Task 2

Q1 What languages have Garbage Collector?

1.How garbage collection works

Garbage collection works by tracking the references to the objects in the memory heap, and identifying the ones that are unreachable or inaccessible by the program. These objects are considered garbage, and can be safely removed from the memory. Depending on the language and the implementation, garbage collection can use different algorithms and strategies to perform this task, such as mark-and-sweep, generational, reference counting, or tracing.

2.Benefits of garbage collection

Garbage collection can simplify the programming process, as it relieves the programmers from the burden of manually managing the memory allocation and deallocation of objects. This can reduce the chances of errors, such as memory leaks, dangling pointers, double frees, or buffer overflows, that can compromise the performance and security of the program. Garbage collection can also improve the memory efficiency, as it can reclaim the unused or wasted memory space, and reduce the memory fragmentation.

3.Drawbacks of garbage collection

Garbage collection is not a perfect solution, and it can also introduce some drawbacks and challenges. One of the main drawbacks is the overhead and unpredictability of the garbage collection process, which can affect the performance and responsiveness of the program. Garbage collection can cause pauses or delays in the execution of the program, as it needs to stop or slow down the program to perform the memory cleanup. This can be problematic for real-time or interactive applications, where latency and consistency are critical. Garbage collection can also consume more memory and CPU resources, as it needs to maintain and update the information about the references and the objects in the memory.

4.Examples of languages with garbage collection

Only c and c++ who deosn't have a GC , they need to be able to run in bare metal with minimal runtime as possible but GC requires a runtime cost.

Q2 What is the 13 principles of Clean Code?

1. Don't Repeat Yourself (DRY)

This principle suggests that code should not have unnecessary duplication. Instead, it should be organized in a way that avoids redundancy and makes it easy to maintain. For example, instead of writing the same calculation in multiple places in the code, create a function that performs the calculation and call that function from the different places where the calculation is needed.

2. Write Everything Twice (WET)

This is an opposite principle of DRY. It suggest that if you find yourself copy-pasting code multiple times, anticipating the identical code forking in different directions later on, having WET code may make that future change easier.

```
class Dog {
 name = "Dog";
 move() {
 console.log("Dog is moving");
 // future implementation
 // console.log("Dog is trotting");
 }
}

class Cat {
 name = "Cat";
 move() {
 console.log("Cat is moving");
 // future implementation
 // console.log("Cat is sneaking");
 }
}
// with DRY, we would have a Animal with move method
// however, dog and cat could be moving in different ways,
// so we want to keep them seperate following WET
```

3. Single Responsibility Principle (SRP)

Each module or function should have only one reason to change. For example, instead of having a function that handles multiple tasks, split it up into multiple functions, each with a single responsibility.

4. Open/closed Principle (OCP)

A module or function should be open for extension but closed for modification. For example, instead of modifying an existing class to add new functionality, create a new class that extends the original class and add the new functionality there.

5. Liskov Substitution Principle (LSP)

Objects of a superclass should be able to be replaced with objects of a subclass without altering the correctness of the program. For example, a subclass should be able to replace its parent class without breaking the program.

```
class Bird {
 fly() {
 console.log("I am flying");
 }
}

class Ostrich extends Bird {
 fly() {
 throw "I can't fly";
 }
}
let bird = new Bird();
bird.fly(); // Output: "I am flying"
```

```
let ostrich = new Ostrich();
ostrich.fly(); // Output: "I can't fly"
let birds = [new Bird(), new Ostrich()];
for (let bird of birds) {
 bird.fly();
}
// Output: "I am flying" "I can't fly"
```

6. Interface Segregation Principle (ISP)

A client should not be forced to implement interfaces it doesn't use.

For example, instead of having a monolithic interface with many methods, split it up into smaller, more specific interfaces.

7. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. For example, instead of having a high-level module depend on a specific implementation of a low-level module, have it depend on an abstraction of the low-level module.

```
// bad example
class UserController {
 constructor() {
 this.userRepository = new MySQLUserRepository();
 }
 ...
}

// good example
class UserController {
 constructor(userRepository) {
 this.userRepository = userRepository;
 }
 ...
}

let mysqlUserRepository = new MySQLUserRepository();
let userController = new UserController(mysqlUserRepository);
```

8. Keep It Simple, Stupid (KISS)

This principle suggests that code should be as simple as possible, and should avoid unnecessary complexity. For example, instead of using a

complex algorithm to solve a problem, use a simpler one that gets the job done.

9. You Aren't Gonna Need It (YAGNI)

This principle suggests that code should not be written until it is actually needed, as it can add unnecessary complexity and make the code harder to maintain. For example, instead of adding a feature that may be needed in the future, focus on the features that are needed now.

10. Fail Fast

This principle suggests that code should fail as early as possible, so that issues can be identified and resolved quickly. For example, instead

of waiting until the end of a function to check for errors, check for errors as soon as possible.

11. Law of Demeter (LoD)

This principle suggests that an object should only communicate with its immediate neighbors and should not reach into the internal state of other objects. For example, instead of accessing the internal state of an object, use a method to get the information you need.

12. Command Query Separation (CQS)

It is a principle that suggests that methods should either be command methods that change the state of an object, or query methods that return information about an object, but not both. For example, instead of having a method that both changes the state of an object

and returns a value, have separate methods for changing the state and returning the value.

13. Composition over Inheritance

It suggests that code should favor composition over inheritance, as composition allows for greater flexibility and easier maintenance. For example, instead of inheriting properties and methods from a parent class, compose objects with the properties and methods they need.

Q3 How to do a (do while)loop in python ?

To create a do while loop in Python, you need to modify the while loop a bit in order to get similar behavior to a do while loop in other languages.

As a refresher so far, a do while loop will run at least once. If the condition is met, then it will run again.

The while loop, on the other hand, doesn't run at least once and may in fact never run. It runs when and only when the condition is met.

The while

```
total = 0
# loop will run at least once
while     :
    # ask the user to enter a number
    num = int(input("Enter a number (or 0 to exit): "))

    # exit the loop if the user enters 0
    if num == 0:
        break
    total += num

# print the total
print("Total:", total)
```

Q4 while loop VS for loop?

Aspect	For Loop	While Loop
Syntax	for item in iterable:	while condition:
Iteration behavior	It iterates through the specified sequence by itself.	It depends on the condition, whether it is 'True' or 'False'.
Number of	The number of iterations	The number of iterations is

iterations	is known or finite.	not known.
Execution	Execution depends on the specific number of iterations.	It repeats the execution based on the specified condition.
Termination	It will terminate once all elements have gone through iteration.	It may result in an infinite loop if the condition never evaluates to 'False'.
Efficiency	It is considered more efficient.	It is less efficient as compared to the other.
Error Potential	It iterates through a sequence, hence, less prone to infinite loops.	It may lead to frequent infinite loops if conditions are not managed carefully.

Q5 What is the same as pass in java ,c++?

In c++ => no, you just use an empty block like: void function(int parameter) { }

In Java => a single semicolon with nothing in front of it indicates an empty statement.

Q6 What is the most common tracing tools in python ?

1.Helios is a purpose-built observability platform that aims to streamline debugging and troubleshooting in Python applications. It offers a comprehensive solution by providing end-to-end visibility into the workflow of an application, empowering developers to identify and resolve issues effectively. By leveraging the powerful context

propagation framework of OpenTelemetry, Helios offers E2E visibility into your system across microservices, serverless functions, databases, and 3rd party APIs, enabling you to quickly identify, reproduce and resolve issues.

2.Uptrace is a feature-rich observability platform that surpasses conventional logging and error-tracking solutions. It offers an all-encompassing approach by providing robust tracing and debugging capabilities specifically designed for Python applications. This empowers developers to identify and resolve issues with ease swiftly. Uptrace's user-friendly interface and many features enhance the debugging process and elevate observability.

3.SigNoz is a powerful open-source Application Performance Monitoring (APM) system that offers extensive visibility into Python applications. With its advanced distributed tracing capabilities, SigNoz empowers developers to seamlessly trace requests and pinpoint performance bottlenecks across various services and components. It provides a comprehensive suite of features dedicated to monitoring, troubleshooting, and optimizing application performance, ensuring a smooth and efficient user experience.