Object Oriented Programming Concepts

(OOP for short)

What is OOP?

- A programming approach that breaks down a problem into objects and focuses on the interactions between objects

- Terminology:
  - **Class** – the code you write to define an object and its properties
  - **Object** – an instance of the class populated with specific state
  - **Attributes** or **data members** – hold data or "state" of the object
  - **Methods** or **member functions** – actions the object can perform

- Why Objects?
  - If we can keep the data and the operations that manipulate it together our code **should** be re-usable and easier to debug/maintain.

Four Pillars of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

~~Four~~ Three Pillars of OOP

- Encapsulation
  - Abstraction
- Inheritance
- Polymorphism

NOTE: Sometimes people may refer to this as the *Three Pillars* where Encapsulation includes Abstraction

Encapsulation

- The bundling of data with the methods that operate on that data
- Some definitions also include **information hiding**
  - hide the internal representation, or state, of an object from the outside
- Control access to data members or methods from other code
  - `private, public, protected` access specifiers (C++/C#/Java/etc.)
  - Python doesn't have access specifiers
- **Implementation level**

Abstraction

- Deciding how external code interacts with your object
- Represents the interface to your object
- Limits the amount of required implementation knowledge for use of an object
- What can an object do, not **HOW** the object do it
- **Design level**

Inheritance

- Deriving a new class that inherits the properties (data members and methods) of the already exist class
  - Base Class (parent) -> Derived Class (child)
- Supports the concept of code reusability and reduces the length of the code in object-oriented programming
- When one or more objects might be the same...but different
- With great power comes great responsibility (more on this later)
  - Consider the **“is a”** relationship

Inheritance Example

- Animal is the base class
- Dog is the derived class

- A **Dog** is a **Animal**

- Dog has all the members of Animal, but also can have its own functions like roll_over().

```
1   class Animal:
2       def __init__(self, sound):
3           self.sound = sound
4
5       def speak(self):
6           print(f"Animal says: {self.sound}")
7
8   class Dog(Animal):
9       def __init__(self):
10          super().__init__("Woof")
11
12      def speak(self):
13          print(f"Dog says: {self.sound}")
14
15      def roll_over(self):
16          print("Dog rolls over.")
17
18  def main():
19      my_chicken = Animal("Cluck")
20      my_chicken.speak()
21
22      my_dog = Dog()
23      my_dog.speak()
24      my_dog.roll_over()
25
26  if __name__ == "__main__":
27      main()
28
```

Polymorphism

- From Greek – “Many Forms”
  - The condition of occurring in several different forms
- Software Design
  - A single interface to entities of different types
- We get the same interface for different types
  - The code that runs, depends on the type

Static Polymorphism

- Determined at compile-time

- Occurs with:
  - Templates (C++)
  - Overloading (function and operator)

C/C++ Overloading

```
1   int library_write_open(int fd);
2   int library_write_open(FILE* file);
3   int library_write_open(const char* filename);
4   int library_write_open(char** buffer, size_t* size);
```

Static Polymorphism

- Determined at compile-time

- Occurs with:
  - Templates (C++)
  - Overloading (function and operator)
    - Limited support with Python modules

Python Overloading

```python
@singledispatch
def add(a, b):
    raise NotImplementedError(f"Unsupported type: {type(a)}")

@add.register(int)
def _(a, b):
    print(a + b)

@add.register(str)
def _(a, b):
    print(a + b)

@add.register(list)
def _(a, b):
    print(a + b)
```

Dynamic Polymorphism

- Determined at run-time
- Used with inheritance
- Derived class **overrides** a base class function
- Dog speak() overrides Animal speak()
  - You can call base class functions within the derived class using super() as in the Dog class __init__() function

```
1   class Animal:
2       def __init__(self, sound):
3           self.sound = sound
4
5       def speak(self):
6           print(f"Animal says: {self.sound}")
7
8   class Dog(Animal):
9       def __init__(self):
10          super().__init__("Woof")
11
12      def speak(self):
13          print(f"Dog says: {self.sound}")
14
15      def roll_over(self):
16          print("Dog rolls over.")
17
18  def main():
19      my_chicken = Animal("Cluck")
20      my_chicken.speak()
21
22      my_dog = Dog()
23      my_dog.speak()
24      my_dog.roll_over()
25
26  if __name__ == "__main__":
27      main()
28
```