            raise IndexError("Index out of range")

        # The original method removes the element at the specified
        position -= 1

        for i in range(position, len(self.__list) - 1):
            self.__list[i] = self.__list[i + 1]

        del self.__list[-1]

    def __str__(self):
        return str(self.__list)

# Create the custom list object
my_list = CustomList([1, 2, 3, 4, 5])

# Print original List
print("Original list:", my_list)

# Remove item at 1st position
my_list.remove(1)

# Print updated list
print("Updated list:", my_list)
```

```
Original list: [1, 2, 3, 4, 5]
Updated list: [2, 3, 4, 5]
```

Inheritance

The ability for a class to take attributes and behaviour from another class.

Key Words

- Base/Parent Class
    - The class that is inherited from
- Child/Sub-Class/Derived-Class
    - The class that inherits the behaviour from the Base Class

Benefits

- We can write code once and reuse it in different places
- We can hide implementation details between derived classes

1. Inheriting From the List class

Every data structure in Python is a class, so like any class, we can inherit from it, this will allow us to make use of basic features while being able to tweak the way these features are implemented in our custom class.

For this example, we will be changing the way that values are added to our custom list when we use the `append()` method, instead of adding values to the end of the list, we will add values to the start of the list.

```
class CustomList(list):

    def append(self, __object) -> None:

        # Create the new list and add the new item to the front
        new_list = []
        new_list.append(__object)

        # Add the existing items to the list
        for i in range(len(self)):
            new_list.append(self[i])

        # Overwrite the old instance
        self.__init__(new_list)


# Initialize List
my_list = CustomList([1,2,3,4])

# Print Original list
print("Original list:", my_list)

# Append 5 to list
my_list.append(5)

# Print Updated list
print("Updated list:", my_list)
```

```
Original list: [1, 2, 3, 4]
Updated list: [5, 1, 2, 3, 4]
```

2. Using Existing Methods

Previously we inherited from the list and changed the way that the `append()` method works to fit what ever weird case we would need to add a new item to the first index of the list.

For this section, we will see that although we have one method in our list, we are still able to perform all of the other operations that come with the `list` class.

```python
class CustomList(list):

    def append(self, __object) -> None:

        # Create the new list and add the new item to the front
        new_list = []
        new_list.append(__object)

        # Add the existing items to the list
        for i in range(len(self)):
            new_list.append(self[i])

        # Overwrite the old instance
        self.__init__(new_list)


# Initialize list
my_list = CustomList([9, 1, 3])

# print original list
print("Original List:", my_list)

# Print sorted list
print("Sorted List:", sorted(my_list))

# Print the first index of the list
print("First Index:", my_list[0])

# Pop the list
print("Popped List:", my_list.pop())
```

```
Original List: [9, 1, 3]
Sorted List: [1, 3, 9]
First Index: 9
Popped List: 3
```

Abstraction

We should only show the user the details that they need to perform their operations and nothing more

Abstraction is a key concept for writing your code and is implemented in `Encapsulation` as one of the goals of `Encapsulation` is to hide the inner workings of a class and giving the user the information that they need.

Abstraction goes a little bit further and says that we should create blue prints for our classes which detail the methods and attributes that should be visible to the user, the user will then be able to pass these blueprints to any functions that need to use these operations.

Key Terms

- Abstract Class

    - A class that can contain concrete methods and method signatures as well as attributes
    - Cannot be instantiated, is can only be passed inherited.
    - Can be used as a parameter allow for any child objects to be passed
- Inheritance
    - A class that only contains method signatures and attributes
    - Cannot be instantiated, is can only be passed inherited.
    - Can be used as a parameter allow for any child objects to be passed

Benefits

- Makes it really easy to swap out dependencies using dependency injection

Notes

- Python does not directly support abstraction like how languages lie C# and Java would, we need to use the `abc` module and some decorators to do this

1. Create Abstract Class

In this section, we will create an abstract class that will will act as a blue print for a person.

This class will include both method signatures and concrete methods along with attributes to show the different ways that it can be applied.

```python
from abc import ABC, abstractmethod

# Abstract Class
class Person(ABC):

    @abstractmethod
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Concrete implementation
    @abstractmethod
    def walk(self):
        return f"{self.name} is walking."

    # Method signature
    @abstractmethod
    def run(self):
        pass

    @abstractmethod
    def eat(self):
        pass

# Child Class
class Child(Person):

    def __init__(self, name, age):
        super().__init__(name, age)

    def walk(self):
        return f"{self.name} is walking."

    def run(self):
        return f"{self.name} is running."

    def eat(self):
        return f"{self.name} is eating."

child_person = Child("John", 12)
```

1.1 Errors When using abstract class wrong

As stated above, you can not call an abstract class directly and you have to implement every method shown in the blueprint.

Lets take a look at the errors that we might get.

1. Direct implementation
2. Incorrect Child class

```
from abc import ABC, abstractmethod

# Abstract Class
class Person(ABC):

    @abstractmethod
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Concrete implementation
    @abstractmethod
    def walk(self):
        return f"{self.name} is walking."

    # Method signature
    @abstractmethod
    def run(self):
        pass

    @abstractmethod
    def eat(self):
        pass


# Implementing
person = Person("jack", 30)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 2
      1 # Implementing
----> 2 person = Person("jack", 30)

TypeError: Can't instantiate abstract class Person with abstract methods __init__, eat, run, walk
```

SEARCH STACK OVERFLOW

```
# Child Class (Left out the walk method)

class Child(Person):

    def __init__(self, name, age):
        super().__init__(name, age)

    def run(self):
        return f"{self.name} is running."

    def eat(self):
        return f"{self.name} is eating."

child_person = Child("John", 12)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 14
     11     def eat(self):
     12         return f"{self.name} is eating."
---> 14 child_person = Child("John", 12)

TypeError: Can't instantiate abstract class Child with abstract method walk
```

SEARCH STACK OVERFLOW

2. Interfaces

Python does not provide a direct way of implementing an interface, we would need to create another abstract class but without any concrete method implementation.

3. Power of Abstraction

In this section, we will be creating 3 abstract classes and a single function that takes the blueprint as a parameter, you will see that without changing anything in the class, we can get different outputs from just changing the child that is being passed.

This example might be a bit obselete because Python is not strongly typed, but in languages like C# and Java, it is a real gamechanger

1. Create classes
2. Create function
3. Pass the different classes to the function

```python
from abc import ABC, abstractmethod

# Abstract class
class Vehicle(ABC):

    @abstractmethod
    def move(self):
        pass

    @abstractmethod
    def reduce_speed(self):
        pass

    @abstractmethod
    def increase_speed(self):
        pass

    @abstractmethod
    def stop(self):
        pass

# Motorcycle class
class Motorcycle(Vehicle):

    def move(self):
        print("Motorcycle is moving")

    def reduce_speed(self):
        print("Motorcyclist squeezes brakes")

    def increase_speed(self):
        print("Motorcyclist rotates throttle")

    def stop(self):
        print("Motorcyclist retracts kick stand")

# Bicycle class
class Bicycle(Vehicle):

    def move(self):
        print("Cyclist pushes off")

    def reduce_speed(self):
        print("Cyclist squeezes brakes")

    def increase_speed(self):
        print("Cyclist increases pedals harder")

    def stop(self):
        print("Cyclist places bicycle on stand")

# Car class
class Car(Vehicle):

    def move(self):
        print("Car is moving")

    def reduce_speed(self):
        print("Driver steps on break pedal")

    def increase_speed(self):
        print("Driver steps on gas pedal")

    def stop(self):
        print("Driver put on the handbrake")


# Create the function

def vehicle_operation(vehicle: Vehicle):
    vehicle.move()
    vehicle.increase_speed()
    vehicle.reduce_speed()
    vehicle.stop()


# initialize classes
motorcycle = Motorcycle()
car = Car()
bicycle = Bicycle()

# pass motorcycle
print("Motorcycle")
vehicle_operation(motorcycle)

# Pass car
print("\ncar")
vehicle_operation(car)

# Pass bicycle
print("\nbicycle")
vehicle_operation(bicycle)
```

```
Motorcycle
Motorcycle is moving
Motorcyclist rotates throttle
Motorcyclist squeezes brakes
Motorcyclist retracts kick stand

car
Car is moving
Driver steps on gas pedal
Driver steps on break pedal
Driver put on the handbrake

bicycle
Cyclist pushes off
Cyclist increases pedals harder
Cyclist squeezes brakes
Cyclist places bicycle on stand
```

Polymorphism

A method call should be able to perform the same operation in different ways.

This principle basicially allows us to reuse the method name so that a similar operation can be performed but with different information being passed.

There are two main ways of making a method polymorphic

- Method overriding
- Method overloading