Why Use Object-Oriented Programming? (Part 1)

Imagine you are in the O&G industry and working with some "well" data.

You need a way to handle this information in your code in a consistent, maintainable, and scalable way.

This is where OOP comes into play.

The "Well" Class and Objects

Well "Class"

Well objects (Instances)

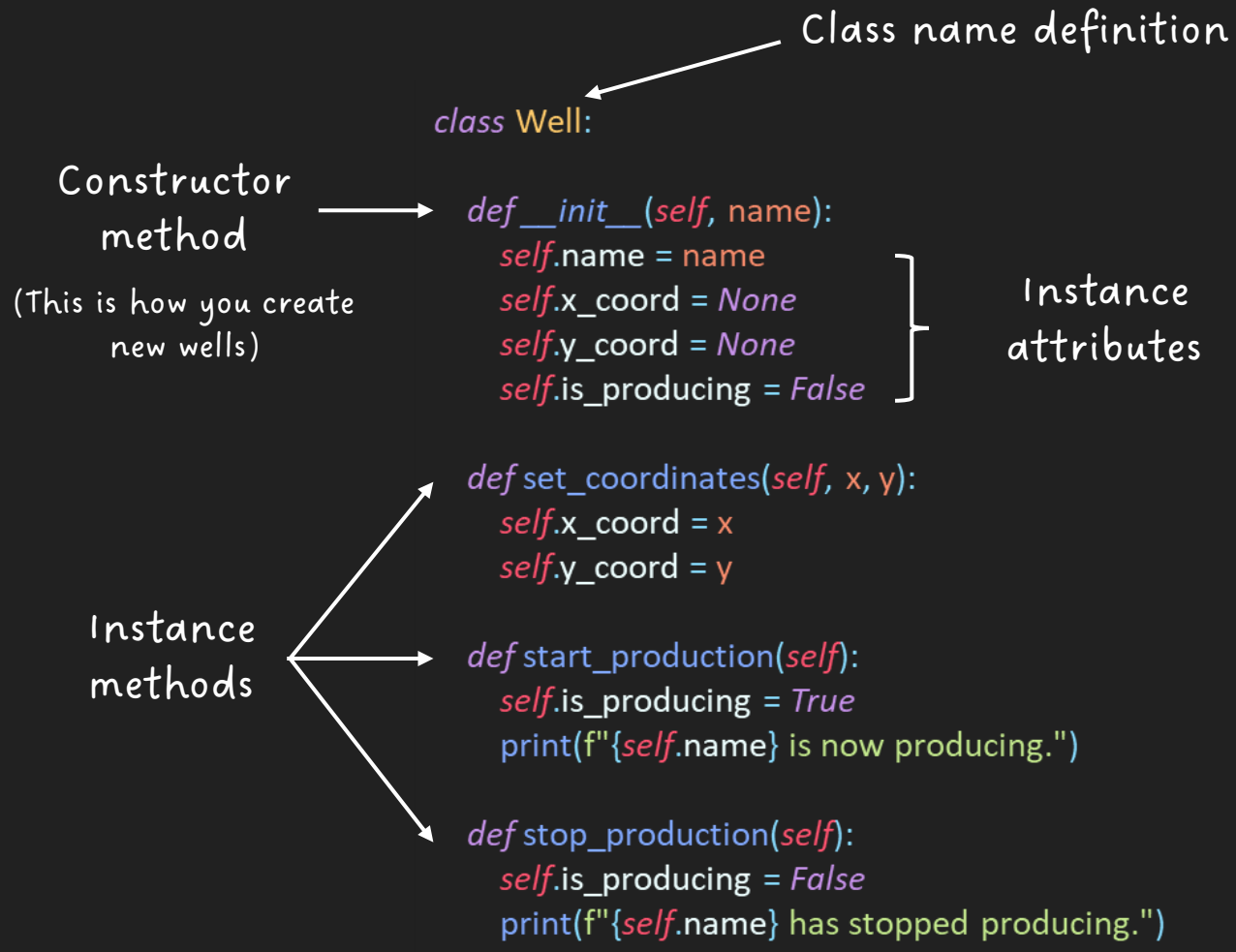
Characteristics (Attributes)

- Name
- Location (X, Y)
- TD
- Production data ...

What can it do? (Methods)

- Produce oil/gas/water
- Stop Producing
- Setup or create some well attributes ☝

Implement the "Well" class in Python




Oscar Cortez

oskrgab

The "self" keyword

```
class Well:

    def __init__(self, name):
        self.name = name
        self.x_coord = None
        self.y_coord = None
        self.is_producing = False

    def set_coordinates(self, x, y):
        self.x_coord = x
        self.y_coord = y

    def start_production(self):
        ...

    def stop_production(self):
        ...
```

In Python, self represents a class's specific instance, allowing access to its unique attributes and methods.

It ensures that each object maintains its state and behaviors and is passed automatically to instance methods when called.

While self is written explicitly in method definitions, it is implicit during method calls.

Definition

```python
class Well:

    def __init__(self, name):
        self.name = name
        self.x_coord = None
        self.y_coord = None
        self.is_producing = False

    def set_coordinates(self, x, y):
        self.x_coord = x
        self.y_coord = y

    def start_production(self):
        self.is_producing = True
        print(f"{self.name} is now producing.")

    def stop_production(self):
        self.is_producing = False
        print(f"{self.name} has stopped producing.")
```

Usage

```python
if __name__ == "__main__":

    well1 = Well("Well 1")
    well1.set_coordinates(10, 20)
    well1.start_production()
    well1.stop_production()

    well2 = Well("Well 2")
    well2.set_coordinates(30, 40)
    well2.start_production()
    well2.stop_production()

    print(well1.__dict__)
    print(well2.__dict__)
```


Oscar Cortez

oskrgab

Common mistakes when using OOP (Part 1)

- **Overusing Classes.** Sometimes, you just need a function or constants.
- **Creating too many properties and methods within the same class.** You can create multiple classes or leverage *inheritance*!
- **Confusing Class and Instance attributes.** In our example, we have just defined instance attributes. We'll see Class attributes later!