# Classes

## *Abstract Classes: Declare Methods without Implementation*

Sometimes you might want different classes to use the same attributes and methods. But the implementation of those methods can be slightly different in each class.

A good way to implement this is to use abstract classes. An abstract class contains one or more abstract methods.

An abstract method is a method that is declared but contains no implementation. The abstract method requires subclasses to provide implementations.

```python
from abc import ABC, abstractmethod

class Animal(ABC):

    def __init__(self, name: str):
        self.name = name
        super().__init__()

    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print(f'{self.name} says: Woof')

class Cat(Animal):
    def make_sound(self):
        print(f'{self.name} says: Meows')

Dog('Pepper').make_sound()
Cat('Bella').make_sound()
```

```
Pepper says: Woof
Bella says: Meows
```

# *classmethod: What is it and When to Use it*

When working with a Python class, if you want to create a method that returns that class with new attributes, use `classmethod`.

Classmethod doesn't depend on the creation of a class instance. In the code below, I use `classmethod` to instantiate a new object whose attribute is a list of even numbers.

```python
class Solver:
    def __init__(self, nums: list):
        self.nums = nums

    @classmethod
    def get_even(cls, nums: list):
        return cls([num for num in nums if num % 2 == 0])

    def print_output(self):
        print("Result:", self.nums)

# Not using class method
nums = [1, 2, 3, 4, 5, 6, 7]
solver = Solver(nums).print_output()
```

```
Result: [1, 2, 3, 4, 5, 6, 7]
```

```python
solver2 = Solver.get_even(nums)
solver2.print_output()
```

```
Result: [2, 4, 6]
```

# *getattr: a Better Way to Get the Attribute of a Class*

If you want to get a default value when calling an attribute that is not in a class, use `getattr()` method.

The `getattr(class, attribute_name)` method simply gets the value of an attribute of a class. However, if the attribute is not found in a class, it returns the default value provided to the function.

```python
class Food:
    def __init__(self, name: str, color: str):
        self.name = name
        self.color = color


apple = Food("apple", "red")

print("The color of apple is", getattr(apple, "color",
"yellow"))
```

```
The color of apple is red
```

```python
print("The flavor of apple is", getattr(apple, "flavor",
"sweet"))
```

```
The flavor of apple is sweet
```

```python
print("The flavor of apple is", apple.sweet)
```

```
---------------------------------------------------------------
-------------

AttributeError                                Traceback (most
recent call last)

/tmp/ipykernel_337430/3178150741.py in <module>
----> 1 print("The flavor of apple is", apple.sweet)
```

```
AttributeError: 'Food' object has no attribute 'sweet'
```

# \_\_call\_\_: *Call your Class Instance like a Function*

If you want to call your class instance like a function, add \_\_call\_\_ method to your class.

```python
class DataLoader:
    def __init__(self, data_dir: str):
        self.data_dir = data_dir
        print("Instance is created")

    def __call__(self):
        print("Instance is called")


data_loader = DataLoader("my_data_dir")
# Instance is created

data_loader()
# Instance is called
```

```
Instance is created
Instance is called
```

# Static method: use the function without adding the attributes required for a new instance

Have you ever had a function in your class that doesn't access any properties of a class but fits well in a class? You might find it redundant to instantiate the class to use that function. That is when you can turn your function into a static method.

All you need to turn your function into a static method is the decorator `@staticmethod`. Now you can use the function without adding the attributes required for a new instance.

```python
import re


class ProcessText:
    def __init__(self, text_column: str):
        self.text_column = text_column

    @staticmethod
    def remove_URL(sample: str) -> str:
        """Replace url with empty space"""
        return re.sub(r"http\S+", "", sample)


text = ProcessText.remove_URL("My favorite page is
https://www.google.com")
print(text)
```

```
My favorite page is
```

# Property Decorator: A Pythonic Way to Use Getters and Setters

If you want users to use the right data type for a class attribute or prevent them from changing that attribute, use the `property` decorator.

In the code below, the first `color` method is used to get the attribute `color` and the second `color` method is used to set the value for the attribute `color`.

```python
class Fruit:
    def __init__(self, name: str, color: str):
        self._name = name
        self._color = color

    @property
    def color(self):
        print("The color of the fruit is:")
        return self._color

    @color.setter
    def color(self, value):
        print("Setting value of color...")
        if self._color is None:
            if not isinstance(value, str):
                raise ValueError("color must be of type string")
            self.color = value
        else:
            raise AttributeError("Sorry, you cannot change a fruit's color!")


fruit = Fruit("apple", "red")
fruit.color
```

```
The color of the fruit is:
```

```
'red'
```

```
fruit.color = "yellow"
```

```
Setting value of color...
```

```
---------------------------------------------------------------
-------------

AttributeError                                 Traceback (most
recent call last)

/tmp/ipykernel_337430/2513783301.py in <module>
----> 1 fruit.color = "yellow"
```

```
/tmp/ipykernel_337430/2891187161.py in color(self, value)
     17              self.color = value
     18         else:
---> 19              raise AttributeError("Sorry, you cannot
change a fruit's color!")
     20
     21
```

```
AttributeError: Sorry, you cannot change a fruit's color!
```

# $\_\_str\_\_$ and $\_\_repr\_\_$: Create a String Representation of a Python Object

If you want to create a string representation of an object, add `__str__` and `__repr__`.

`__str__` shows readable outputs when printing the object. `__repr__` shows outputs that are useful for displaying and debugging the object.

```python
class Food:
    def __init__(self, name: str, color: str):
        self.name = name
        self.color = color

    def __str__(self):
        return f"{self.color} {self.name}"

    def __repr__(self):
        return f"Food({self.color}, {self.name})"


food = Food("apple", "red")

print(food)  # str__
```

```
red apple
```

```python
food  # __repr__
```

```
Food(red, apple)
```