# Basics

1. Question 1

* Polymorphism - Polymorphism describes a situation where different objects may be referred to by the same name, and all entities share the same interface of actions. In OOP, Polymorphism may be implemented using inheritance. For example, the base type Shape may define the *area* and *perimeter* functions, and the inheriting types Circle, Triangle and Square each separate those functions.
* OOP - A programming paradigm focused on *objects* rather than functions. Objects include attributes (object variables) and methods (functions which operate on the object’s variables).
* Functional Programming - A programming paradigm which focuses on using functions and immutable data structures. There’s a main focus on using pure functions, functions who depend only on their input, and whose sole external impact is their return value. Moreover, there is a focus on First Class Functions (functions treated as variables) and Higher Order Functions (functions which receive other functions as variables).

1. Question 2
   1. *Composition* is a concept which describes the situation where an object contains sub-objects as instances and uses their methods. It is referred to as a has-a relationship.

*Inheritance* is where objects are derived from a base-object and inherit its methods and may override them. It is referred to as an is-a relationship.

Pros of composition over inheritance: increased flexibility in changing the base class, access control to the base class.

However, when the relationship is definitely an is-a, inheritance is a better match than composition, as it allows more efficient code reuse.

* 1. A *mutable* object may be changed after its creation whereas an *immutable* object cannot.

When a mutable object is changed, all references are changed, whereas that is not possible with immutable objects, that may be an advantage but also requires more careful usage of mutable objects.

The use of immutable objects may be more memory intensive as copies of the data are created.

* 1. Statically typed languages perform type checking during compilation while dynamically typed languages perform it during runtime.

Static typing makes programs faster and also may make debugging faster by not allowing scripts to run with bugs.

Dynamic typing speeds up coding as types don’t have to be specified and allows for greater flexibility as the same variable may be of different types in different runs.

1. Duck typing is a method of typing in dynamically typed languages, where instead of specifying the type of the input, it is assumed that he has all of the required methods and attributes.
2. Public members may be accessed by everyone, private members may only be accessed by their class, and protected members may be accessed by class and derived classes.

In python, by default all members are public, by putting a single underscore at the beginning of the name a member is signed as protected, but it may still be accessed. By putting two underscores at the beginning, a member is defined to be private, it may still be accessed, but instead of by “.\_\_[name]”, by “.\_[class]\_\_[name]”.

1. Instance methods are normal object methods, their first input is the *self* parameter, it allows access to the object’s attributes and class attributes.

Class methods are marked with the @classmethod decorator, their first argument is the *cls* parameter, and it allows them access to the class attributes, rather than an object’s attributes.

Static methods are marked with the @staticmethod decorator, they don’t get any parameter by default and therefore don’t have access to an object’s or a class’s arguments.

# Design Concepts

1. SOLID stands for the following:
   1. Single Responsibility Principle - each class should have only one reason to change. This means that each class should solve a single problem.
   2. Open-Closed Principle - well written classes should be extendable without changing the existing code. Open for extension, closed for modification.
   3. Liskov Substitution Principle - every derived class should be substitutable for its parent class, i.e. the derived class should merely extend the base class’s functionality instead of changing it.
   4. Interface Segregation Principle - It’s better to have many small interfaces rather than a few big ones.
   5. Dependency Inversion Principle - High level modules should not depend on low level modules, rather they both should depend on abstractions. That way low level modules may be changed without affecting the high level modules.
2. Having multiple constructors allows higher flexibility when constructing objects, such as not having all of the required input variables.

It is possible to overcome the lack of multiple constructors using default arguments and optional arguments (\*args and \*\*kwargs). Where in Java you would have a constructor with and without the argument A, in Python you would have A be None by default, and infer its value if it’s not given.

1. A design pattern is a blueprint for a solution for a recurring problem in programming.
   1. Singleton - A singleton is a design pattern which solves the problem of having only a single instance of some class. Unlike global variables, it supports lazy initialization.
   2. Observer - A design pattern for the situation that multiple clients need to observe a single subject and react to its state changes. The *observers* must implement a reaction function and the *subject* needs to maintain a list of observers and notify them in case of a state change. Different derived classes of the observer class may describe different types of clients.
   3. Factory - A design pattern for when there are multiple subclasses for the same base class, and it is desired to construct one of the subclasses, without specifying which, depending on the inputs.
   4. Abstract Factory - A factory of factories. This design pattern is useful when one wishes to create families of related objects. The abstract factory creates factories that create objects with some common “theme”
   5. Builder - This design pattern breaks down the construction of complex objects to a set of simpler functions.
   6. Strategy - A design pattern which allows us to solve a problem using different solutions. All *Strategy* objects implement the same interface, but each implements a different solution to the problem.
   7. Composite - This design pattern is useful when one wishes to treat a group object (in a tree structure) the same as a single object. An interface is defined for both the *leaf* object and a *composite* object, which holds a tree of leaves. When a function is used on the composite, it forwards it to its leaves.
   8. Iterator - A design pattern which allows one to traverse the elements of some collection without exposing its interface. Different iterator implementations could provide different methods of traversal (such as BFS, DFS, etc).

# Python Specifics

1. Differences:
   1. Python 3 requires parentheses when printing while Python 2 does not.
   2. Integer division in Python 2 always results in an integer, where in Python 3 it results in a float.
   3. Python 2’s *xrange* function was replaced in Python 3 by the *range* function, which is faster.

The problem between converting code is that some built in functions are different in the two versions and some packages don’t support both versions.