**ADVANCED OOP PYTHON PYQs**

*Authored by Arkajyoti Naskar*

**Answer any 6 questions : 6 X 5 = 30**

**1. Write a program to perform addition, subtraction, multiplication, integer division, floor division and modulo division on two integer and float using Python. (5)**

**# Arithmetic operations on integers**

**a = 10**

**b = 3.5**

**addition\_int = a + b**

**print("Addition:", addition\_int) #13.5**

**subtraction\_int = a - b**

**print("Subtraction:", subtraction\_int) #6.5**

**multiplication\_int = a \* b**

**print("Multiplication:", multiplication\_int) #35.0**

**integer\_division = a // b**

**print("Integer Division:", integer\_division) #2.0**

**floor\_division = a / b**

**print("Floor Division:", floor\_division) #2.8571…**

**modulo\_division = a % b**

**print("Modulo Division:", modulo\_division) #3.0**

**2. Describe in detail about lambda functions or anonymous function. Describe list slicing with examples in Python. 2+3=5**

Python Lambda Functions are anonymous function means that the function is without a name. As we already know that the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in Python. Example:- n= lambda n:n\*n\*n returns n cube. lambda x,y,z:(x+y+z)/3 returns average of three values, lambda s:s.trim().upper() returns s trimmed and in uppercase.

Lambda functions can be used wherever function objects are required. This function can have any number of arguments but only one expression, which is evaluated and returned. They are syntactically restricted to a single expression. Like nested function definitions. Another use is to pass a small function as an argument.

Lambda functions can be generated inline. Example - print((lambda l:sum(l)/len(l))(lst1)) prints sum of lst1 divided by length of lst1. Lambda functions can be used with map, reduce, filter. Example-lst2=map(lambda n:n\*n, lst1)

**List Slicing:**

List slicing refers to accessing a specific portion or a subset of the list for some operation while the original list remains unaffected. The slicing operator in Python can take 3 parameters out of which 2 are optional depending on the requirement.

In Python, list slicing is a common practice and it is the most used technique for programmers to solve efficient problems. Consider a python list, In-order to access a range of elements in a list, you need to slice a list. One way to do this is to use the simple slicing operator i.e. colon(:), With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list

list\_name[start:stop:steps]

Another ways for list slicing is

By passing just the start or stop parameter: my\_first\_list[3:]

By passing the start and stop parameter: my\_first\_list[1 : 4]

By passing the start, stop, and steps parameters: my\_first\_list[0 : 3 : 1]

**3. “Tuples are immutable”. Explain with Examples. Write a Python program to create a histogram from a given list of integers. 3+2=5**

Tuple is one of 4 built-in data types in Python used to store collections of data, A tuple is a collection which is ordered and unchangeable. Tuples are written with round brackets. Tuples are immutable data structures in Python, which means that once a tuple is created, its elements cannot be changed and ordered and allow duplicate values. Tuples in Python.

* We can find items in a tuple since finding any item does not make changes in the tuple.
* One cannot add items to a tuple once it is created.
* Tuples cannot be appended or extended.
* We cannot remove items from a tuple once it is created.

**def histogram(lst):**

**for item in lst:**

**print("\* " \* item)**

**length = int(input("Enter List Length : "))**

**lst = []**

**for \_ in range(length):**

**temp = int(input("Enter List Item : "))**

**lst.append(temp)**

**print("Histogram is ... ")**

**histogram(lst)**

**4. Difference between built in exceptions and handling exception in Python. Write a program to catch a Divide by zero exception. Add a finally block too. 3+2=5**

**Built-in Exception:** All instances in Python must be instances of a class that derives from BaseException. Two exception classes that are not related via subclassing are never equivalent, even if they have the same name. The built-in exceptions can be generated by the interpreter or built-in functions.

There are several built-in exceptions in Python that are raised when errors occur. These built-in exceptions can be viewed using the local() built-in functions as follows:

>>> locals()['\_\_builtins\_\_']

This returns a dictionary of built-in exceptions, functions and attributes.

These exceptions are pre-defined in the language and can be raised automatically when certain conditions are met. Some common built-in exceptions include ZeroDivisionError, TypeError, ValueError, and FileNotFoundError etc.

**Exception handling:** Exception handling allows you to gracefully handle and recover from exceptions that occur during program execution. It involves using the try, except, and finally blocks to catch and handle specific exceptions. The try block encloses the code that may potentially raise an exception, while the except block is used to catch and handle specific exceptions. The finally block, if specified, will always execute regardless of whether an exception was raised or not.

**try:**

**dividend = 10**

**divisor = 0**

**result = dividend / divisor**

**print("Result:", result)**

**except ZeroDivisionError:**

**print("Error: Division by zero is not allowed.")**

**finally:**

**print("Program execution completed.")**

**\*5. How does Generalization differ from Encapsulation. Explain with examples in Python. What is monkey patching in Python? 3+2=5**

**Encapsulation** is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. It allows you to hide the internal details of an object and expose only the necessary functionality through methods. Encapsulation helps achieve data abstraction, information hiding, and modularity.

**Generalization:** Generalization is a mechanism in OOP that allows you to create a hierarchy of classes where more specific classes inherit the properties and behaviors of more general classes. It enables you to define common attributes and methods in a base class and then extend or specialize them in derived classes.

Monkey patching is the technique of dynamic modification of a piece of code at the run time. Actually by doing monkey patch we change the behavior of code but without affecting the original source code.

|  |
| --- |
| # monk.py  **class** A:  **def** func(self):            print ("func() is being called") |

We use above module (monk) in below code and change behavior of func() at run-time by assigning different value.

|  |
| --- |
| **import** monk  **def** monkey\_f(self):  **print** ("monkey\_f() is being called")  # replacing address of "func" with "monkey\_f"  monk.A.func **=** monkey\_f  obj **=** monk.A()  # calling function "func" whose address got replaced  # with function "monkey\_f()"  obj.func() |

Examples:

Output :monkey\_f() is being called

**6. What is multithreading. How to create a thread in Python. 2+3=5**

Multithreading is a programming concept that involves the execution of multiple threads within a single process. Each thread represents a separate flow of execution and can perform tasks concurrently with other threads, providing the illusion of parallelism. Threads share the same memory space and resources of a process, but they have their own program counters, registers, and stack space. . In Python, the Global Interpreter Lock (GIL) prevents the threads from running simultaneously. If 2 processes are called simultaneously, one may execute before other finishes executing and mess up the output. For this we use the join method.

Creating a thread in Python can be achieved by using the threading module.

**import threading**

**# Function to be executed by a thread**

**def print\_numbers():**

**for i in range(1, 6):**

**print(i)**

**# Create a new thread**

**thread = threading.Thread(target=print\_numbers)**

**# Start the thread**

**thread.start()**

**# Wait for the thread to complete**

**thread.join()**

**# The main thread resumes execution after the other thread finishes**

**print("Thread execution completed.")**

**7. Write a Python script to display current date and time. What is the difference between append() and extend() methods? 2+3=5**

**import datetime**

**# Get current date and time**

**current = datetime.datetime.now()**

**# Display current date and time**

**print("Current date and time:")**

**print(current.strftime("%d-%m-%y %H:%M:%S"))**

Effect: .append() adds a single element to the end of the list while .extend() can add multiple individual elements to the end of the list.

Argument: .append() takes a single element as argument while .extend() takes an iterable as argument (list, tuple, dictionaries, sets, strings).

**# append() example**

**my\_list = [1, 2, 3]**

**my\_list.append(4)**

**print(my\_list) # Output: [1, 2, 3, 4]**

**# extend() example**

**my\_list = [1, 2, 3]**

**my\_list.extend([4, 5, 6])**

**print(my\_list) # Output: [1, 2, 3, 4, 5, 6]**

**8. What is self-keyword in Python? Explain with an example. Why would you use NumPy arrays instead of lists in Python? 2+3=5**

**Self:** In Python, the self keyword is used as a reference to the instance of a class within the class itself. It represents the current object being accessed or manipulated.  It binds the attributes with the given arguments.

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on. Self is always pointing to Current Object.

**I’ll use NumPy arrays instead of lists in Python because-**

1. NumPy may use much less memory to store data than regular Python lists. The difference is mostly due to "indirectness" -- a Python list is an array of pointers to Python objects, at least 4 bytes per pointer plus 16 bytes for even the smallest Python object (4 for type pointer, 4 for reference count, 4 for value -- and the memory allocators rounds up to 16). A NumPy array is an array of uniform values -- single-precision numbers takes 4 bytes each, double-precision ones, 8 bytes.
2. Python lists are an array of pointers referring to objects in memory. Each time we refer to a memory object, Python first retrieves the pointer and then goes to the memory location of the pointer to access that object—this causes a significant performance decrease in lists. However, NumPy arrays are homogeneous. They store only one type of data in continuous memory locations by which the access time of an object takes very little or no time.
3. Access in reading and writing items is also faster with NumPy. NumPy arrays are more efficient for numerical computations and large datasets compared to regular Python lists. NumPy provides a wide range of mathematical functions and operations that can be applied directly to arrays, making it easier to perform complex calculations and manipulations on large datasets.

**\*9. How will you read a random line in a file? Define pickling and unpickling in Python. 2+3=5**

**import random**

**def random\_line(fname):**

**lines = open(fname).read().splitlines()**

**return random.choice(lines)**

**print(random\_line('test.txt'))**

**Pickling and unpickling** in Python refer to the process of serializing (pickling) and deserializing (unpickling) Python objects into a byte stream and vice versa. This allows objects to be stored or transmitted in a compact format, which can be later reconstructed to their original form.

Pickling: Pickling is the process of converting a Python object into a byte stream. The pickle module in Python provides functions to perform pickling. It serializes the object hierarchy, including nested objects, into a binary format that can be written to a file or transferred over a network.

Unpickling: Unpickling is the reverse process of pickling. It involves converting a byte stream back into a Python object. The pickle module provides the load() or loads() function to deserialize the byte stream and reconstruct the original object.

**import pickle**

**# Pickling**

**data = [1, 2, 3, 4, 5]**

**with open('data.pkl', 'wb') as file:**

**pickle.dump(data, file)**

**# Unpickling**

**with open('data.pkl', 'rb') as file:**

**loaded\_data = pickle.load(file)**

**print(loaded\_data) # Output: [1, 2, 3, 4, 5]**

**10. Compare List and Tuple sequences in Python. 3.5 [CO1]**

Lists and tuples are both sequence types in Python, but they have some differences in terms of mutability, syntax, and usage. Here's a comparison between lists and tuples:

**Mutability:**

Lists are mutable, which means you can modify, add, or remove elements after the list is created. You can change individual elements, extend the list, or remove items using methods like append(), extend(), insert(), remove(), etc.

Tuples, on the other hand, are immutable. Once a tuple is created, you cannot change its elements. Tuples are fixed in size and cannot be modified, added to, or removed from.

**Syntax:**

Lists are defined using square brackets [], and elements are separated by commas. For example: my\_list = [1, 2, 3]

Tuples are defined using parentheses (), and elements are also separated by commas. For example: my\_tuple = (1, 2, 3)

**Usage**:

Lists are commonly used when you have a collection of items that may need to be modified or updated over time. They are suitable for scenarios where you need to store and manipulate dynamic data.

Tuples are often used when you have a collection of items that should remain fixed and not change. They are useful for representing items that are conceptually related and should be treated as a single entity. Tuples are also useful for returning multiple values from a function.

**Performance:**Lists generally have a slightly higher memory overhead compared to tuples because lists are mutable, requiring extra memory for potential modifications.

Tuples are generally faster to access than lists because they are immutable, allowing for some performance optimizations.

**11. Discuss the following methods associated with the file object**

**a) read() b) readline() c) readlines() d) tell() e) seek() f) write(). 3.5 [CO1]**

a) **read():** This method reads and returns the entire contents of a file as a single string. It takes an optional parameter specifying the number of characters to read. If no parameter is provided, it reads the entire file.

**Example:**

**with open('file.txt', 'r') as file:**

**content = file.read()**

**print(content)**

b) **readline():** This method reads and returns a single line from a file. It reads characters from the current position until it encounters a newline character or reaches the end of the file. With subsequent calls to readline(), it continues reading the next lines.

**Example :**

**with open('file.txt', 'r') as file:**

**line1 = file.readline()**

**line2 = file.readline()**

**print(line1)**

**print(line2)**

c) **readlines():** This method reads all lines from a file and returns them as a list of strings. Each string represents a line, including the newline character at the end of each line. It is useful when you want to process the lines individually.

**Example :**

**with open('file.txt', 'r') as file:**

**lines = file.readlines()**

**for line in lines:**

**print(line)**

d) **tell():** This method returns the current position (in bytes) within the file. The position represents the next byte that will be read or written. It can be useful for keeping track of the file's current position during reading or writing operations.

**with open('file.txt', 'r') as file:**

**file.read(10)**

**# Read 10 characters**

**position = file.tell()**

**print("Current position:", position)**

e) **seek():** This method changes the current position within the file. It takes two parameters: offset (the number of bytes to move) and whence (the reference point from where the offset is applied). Common values for whence are 0 (beginning of the file), 1 (current position), and 2 (end of the file).

**with open('file.txt', 'r') as file:**

**file.seek(5) # Move to the 6th byte in the file**

**data = file.read()**

**print(data)**

f) **write():** This method writes a string or a sequence of bytes to a file. It appends the content at the current position or overwrites existing content if the file is opened in write or append mode. It returns the number of characters or bytes written.

**Example :**

**with open('file.txt', 'w') as file:**

**file.write('Hello, world!')**

**\*12. Explain try, catch and finally block with an example in Python.3.5 [CO1]**

The try, except, and finally blocks are used in Python for handling exceptions. They allow you to catch and handle potential errors or exceptions that may occur during the execution of a program. The try block encloses the code that may potentially raise an exception, while the except block is used to catch and handle specific exceptions. The finally block, if specified, will always execute regardless of whether an exception was raised or not.

**try:**

**dividend = 10**

**divisor = 0**

**result = dividend / divisor**

**print("Result:", result)**

**except ZeroDivisionError:**

**print("Error: Division by zero is not allowed.")**

**finally:**

**print("Program execution completed.")**

**13. Write a Python program to read lines from a file, break into tokens and**

**convert the tokens to unique numerical values using Python dictionary.3.5**

**def convert\_tokens\_to\_numeric(file\_path):**

**token\_dict = {}**

**unique\_value = 1**

**with open(file\_path, 'r') as file:**

**for line in file:**

**tokens = line.strip().split() # Split line into tokens**

**for token in tokens:**

**if token not in token\_dict:**

**token\_dict[token] = unique\_value**

**unique\_value += 1**

**return token\_dict**

**# Example usage**

**file\_path = 'input.txt' # give full path**

**result\_dict = convert\_tokens\_to\_numeric(file\_path)**

**print(result\_dict)**

**14. Discuss different mutable and immutable data types in Python.7 [CO1]**

In Python, data types can be classified as either mutable or immutable. This classification determines whether the object's state can be modified after it is created. Here's a discussion of different mutable and immutable data types in Python:

**Mutable Data Types:**

**Lists:** Lists are one of the most commonly used mutable data types in Python. They are ordered collections of elements that can be modified, extended, or shrunk. You can add or remove elements using methods like append(), extend(), insert(), remove(), and more.

**Sets:** Sets are unordered collections of unique elements. They can be modified by adding or removing elements. Sets provide methods such as add(), remove(), discard(), pop(), and set operations like union, intersection, difference, and more.

**Dictionaries:** Dictionaries are key-value pairs that store data. They are mutable and allow you to add, update, or delete key-value pairs. Dictionaries provide methods such as update(), pop(), del, and more for modifying the data.

**Bytearrays:** Bytearrays are mutable sequences of bytes. They are similar to strings but can be modified by changing the value of individual bytes.

**Immutable Data Types:**

**Numeric Data Types :** Python’s built-in numeric data types such as integers, booleans, floats, complex numbers, fractions, and decimals are immutable. Once it’s created , its value cannot be changed.

**Tuples:** Tuples are ordered collections of elements, similar to lists. However, tuples are immutable, and their elements cannot be modified once they are created. Tuples are defined using parentheses.

**Strings:** Strings are sequences of characters and are immutable in Python. Once a string is created, its characters cannot be modified individually.

**Frozensets:** Frozensets are immutable versions of sets. They behave like sets, but once created, their elements cannot be modified.

**Bytes:** Bytes are immutable sequences of bytes. They are used to represent binary data and are commonly used in I/O operations.

Immutable data types ensure data integrity and can be useful in scenarios where you want to guarantee that the data remains unchanged. Mutable data types provide flexibility for modifying data when needed.

**\*15. Search for palindrome and unique words in a text using class method and string method. 4+3=7/5+5=10 [CO4]**

**class TextAnalyzer:**

**def \_\_init\_\_(self, text):**

**self.text = text**

**@classmethod**

**def find\_palindromes(cls,text):**

**palindromes = []**

**words = text.split()**

**for word in words:**

**if word.lower() == word.lower()[::-1]:**

**palindromes.append(word)**

**return palindromes**

**@staticmethod**

**def find\_unique\_words(text):**

**words = text.split()**

**unique\_words = set(words)**

**return unique\_words**

**# Example usage**

**text = "level kayak python radian madam anaconda civic"**

**# Using class method to find palindromes**

**palindromes = TextAnalyzer.find\_palindromes(text)**

**print("Palindromes:", palindromes)**

**# Using string method to find unique words**

**unique\_words = TextAnalyzer.find\_unique\_words(text)**

**print("Unique Words:", unique\_words)**

**16. In Python, explain the accessibility of the members of a class across the other classes within and outside the package. Explain multiple inheritance in Python with an example. 3+4=7 [CO1]**

In Python, the accessibility of members (attributes and methods) of a class across other classes depends on their visibility, which is controlled by access modifiers. There are three levels of visibility in Python:

Public: Public members are accessible from anywhere, both within the class and outside the class or package. They can be accessed using the dot notation (object.member).

Protected: Protected members are denoted by a single underscore \_ prefix. They are intended to be internal to the class or its subclasses. While not enforced by the language, it is a convention that protected members should not be accessed directly from outside the class or its subclasses.

Private: Private members are denoted by a double underscore \_\_ prefix. They are intended to be used only within the class itself. Private members are name-mangled by the interpreter, which means they are renamed to incorporate the class name. While it is still possible to access private members using the mangled name, it is considered a bad practice.

Multiple inheritance: it allows a class to inherit attributes and methods from multiple parent classes. In Python, a class can inherit from multiple classes by specifying them in the class definition within parentheses. The order of the parent classes matters, as it determines the method resolution order (MRO) when resolving method calls.

**class Parent1:**

**def greet(self):**

**print("Hello from Parent1")**

**class Parent2:**

**def greet(self):**

**print("Hello from Parent2")**

**class Child(Parent1, Parent2):**

**pass**

**child = Child()**

**child.greet()**

In this example, we have two parent classes, Parent1 and Parent2, both having a greet method. The Child class inherits from both parent classes. When we create an instance of Child and call the greet method, Python uses the MRO to determine the order of method resolution.

**17. What is operator overloading? Write a class in Python to represent complex**

**numbers with necessary constructors. Write methods or functions for the following:**

**Overloading the operator “+”.**

**Overloading the operator “\*”.**

**Overloading the operator “<<” so that a complex number is displayed in “a+ib”**

**form.**

**Overload “>” operator. A complex number “a+ib” is greater than “c+id” if “a2+b2”**

**is greater than “c2+d2”. 3+4=7 [CO1]**

Operator overloading is a feature in Python that allows operators to be used with objects of user-defined classes. By overloading operators, you can define the behavior of the operators for your custom classes.

**class ComplexNumber:**

**def \_\_init\_\_(self, real, imaginary):**

**self.real = real**

**self.imaginary = imaginary**

**def \_\_add\_\_(self, other):**

**if isinstance(other, ComplexNumber):**

**return ComplexNumber(self.real + other.real, self.imaginary + other.imaginary)**

**else:**

**raise TypeError("Unsupported operand type for +")**

**def \_\_mul\_\_(self, other):**

**if isinstance(other, ComplexNumber):**

**real = self.real \* other.real - self.imaginary \* other.imaginary**

**imaginary = self.real \* other.imaginary + self.imaginary \* other.real**

**return ComplexNumber(real, imaginary)**

**else:**

**raise TypeError("Unsupported operand type for \*")**

**def \_\_str\_\_(self):**

**return f"{self.real}+{self.imaginary}i"**

**def \_\_gt\_\_(self, other):**

**if isinstance(other, ComplexNumber):**

**self\_magnitude = self.real \*\* 2 + self.imaginary \*\* 2**

**other\_magnitude = other.real \*\* 2 + other.imaginary \*\* 2**

**return self\_magnitude > other\_magnitude**

**else:**

**raise TypeError("Unsupported operand type for >")**

**# Example**

**c1 = ComplexNumber(3, 4)**

**c2 = ComplexNumber(1, 2)**

**c3 = c1 + c2**

**print(c3) # Output: 4+6i**

**c4 = c1 \* c2**

**print(c4) # Output: -5+10i**

**print(c1 > c2) # Output: True**

**19. Create book, ebook, journal classes to use inheritance with title, publisher, page, year of publishing details. 5**

**class Publication:**

**def \_\_init\_\_(self, title, publisher, pages, year):**

**self.title = title**

**self.publisher = publisher**

**self.pages = pages**

**self.year = year**

**class Book(Publication):**

**def \_\_init\_\_(self, title, publisher, pages, year, author):**

**super().\_\_init\_\_(title, publisher, pages, year)**

**self.author = author**

**def display\_details(self):**

**print("Book Details:")**

**print("Title:", self.title)**

**print("Author:", self.author)**

**print("Publisher:", self.publisher)**

**print("Pages:", self.pages)**

**print("Year:", self.year)**

**print()**

**class Ebook(Publication):**

**def \_\_init\_\_(self, title, publisher, pages, year, format):**

**super().\_\_init\_\_(title, publisher, pages, year)**

**self.format = format**

**def display\_details(self):**

**print("Ebook Details:")**

**print("Title:", self.title)**

**print("Format:", self.format)**

**print("Publisher:", self.publisher)**

**print("Pages:", self.pages)**

**print("Year:", self.year)**

**print()**

**class Journal(Publication):**

**def \_\_init\_\_(self, title, publisher, pages, year, volume, issue):**

**super().\_\_init\_\_(title, publisher, pages, year)**

**self.volume = volume**

**self.issue = issue**

**def display\_details(self):**

**print("Journal Details:")**

**print("Title:", self.title)**

**print("Volume:", self.volume)**

**print("Issue:", self.issue)**

**print("Publisher:", self.publisher)**

**print("Pages:", self.pages)**

**print("Year:", self.year)**

**print()**

**# Example usage:**

**book = Book("The Great Gatsby", "Scribner", 180, 1925, "F. Scott Fitzgerald")**

**book.display\_details()**

**ebook = Ebook("Python Tricks", "No Starch Press", 400, 2017, "PDF")**

**ebook.display\_details()**

**journal = Journal("Nature", "Nature Publishing Group", 50, 2022, "Volume 596", "Issue 7871")**

**journal.display\_details()**

**\*18. Discuss advantages of Python over Java as an Object-Oriented Programming Language. 7/10 [CO1]**

1. It is simpler and easier to use than Java. Python is a scripting language, but Java inherits much of the complexity and syntax of systems languages such as C++.

2. Dynamic typing Because Python is dynamically typed, it's fast and friendly for development. Additionally, it uses asynchronous code to tackle situations and challenges faster because each unit of code runs separately.

3. It offers extensive libraries: Its large library supports common tasks and commands. It also contains code that can be used for many different purposes, ranging from generating documentation to unit testing to CGI.

4.It offers a more flexible approach to programming: Python supports a variety of programming styles and has multiple paradigms. Easy integration with and extensibility using C and Java

5. In Java, multiple inheritances are partially done through interfaces, whereas Python supports both single and multiple inheritances.

6. Python Codes are 5-10 times more concise and more versatile, easier to read, and has a simpler syntax.

7. Easy to study, reusable and maintainable language, readability, coherence and software quality.

8. Expansibility, Component integration, Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms.

9. Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA.

10. Python’s standard pickle module provides a simple object persistence system—it allows programs to easily save and restore entire Python objects to files and file-like objects.

**19. How multiple threads are created and ended in Python? How will they synchronize?**

**Discuss these with an example to solve the producer-consumer problem. Producer and Consumer are the two entities here who share the same buffer. The producer can either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. 7 [CO2]**

* 2 ways for Inter Threaded Communication- using Event (wait,set and clear) and using Condition(acquire release wait notify notifyall)
* In event, wait method waits for the flag to be set before exection of codes below it. set and clear are used to change the flags.

Example-

def fun1():

while True:

ev.wait();#wait for flag to be clear

#do corresponding work in the thread

ev.clear() #clear the flag again

def fun2():

while True:

#perform work

#set the flag

#ev.set()

ev=Event()

th1=threading.Thread(target=fun1)

th2=threading.Thread(target=fun2) #fun2 executes its work first and then fun 1 work is executed once fun2 work finishes

* Condition uses lock release acquire wait notify and notifyall. The wait method release the lockuntil a notify wakes it up in another thread. Acquire and release have previous functionalities(block and unblock like your ex), acquire waits if the current thread is blocked.

Producer/Consumer problem-

#Producer thread

cond.acquire()

#produce one item

cond.notify()

cond.release()

#Consumer thread

cond.acquire()

while(item unavailable):

cond.wait

cond.release()

* TO sum it up, for multithreading, we have lock,rlock which can be utilised by acquire and release. For interthreaded communication we use condition, which has got acquire,release,wait,notify and notifyAll, we also have Event() which is used by set wait and clear. Rest notify and notifyall concepts are same as in JAva.
* To achieve producer consumer by using Events-

ev=Event()

#we have set wait and clear

#Producer code

ev.set()

#Create object

ev.clear()

#Consumer code

ev.set()

if(Item unavailable):

ev.wait()

ev.clear()

**20. Discuss the following list functions- a) len(), b) sum(), c) any(), d) all(), e)sorted(). Write first seven Fibonacci numbers using generator next function/yield in python. Trace and memorize the function.(5+5=10)**

Certainly! Let's discuss each of the list functions you mentioned:

1. **len():** The len() function is used to determine the length or the number of elements in a list. It takes a list as an argument and returns an integer representing the number of elements in the list.

For example: my\_list = [1, 2, 3, 4, 5] length = len(my\_list) print(length) # Output: 5

1. **sum():** The sum() function is used to calculate the sum of all the elements in a list. It takes a list of numbers as an argument and returns the sum of those numbers.

my\_list = [1, 2, 3, 4, 5] total\_sum = sum(my\_list) print(total\_sum) # Output: 15

**c) any():** The any() function is used to check if at least one element in a list satisfies a given condition. It takes an iterable (such as a list) as an argument and returns True if at least one element in the iterable evaluates to True, otherwise it returns False. For example:

my\_list = [False, False, True, False] result = any(my\_list) print(result) # Output: True

In this example, the any() function returns True because at least one element in my\_list (the third element) is True.

**d) all():** The all() function is used to check if all the elements in a list satisfy a given condition. It takes an iterable (such as a list) as an argument and returns True if all elements in the iterable evaluate to True, otherwise it returns False. For example:

my\_list = [True, True, True, True] result = all(my\_list) print(result) # Output: True

In this example, the all() function returns True because all elements in my\_list are True.

**e) sorted():** The sorted() function is used to sort the elements of a list in ascending order (by default) and return a new sorted list. It takes an iterable as an argument and returns a new list containing the sorted elements. The original list remains unchanged. For example:

my\_list = [5, 2, 1, 4, 3] sorted\_list = sorted(my\_list) print(sorted\_list) # Output: [1, 2, 3, 4, 5]

In this example, the sorted() function sorts the elements of my\_list in ascending order and returns a new list [1, 2, 3, 4, 5].

**def fibonacci\_generator(): # function to generate fibonacci numbers.**

**a, b = 0, 1**

**yield a**

**yield b**

**while True:**

**c = a + b**

**yield c**

**a, b = b, c**

**fib\_itr= fibonacci\_generator()#A generator for Fibonacci numbers**

**for \_ in range(7):**

**print(next(fib\_itr))# Print next Fibonacci number from generator**

**21.Create a list of tuples, each tuple should contain an item and its price in float, write a program to sort the tuples in descending order by price. Use operator: itemgetter().**

**Write a program that proves that the dictionary returned by globals() can be used to manipulate values of variables in it. (5+5=10)**

To sort the list of tuples in descending order by price using the itemgetter() operator, you can follow the example below:

**from operator import itemgetter**

**# Define the list of tuples**

**items = [("Apple", 2.5), ("Banana", 1.75), ("Grapes", 2.0), ("Orange", 3.25)]**

**# Sort the list in descending order by price**

**sorted\_items = sorted(items, key=itemgetter(1), reverse=True)**

**# Print the sorted list**

**for item in sorted\_items:**

**print(item)**

In this example, we import the itemgetter() function from the operator module. The itemgetter(1) is used as the key argument for the sorted() function, specifying that we want to sort the tuples based on the second element (the price). The reverse=True parameter is provided to sort the tuples in descending order.

The output will be:

('Orange', 3.25) ('Apple', 2.5) ('Grapes', 2.0) ('Banana', 1.75)

As you can see, the tuples are sorted in descending order based on the price.

-Here's an example program that demonstrates how you can use the globals() dictionary to manipulate the values of variables:**# Declare some variablesa = 10b = "Hello, World!"c = [1, 2, 3]# Access the globals() dictionary and update variable valuesglobals()["a"] = 20globals()["b"] = "Hello!"globals()["c"].append(4)# Print the updated variable valuesprint("Updated values:")print("a =", a)print("b =", b)print("c =", c)**In this example, we have three variables: a, b, and c. We use the globals() function to access the dictionary representing the global namespace, which includes all the defined variables. By manipulating the values within this dictionary using the variable names as keys, we can update the values of the corresponding variables.The output will be:

a = 20b = Hello!c = [1, 2, 3, 4]

**22. Explain different functional programming features in Python. Write a program to implement a stack data structure of specific size. If the stack is full and we are trying to push an item then IndexError exception should be raised. Similarly, if the stack is empty then an IndexError exception should be raised.(5+5=10)**

**Different Functional Programming features in Python are-**

**Map** applies a function to each element in the sequence and returns a new sequence containing the result.

**def multiply(x, y):**

**return x\*y**

**products =map(multiply, [1,2],[3,4]) # [1\*3,2\*4]=[3,8]**

**Filter** operation applies a function to all the elements of a sequence and returns sequence of those elements for which result returns True.

**def func(n): n%5==0?True:False**

**lst1=[45,23,42,65]**

**lst2=filter(func,lst1) #returns [45,65]**

**Reduce** operation performs a rolling computation to sequential values in a sequence and prints the result. It returns a single value, calls a binary function on the first two items and then the result with the third and iterates so on.

**def getsum(x,y):**

**return x+y**

**lst=[1,2,3,4,5]**

**s=reduce(getsum,lst) # returns 1+2+3+4+5 =15**

Here's an example program that implements a stack data structure with a specific size and raises IndexError exceptions when appropriate:

**class Stack:**

**def \_\_init\_\_(self, size):**

**self.size = size**

**self.stack = []**

**def push(self, item):**

**if len(self.stack) == self.size:**

**raise IndexError("Stack is full. Cannot push item.")**

**self.stack.append(item)**

**def pop(self):**

**if len(self.stack) == 0:**

**raise IndexError("Stack is empty. Cannot pop item.")**

**return self.stack.pop()**

**def is\_empty(self):**

**return len(self.stack) == 0**

**def is\_full(self):**

**return len(self.stack) == self.size**

**# Example usage: stack = Stack(3)**

**# Pushing items into the stack**

**stack.push("Item 1")**

**stack.push("Item 2")**

**stack.push("Item 3")**

**# Trying to push when the stack is full**

**try:**

**stack.push("Item 4")**

**except IndexError as e:**

**print("Error:", str(e))**

**# Popping items from the stack**

**print(stack.pop()) # Output: Item 3**

**print(stack.pop()) # Output: Item 2**

**print(stack.pop()) # Output: Item 1**

**# Trying to pop when the stack is empty**

**try:**

**print(stack.pop())**

**except IndexError as e:**

**print("Error:", str(e))**

**23. Why Python is called as dynamic and strongly typed language? Discuss the ord(), hex(), oct(), complex(), and float() type conversion functions with examples(5+5=10)**

**Dynamic Typing:** In Python, variables are not explicitly declared with a specific data type. Instead, the type of a variable is determined at runtime based on the value assigned to it. This flexibility allows variables to hold different types of values over the course of a program's execution. For example, a variable initially assigned an integer value can later be reassigned a string value without requiring any explicit type conversions. This dynamic behaviour is one of the reasons Python is considered a dynamically typed language.

In a dynamically typed language, the interpreter does not assign a type to the variable per se because the type can change at runtime. If you ask a variable its type, it will give you the type of the object it is currently assigned to at that moment.

**x = 4**

**print(type(4)) # at this moment, x points to an integer**

**x = "Hello, world"**

**print(type(x)) # and at this moment, x points to a string**

**Strongly typed:** If a language is Strongly typed than it should follow some important rule that While the language is compiled or interpreted it must keep track of all the variables and constants that they are assigned to some data-type. In python programming language variables are not declared with the type of variable. But as the code interpreted the interprets keep track of all the type of variable in it.

Hence the variables initialize are assigned to the type of the category they belong it follows the most important rule of being a strongly typed language. And so, we can say that the language is strongly typed.

Strong typing means that Python enforces strict rules regarding type compatibility and prevents certain operations between incompatible types. For example, you cannot perform arithmetic operations between a string and an integer directly without explicitly converting one of the operands. This helps prevent unintended consequences and enhances code reliability.

**1 + "1" # TypeError in Python**

**1 + "1" // "11" in JavaScript**

Let's discuss each of the type conversion functions you mentioned along with examples:

**a) ord():** The ord() function is used to get the Unicode code point of a character. It takes a single character as an argument and returns an integer representing its Unicode code point. For example:

**char = 'A'**

**unicode\_code = ord(char)**

**print(unicode\_code) # Output: 65**

In this example, the ord() function returns the Unicode code point of the character 'A', which is 65.

**b) hex():** The hex() function is used to convert an integer to a lowercase hexadecimal string prefixed with "0x". It takes an integer as an argument and returns a string representation of the hexadecimal value.

**number = 15**

**hex\_string = hex(number)**

**print(hex\_string) # Output: 0xf**

In this example, the hex() function converts the integer 15 to its hexadecimal representation '0xf'.

**c) oct():** The oct() function is used to convert an integer to an octal (base 8) string prefixed with "0o". It takes an integer as an argument and returns a string representation of the octal value. For example:

**number = 10**

**oct\_string = oct(number)**

**print(oct\_string) # Output: 0o12**

In this example, the oct() function converts the integer 10 to its octal representation '0o12'.

**d) complex():** The complex() function is used to create a complex number. It takes two arguments, representing the real and imaginary parts, and returns a complex number. If only one argument is provided, it represents the real part, and the imaginary part is set to 0. For example:

**num1 = complex(2, 3)**

**num2 = complex(5)**

**print(num1) # Output: (2+3j)**

**print(num2) # Output: (5+0j)**

In this example, we create complex numbers num1 with a real part of 2 and an imaginary part of 3, and num2 with a real part of 5 and an imaginary part of 0.

**e) float():** The float() function is used to convert a string or an integer to a floating-point number. It takes a single argument and returns the corresponding floating-point value. For example:

**number\_str = "3.14"**

**float\_number = float(number\_str)**

**print(float\_number) # Output: 3.14**

**number\_int = 5**

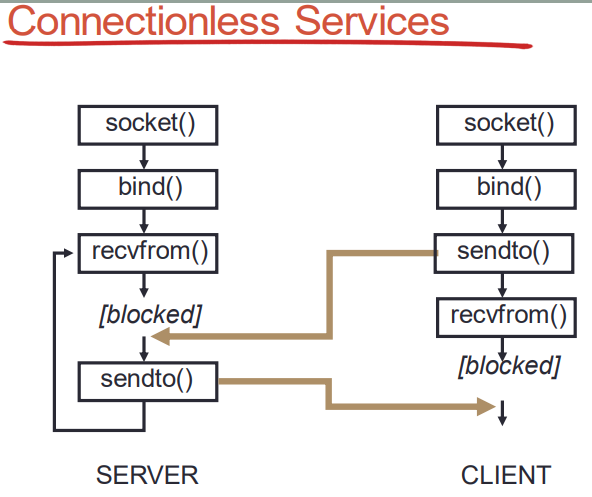
**float\_number = float(number\_int)**

**print(float\_number) # Output: 5.0**

In this example, the float() function converts a string "3.14" and an integer 5 to their corresponding floating-point values.

**24. Write a Python program to design a simple connectionless server, explaining the connectionless service. 7 [CO2]**

How to get UDP- For UDP for the server everything is same till the point we receive details from the specific client socket. We send a message via the client socket to the server socket and then we extract the socket data from it if we want to send a message to the client from the server afterwards. We do not connect directly to the socket from the client and therefore we do not receive a direct connection request in the server side so that the operation socketdetails,addr=s.accept() can be executed, instead we do socketdetails,addr=s.recvfrom(portnumber)



**from socket import socket, AF\_INET, SOCK\_DGRAM**

**# Server configuration**

**HOST = '127.0.0.1' # Server hostname**

**PORT = 12345 # Server port number**

**# Create a UDP socket**

**s = socket(AF\_INET, SOCK\_DGRAM)**

**# Bind the socket to a specific host and port**

**s.bind((HOST, PORT))**

**print('Server listening on {}:{}'.format(HOST, PORT))**

**# Server loop**

**while True:**

**# Receive data from the client**

**data, address = s.recvfrom(1024)**

**message = data.decode() # Decode the received data**

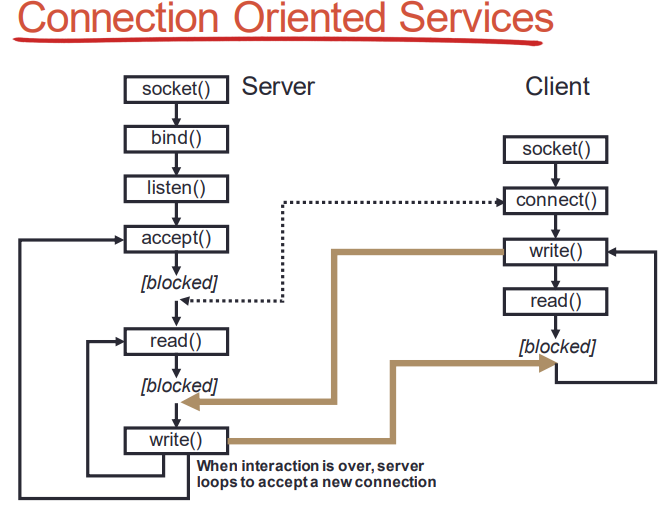
**# Process the received message**

**print("Connection From",address)**

**# Send the response back to the client**

**s.sendto(message.upper().encode(), address)**

**25. Write a Python program to design a simple connection-oriented server, explaining the connection-oriented service. (10)**



**from socket import socket, AF\_INET, SOCK\_STREAM**

**# Server configuration**

**HOST = '127.0.0.1' # Server hostname**

**PORT = 12345 # Server port number**

**BACKLOG = 5 # Number of pending connections in the queue**

**s = socket(AF\_INET, SOCK\_STREAM) # Create a TCP socket**

**s.bind((HOST, PORT)) # Bind the socket to a specific host and port**

**s.listen(BACKLOG) # Listen for incoming connections**

**print('Server listening on {}:{}'.format(HOST, PORT))**

**while True:**

**connection\_socket, addr = s.accept() # Accept a client connection**

**# Server waits on accept() for incoming requests, new socket created on return.**

**print('Connected to client:', addr)**

**data = connection\_socket.recv(1024)# Receive data from the client**

**message = data.decode().upper()#Read bytes from socket(not address)**

**print('Received from client:', message)**

**# Send the response back to the client**

**connection\_socket.sendall(message.encode())**

**connection\_socket.close() # Close connection to this client**

A connection-oriented service is a communication protocol that provides a reliable and ordered data transmission between two endpoints. It establishes a dedicated connection between the client and the server, ensuring that data is delivered in the correct order and without loss or corruption.

Basically, create a server for TCP by creating a socket first and then binding the socket to a specific port. Now the difference between a socket and a port for our perspective is that if we bind a socket to a port, we can use that socket to listen to all the incoming and send messages via that port. Therefore, we first bind the server to a port and then we start listening for connection requests from it. Then we run an infinite loop in which when a accept a request and extract the socket details and address from it. Then we send further messages to that address and receive messages in bytecode format and utf 8 encoding. For the client side, we first create socket and then connect to the same port as the server. Then we send and receive messages in bytecode and utf8.

**26. Create a GUI application in Python that provides an Entry field where the user can provide the name of a text file. Open the file and read it, displaying its contents in a Label. You can also replace the Entry widget with menu that has a File Open option that pops up a window to allow the user to specify the file to read. Also add an Exit or Quit option to the menu rather than having a QUIT button. (10)**

**import tkinter as tk**

**from tkinter import filedialog**

**def open\_file():**

**# Open a file dialog to choose a text file**

**filepath = filedialog.askopenfilename(filetypes=[("Text Files", "\*.txt")])**

**# Read the contents of the selected file**

**with open(filepath, 'r') as file:**

**file\_contents = file.read()**

**# Update the label with the file contents**

**label.config(text=file\_contents)**

**def exit\_application():**

**root.destroy()**

**root = tk.Tk() # Create the main application window**

**root.title("File Reader")**

**# Create a menu bar**

**menu\_bar = tk.Menu(root)**

**root.config(menu=menu\_bar)**

**# Create a File menu**

**file\_menu = tk.Menu(menu\_bar, tearoff=False)**

**menu\_bar.add\_cascade(label="File", menu=file\_menu)**

**# Add Open option to the File menu**

**file\_menu.add\_command(label="Open", command=open\_file)**

**# Add Exit option to the File menu**

**file\_menu.add\_command(label="Exit", command=exit\_application)**

**# Create a label to display the file contents**

**label = tk.Label(root, wraplength=400)**

**label.pack(padx=10, pady=10)**

**root.mainloop()# Start the main event loop**