1. Why is using "**while True**" not a good idea? Is there any situation in that we have to use it?

In general, using “while True” makes the source code much harder to maintain and read (it’s impossible to see what is the break condition at a glance) and it encourages other bad programming practices such as using complicated conditions and “break” statements to exit out of the loop. It might be acceptable under very specific situations such as when you have multiple threads and you have a “worker” thread always running in the background or when waiting for communication from another component, but even then, it should be used at a minimum.

1. Why is using the "**eval"**and**"exec"** functions a bad practice and may cause a security issue?

Eval is a function used to evaluate an expression and get it executed as long as it is valid python syntax. Exec, on the other hand, is used to dynamically execute python programs by passing a string or some object code. The problem is that both of these functions can be exploited and abused because you could potentially run any unsafe code or script with them and this leads to security issues since the programmer calling them has no control on what the parameters passed to them will do.

1. What is pdb (the Python Debugger)?

This is a module that can be used with Python scripts to have an interactive source code debugger so that programmers can set breakpoints and perform stepping through the code and other more advanced debugging techniques such as inspecting stack frames. This is particularly useful if the programmer is not using an IDE, but rather the terminal window to code their python programs.

1. Why do we use "\_" before the name of a variable?

Placing a leading underscore before the variable’s name makes the variable “non-public” and therefore for internal use only (meaning, that no other classes should be calling on these variables).

1. What are Dunder/Magic Methods in Python? Explain with an example.

The dunder or magic methods in Python are those that are defined by prepending two underscores to the method’s name. These methods are defined by built-in classes in Python and can be overloaded. For example, the methods “\_\_eq\_\_” (for equality or comparison) and “\_\_add\_\_” (for adding an item).

1. What is the "self" in your code? Why do we use it in a method's header?

Using “self” in Python code refers to a representation of the instance of the class that calls the method or property. For example, when one says “self.my\_variable”, the programmer is effectively calling a property called “my\_variable” from the current instance of the class that is running. We use it in the method’s header to indicate that it is a non-static method (the method can only be used if the class that includes it has been instantiated).

1. What is the difference between a **list** and a **tuple**? When do we use them?

A list is a generic representation of a collection of objects that is mutable which means it can be modified after it is instantiated (you can add or remove items after creation). A tuple, on the other hand, is an ordered immutable collection (meaning, once they have been declared in instantiation, they cannot be changed). Thus, we should use lists when we expect contents to change constantly or frequently and tuple when we expect contents whose values do not change frequently.

1. What is **encapsulation**? Why do we use it?

Encapsulation is one of the principles of object-oriented programming by which access to properties and/or methods of classes may be restricted by various levels of restriction to prevent external objects from making modifications to those components. We use it to contain changes and force classes to keep limits in what they can and cannot do.

1. What is a constructor in a class and how can we define it?

The constructor is the first method in any class that gets called when a programmer creates an instance of that class. In python, it is defined by using the keyword “\_\_init\_\_”, for example “def \_\_init\_\_(self):” Unless the class is static, all classes call their constructor when they are initialized.

1. What are **Overriding** and **Overloading**?

Overriding is a type of run-time Polymorphism where a child class provides a different implementation to a method that its parent class already has defined. This allows for the child class to have different behavior than its parent class. Overloading, on the other hand, is a type of compile-time Polymorphism where the same name can be used for two or more methods with different parameters for each of those methods: this allows to consolidate functionality into one theme (the name of the methods), but with different implementations as parameters provided.

1. What is **Polymorphism**? Explain it with examples.

Polymorphism is another core concept in object-oriented programming by which objects can have more than one form or representation with additional functionality for those additional representations. For example, a super class could represent a generic “Feline” object with basic features any feline could share, but you could also have sub classes such as “cat” and “lion” that represent specific types of felines with more distinctive behaviors.

1. What are the differences between a **list** and a **set**?

In a Python list, we can have a collection of many objects and these objects can be duplicates of other objects in the same list whereas in a set all objects are unique meaning only one instance of a value can be found in a set. For example, in a list you could have [1,2,3,1,2,3] where 1,2,3 repeat whereas in a set you could only have 1,2,3 once.

1. What is the difference between the Python **identity** operator (**is**/**is not**) and the **equality** operator (**==**/**!=**)? Where to use them and where not?

The identity operator is used to compare the identity of two objects and know if two references refer to the same object whereas the equality operator is used to compare the value of two objects. In conditional statements when we want to compare the value of variables, we should use the equality operator and when we are checking if two instances are referencing the same object.

1. What is the difference between OOP and Functional programming?

OOP stands for object-oriented programming and it is a programming paradigm that declares pieces of functionality in a program in terms of objects so that modeling these programs is easier, uses encapsulation, and takes up concepts from the real world and abstract them out into reusable components. Functional programming, on the other hand, is concerned with writing programs with the goal of achieving single functionality by breaking logic into functions to perform calculations with simple inputs/outputs.

1. What are iterators, generators, and decorators in Python?

In Python, iterators are classes that implement “\_\_iter\_\_” and “\_\_next\_\_” methods such that these classes, when instantiated into objects, can be iterated through with a for loop or with a next function. An iterator is a method that uses the “yield” statements to iterate through properties or variables. A decorator allows to modify the behavior of a Python function or class.

1. What are passing by value and passing by reference? Explain it with an example.

When a parameter is passed by value, a copy of that variable is made such that any modifications to the copy won’t affect the original variable. On the other hand, passing by reference means that the parameter is passed as a pointer (its address in memory) and therefore a change inside the method that receives that variable will have an impact on the original variable. Example:

# Passing by Value

def increase\_counter(counter: int) -> None:

counter += 1

print (“The counter is “ + counter)

counter = 0

# Before calling the function…

print(counter) # this prints 0

increase\_counter(counter) # this now prints 1

# After calling the function…

print(counter) # this prints 0

# Passing by Reference

def add\_to\_list(my\_list: List[int]) -> None:

my\_list.append(4)

my\_unique\_list = [1, 2]

# Before calling the function…

print(my\_unique\_list) # this prints [1, 2]

add\_to\_list(my\_unique\_list)

# After calling the function…

print(my\_unique\_list) # this prints [1, 2, 4]

1. What is a **round-off** error in python? Explain it with an example.

A round-off error in Python is the difference between the approximation of a number used in calculating something and its correct true value. In Python, the most common round-off error is with float numbers. In Python, adding three numbers such as 0.1 + 0.2 + 0.3 == 0.6 returns False as a result. This demonstrates a round-off error. In above case, using a Python function such as round would help, example round(0.1 + 0.2 + 0.3, 5) == round (0.6, 5) would return True.

1. Explain **map()**, **filter()**, and **reduce()** in Python with Examples.

These are higher-order functions because they receive another function as an argument and then apply that function in some way.

With map() function, a function can be passed to it to transform each element in an iterable object. For example:

old\_list = [1, 2, 3, -1, -5]

new\_list = list(map(str, old\_list)) # Here, the map function takes the str function and applies it to each value in the old\_list list. Thus, new\_list becomes a list of strings with the string representation of each number so new\_list = [“1”, “2”, “3”, “-1”, “-5”]

With filter(), a function that returns Boolean is expected and only the elements of the list that meet the condition (for which true is returned) are returned. In below example, these elements are also being added to new\_list.

def isPositive (number: int) -> bool:

if number % 2 == 0:

return True

else:

return False

new\_list = list(filter(isPositive, old\_list) # With filter function, the new\_list = [1, 2, 3]

With reduce(), a function is passed and reduce will reduce the list to a single value by combining elements. In below example, a lambda expression (considered a function) is used to multiply two values and returns its result. When applied by the reduce() function, the reduce() function will return one combined result. Example:

result = reduce((lambda x, y: x \* y), [1,2,3,4])

print(result) # this prints 24