|  |
| --- |
| [Type the company name] |
| Software Test Engineer |
| [Type the document subtitle] |

|  |
| --- |
| Igor Stanković  [Pick the date] |

# OOP

Object-Oriented Programming (OOP) is a programming paradigm in computer science 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. There are many object-oriented programming languages, including JavaScript, C++, Java, and Python.

Benefits of OOP for software engineering

* OOP models complex things as reproducible, simple structures
* Reusable, OOP objects can be used across programs
* Polymorphism allows for class-specific behavior
* Easier to debug, classes often contain all applicable information to them
* Securely protects sensitive information through encapsulation

## Four Principles of OOP

The four pillars of object-oriented programming are:

* **Inheritance**: child classes inherit data and behaviors from the parent class
* **Encapsulation**: containing information in an object, exposing only selected information
* **Abstraction**: only exposing high-level public methods for accessing an object
* **Polymorphism**: many methods can do the same task

### Inheritance

Inheritance allows classes to inherit features of other classes. Put another way, parent classes extend attributes and behaviors to child classes. Inheritance supports reusability.

If basic attributes and behaviors are defined in a parent class, child classes can be created, extending the functionality of the parent class and adding additional attributes and behaviors.

For example, herding dogs have the unique ability to herd animals. In other words, all herding dogs are dogs, but not all dogs are herding dogs. We represent this difference by creating a child class HerdingDog from the parent class Dog, and then adding the unique herd() behavior.

The benefits of inheritance are programs can create a generic parent class and then create more specific child classes as needed. This simplifies programming because instead of recreating the structure of the Dog class multiple times, child classes automatically gain access to functionalities within their parent class.

### Encapsulation

Encapsulation means containing all important information inside an object, and only exposing selected information to the outside world. Attributes and behaviors are defined by code inside the class template.

Then, when an object is instantiated from the class, the data and methods are encapsulated in that object. Encapsulation hides the internal software code implementation inside a class and hides the internal data of inside objects.

Encapsulation requires defining some fields as private and some as public.

* **Private**/ Internal interface: methods and properties accessible from other methods of the same class.
* **Public** / External Interface: methods and properties accessible from outside the class.

The benefits of encapsulation are summarized here:

* Adds security: Only public methods and attributes are accessible from the outside
* Protects against common mistakes: Only public fields & methods are accessible, so developers don’t accidentally change something dangerous
* Protects IP: Code is hidden in a class; only public methods are accessible by the outside developers
* Supportable: Most code undergoes updates and improvements
* Hides complexity: No one can see what’s behind the object’s curtain!

### Abstraction

Abstraction is an extension of encapsulation that uses classes and objects, which contain data and code, to hide the internal details of a program from its users. This is done by creating a layer of abstraction between the user and the more complex source code, which helps protect sensitive information stored within the source code.

* Reduces complexity and improves code readability
* Facilitates code reuse and organization
* Data hiding improves data security by hiding sensitive details from users
* Enhances productivity by abstracting away low-level details

Abstraction also serves an important security role. By only displaying selected pieces of data and only allowing data to be accessed through classes and modified through methods, we protect the data from exposure. To continue with the car example, you wouldn’t want an open gas tank while driving a car.

The benefits of abstraction are summarized below:

* Simple, high-level user interfaces
* Complex code is hidden
* Security
* Easier software maintenance
* Code updates rarely change the abstraction

### Polymorphism

Polymorphism means designing objects to share behaviors. Using inheritance, objects can override shared parent behaviors with specific child behaviors. Polymorphism allows the same method to execute different behaviors in two ways: method overriding and method overloading.

**Method Overriding**

Runtime polymorphism uses method overriding. In method overriding, a child class can implement differently than its parent class. In our dog example, we may want to give TrackingDog a specific type of bark different than the generic dog class.

**Method Overloading**

Compile Time polymorphism uses method overloading. Methods or functions may have the same name but a different number of parameters passed into the method call. Different results may occur depending on the number of parameters passed in.

The benefits of Polymorphism are:

* Objects of different types can be passed through the same interface
* Method overriding
* Method overloading

## Building blocks of OOP

Next, we’ll take a deeper look at each of the fundamental building blocks of an OOP program used above:

* Classes
* Objects
* Methods
* Attributes

### Classes

In a nutshell, classes are essentially user-defined data types. Classes are where we create a blueprint for the structure of methods and attributes. Individual objects are instantiated from this blueprint.

Classes contain fields for attributes and methods for behaviors.

### Objects

Objects are, unsurprisingly, a huge part of OOP! Objects are instances of a class created with specific data.

### Attributes

Attributes are the information that is stored. Attributes are defined in the Class template. When objects are instantiated, individual objects contain data stored in the Attributes field.

The state of an object is defined by the data in the object’s attributes fields. For example, a puppy and a dog might be treated differently at a pet camp. The birthday could define the state of an object and allow the software to handle dogs of different ages differently.

### Methods

Methods represent behaviors. Methods perform actions; methods might return information about an object or update an object’s data. The method’s code is defined in the class definition.

# Design Patterns

By definition, Design Patterns are reusable solutions to commonly occurring problems in the context of software design. They are divided into three categories Creational, Behavioral and Structural patterns.

## Creational Design Patterns:

These type of pattern support the creation of objects. Because in certain situations there are more elegant ways than using the new operator.

### Singleton Pattern

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to make sure there is no more than one instance; it does it by instantiating itself and in the same time it provides a global point of access to that instance. By doing it, the singleton class ensures the same instance can be used from everywhere, preventing direct invocation of the singleton constructor.

**Intent**

* Ensure that only one instance of a class is created.
* Provide a global point of access to the object.

**Implementation**

The implementation involves a static member in the Singleton class which keeps the reference to the instance, a private constructor and a static public method that returns the static member reference.

The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

class Singleton {

private static Singleton instance;

private Singleton()

}

public static synchronized Singleton getInstance(){

if (instance == null)

instance = new Singleton();

return instance;

}

public void doSomething()

{

}

}

You can notice in the above code that getInstance method ensures that only one instance of the class is created. The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through the getInstance method. The getInstance method is used also to provide a global point of access to the object and it can be used in the following manner:

### Factory

The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java and C#. It comes in different variants and implementations. If you are searching for it, most likely, you'll find references about the GoF patterns: Factory Method and Abstract Factory.

The implementation is really simple

* The client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.
* The factory instantiates a new concrete product and then returns to the client the newly created product(casted to abstract product class).
* The client uses the products as abstract products without being aware about their concrete implementation.

Applicability & Examples

Probably the factory pattern is one of the most used patterns.

For example a graphical application works with shapes. In our implementation the drawing framework is the client and the shapes are the products. All the shapes are derived from an abstract shape class (or interface). The Shape class defines the draw and move operations which must be implemented by the concrete shapes. Let's assume a command is selected from the menu to create a new Circle. The framework receives the shape type as a string parameter, it asks the factory to create a new shape sending the parameter received from menu. The factory creates a new circle and returns it to the framework, casted to an abstract shape. Then the framework uses the object as casted to the abstract class without being aware of the concrete object type.

### Factory Method Pattern

Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects. One type of responsibility is creating such objects. The library knows when an object needs to be created, but not what kind of object it should create, this being specific to the application using the library.

The Factory method works just the same way: it defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time. A simple real life example of the Factory Method is the hotel. When staying in a hotel you first have to check in. The person working at the front desk will give you a key to your room after you've paid for the room you want and this way he can be looked at as a 'room' factory. While staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a 'phone-call' factory, because he controls the access to calls, too.

**Intent**

Defines an interface for creating objects, but let subclasses to decide which class to instantiate

Refers to the newly created object through a common interface

### Abstract Factory Pattern

Modularization is a big issue in today's programming. Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information. Take the case of a information manager which manages phone number. Phone numbers have a particular rule on which they get generated depending on areas and countries. If at some point the application should be changed in order to support adding numbers form a new country, the code of the application would have to be changed and it would become more and more complicated.

In order to prevent it, the Abstract Factory design pattern is used. Using this pattern a framework is defined, which produces objects that follow a general pattern and at runtime this factory is paired with any concrete factory to produce objects that follow the pattern of a certain country. In other words, the Abstract Factory is a super-factory which creates other factories (Factory of factories).

**Intent**

Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

### Builder Pattern

The more complex an application is the complexity of classes and objects used increases. Complex objects are made of parts produced by other objects that need special care when being built. An application might need a mechanism for building complex objects that is independent from the ones that make up the object. If this is the problem you are being confronted with, you might want to try using the Builder (or Adaptive Builder) design pattern.

This pattern allows a client object to construct a complex object by specifying only its type and content, being shielded from the details related to the object's representation. This way the construction process can be used to create different representations. The logic of this process is isolated form the actual steps used in creating the complex object, so the process can be used again to create a different object form the same set of simple objects as the first one.

**Intent**

Defines an instance for creating an object but letting subclasses decide which class to instantiate

Refers to the newly created object through a common interface

### Prototype Pattern

Today’s programming is all about costs. Saving is a big issue when it comes to using computer resources, so programmers are doing their best to find ways of improving the performance When we talk about object creation we can find a better way to have new objects: cloning. To this idea one particular design pattern is related: rather than creation it uses cloning. If the cost of creating a new object is large and creation is resource intensive, we clone the object.

The Prototype design pattern is the one in question. It allows an object to create customized objects without knowing their class or any details of how to create them. Up to this point it sounds a lot like the Factory Method pattern, the difference being the fact that for the Factory the palette of prototypical objects never contains more than one object.

**Intent**

* specifying the kind of objects to create using a prototypical instance
* creating new objects by copying this prototype

**Object Pool Pattern**

Performance can be sometimes the key issue during the software development and the object creation(class instantiation) is a costly step. While the Prototype pattern helps in improving the performance by cloning the objects, the Object Pool pattern offer a mechanism to reuse objects that are expensive to create.

Clients of an object pull "feel" like they are owners of a service although the service is shared among many other clients.

**Intent**

* reuse and share objects that are expensive to create

## Behavioral Patterns

### Chain of Responsibility

In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also happen to be denied access to the object which needs to handle the event. In this case there are two possibilities: there is the beginner/lazy approach of making everything public, creating reference to every object and continuing from there and then there is the expert approach of using the Chain of Responsibility.

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of coin, the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.

**Intent:**

It avoids attaching the sender of a request to its receiver, giving this way other objects the possibility of handling the request too.

The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it.

### Command Pattern

“An object that contains a symbol, name or key that represents a list of commands, actions or keystrokes”. This is the definition of a macro, one that should be familiar to any computer user. From this idea the Command design pattern was given birth.

The Macro represents, at some extent, a command that is built from the reunion of a set of other commands, in a given order. Just as a macro, the Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command design pattern provides the options to queue commands, undo/redo actions and other manipulations.

**Intent**

- encapsulate a request in an object

- allows the parameterization of clients with different requests

- allows saving the requests in a queue

### Interpreter

The Interpreter is one of the Design Patterns published in the GoF which is not really used. Ussualy the Interpreter Pattern is described in terms of formal grammars, like it was described in the original form in the GoF but the area where this design pattern can be applied can be extended.

**Intent**

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

### Iterator

One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.

But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.

The idea of the iterator pattern is to take the responsibility of accessing and passing trough the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

**Intent**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The abstraction provided by the iterator pattern allows you to modify the collection implementation without making any changes outside of collection. It enables you to create a general purpose GUI component that will be able to iterate through any collection of the application.

### Mediator Pattern

In order to have a good object oriented design we have to create lots of classes interacting one with each other. If certain principles are not applied the final framework will end in a total mess where each object relies on many other objects in order to run. In order to avoid tight coupled frameworks, we need a mechanism to facilitate the interaction between objects in a manner in that objects are not aware of the existence of other objects.

Let's take the example of a screen. When we create it we add all sort of controls to the screen. This control need to interact with all the other control. For example when a button is pressed it must know if the data is valid in other controls. As you have seen if you created different applications using forms you don't have to modify each control class each time you add a new control to the form. All the operations between controls are managed by the form class itself. This class is called mediator.

**Intent**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

### Memento Pattern

It is sometimes necessary to capture the internal state of an object at some point and have the ability to restore the object to that state later in time. Such a case is useful in case of error or failure. Consider the case of a calculator object with an undo operation such a calculator could simply maintain a list of all previous operation that it has performed and thus would be able to restore a previous calculation it has performed. This would cause the calculator object to become larger, more complex, and heavyweight, as the calculator object would have to provide additional undo functionality and should maintain a list of all previous operations. This functionality can be moved out of the calculator class, so that an external (let's call it undo manager class) can collect the internal state of the calculator and save it. However providing the explicit access to every state variable of the calculator to the restore manager would be impractical and would violate the encapsulation principle.

**Intent**

The intent of this pattern is to capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

### Observer Pattern

We can not talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.

**Intent**

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

### Strategy

There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.

**Intent**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

### Template Method

If we take a look at the dictionary definition of a template we can see that a template is a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used.

On the same idea is the template method is based. A template method defines an algorithm in a base class using abstract operations that subclasses override to provide concrete behavior.

**Intent**

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

- Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

### Visitor Pattern

Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type.

A possible approach to apply a specific operation on objects of different types in a collection would be the use if blocks in conjunction with 'instanceof' for each element. This approach is not a nice one, not flexible and not object oriented at all. At this point we should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

**Intent**

Represents an operation to be performed on the elements of an object structure.

Visitor lets you define a new operation without changing the classes of the elements on which it operates.

### Null Object Pattern

There are some cases when a system has to use some functionality and some cases when it doesn't. Let's say we have to implement a class that should send the results to a log file or to the console. But this is just an additional option and the data is logged depending on the configuration values.

If there are cases when the client module does not have to log any data then it has to check the configuration parameter in and if block and then to call or not the Logger class. But as we know the 'if' block is not an elegant solution.

**Intent**

Provide an object as a surrogate for the lack of an object of a given type.

The Null Object Pattern provides intelligent do nothing behavior, hiding the details from its collaborators.

## Structural Patterns

### Adapter Pattern

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

### Bridge Pattern

Sometimes an abstraction should have different implementations; consider an object that handles persistence of objects over different platforms using either relational databases or file system structures (files and folders). A simple implementation might choose to extend the object itself to implement the functionality for both file system and RDBMS. However this implementation would create a problem; Inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementation independently.

**Intent**

The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently.

### Composite Pattern

There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system. A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files. Note that a folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object. Note also that since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, it would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface.

**Intent**

The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.

Composite lets clients treat individual objects and compositions of objects uniformly.

### Decorator Pattern

Extending an object's functionality can be done statically (at compile time) by using inheritance however it might be necessary to extend an object's functionality dynamically (at runtime) as an object is used.

Consider the typical example of a graphical window. To extend the functionality of the graphical window for example by adding a frame to the window, would require extending the window class to create a FramedWindow class. To create a framed window it is necessary to create an object of the FramedWindow class. However it would be impossible to start with a plain window and to extend its functionality at runtime to become a framed window.

**Intent**

The intent of this pattern is to add additional responsibilities dynamically to an object.

### Flyweight Pattern

The Flyweight Pattern is a structural design pattern that allows programs to efficiently share a large number of objects by minimizing memory usage. It achieves this by separating the intrinsic state (shared) from the extrinsic state (external) of the object.

Some programs require a large number of objects that have some shared state among them. Consider for example a game of war, where there is a large number of soldier objects; a soldier object maintain the graphical representation of a soldier, soldier behavior such as motion, and firing weapons, in addition soldier's health and location on the war terrain. Creating a large number of soldier objects is a necessity however it would incur a huge memory cost. Note that although the representation and behavior of a soldier is the same their health and location can vary greatly.

**Intent**

The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

### Proxy Pattern

Sometimes we need the ability to control the access to an object. For example if we need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely. Until that point we can use some light objects exposing the same interface as the heavy objects. These light objects are called proxies and they will instantiate those heavy objects when they are really need and by then we'll use some light objects instead.

This ability to control the access to an object can be required for a variety of reasons: controlling when a costly object needs to be instantiated and initialized, giving different access rights to an object, as well as providing a sophisticated means of accessing and referencing objects running in other processes, on other machines.

Consider for example an image viewer program. An image viewer program must be able to list and display high resolution photo objects that are in a folder, but how often do someone open a folder and view all the images inside. Sometimes you will be looking for a particular photo, sometimes you will only want to see an image name. The image viewer must be able to list all photo objects, but the photo objects must not be loaded into memory until they are required to be rendered.

**Intent**

The intent of this pattern is to provide a "Placeholder" for an object to control references to it.

# Solid Principles

## Single Responsibility Principle (SRP)

Every module, class, or function in a computer program should have responsibility for a single part of that program’s functionality. Also, they should encapsulate that part, and their services should be narrowly aligned with that responsibility.

SRP is closely related to the concepts of Coupling (low) and Cohesion (high). SRP does not necessarily mean that your class should only have one method or property, but rather that the functionality should be related to a single responsibility (and have only one reason for changing).

With SRP, classes become smaller and cleaner, making them easier to maintain.

## Open-Close Principle (OCP)

OCP states that “software entities such as modules, classes, functions, etc. should be open for extension, but closed for modification.” In simple words, one module/class should be developed in such a way that it allows its behavior to be extended without needing to alter its source code.

How to apply OCP:

Add the new functionalities by creating new derived classes which should be inherited from the original base class.

Allow the client to access the original class with an abstract interface through compositional design patterns like Strategy.

So, instead of changing the existing functionality, create new derived classes and leave the original class implementation as it is.

Problems of not following OCP

If you allow a class or function to add new logic, you must test the entire functionality of the application, including both new and existing functionality. You must also inform the QA team about future changes so that they can prepare for regression testing as well as new feature testing.

For example, suppose that we have implemented a mechanism for applying a discount to the final amount value on an invoice. There are two types of discounts: one that is only applicable to final Invoices and one that applies to proposed Invoices. The OCP violation occurs when we need to add new discount type(s), and we need to change the original implementation of the Invoice class.

## Liskov Substitution Principle (LSP)

The LSP is a Substitutability principle in OOP (Object-Oriented Programming). The third SOLID principle states that if S is a subtype of T, then objects of type T should be replaced with objects of type S.

So, if we can successfully replace the object/instance of a parent class with an object/instance of the child class, without affecting the behavior of the base class instance, then we are following LSP.

When this principle is violated, it usually leads to a lot of extra conditional logic scattered throughout the application, checking to see if an object is of a specific type.

As the application grows, the duplicated and scattered code becomes a breeding ground for bugs. The partial implementation of interfaces or base class functionality, leaving unimplemented methods or properties to throw an exception, is a very common violation of this principle (for example: NotImplementedException).

In code that you know will only be used by one client that you can monitor, this is fine. But in a shared codebase, or worse, in framework code that is shipped to third parties, such implementations should be avoided.

If a given interface has more features than you need, use the Interface Segregation Principle (ISP) to create a new interface that only has the features that your client code needs and that you can fully implement.

As a parallel example, consider the case where a father is a teacher and his son is a doctor. The son cannot simply replace his father, even though both belong to the same family. Another ‘Apples and Oranges’ example can be found in the GitHub repository here.

## Interface Segregation Principle (ISP)

The ISP states that “Clients should not be forced to implement any methods they do not use. Rather than one fat interface, numerous little interfaces are preferred, based on groups of methods, with each interface serving one submodule.“

That definition can be split into two parts:

* No class should be forced to implement any interface method(s) that it does not use.
* Rather than creating large interfaces, create multiple smaller interfaces to allow clients to focus on the methods that are relevant to them.

## Dependency Inversion Principle (DIP)

DIP, the fifth SOLID principle, states that high-level modules/classes should not depend on low-level modules/classes. Instead, both should depend upon abstractions.

Secondly, abstractions should not depend on details; details should depend upon abstractions.

Always try to keep the high-level module and the low-level module as loosely coupled as possible.

When a class knows about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So, we must keep these high-level and low-level modules/classes loosely coupled as much as possible.

To do that, we need to make both of them dependent on abstractions instead of knowing each other. The source code can be found here.

# K.I.S.S Principle

KISS (Keep it simple stupid) is an important term in programming because it puts in your subconscious that every process you’re creating should be as simple as possible and also equally as efficient.

In the programming context, there are a few points to note whenever we want to reduce complexity.

* Ensure your variable names describes the variable it holds properly.
* Ensure your method names translates to the purpose of that method.
* Write comments within your method where necessary.
* Ensure your classes has a single responsibility.
* Avoid global states and behaviors like as much as you can.
* Delete instances, methods or redundant processes within the code base that are not in use.

We should try to reduce complexity, while maintaining an efficient system. The British computer scientist, M. A Jackson actually wrote that,

Programmers often take refuge in an understandable, but disastrous, inclination towards complexity and ingenuity in their work.

Why is the KISS principle important in programming?

* One major challenge Developers face is working on an existing code base. But when the KISS principle is applied it tackles that issue.
* The KISS principle facilitates continuity when needed and gives room for other people to understand the process.
* Simpler processes allow for greater efficiency in automated testing. It is easier to test a simple system than a complex one.

## What is DRY Development?

DRY, which stands for ‘don’t repeat yourself,’ is a principle of software development that aims at reducing the repetition of patterns and code duplication in favor of abstractions and avoiding redundancy.

Popularized by the book, The Pragmatic Programmer, the DRY principle states that, “every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” Using the principle, logic or algorithms that have certain functionality should only appear once in an application.

# OOP Questions and Answers

**1. What is the difference between OOP and SOP?**

|  |  |
| --- | --- |
| Object-Oriented Programming | Structural Programming |
| Object-Oriented Programming is a type of programming which is based on objects rather than just functions and procedures | Provides logical structure to a program where programs are divided functions |
| Bottom-up approach | Top-down approach |
| Provides data hiding | Does not provide data hiding |
| Can solve problems of any complexity | Can solve moderate problems |
| Code can be reused thereby reducing redundancy | Does not support code reusability |

**2. What is Object Oriented Programming?**

Object-Oriented Programming(OOPs) is a type of programming that is based on objects rather than just functions and procedures. Individual objects are grouped into classes. OOPs implements real-world entities like inheritance, polymorphism, hiding, etc into programming. It also allows binding data and code together.

**. Why use OOPs?**

* OOPs allows clarity in programming thereby allowing simplicity in solving complex problems
* Code can be reused through inheritance thereby reducing redundancy
* Data and code are bound together by encapsulation
* OOPs allows data hiding, therefore, private data is kept confidential
* Problems can be divided into different parts making it simple to solve
* The concept of polymorphism gives flexibility to the program by allowing the entities to have multiple forms

**4. What are the main features of OOPs?**

* Inheritance
* Encapsulation
* Polymorphism
* Data Abstraction

To know more about OOPs in JAVA, Python, and C++ you can go through the following blogs:

* [**JAVA** OOPs Concepts](https://www.edureka.co/blog/object-oriented-programming/)
* [**Python** OOPs Concepts](https://www.edureka.co/blog/python-class/)
* [**C++** OOPs Concepts](https://www.edureka.co/blog/object-oriented-programming-in-cpp/)

**Classes and Objects OOPs Interview Questions and Answers**

**5. What is an object?**

An object is a real-world entity which is the basic unit of OOPs for example chair, cat, dog, etc. Different objects have different states or attributes, and behaviors.

**6. What is a class?**

A class is a prototype that consists of objects in different states and with different behaviors. It has a number of methods that are common the objects present within that class.

**7. What is the difference between a class and a structure?**

**Class:**User-defined blueprint from which objects are created. It consists of methods or set of instructions that are to be performed on the objects.

**Structure:**A structure is basically a user-defined collection of variables which are of different data types.

### ****8. Can you call the base class method without creating an instance?****

Yes, you can call the base class without instantiating it if:

* It is a static method
* The base class is inherited by some other subclass

### ****9. What is the difference between a class and an object?****

|  |  |
| --- | --- |
| Object | Class |
| A real-world entity which is an instance of a class | A class is basically a template or a blueprint within which objects can be created |
| An object acts like a variable of the class | Binds methods and data together into a single unit |
| An object is a physical entity | A class is a logical entity |
| Objects take memory space when they are created | A class does not take memory space when created |
| Objects can be declared as and when required | Classes are declared just once |

To know more about objects and classes in JAVA, Python, and C++ you can go through the following blogs:

* [Objects in **Java**](https://www.edureka.co/blog/java-object/)
* [Class in **Java**](https://www.edureka.co/blog/java-objects-and-classes/)
* [Objects and classes in **Python**](https://www.edureka.co/blog/python-class/)
* [Objects in **C++**](https://www.edureka.co/blog/object-oriented-programming-in-cpp/#Objects)

#### ****10. What is inheritance?****

Inheritance is a feature of OOPs which allows classes inherit common properties from other classes. For example, if there is a class such as ‘vehicle’, other classes like ‘car’, ‘bike’, etc can inherit common properties from the vehicle class. This property helps you get rid of redundant code thereby reducing the overall size of the code.

#### ****11. What are the different types of inheritance?****

* Single inheritance
* Multiple inheritance
* Multilevel inheritance
* Hierarchical inheritance
* Hybrid inheritance

#### ****12. What is the difference between multiple and multilevel inheritance?****

|  |  |
| --- | --- |
| Multiple Inheritance | Multilevel Inheritance |
| Multiple inheritance comes into picture when a class inherits more than one base class | Multilevel inheritance means a class inherits from another class which itself is a subclass of some other base class |
| Example: A class defining a child inherits from two base classes Mother and Father | Example: A class describing a sports car will inherit from a base class Car which inturn inherits another class Vehicle |

#### ****13. What is hybrid inheritance?****

Hybrid inheritance is a combination of multiple and multi-level inheritance.

#### ****14. What is hierarchical inheritance?****

Hierarchical inheritance refers to inheritance where one base class has more than one subclasses. For example, the vehicle class can have ‘car’, ‘bike’, etc as its subclasses.

#### ****15. What are the limitations of inheritance?****

* Increases the time and effort required to execute a program as it requires jumping back and forth between different classes
* The parent class and the child class get tightly coupled
* Any modifications to the program would require changes both in the parent as well as the child class
* Needs careful implementation else would lead to incorrect results

To know more about inheritance in Java and Python, read the below articles:

* [Inheritance in Java](https://www.edureka.co/blog/inheritance-in-java/)
* [Inheritance in Python](https://www.edureka.co/blog/inheritance-in-python/)

#### ****16. What is a superclass?****

A superclass or base class is a class that acts as a parent to some other class or classes. For example, the Vehicle class is a superclass of class Car.

**17. What is a subclass?**

A class that inherits from another class is called the subclass. For example, the class Car is a subclass or a derived of Vehicle class.

**8. What is polymorphism?**

Polymorphism refers to the ability to exist in multiple forms. Multiple definitions can be given to a single interface. For example, if you have a class named Vehicle, it can have a method named speed but you cannot define it because different vehicles have different speed. This method will be defined in the subclasses with different definitions for different vehicles.

**19. What is static polymorphism?**

Static polymorphism (static binding) is a kind of polymorphism that occurs at compile time. An example of compile-time polymorphism is method overloading.

**20. What is dynamic polymorphism?**

Runtime polymorphism or dynamic polymorphism (dynamic binding) is a type of polymorphism which is resolved during runtime. An example of runtime polymorphism is method overriding.

**21. What is method overloading?**

Method overloading is a feature of OOPs which makes it possible to give the same name to more than one methods within a class if the arguments passed differ.

**22. What is method overriding?**

Method overriding is a feature of OOPs by which the child class or the subclass can redefine methods present in the base class or parent class. Here, the method that is overridden has the same name as well as the signature meaning the arguments passed and the return type.

**23. What is operator overloading?**

Operator overloading refers to implementing operators using user-defined types based on the arguments passed along with it.

**24. Differentiate between overloading and overriding.**

|  |  |
| --- | --- |
| Overloading | Overriding |
| Two or more methods having the same name but different parameters or signature | Child class redefining methods present in the base class with the same parameters/ signature |
| Resolved during compile-time | Resolved during runtime |

To know more about polymorphism in Java and Python, read the below articles:

* [Polymorphism in Java](https://www.edureka.co/blog/polymorphism-in-java/)
* [Polymorphism in Python](https://www.edureka.co/blog/object-oriented-programming-python/#Polymorphism)

**25. What is encapsulation?**

Encapsulation refers to binding the data and the code that works on that together in a single unit. For example, a class. Encapsulation also allows data-hiding as the data specified in one class is hidden from other classes.

**26. What are ‘access specifiers’?**

[Access specifiers or access modifiers are keywords](https://www.edureka.co/blog/access-modifiers-in-java/) that determine the accessibility of methods, classes, etc in OOPs. These access specifiers allow the implementation of encapsulation. The most common access specifiers are public, private and protected. However, there are a few more which are specific to the programming languages.

**27. What is the difference between public, private and protected access modifiers?**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Accessibility from own class | Accessibility from derived class | Accessibility from world |
| Public | Yes | Yes | Yes |
| Private | Yes | No | No |
| Protected | Yes | Yes | No |

To know more about encapsulation read along:

* [Encapsulation in Java](https://www.edureka.co/blog/object-oriented-programming/#encapsulation)
* [Encapsulation in C++](https://www.edureka.co/blog/encapsulation-in-cpp/)
* [Encapsulation in Python](https://www.edureka.co/blog/object-oriented-programming-python/#Encapsulation)

**Data abstraction**

**28. What is data abstraction?**

Data abstraction is a very important feature of OOPs that allows displaying only the important information and hiding the implementation details. For example, while riding a bike, you know that if you raise the accelerator, the speed will increase, but you don’t know how it actually happens. This is [data abstraction](https://www.edureka.co/blog/data-abstraction-in-cpp/) as the implementation details are hidden from the rider.

**29. How to achieve data abstraction?**

Data abstraction can be achieved through:

* Abstract class
* Abstract method

**30. What is an abstract class?**

An abstract class is a class that consists of abstract methods. These methods are basically declared but not defined. If these methods are to be used in some subclass, they need to be exclusively defined in the subclass.

**1.** **Can you create an instance of an abstract class?**

No. Instances of an abstract class cannot be created because it does not have a complete implementation. However, instances of subclass inheriting the abstract class can be created.

**32. What is an interface?**

It is a concept of OOPs that allows you to declare methods without defining them. Interfaces, unlike classes, are not blueprints because they do not contain detailed instructions or actions to be performed. Any class that implements an interface defines the [methods of the interface](https://www.edureka.co/blog/java-interface/).

**33. Differentiate between data abstraction and encapsulation.**

|  |  |
| --- | --- |
| Data abstraction | Encapsulation |
| Solves the problem at the design level | Solves the problem at the implementation level |
| Allows showing important aspects while hiding implementation details | Binds code and data together into a single unit and hides it from the world |

To know more about data abstraction, below articles might help you:

* [Abstraction in **Java**](https://www.edureka.co/blog/java-abstraction/)
* [Abstraction in **Python**](https://www.edureka.co/blog/object-oriented-programming-python/#Abstraction)

**34. What are virtual functions?**

Virtual functions are functions that are present in the parent class and are overridden by the subclass. These functions are used to achieve runtime polymorphism.

**35. What are pure virtual functions?**

Pure virtual functions or [abstract functions](https://www.edureka.co/blog/virtual-function-in-cpp/) are functions that are only declared in the base class. This means that they do not contain any definition in the base class and need to be redefined in the subclass.

**36. What is a constructor?**

A constructor is a special type of method that has the same name as the class and is used to initialize objects of that class.

**37. What is a destructor?**

A destructor is a method that is automatically invoked when an object is destroyed. The destructor also recovers the heap space that was allocated to the destroyed object, closes the files and database connections of the object, etc.

**38. Types of constructors**

[Types of constructors](https://www.edureka.co/blog/python-constructors/) differ from language to language. However, all the possible constructors are:

* Default constructor
* Parameterized constructor
* Copy constructor
* Static constructor
* Private constructor

**39. What is a copy constructor?**

A [copy constructor](https://www.edureka.co/blog/constructor-in-java/) creates objects by copying variables from another object of the same class. The main aim of a copy constructor is to create a new object from an existing one.

**40. What is the use of ‘finalize’?**

Finalize as an object method used to free up unmanaged resources and cleanup before Garbage Collection(GC). It performs memory management tasks.

**41. What is Garbage Collection(GC)?**

GC is an implementation of automatic memory management. The Garbage collector frees up space occupied by objects that are no longer in existence.

**42. Differentiate between a class and a method.**

|  |  |
| --- | --- |
| Class | Method |
| A class is basically a template that binds the code and data together into a single unit. Classes consist of methods, variables, etc | Callable set of instructions also called a procedure or function that are to be performed on the given data |

**43. Differentiate between an abstract class and an interface?**

|  |  |  |
| --- | --- | --- |
| Basis for comparison | Abstract Class | Interface |
| Methods | Can have abstract as well as other methods | Only abstract methods |
| Final Variables | May contain final and non-final variables | Variables declared are final by default |
| Accessibility of Data Members | Can be private, public, etc | Public by default |
| Implementation | Can provide the implementation of an interface | Cannot provide the implementation of an abstract class |

**44. What is a final variable?**

A variable whose value does not change. It always refers to the same object by the property of non-transversity.

### ****45. What is an exception?****

An exception is a kind of notification that interrupts the normal execution of a program. Exceptions provide a pattern to the error and transfer the error to the exception handler to resolve it. The state of the program is saved as soon as an exception is raised.

### ****46. What is exception handling?****

Exception handling in Object-Oriented Programming is a very important concept that is used to manage errors. An exception handler allows errors to be thrown and caught and implements a centralized mechanism to resolve them.

**47. What is the difference between an error and an exception?**

|  |  |
| --- | --- |
| **Error** | **Exception** |
| Errors are problems that should not be encountered by applications | Conditions that an application might try to catch |

**48. What is a try/ catch block?**

A try/ catch block is used to handle exceptions. The try block defines a set of statements that may lead to an error. The catch block basically catches the exception.

**49. What is a finally block?**

A finally block consists of code that is used to execute important code such as closing a connection, etc. This block executes when the try block exits. It also makes sure that finally block executes even in case some unexpected exception is encountered.

**OOPs Interview Questions – Limitations of OOPs**

**50. What are the limitations of OOPs?**

* Usually not suitable for small problems
* Requires intensive testing
* Takes more time to solve the problem
* Requires proper planning
* The programmer should think of solving a problem in terms of objects

**56. What are the characteristics of an abstract class?**

1. A class having at least one pure virtual function is called an Abstract class.
2. An Abstract class cannot have objects created, i.e., an abstract class cannot be instantiated, but Object references can be created.
3. An Abstract class can have non-abstract functions and pure virtual functions also.
4. The pure virtual function can have its implementation code in the derived class; otherwise, the derived class will also be considered an abstract Class

**57. What is constructor chaining?**

Constructor chaining is a method to call one constructor from another concerning a current object reference. It can be done in two ways: –

1. Using the “this” keyword, the reference can be made to the constructor in the current class.
2. To call the constructor from the base class “super” keyword will be used.

**58. What is Coupling in OOP, and why is it helpful?**

The degree of dependency between the components is called coupling.

**Types of Coupling**

A. **Tight Coupling** – If the dependency between components is high, these components are called tightly coupled.

B.  **Loose Coupling** – If the dependency between components is low, it is called loose coupling. Loose coupling is preferred because of the following reasons:-

1. It increases the maintainability of code
2. It provides reusability of code

**59. Name the operators that cannot be overloaded**

All the operators except the + operator cannot be overloaded.

**60. What is Cohesion in OOP?**

The modules having well-defined and specific functionality are called cohesion.

**Advantages**

It improves the maintainability and reusability of code.

**61. What are the levels of data abstraction?**

Highlighting the set of services by hiding internal implementation details is called abstraction.

By using abstract Class and interface, we can implement abstraction

**62. What are the types of variables in OOP?**

Variables are basic units to store data in RAM for Java programs.

Variables should be declared before using them in Java programming. Variable initialization can be static or dynamic. The syntax for variable declaration and static initialization is: –

***Types of variables***

**Primitive Variables:**It is used to represent primitive values like int, float, etc.

Reference Variables: It is used to refer to objects in Java.

**Instance Variables:** Variables whose value varied from object to object are instance variables. For every object, a separate copy of the instance variable is created. Instance variables are declared within the Class and outside any method/block/constructor

**Static variables:** For static Variables, a single copy of the variable is created, and that copy is shared between every Class object. The static variable is created during class loading and destroyed at class unloading.

Static variables can be accessed directly from the static and instance area. We are not required to perform initialization explicitly for static variables, and JVM will provide default values.

**Local Variables:** Variables declared inside a method or block or constructor are local variables. Hence the scope of local variables is the same as the block’s scope in which we declared that variable.

JVM doesn’t provide default values, and before using that variable, the initialization should be performed explicitly.

**63. What do you understand by Garbage Collection in the OOPs world?**

Garbage collection is a memory recovery technique included in programming languages like C# and Java. A GC-enabled programming language contains one or more garbage collectors that automatically free up memory space allocated to objects that are no longer needed by the program.

**64. Is it possible to run a Java application without implementing the OOPs concept?**

No, since Java programmes are founded on the concept of object-oriented programming models, or OOPs, a Java application cannot be implemented without it.

# Python

**1. What is Python?**

Python is a high-level, interpreted programming language known for its simplicity and

readability. It emphasizes code readability and encourages a clean and concise coding style.

**2. What are the key features of Python?**

Key features of Python include its easy-to-read syntax, dynamic typing, automatic memory management, extensive standard library, and support for multiple programming paradigms.

**3. How is Python different from other programming languages?**

Python stands out with its simplicity, readability, and easy-to-understand syntax. It has a

large and active community, extensive libraries, and is widely used in various domains such as web development, data analysis, and scientific computing.

**4. What is PEP 8?**

PEP 8 is the official style guide for Python code. It provides guidelines on how to format

Python code to enhance readability and maintain consistency across projects.

**5. What are Python modules?**

Python modules are files containing Python code that define functions, classes, and

variables. They allow code reuse and organization, making it easier to manage and maintain larger projects.

**6. What is a Python package?**

A Python package is a way to organize related modules into a directory hierarchy. It allows for a logical grouping of modules, making it easier to manage and distribute code.

**7. How do you comment in Python?**

Comments in Python are denoted by the # character. Anything after the # is considered a comment and is ignored by the Python interpreter.

**8. What are Python data types?**

Python supports various data types, including integers, floating-point numbers, strings,

lists, tuples, dictionaries, and booleans. Each data type has its own characteristics and uses.

**9. What is type conversion in Python?**

Type conversion, also known as type casting, is the process of converting one data type into another. Python provides built-in functions like int(), float(), str(), etc., to perform type conversion.

**10. What is string interpolation in Python?**

String interpolation in Python allows you to embed expressions or variables within a string,making it easier to construct dynamic strings. It can be done using f-strings or the format()

method.

**11. What are Python conditional statements?**

Python conditional statements, such as if, elif, and else, allow you to perform different

actions based on certain conditions. They control the flow of the program based on the

truthfulness of the conditions.

12. What are Python loops?

Python loops, like for and while, enable you to execute a block of code repeatedly. They

iterate over a sequence or execute until a specific condition is met.

**13. What is the difference between range() and xrange() in Python 2?**

In Python 2, range() generates a list of numbers, while xrange() returns an iterator. xrange() is more memory-efficient for large ranges because it generates values on the fly.

**14. What are Python functions?**

Python functions are reusable blocks of code that perform a specific task. They help in code organization, reusability, and modularity. Functions can accept arguments and return values.

**15. What is the difference between a function and a method in Python?**

In Python, a function is a standalone block of code that can be called independently. A

method, on the other hand, is a function that is associated with an object or a class and can access the object's data.

**16. How do you define a function in Python?**

A function in Python is defined using the def keyword, followed by the function name,

parentheses for parameters (if any), and a colon. The function body is indented below.

**17. What is the \_\_init\_\_ method used for?**

The \_\_init\_\_ method is a special method in Python classes that is automatically called when an object is created from the class. It is used to initialize the object's attributes and perform setup tasks.

**18. What is object-oriented programming (OOP)?**

Object-oriented programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. It emphasizes encapsulation, inheritance, and polymorphism.

**19. What are Python classes and objects?**

In Python, a class is a blueprint that defines the properties and behaviors of objects. An

object is an instance of a class. It represents a specific entity and can interact with other

objects.

**20. How do you create an object in Python?**

An object is created by calling the class as if it were a function. The class acts as a

constructor, initializing the object and returning it.

**21. What is inheritance in Python?**

Inheritance is a mechanism in Python that allows a class to inherit properties and methods from another class. It enables code reuse and supports the creation of hierarchical class structures.

**22. What is method overriding?**

Method overriding is the process of defining a method in a subclass that has the same name as a method in its superclass. The subclass method overrides the implementation of the superclass method.

**23. What is method overloading?**

Method overloading is not directly supported in Python. However, you can achieve similar functionality by defining a single method with default argument values or using variablelength arguments.

**24. What is encapsulation in Python?**

Encapsulation is the process of bundling data and methods together within a class. It allows for data hiding and controlling access to the object's attributes using getter and setter methods.

**25. What is polymorphism in Python?**

Polymorphism is the ability of an object to take on multiple forms or have multiple

behaviors. In Python, polymorphism is achieved through method overriding and method

overloading (using default argument values or variable-length arguments).

**26. What is a generator in Python?**

A generator in Python is a function that returns an iterator. It allows you to generate a

sequence of values on-the-fly, conserving memory and improving performance.

*# Generator function to generate square numbers up to a given limit*def square\_generator(limit):  
 num = 1  
 while num <= limit:  
 yield num \*\* 2  
 num += 1  
  
*# Using the generator to print square numbers up to 10*for square in square\_generator(10):  
 print(square)

Yield returns generator object that can be loop trough for loop or we can call next on generator object. When last element is couth on the next element we get StopInteration error.

**27. What are decorators in Python?**

Decorators are a way to modify the behavior of a function or class without directly changing its source code. They are defined using the @decorator\_name syntax and can be used for tasks like logging, timing, or modifying function arguments.

Some built-in decorators:

* **@staticmethod**: This decorator is used to declare a static method within a class. Static methods do not operate on instances of the class and are primarily used for utility functions.
* **@classmethod**: This decorator is used to declare a class method within a class. Class methods take a reference to the class as their first argument and can access or modify class-level variables.
* **@property**: This decorator is used to define properties in a class. It allows you to define methods that can be accessed like attributes.

Personal decorator:

def log\_function\_calls(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Calling function: {func.\_\_name\_\_}")  
 print(f"Arguments: {args}, {kwargs}")  
 result = func(\*args, \*\*kwargs)  
 print(f"Return value: {result}")  
 return result  
 return wrapper

\*args and \*\*kwargs are special syntax in Python that allow you to pass a variable number of positional arguments and keyword arguments to a function, respectively.

Here's what they mean:

\*args: It stands for "arguments" and is used to pass a variable number of positional arguments to a function. When a function parameter is prefixed with \*, it collects all the positional arguments passed to the function into a tuple. This allows you to define functions that can accept any number of positional arguments.

\*\*kwargs: It stands for "keyword arguments" and is used to pass a variable number of keyword arguments to a function. When a function parameter is prefixed with \*\*, it collects all the keyword arguments passed to the function into a dictionary. This allows you to define functions that can accept any number of keyword arguments.

def example\_function(\*args, \*\*kwargs):  
 print("Positional arguments (\*args):", args)  
 print("Keyword arguments (\*\*kwargs):", kwargs)  
  
*# Example usage of the function*example\_function(1, 2, 3, name='John', age=30)

**28. What is a lambda function in Python?**

A lambda function is an anonymous function in Python that is defined using the lambda

keyword. It is a shorthand way to create small, one-line functions without explicitly defining a function using def.

add = lambda x, y: x + y  
print(add(3, 5)) *# Output will be 8*

**29. What is a module in Python?**

A module in Python is a file containing Python definitions and statements. It can be

imported and used in other Python programs to access its functions, classes, and variables.

**30. How do you import modules in Python?**

Modules can be imported in Python using the import keyword followed by the module name. You can also import specific objects from a module using the from module\_name import object\_name syntax.

**31. What is a virtual environment in Python?**

A virtual environment in Python is a self-contained directory that contains a specific version of Python interpreter and installed packages. It allows you to isolate Python environments for different projects and manage their dependencies.

* python3 -m venv myenv
* myenv\Scripts\activate
* p install package\_name

**32. What are exceptions in Python?**

Exceptions in Python are events that occur during the execution of a program that disrupt the normal flow of the code. They can be handled using try-except blocks to gracefully handle errors and exceptions.

Create own exception:

class CustomError(Exception):  
 *"""Custom exception class."""* def \_\_init\_\_(self, message="An error occurred"):  
 self.message = message  
 super().\_\_init\_\_(self.message)  
  
  
*# Example usage:*def divide(x, y):  
 if y == 0:  
 raise CustomError("Division by zero is not allowed")  
 return x / y  
  
  
try:  
 result = divide(10, 0)  
except CustomError as e:  
 print("Custom Error:", e.message)

**33. What is error handling in Python?**

Error handling in Python involves using try-except blocks to catch and handle exceptions that may occur during the execution of the code. It allows for graceful recovery from errors and prevents the program from crashing.

**34. What is the purpose of the try-except-else-finally block in Python?**

The try-except-else-finally block in Python is used for exception handling. The try block

contains the code that may raise an exception. The except block is used to handle specific exceptions. The else block is executed if no exceptions occur. The finally block is always executed, regardless of whether an exception occurred or not.

**35. What are the built-in data structures in Python?**

Python provides several built-in data structures, including lists, tuples, dictionaries, sets,

and strings. These data structures offer different ways to store, manipulate, and retrieve

data.

**36. What is a list in Python?**

A list in Python is an ordered collection of items that can be of different data types. It is

mutable, meaning its elements can be modified. Lists are denoted by square brackets [ ] and can contain elements separated by commas.

**37. What is a tuple in Python?**

A tuple in Python is an ordered collection of items similar to a list. However, tuples are

immutable, meaning their elements cannot be changed once assigned. Tuples are denoted by parentheses ( ) and can contain elements separated by commas.

**38. What is a dictionary in Python?**

A dictionary in Python is an unordered collection of key-value pairs. It is mutable and allows fast access to values based on their associated keys. Dictionaries are denoted by curly braces { } and use colons : to separate keys and values.

**39. What is a set in Python?**

A set in Python is an unordered collection of unique elements. It is mutable and provides mathematical set operations like union, intersection, and difference. Sets are denoted by curly braces { } or the set() function.

**40. What is a string in Python?**

A string in Python is a sequence of characters enclosed in single quotes, double quotes, or triple quotes. It is immutable, meaning its individual characters cannot be changed. Strings can be manipulated and operated upon in various ways.

**41. How do you concatenate strings in Python**?

Strings can be concatenated in Python using the + operator or by using the .join() method. The + operator concatenates two strings, while the .join() method concatenates multiple

strings using a specified delimiter.

**42. How do you format strings in Python?**

Strings can be formatted in Python using the % operator, the str.format() method, or fstrings (formatted string literals). These methods allow you to insert values into

placeholders within a string.

**43. What are file handling operations in Python?**

File handling operations in Python involve reading from and writing to files. Python provides

built-in functions and methods to open, read, write, and close files.

**44. How do you open and close a file in Python?**

Files can be opened in Python using the open() function, which takes the file name and the mode of operation as arguments. The close() method is used to close an opened file and free up system resources.

**45. What are the different file modes in Python?**

The different file modes in Python include "r" for reading, "w" for writing (overwriting

existing content), "a" for appending, "x" for exclusive creation (fails if the file already exists), and "b" for binary mode.

**46. What is exception handling in file operations?**

Exception handling in file operations involves handling potential errors that may occur while performing file-related operations. This ensures that the program handles file-related exceptions gracefully and avoids crashes or data loss.

**47. What is a context manager in Python?**

A context manager in Python is an object that defines the methods \_\_enter\_\_() and

\_\_exit\_\_() to enable the with statement. It allows for resource allocation and deallocation, such as automatically closing a file after use.

**48. What are megic functions in Python?**

In Python, "magic functions" are special methods that start and end with double underscores, also known as dunder methods. These methods allow you to define functionality that integrates with Python's language constructs. They are called "magic" because they allow you to perform various operations that seem like built-in language features, but they are implemented through these special methods.

Here are a few common magic methods:

1. **\_\_init\_\_(self, ...)**: Initializes an object. This is called when an instance of the class is created.
2. **\_\_str\_\_(self)**: Returns the string representation of an object when **str()** is called on it.
3. **\_\_repr\_\_(self)**: Returns the "official" string representation of an object. This is what is displayed when an object is inspected in the interpreter.
4. **\_\_len\_\_(self)**: Returns the length of an object. This allows you to use the **len()** function on your objects.
5. **\_\_getitem\_\_(self, key)**: Defines behavior for when an item is accessed using the square bracket notation (**obj[key]**).
6. **\_\_call\_\_(self, ...)**: Enables the instance of the class to be called as a function.
7. **\_\_iter\_\_(self)**: Returns an iterator object.
8. **\_\_next\_\_(self)**: Retrieves the next item from the iterator.
9. **\_\_enter\_\_(self)**, **\_\_exit\_\_(self, exc\_type, exc\_value, traceback)**: Implement context management protocols for an object.
10. **\_\_setattr\_\_(**self, name, value): Called when an attribute assignment is attempted (obj.name = value).
11. **\_\_getattr\_\_**(self, name): Called when an attempt to access a non-existent attribute is made (obj.name).
12. **\_\_delattr\_\_(**self, name): Called when an attribute deletion is attempted (del obj.name).
13. **\_\_eq\_\_**(self, other): Defines behavior for equality comparison (obj == other).
14. **\_\_lt\_\_(**self, other), \_\_le\_\_(self, other), \_\_gt\_\_(self, other), \_\_ge\_\_(self, other): Defines behavior for comparison operators (<, <=, >, >=).
15. **\_\_add\_\_(**self, other), \_\_sub\_\_(self, other), \_\_mul\_\_(self, other), \_\_truediv\_\_(self, other): Defines behavior for arithmetic operators (+, -, \*, /).
16. **\_\_contains\_\_(**self, item): Called to implement membership test operators (in, not in).
17. **\_\_hash\_\_(**self): Returns a hash value for the object, which is used in hashing-based data structures like dictionaries and sets.
18. **\_\_format\_\_(**self, format\_spec): Returns a formatted string representation of the object, based on the format\_spec.
19. **\_\_bool\_\_(**self): Defines the truth value of the object when it is used in a boolean context (if obj:).
20. **\_\_index\_\_(**self): Called to convert the object to an integer, used for indexing in sequences like lists or tuples.

**49. What is a list comprehension in Python?**

A list comprehension in Python is a concise way to create lists based on existing lists or

other iterable objects. It allows you to combine looping and conditional logic in a single line of code.

**50. What is the pass statement in Python?**

The pass statement in Python is a placeholder statement that does nothing. It is used as a syntactic placeholder when a statement is required by the Python syntax, but no action is needed.

**51. What is the purpose of the self parameter in Python?**

The self parameter is used as a reference to the current instance of a class in Python. It

allows accessing the attributes and methods of that instance within the class definition.

**52. What is the difference between a shallow copy and a deep copy in Python?**

In Python, a shallow copy creates a new object that references the original data, while a

deep copy creates a new object with completely independent copies of the original data.

Modifying the original data does not affect the deep copy, but it can affect the shallow copy.

**53. What are the advantages of using Python for web development?**

Python offers several advantages for web development, including a wide range of

frameworks (such as Django and Flask), a large community, extensive libraries, and easy

integration with other technologies.

**54. What is the Global Interpreter Lock (GIL) in Python?**

The Global Interpreter Lock (GIL) is a mechanism in the CPython interpreter (the reference implementation of Python) that allows only one thread to execute Python bytecode at a time. This restricts the parallel execution of Python threads and can impact performance in certain scenarios.

**55. What is a metaclass in Python?**

A metaclass in Python is a class that defines the behavior and structure of other classes. It allows you to customize class creation, modify attributes, and add additional functionality to classes.

56. How do you handle file I/O errors in Python?

File I/O errors in Python can be handled using exception handling. By using try-except

blocks around file-related operations, you can catch specific exceptions like

FileNotFoundError or PermissionError and handle them gracefully.

**57. What is the purpose of the \_\_name\_\_ variable in Python?**

The \_\_name\_\_ variable in Python is a built-in variable that represents the current module's

name. It can be used to determine whether a module is being run as the main script or

imported as a module.

**58. What is the difference between a shallow comparison and a deep comparison in Python?**

In Python, a shallow comparison checks if two objects have the same memory address,

while a deep comparison checks if the objects have the same values. Shallow comparisons can be done using the is operator, while deep comparisons are typically done using the == operator.

**59. What are the advantages of using virtual environments in Python?**

Virtual environments in Python provide a dedicated environment for each project, allowing you to isolate project dependencies, avoid conflicts between packages, and maintain

project-specific versions of Python and packages.

**60. What is the purpose of the \_\_main\_\_ block in Python?**

The \_\_main\_\_ block in Python is used to define the entry point of a Python program. The code inside the if \_\_name\_\_ == "\_\_main\_\_": block will only execute if the script is run directly, not when it is imported as a module.

**61. What is the purpose of the \_\_str\_\_ method in Python?**

The \_\_str\_\_ method in Python is a special method that returns a string representation of an object. It is used to provide a human-readable representation of the object when the str() function is called or when the object is printed.

**62. What is the purpose of the \_\_repr\_\_ method in Python?**

The \_\_repr\_\_ method in Python is a special method that returns a string representation of an object that can be used to recreate the object. It is used to provide a detailed and

unambiguous representation of the object.

**63. What is the difference between the \_\_str\_\_ and \_\_repr\_\_ methods in Python?**

The \_\_str\_\_ method is intended to provide a human-readable string representation of an object, while the \_\_repr\_\_ method is intended to provide a detailed and unambiguous string representation that can be used to recreate the object.

**64. What is the purpose of the super() function in Python?**

The super() function in Python is used to call a method in a superclass or parent class. It is often used in method overriding to invoke the superclass's implementation of the method

before adding additional functionality in the subclass.

**65. What is the purpose of the \_\_getitem\_\_ method in Python?**

The \_\_getitem\_\_ method in Python is a special method that allows objects to define

behavior for indexing and slicing operations. It is called when an item is accessed using

square brackets ([]) and supports accessing items by index or slicing.

**66. What is the purpose of the \_\_setitem\_\_ method in Python?**

The \_\_setitem\_\_ method in Python is a special method that allows objects to define

behavior for assigning values to items using square brackets ([]). It is called when an item is assigned a value using indexing.

**67. What is the purpose of the \_\_len\_\_ method in Python?**

The \_\_len\_\_ method in Python is a special method that returns the length of an object. It is

called when the len() function is used on an object.

**68. What is the purpose of the \_\_iter\_\_ method in Python?**

The \_\_iter\_\_ method in Python is a special method that returns an iterator object. It is used

to make an object iterable, meaning it can be looped over using a for loop or used with other

iterator-related functions and constructs.

**69. What is the purpose of the \_\_next\_\_ method in Python?**

The \_\_next\_\_ method in Python is a special method that returns the next item in an iterator. It is called by the next() function and is used in conjunction with the \_\_iter\_\_ method to

create custom iterators.

**70. What is the purpose of the @property decorator in Python?**

The @property decorator in Python is used to define a method as a getter for a class

attribute. It allows accessing the attribute as if it were a normal attribute, while internally

calling the getter method.

**71. What is the purpose of the @staticmethod decorator in Python?**

The @staticmethod decorator in Python is used to define a static method in a class. Static

methods do not require an instance of the class to be called and can be accessed directly

from the class itself.

**72. What is the purpose of the @classmethod decorator in Python?**

The @classmethod decorator in Python is used to define a class method. Class methods

receive the class itself as the first parameter, allowing them to access and modify classlevel attributes and perform operations specific to the class.

**73. What is the purpose of the \_\_call\_\_ method in Python?**

The \_\_call\_\_ method in Python is a special method that allows an object to be called as if itwere a function. It is called when parentheses are used to invoke the object.

**74. What is the purpose of the \*args and \*\*kwargs parameters in Python?**

The \*args parameter in Python allows a function to accept a variable number of positional arguments as a tuple, while the \*\*kwargs parameter allows a function to accept a variable number of keyword arguments as a dictionary. This flexibility allows functions to handle

different numbers and types of arguments.

**75. What are decorators in Python?**

Decorators in Python are a way to modify or enhance the behavior of functions or classes without directly modifying their source code. Decorators are implemented as functions that

wrap around the target function or class and add additional functionality.

**76. What is the purpose of the @classmethod decorator in Python?**

The @classmethod decorator in Python is used to define a class method. Class methods receive the class itself as the first parameter, allowing them to access and modify classlevel attributes and perform operations specific to the class.

**77. What is a lambda function in Python?**

A lambda function in Python is an anonymous function that can be defined in a single line. It is often used for simple, one-time operations and does not require a formal def statement.

**78. What are modules in Python?**

Modules in Python are files that contain Python code and definitions. They can be imported and used in other Python programs to provide reusable functionality.

**79. What are packages in Python?**

Packages in Python are a way to organize related modules into a directory hierarchy. They allow for better organization and modularization of code, making it easier to manage large projects.

**80. What is the purpose of the \_\_init\_\_.py file in a package?**

The \_\_init\_\_.py file in a package serves as an indicator that the directory is a Python

package. It can be empty or contain initialization code that is executed when the package is imported.

**81. What is the purpose of the sys module in Python?**

The sys module in Python provides access to system-specific parameters and functions. It allows interaction with the Python interpreter and provides information about the runtime environment.

**82. What is the purpose of the os module in Python?**

The os module in Python provides a way to interact with the operating system. It allows

performing various operations related to file and directory manipulation, process

management, and environment variables.

**83. What is the purpose of the datetime module in Python?**

The datetime module in Python provides classes for manipulating dates and times. It allows creating, formatting, and performing operations on dates and times.

**84. What are decorators in Python?**

Decorators in Python are a way to modify or enhance the behavior of functions or classes without directly modifying their source code. Decorators are implemented as functions that wrap around the target function or class and add additional functionality.

**85. What is the purpose of the @property decorator in Python?**

The @property decorator in Python is used to define a method as a getter for a class

attribute. It allows accessing the attribute as if it were a normal attribute, while internally

calling the getter method.

**86. What is the purpose of the @staticmethod decorator in Python?**

The @staticmethod decorator in Python is used to define a static method in a class. Static methods do not require an instance of the class to be called and can be accessed directly

from the class itself.

**87. What is the purpose of the @classmethod decorator in Python?**

The @classmethod decorator in Python is used to define a class method. Class methods

receive the class itself as the first parameter, allowing them to access and modify classlevel attributes and perform operations specific to the class.

**88. Explain threading in python?**

Threading in Python allows you to run multiple threads (smaller units of a process) concurrently within a single process. This enables you to perform multiple tasks simultaneously, which can be useful for tasks such as I/O-bound operations and parallel processing.

*# Define a function that will be executed in a separate thread*def print\_numbers():  
 for i in range(5):  
 print(f"Number: {i}")  
 time.sleep(1)  
  
*# Create a thread object with the target function*thread = threading.Thread(target=print\_numbers)  
  
*# Start the thread*thread.start()  
  
*# Main thread continues to execute while the new thread runs concurrently*for letter in 'ABCDE':  
 print(letter)  
 time.sleep(1)  
  
*# Wait for the thread to finish (optional)*thread.join()  
  
print("Main thread and the new thread have finished execution.")

Threading can be useful for tasks that involve I/O operations or tasks that can be parallelized. However, due to Python's Global Interpreter Lock (GIL), threading may not provide significant performance improvements for CPU-bound tasks. For CPU-bound tasks, you may want to consider using the multiprocessing module, which allows true parallel execution

Using Thread-safe Data Structures: Python's queue.Queue and queue.LifoQueue classes from the queue module provide thread-safe implementations of FIFO and LIFO queues, respectively. These data structures are designed to be accessed by multiple threads concurrently without the need for explicit locking.by spawning multiple processes.

**89. Multiprocessing** in Python is a module that allows you to create and manage multiple processes concurrently. It is particularly useful for CPU-bound tasks where the Global Interpreter Lock (GIL) in Python's threading model limits concurrency.

The multiprocessing module provides various classes and functions for creating and managing processes, communication between processes, and synchronization. Some important components of the multiprocessing module include:

* Process: Represents an individual process.
* Pool: Manages a pool of worker processes.
* Queue and Pipe: Allow communication between processes.
* Locks, Semaphores, and Events: Provide synchronization mechanisms for coordinating between processes.

When using multiprocessing, keep in mind that each process has its own memory space, and data is not shared between processes by default. However, you can use mechanisms such as queues, pipes, and shared memory to exchange data between processes when necessary

Multiprocessing is particularly effective for CPU-bound tasks and can leverage multiple CPU cores to achieve parallelism and improve performance. However, creating and managing processes incurs more overhead compared to threads, so multiprocessing may not be suitable for all use cases.

**90. What is locking mechanism?**

Locking is a mechanism used in concurrent programming to synchronize access to shared resources, ensuring that only one thread or process can access the resource at any given time. In Python, locks are typically implemented using the threading.Lock or multiprocessing.Lock classes, depending on whether you're working with threads or processes.

**The Global Interpreter Lock (GIL)** is a mutex (mutual exclusion) that protects access to Python objects, preventing multiple native threads from executing Python bytecodes simultaneously. It ensures that only one thread executes Python bytecode at any given time, even in a multi-threaded Python program.

The GIL has implications for multi-threaded Python programs:

* Limitation on Multi-Core CPU Usage: Because of the GIL, multi-threaded Python programs cannot fully utilize multiple CPU cores for CPU-bound tasks. This is because, despite multiple threads being created, only one thread can execute Python bytecode at any moment, effectively limiting parallelism.
* I/O-bound Operations: While the GIL restricts CPU-bound tasks, it has less impact on I/O-bound operations such as network I/O, disk I/O, and other external system calls. In these cases, threads can release the GIL while waiting for I/O operations to complete, allowing other threads to execute Python code in the meantime.
* Impact on CPU-bound Performance: CPU-bound tasks that heavily rely on computation may not see significant performance improvements from threading due to the GIL. In such cases, alternative approaches like multiprocessing or using external libraries (e.g., NumPy, Cython) that release the GIL during computation can be more effective.
* Memory Management: The GIL simplifies memory management and makes the CPython interpreter (the standard Python interpreter) implementation easier to maintain and debug. However, it also introduces limitations in terms of concurrency.

It's important to note that the GIL is specific to the CPython interpreter, which is the reference implementation of Python. Other Python implementations such as Jython and IronPython do not have a GIL.

**91. What is the purpose of the \_\_init\_\_.py file in a package?**

The \_\_init\_\_.py file in a package serves as an indicator that the directory is a Python

package. It can be empty or contain initialization code that is executed when the package is imported.

**92. What is the purpose of the sys module in Python?**

The sys module in Python provides access to system-specific parameters and functions. It allows interaction with the Python interpreter and provides information about the runtime environment.

**93. What is the purpose of the os module in Python?**

The os module in Python provides a way to interact with the operating system. It allows

performing various operations related to file and directory manipulation, process

management, and environment variables.

**94. What is the purpose of the datetime module in Python?**

The datetime module in Python provides classes for manipulating dates and times. It allows creating, formatting, and performing operations on dates and times.

**95. What is the purpose of the random module in Python?**

The random module in Python provides functions for generating random numbers. It allows you to generate random integers, floating-point numbers, and make random selections from lists.

**96. What is the purpose of the json module in Python?**

The json module in Python provides functions for working with JSON (JavaScript Object

Notation) data. It allows encoding Python objects into JSON strings and decoding JSON

strings into Python objects.

**97. What is the purpose of the pickle module in Python?**

The pickle module in Python provides functions for serializing and deserializing Python

objects. It allows you to convert Python objects into a binary format that can be stored ortransmitted, and then restore them back into objects.

**98. What are generators in Python?**

Generators in Python are functions that can be paused and resumed, allowing them to

produce a sequence of values over time. They are memory-efficient and provide a

convenient way to iterate over large or infinite sequences.

**99. What is the purpose of the yield keyword in Python?**

The yield keyword in Python is used in the context of generators. It allows a generator

function to temporarily pause and yield a value to the caller, without losing its internal state.

The generator can then be resumed to continue execution from where it left off.

**100. What is the purpose of the zip() function in Python?**

The zip() function in Python is used to combine multiple iterables (such as lists or tuples) into a single iterable of tuples. It pairs up corresponding elements from each iterable, stopping when the shortest iterable is exhausted.