**OOP - Dry Part**

**Basics**

**1.**

1. **Polymorphism** - concept that you can access objects of different types through the same interface.
2. **Object Oriented programming (OOP)** is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.
3. **Functional programming (also called FP)** is a way of thinking about software construction by creating pure functions. It avoids concepts of shared state, mutable data observed in Object Oriented Programming. Functional languages emphasis on expressions and declarations rather than execution of statements. Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function.

2.

1. **Composition vs Inheritance:**

1. **Composition** is the design technique in object-oriented programming to implement **has-a** relationship between objects. For instance, if a person(class) has a Job (Class). **Inheritance** is the design technique in object-oriented programming to implement **is-a** relationship between objects. For example, dog (Class) is an animal(Class)
2. **Pros composition**:

First, Inheritance is tightly coupled whereas composition is loosely coupled**.**

Second**,** there is no access control in inheritance whereas access can be restricted in composition.

Third, Composition provides flexibility in invocation of methods that is useful with multiple subclass scenario.

**Pros Inheritance:**

First, facilitates easy reusability of code from parent class without having to copy it.

Second, provides a clear, hierarchical structure so you can break a model into simple, accessible components.

1. **Mutability vs Immutability:**
   1. Mutable object can be changed after it's created, and an immutable object can't.
   2. Immutable types are safer from bugs, easier to understand, and more ready for change. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. Getting good performance is one reason why we use mutable objects. Another is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure.
2. **Static Typing vs Dynamic Typing:**
   1. A programming language is statically typed if the type of a variable is known at compile time. A language is dynamically typed if the type of a variable is checked during run-time.
   2. **Pros of static typing are:**

First, we’ll be able to fix a lot of errors even before running the program.

Second, the execution of the code will be faster compared to the dynamically typed languages.

**Pros of dynamic typing are:**

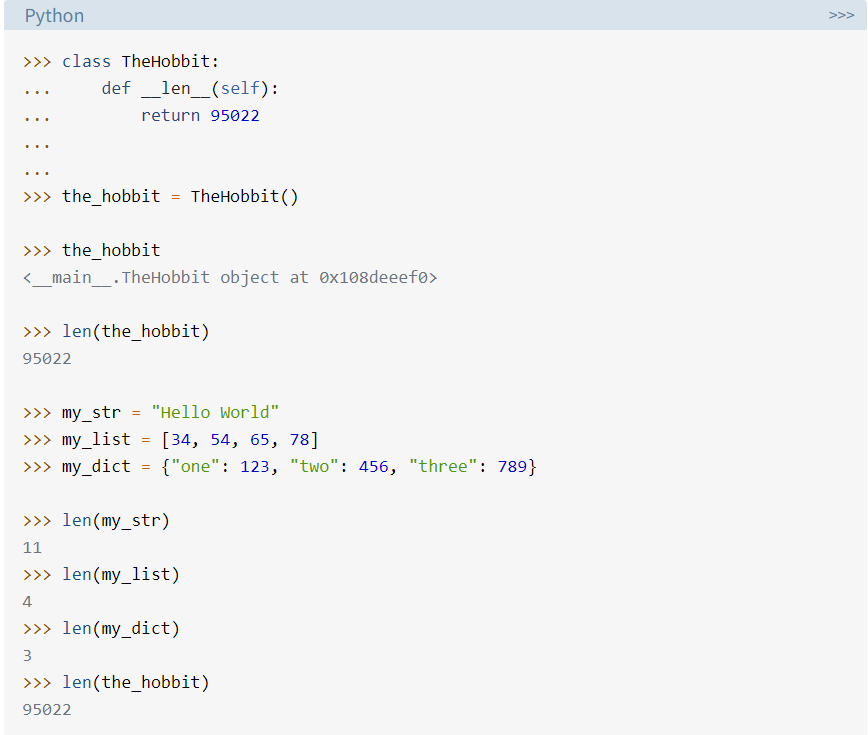
First, a dynamically typed language has the capability to identify the type of each variable during run-time.

Second, it is easy to write code without worrying about the type of variables. However, there are chances where you may find yourself searching for a bug that is due to the interpreter misinterpreting the type of a variable.

* 1. Statically typed languages are for example C++, C#. Dynamically typed languages are Python, JavaScript.

3.

Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute.

For example, you can call len() on any Python object that defines a .\_\_len\_\_() method:

4. **Public** - Access is not restricted.

**Protected** - Access is limited to the containing class or types derived from the

containing class.

**Private** - Access is limited to the containing type.

All members in a Python class are public by default. Any member can be accessed from outside the class environment.

Python's convention to make an instance variable protected is to add a prefix \_ (single underscore) to it. This effectively prevents it from being accessed unless it is from within a sub-class.

The double underscore \_\_ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError.

5. **Instance method** is the basic, no-frills method type you’ll use most of the time. The method can accept more than just one parameter self, which points to an instance of a Class when the method is called. instance methods can freely access attributes and other methods on the same object. The instance method acts on an object’s attributes. It can modify the object state by changing the value of instance variables.

**Class method** - Instead of accepting a self parameter, class methods take a cls parameter that points to the class—and not the object instance—when the method is called. Furthermore, it can’t modify object instance state. That would require access to self. However, class methods can still modify class state that applies across all instances of the class. It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances. We generally use the class method to create factory methods. Factory methods return class objects ( similar to a constructor ) for different use cases.

**Static method** - This type of method takes neither a self nor a cls parameter (but of course it’s free to accept an arbitrary number of other parameters). Therefore, a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they’re primarily a way to namespace your methods. Static methods are great signal to show that a particular method is independent from everything else around it. We generally use static methods to create utility functions.

**Design Concepts**

1. **SOLID** design principles:

**S**ingle Responsibility Principle - The idea behind the SRP is that every class, module, or function in a program should have one responsibility/purpose in a program. As a commonly used definition, "every class should have only one reason to change”.

Benefits:

1. it makes your software easier to implement and prevents unexpected side-effects of future changes.
2. In the end, you need to change your class more often, and each change is more complicated, has more side-effects, and requires a lot more work than it should have. So, it’s better to avoid these problems by making sure that each class has only one responsibility.
3. Classes, software components and microservices that have only one responsibility are much easier to explain, understand and implement than the ones that provide a solution for everything.

**O**pen/Closed principle **-** Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending on classes. the **Polymorphic** Open/Closed Principle. It uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them.

**L**iskov substitution principle - The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception if it gets called with an object of the subclass.

**I**nterface Segregation Principle - the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

**D**ependency Inversion Principle - High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

1. Just like any regular method, a constructor may also have overloaded definitions. Constructor overloading provides flexibility in object creation, which is worth its weight in gold in many situations.

The star or asterisk symbol (\*) after argument means that start is a keyword-only argument. To pass a value to a keyword-only argument, you need to use the argument’s name explicitly.

The special method .\_\_call\_\_() turns the instances of a class into callable objects. In other words, you can call the instances of a class like you call any regular function.

We can use isinstance() is the object that you want to type check. Hence, passing different types of objects to the constructor which allow us to use multiple constructors.

Using @classmethod makes it possible to add as many explicit constructors as you need to a given class.

1. Design Patterns : Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

**Singleton -** Ensure that a class has just a single instance and provide a global access point to that instance.

**Observer** - Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called Observer and the object that is being watched is called Subject.

**Factory -** The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class**.**

**Abstract Factory -** Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

**Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**Strategy** pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

**Composite** - When we need to create a structure in a way that the objects in the structure have to be treated the same way, we can apply composite design pattern.

**Iterator** pattern is used to provide a standard way to traverse through a group of Objects. Iterator pattern is useful when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program. The logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily.