OOP – Dry Part

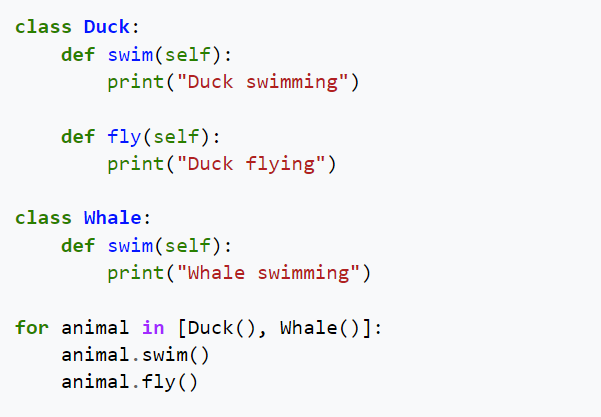
**Basics:**

1. a) Polymorphism is the concept of using certain interfaces, function etc. for various types.  
   For example, if we have a certain function that behaves for a class named Animal, the function will still work for subclasses Cat, Dog.  
     
   b) OOP (Object Oriented Programing) is a paradigm of coding that focuses on objects and data, and not on the logic in the code.  
   The model refers to all structures in the code as objects and examines the relations between them.  
   Each function is paired with a certain object and is used to manipulate it in some way.  
     
   c) Functional Programming is another paradigm of coding that focuses on using only functions (instead of other logical structures) in order to solve computable problems.  
   Functions are used for every action in the code, and are the building blocks of the entire language.
2. a) Composition vs Inheritance
   * 1. The main aspect of composition is that it represents a relationship of ‘has a’, meaning that if we say a certain class is called a composite when it contains an object (or **has an object)** of a different component class.  
        For example, one can look at a class named House, it is a composite class due to the fact it contains a component class that represents an air conditioner.

Inheritance is again a concept that represents a connection between classes, but in this case the classes have an ‘is a’ relationship.  
This means, that if a class inherits another class, it can act as the ‘father’ class, because it’s an extension or derivation of the father class and its specifics.  
By looking at the same example, we notice that there is an ‘is a’ relationship between the same father class – House, and a different class called Villa, that represents a certain kind of house.

* + 1. As can be inferred, the main difference between the concepts is that one represents a relationship that doesn’t inherently has a connection, such as an air conditioner and a house, the connection is more incidental.  
       Inheritance represents a deeper connection, in which the child class is derived from the father class.  
         
       The main pros of inheritance are that it’s very powerful and represents a strong tool in a language.  
       Inheritance is the gateway to polymorphism which contributes greatly to the readability of the code, its conciseness and logical clarity – by using fewer types, that have similar characteristics we can save space, have less mistakes, and better explain certain patterns in our code.  
       The main drawback is speed – the close relationship between the classes means they are not entirely independent, and for a certain subclass we need to load the base class in order to execute it.  
       In addition, changes in the base class may very well affect the subclasses, and handling it might prove to be somewhat difficult.  
         
       The main pros of composition are that it can add a lot of positive complexity and expressiveness to the code (various connection between entirely different types of classes).  
       Composition can often avoid some of the problems possible with inheritance, because it’s easier to be altered later in the process, and in addition it can avoid breaking encapsulation and exposing access.  
       For example, a child class will have access to non-public information of the base class, which can be dangerous in some cases, a thing which can be easily preventable in composition.  
         
       The cons are that eventually composition isn’t compatible with the concept of polymorphism, and we can’t use the properties of inherited classes that can often make our work seem less complicated.
    2. A mutable object is a type of object that can be changed or mutated after its creation – the programmer can access its inner state and alter the values.  
       For example, a regular array is a mutable object.  
         
       An immutable object is the exact opposite – a programmer trying to alter its inner values will be unable to do so after its creation.  
       For example, a tuple in python is immutable.
    3. The main pros of immutability is the confidence and stability in its values – it can be given as an argument to a function and a programmer will be confident it won’t alter its value, and this will be true throughout the program, we won’t need to debug the changes in the code, as it won’t change.  
         
       The pros of mutability are the fact that it’s dynamic, we’ll be able to use arrays (for example) for different purposes, and won’t need to assign large chunks of memory for every small change in our data structure.  
       The code will be much more dynamic and flexible, because we’ll be able to reuse our data structures.
    4. Dynamically typed languages perform type checking at runtime, and don’t require to define the type of the variables when declaring them, they won’t point out that a certain type isn’t compatible with a usage of a function during compilation, but only in runtime.  
         
       Statically typed languages perform type checking during compilation, and do require to define the type of the variables when declaring them.  
       Thus a wrong usage of a variable will be known (in some cases) during compilation time.
    5. The pros of dynamical typing is the aspect of flexibility – we use less forced casting, easier to use polymorphism with, the programmer has much more discretion in the writing of the code, can reduce extra code and improve readability.  
         
       The cons are mostly connected to proneness to errors – by not checking types and “counting on the programmer”, there’s a danger of buggy behavior, types might be non-compatible with their usage in the code, and bugs in runtime are often harder to catch.  
         
       The pros and cons of static typing are mostly the opposite.
    6. Dynamically typed – Python, JS  
       Statically typed – C, C++

1. Duck typing works by the hypothesis that the sentence “If it walks like a duck and it quacks like a duck, then it must be a duck" is indicative of whether an object can be used for a certain task.  
   Meaning that if an object is compatible with a function call, and has every field or feature needed for a certain operation, we can use it for that operation.  
   An example taken from Wikipedia:



The results will prove that the 3 first invocations work (twice for duck), because both objects have a swim() method, but the whale doesn’t have a fly() method, so we won’t be able to invoke the method on the object.  
Showing exactly the concept we wanted to explain – The function works if it’s defined, no matter the object, and doesn’t work otherwise.

1. Private – A private field or method can only be reached within the same class.  
   Protected – Can be reached within the same class and its parent classes or subclasses.  
   Public – Can be reached in the entire universe of the code.  
     
   In python, variables are public by default, but by adding an underscore before the name of a variable we can make it protected, and by adding two underscores we can make it private.  
   Those definitions are less ingrained in the language than in different languages and can be passed, but still somewhat fulfill the need of such private and protected variables.
2. An instance method is the regular type of method we know in python classes, it gets self as a parameter and thus acts as a method that alters a certain instance of a class.  
   A Class method is a method that is called on a class itself and not on an a specific object instance, it is bound to a class but not to an object of a class, it can modify the state of the class (certain attributes of the class that are shared with all instances of a class).  
   A static method is very similar to a class method in terms of being called on a class and not on an instance, the difference being that it’s can’t access the state of the class and can’t take any parameters.

**Design Concepts:**

1. SOLID is an acronym for the first five object-oriented design (OOD) principles(written by Uncle Bob), and stands for Single-responsiblity Principle, Open-closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion.  
   These principles help us maintain a healthy code environment as the project grows and develops.  
     
   The Single-responsibility Principle rule means that a class should change only for one reason, and have only one job and reason to exist.  
   In other words, Each class must not be responsible of two different aspects of the code.  
   An example for this can be a module that compiles and prints a report, it can be changed for two reasons.  
   An easy fix for this problem would be to split this module into two, thus keeping the code readable.  
     
   The Open-Closed Principle says that Objects or entities should be open for extension but closed for modification, meaning that a class should be extendable without modifying the class itself.  
   An example for this can be a certain class shape which has multiple subclasses, and we wish to calculate their area, we do it using a function defined in the superclass.  
   If we wish to add extra subclasses, we’ll have to go to the superclass and alter the area function, something that doesn’t stand in line with the principle, thus the better option will be to just implement the function individually for each shape.  
     
   The Liskov Substitution Principle means that every subclass or derived class should be substitutable for their base or parent class.  
   In other words, the ideal is to have the objects of our subclasses behaving the same way as the objects of our superclass.  
   An example for that can be a certain superclass that implements a function, but has other parameters than the same function in a subclass, causing the subclass to not be able to replace the superclass.  
   There are a few solution to this, such as using an abstract class that will function as the ‘father’ of both the previous super and subclass (ending the is-a relation between them), and defining a suitable function in each of the new ‘brother’ classes.  
     
   The Interface Segregation Principle Interface segregation principle states that a client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.  
   Interfaces should strive to answer the common denominator and not force their derived classes to implement functions they have no use in.  
   An example of such case can be an interface of a car that has a function of ‘fillGas()’, but there’s a class named ‘ElectricCar’ that implements it, and thus doesn’t need to fill gas.  
   Solutions for this include adding a different interface, or not including such a function in the first place.  
     
   Lastly, the Dependency Inversion principle means that the high-level module must not depend on the low-level module, but they should depend on abstractions.  
   Abstractions should not depend on details. Details should depend upon abstractions.  
   An example for that can include the function ‘navigate()’ for google maps, that starts navigation, but in order for the function to include every possibility, we need to give it a certain string as a parameter (‘car’ or ‘bike’, etc.), an action that makes our code clumsy, and dependent on details.  
   Instead of that we can use extra layers of instructions by including an interface that will be implemented by all transport options, and thus navigate() will not depend on the lower level.
2. The advantage of using multiple constructors is that it allows us to have more flexibility with the details needed for us to create a class, and to perform different actions on the instantiation of a class.  
   One option for this in python is by overloading, using checking of the function parameters.  
   Meaning that we’ll check whether \*args is int, string, etc. and will act accordingly.  
   Another option can include using default values, meaning that if we want to create a certain class with three different parameters, by using default values we can allow flexibility in terms of values given as a default or not, thus eliminating somewhat the need for a few constructors.  
   Another option is using class methods as we saw them before, meaning that we use a certain abstraction, and instead of ‘creating different init functions’, we just use class methods that return classes and each time instantiate them with different values, thus actually creating different constructors.
   1. The singleton pattern is a software design pattern that restricts the instantiation of a class to a singular instance, and makes sure that everyone globally have an access to the one instance of the singleton.  
      It is usually needed in cases of a certain shared resource between different classes, a case in which the system will have to be up to date, synchronized and provide access for everyone.
   2. The Observer pattern is a pattern in which an object, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.  
      An example for the importance of this pattern can be a synchronized behavior in a certain app – when a user types on the screen, some things should happen.  
      In order for the objects in charge of those different functions to not being have to check the state of the screen each certain amount of time, we can define them as observers, and notify them when an event of importance happens.
   3. In the Factory pattern, we can create an object without exposing the creation logic to the client and refer to newly created object using a common interface.  
      By using this method we attain a few different advantages, first is we have a better way of hiding the implementation of the creation from the user, we can create objects in an easier way, and most importantly, every subtype that needs a different constructor can customize it by itself, by implementing an interface that inherits from the original interface defining the super type.
   4. The Abstract Factory pattern is somewhat similar to the regular Factory pattern, but with a main difference.  
      Here, we have no option of implementing the superclass’ interface, as it is abstract, but we must inherit from the interface, implement a similar building method and thus use as a factory for a certain subtype.  
      For example, we can look at a father class of “furniture”, from which the classes Chair, Table and Couch inherit, and by using the concept of a factory introduced before, we use a certain modification of it linked to abstraction.
   5. The Builder pattern allows to construct complex objects step by step. The pattern lets you produce different types and representations of an object using the same construction code.  
      This can be helpful for a complex object that needs many features for its construction, we can look at a Car object – it can have windows, clutch, a nitro option, smart screens, etc.  
      Instead of having a huge constructor that contains many arguments indicating whether a feature exists or not, we can transfer the problem to different function.  
      For example, we’ll have a function – “Build clutch” if the car is manual, but we won’t have to call it at all if the car is automatic.
   6. The Strategy pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.  
      Meaning that based on runtime information, we’ll be able to pick one of the algorithms, or “strategies”, according to the expected behavior.  
      A real-world example can be Google Maps – it chooses which algorithm of navigation to use, on a runtime basis, based on the preference of the transport of the user, the algorithm will be different for a driver and a cycler.
   7. The Composite pattern allows composing objects into tree structures and then work with these structures as if they were individual objects.  
      Meaning that if we have a certain part of types in our code that can be viewed as a tree in a hierarchical sense, we’ll able to treat them as one for some purposes.  
      For example, we can look at a certain branch in a company, and want to know the amount that’ll go towards salaries there, and a composite pattern will help us do it more easily.
   8. The Iterator pattern lets you traverse elements of a collection without exposing its underlying representation.  
      If one wishes to traverse the elements of a tree, or a stack, or some more complex structure, it isn’t always clear what the order of the access should be, and whether the programmer should even know that.  
      An iterator is in charge of the entire concept of traversal, it manages the order, current placement and the amount of items left, and thus makes it easier for the programmer to traverse the elements of a list, and also keeps the algorithm in charge of it private.

**Python Specifics:**

1. Python 3 is a newer version than Python 2, released 8 after, and is intended to improve some key issues in the language structure.  
   The improvements occurred in a few areas – syntax, Python 3 has better readability and is more intuitive, performance, where python 3 is faster and performs better, and Unicode support.  
   In addition, several functions and conventions were changed, such as the print and the range function.  
   There’s a significant problem regarding backwards compatibility between the versions.  
   A code written in python 2 can be transferred to python 3 without major problems, but the other way around isn’t that simple.  
   Python 3 isn’t backward compatible with python 2, and thus every code written in python 3 will need to be heavily altered in order to be used in python 3.  
   In addition, libraries are also mostly incompatible between the versions, and a library code from python 2 likely can’t be used in python 3, and the other way around.  
   Thus, transferring code between the versions is difficult and causes problems to people who wrote large projects in the previous version.
2. 3.4 – The changes were mainly the following:  
   they made Pip always available, made file descriptors non-inheritable, improved Cpython (the connection between C and Python), and added several important libraries including enums, pathlib, asyncio and statistics.

It also improved already included libraries such as multiprocessing, IP handling and email, and finally added multiple modifications in the field of security.  
  
3.5 – The changes included:  
New operators and syntax features such as @ as a matrix multiplication operator, and % to be used in order to format bytes.  
It also included more improvements in the fields of security, built-in libraries, and the usage of Cpython.  
  
3.6 – Included multiple syntax features such as formatted string literals and underscores in numerical literals, added several asynchronous features.  
Changes were added in terms of cpython, security and some other libraries, and had many standard libraries upgrades.  
There were also some key changes in terms of windows, and some default settings.  
  
3.7 – Included small syntax changes, mostly related to backward compatibility such as making async and await keywords.  
It added an important debugging function – breakpoint(), and also added more options of typing and the usage of generic variables.  
In addition to that, there were also some changes in the standard library, cpython and in the C API.

3.8 – New features such as an assignment expression (:= inside of a larger expression), type hinting generics in standard collections, changes in terms of parameter specifications inside of functions, improvement in the f’ syntax, better Python Initialization Configuration.

3.9 - union operators were added to dictionaries, string methods to remove prefixes and suffixes, flexible function and variable annotations.  
In addition a new parser was added.  
  
3.10 – Multiple typing features were added, such as the addition of union types, parameter specification variables, and explicit type aliases.  
Many structural improvements and removals in the language itself such as adding encoding warnings.

3.11 – Again many important deprecations, multiple library modules were deprecated, several changes around exceptions, and many typing features added such as the ‘self’ type, and variadic generics.