Design Patterns - Python OOP Refresher

Let's do a quick refresher of the Python language (3.5+) with reference to Object Oriented Programming.

We will discuss the following main concepts:

1. Classes and Objects
2. Encapsulation
3. Inheritance
    a. Interface Contracts
    b. Abstract classes
4. UML Diagrams for Python OOP

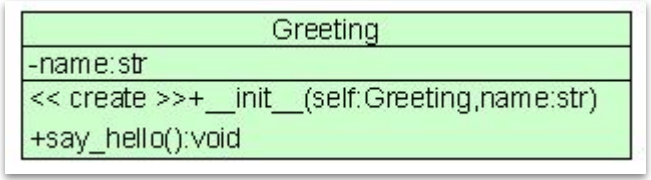
Design Patterns - Python OOP Refresher

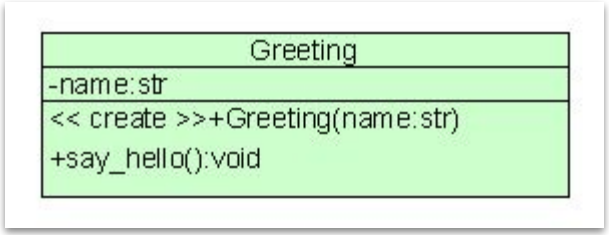
Classes and Objects

1. Classes are simple (or more complex) **recipes** that are used to create two main things:
   a. Provide a Data container
      i. Variables
      ii. Constants
   b. Provide operations on data
      i. Functions/Methods
2. We then can use the Classes to create instances of objects which hold the specific data that we can operate on.

```python
# Define the Greeting class
class Greeting:
    # Constructor for the Greeting class
    def __init__(self, name):
        # Initialize 'name' with the provided value
        self.name = name

    # Define our greeting method
    def say_hello(self):
        # Print a personalized greeting message
        # using the 'name' attribute
        print(f"Hello, {self.name}!")

# Create an object of the Greeting class,
# initializing it with the name 'John'
greeting = Greeting("John")

# Call the 'say_hello' method on the 'greeting'
# object to print the greeting message
greeting.say_hello()
```

Design Patterns - Python OOP Refresher






```
greeting = Greeting('John')
```

```
Greeting greeting = Greeting('John')
```

```python
# Define the Greeting class
class Greeting:
    # Constructor for the Greeting class
    def __init__(self, name):
        # Initialize 'name' with the provided value
        self.name = name

    # Define our greeting method
    def say_hello(self):
        # Print a personalized greeting message
        # using the 'name' attribute
        print(f"Hello, {self.name}!")

# Create an object of the Greeting class,
# initializing it with the name 'John'
greeting = Greeting("John")

# Call the 'say_hello' method on the 'greeting'
# object to print the greeting message
greeting.say_hello()
```

Design Patterns - Python OOP Refresher

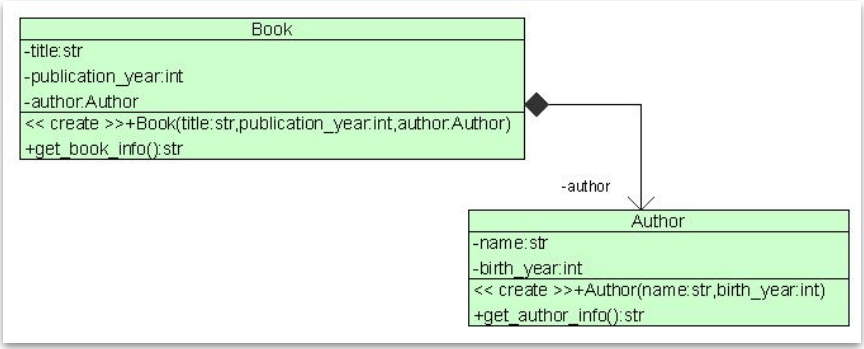
Encapsulation

In general a well designed **class already achieves encapsulation** in the sense that it **gathers** all the relevant **data** and **functionality**. It helps us with controlling complexity as well.
Do have these points in mind:

1. Create **classes <u>for all</u> the objects** you need in your code.
2. Create collections of related objects **<u>so that they can be treated as units</u>**.

```python
class Author:
  def __init__(self, name, birth_year):
      self.name = name
      self.birth_year = birth_year
  def get_author_info(self):
      return f"{self.name} (born {self.birth_year})"

class Book:
  def __init__(self, title, pub_year, author: Author):
      self.title = title
      self.publication_year = pub_year
      self.author = author

  def get_book_info(self):
      return f"'{self.title}' by
{self.author.get_author_info()}, published in
{self.publication_year}"

# Create an Author object
author_obj = Author("George Orwell", 1903)
# Create a Book object aggregating the Author object
book_obj = Book("1984", 1949, author_obj)

# Print the book information, with included author
print(book_obj.get_book_info())
```

Design Patterns - Python OOP Refresher




```python
class Author:
  def __init__(self, name, birth_year):
      self.name = name
      self.birth_year = birth_year
  def get_author_info(self):
      return f"{self.name} (born {self.birth_year})"

class Book:
  def __init__(self, title, pub_year, author: Author):
      self.title = title
      self.publication_year = pub_year
      self.author = author

  def get_book_info(self):
      return f"'{self.title}' by
{self.author.get_author_info()}, published in
{self.publication_year}"

# Create an Author object
author_obj = Author("George Orwell", 1903)
# Create a Book object aggregating the Author object
book_obj = Book("1984", 1949, author_obj)

# Print the book information, with included author
print(book_obj.get_book_info())
```

Design Patterns - Python OOP Refresher

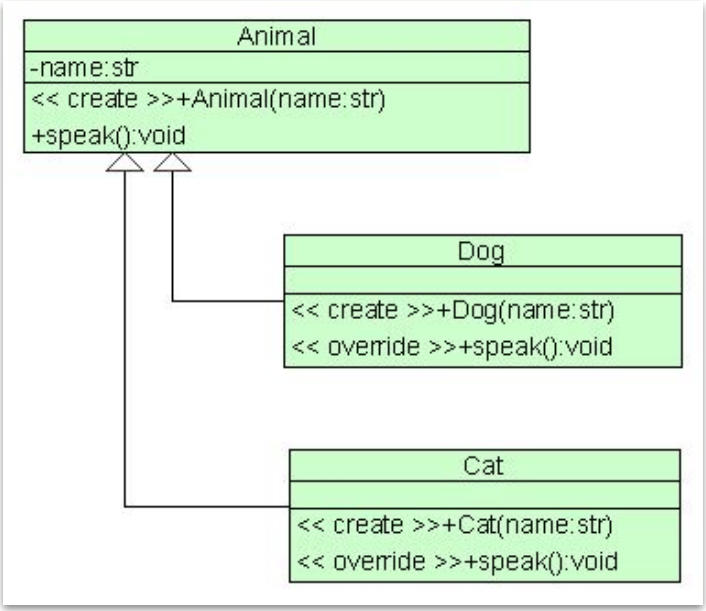
Inheritance

Allows us to generalize. We can think of it as a tree which grows more complex as we keep on extending its branches.

1. We start with some property or behaviour that is **present in different instances or types** of entities.
2. We then create new and more specialized versions of the **'parent'** by inheriting either data and/or behaviour in the **'children'.**

```python
# Base (parent) class
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(f"{self.name} makes a sound.")

# Derived (child) class
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

# Derived (child) class
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")

# Create objects of the derived c
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Call the 'speak' method on the d
dog.speak()
cat.speak()
```

Design Patterns - Python OOP Refresher




```python
# Base (parent) class
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(f"{self.name} makes a sound.")

# Derived (child) class
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

# Derived (child) class
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")

# Create objects of the derived c
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Call the 'speak' method on the d
dog.speak()
cat.speak()
```