`Differences between functional programming and object-oriented programming`

`Example:`

Let's say we are building a program that models a bank account. In a procedural approach, we might define a module that contains functions to interact with the bank account. We might also define a global variable to store the current balance of the account, since we need to access and modify it from multiple functions.

Here's an example of how we might implement **the bank account module using procedural programming with global data**:

```
balance = 0

def deposit(amount):
    global balance
    balance += amount

def withdraw(amount):
    global balance
    if balance < amount:
        print("Insufficient funds")
    else:
        balance -= amount

def get_balance():
    global balance
    return balance
```

In this implementation, we define a global variable **`balance`** to store the current balance of the account. We also define three functions: **`deposit`**, **`withdraw`**, and **`get_balance`**, which modify and access the global **`balance`** variable.

Using an object-oriented approach to implementing the same functionality.

Instead of using **global data**, we would **encapsulate the data and behavior of the bank account within an object.** We would define **a class** to represent **the bank account**, and define **methods** on that **class to interact with the account**.

```
class BankAccount:
    def __init__(self):
        self.balance = 0

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

    def withdraw(self, amount):
        if self.balance < amount:
            print("Insufficient funds")
        else:
            self.balance -= amount

    def get_balance(self):
        return self.balance
```

In this implementation, we define a **`BankAccount` class** that contains the data and behavior of the bank account. We define an `__init__` method to initialize the balance to 0, and define three other methods: `deposit`, `withdraw`, and `get_balance`, which modify and access the `balance` attribute of the object.

To use this class, we would create an instance of the `BankAccount` class and call its methods:

```
account = BankAccount()
account.deposit(100)
account.withdraw(50)
print(account.get_balance())
>>> prints 50
```

In summary, **the procedural implementation uses a global variable to store the bank account balance, which can be modified by any part of the program.** The **object-oriented implementation encapsulates the balance within the `BankAccount` object, which can only be modified by calling methods on that object.** This makes the code more modular and easier to reason about.

Encapsulation is one of the fundamental concepts of object-oriented programming (OOP). It refers to the practice of bundling together data and functions that operate on that data within a single unit, called an object.

Encapsulation is important because it allows us to group related data and functions together in a way that makes sense. For example, if we were building a program that modeled a car, we might encapsulate all the data and functions related to the car's engine within an `Engine` object, and all the data and functions related to the car's transmission within a `Transmission` object.

By encapsulating the data and functions within an object, we can control access to that **data** and **functions**. We can specify which functions are allowed to access and modify the data, and which functions are not. *This helps to keep our code organized and makes it easier to maintain and modify.*

Encapsulation is often compared to a black box, where the internal workings of the object are hidden from the outside world. The only way to interact with the object is through its public interface, which consists of the functions that are exposed by the object. This provides a layer of abstraction that makes it easier to reason about the behavior of the program, since we don't need to know the details of how the object works internally.

In summary, encapsulation is the practice of bundling together data and functions that operate on that data within an object. This helps to control access to the data and functions, and provides a layer of abstraction that makes it easier to reason about the behavior of the program.

Objects: characteristics(name, address, phone number, hourly pay) and actions(sell item, take item)

Objects:

➔ attributes(color, size, style, price)

➔ methods(change_price)

Object-Oriented Programming (OOP) Vocabulary

- **class** - a blueprint consisting of methods and attributes
- **object** - an instance of a class. It can help to think of objects as something in the real world like a yellow pencil, a small dog, a blue shirt, etc. However, as you'll see later in the lesson, objects can be more abstract.
- **attribute** - a descriptor or characteristic. Examples would be color, length, size, etc. These attributes can take on specific values like blue, 3 inches, large, etc.
- **method** - an action that a class or object could take
- **OOP** - a commonly used abbreviation for object-oriented programming
- **encapsulation** - one of the fundamental ideas behind object-oriented programming is called **encapsulation**: you can combine functions and data all into a single entity. *In object-oriented programming, this single entity is called a class.* **Encapsulation** allows you to hide implementation details much like how the scikit-learn package hides the implementation of machine learning algorithms.