Python Coding Interview Questions - KP

Compiled Languages

Compiled languages are converted directly into machine code that the processor can execute.

- tend to be faster and more efficient to execute than interpreted languages.
- they also give the developer more control over hardware aspects, like memory management and CPU usage.
- a "build" step – they need to be manually compiled first. You need to "rebuild" the program every time you need to make a change.

Examples of pure compiled languages are C, C++, Rust, and Go.

Interpreted Languages

Interpreters run through a program line by line and execute each command.

Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time compilation, that gap is shrinking.

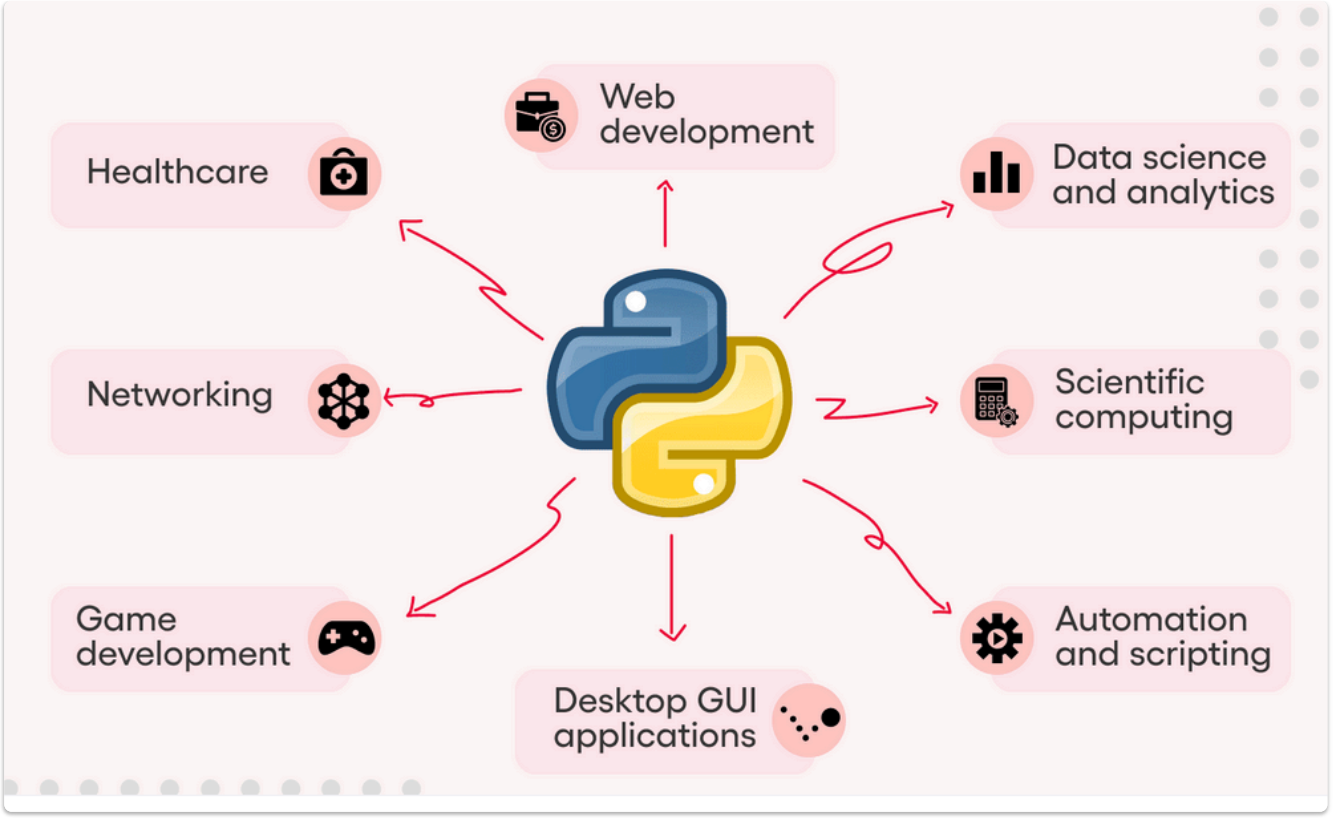
> *Just-in-time compilation is a method for improving the performance of interpreted programs. During execution the program may be compiled into native code to improve its performance. It is also known as dynamic compilation.*

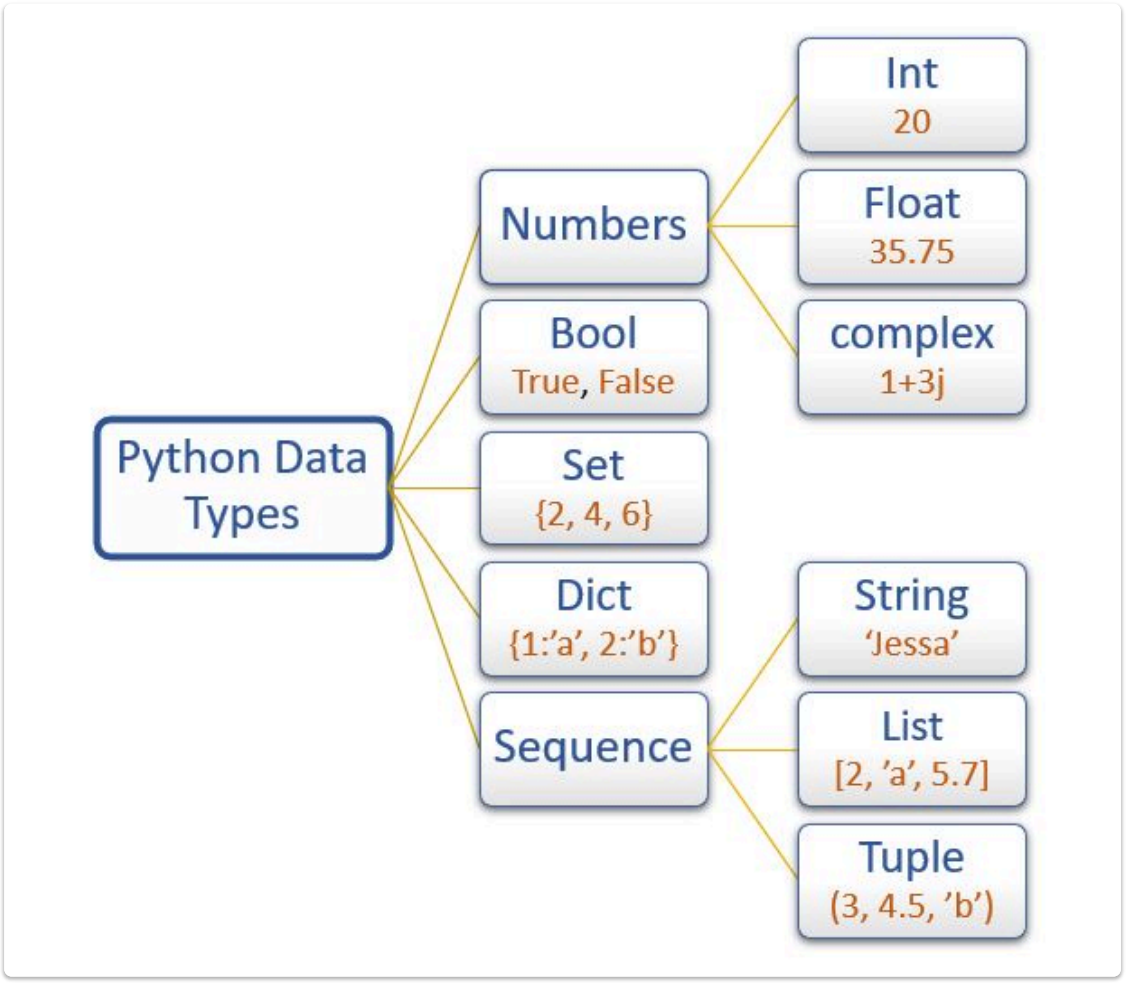
Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

Java can be considered both a compiled and an interpreted language. Java source code is directly converted to binary byte code, which is then executed by the JVM (Java Virtual Machine) which is a software based interpreter.

1. Why Python?

- Python is a high-level, interpreted programming language that is object-oriented and is dynamically typed.
- Python works with Windows, Mac, Linux, Raspberry Pi, and other platforms.
- Compared to other programming languages, Python lets developers write programs with fewer lines.





Data type	Description	Example
`int`	To store integer values	`n = 20`
`float`	To store decimal values	`n = 20.75`
`complex`	To store complex numbers (real and imaginary part)	`n = 10+20j`

Data type	Description	Example
`str`	To store textual/string data	`name = 'Jessa'`
`bool`	To store boolean values	`flag = True`
`list`	To store a sequence of mutable data	`l = [3, 'a', 2.5]`
`tuple`	To store sequence immutable data	`t =(2, 'b', 6.4)`
`dict`	To store key: value pair	`d = {1:'J', 2:'E'}`
`set`	To store unorder and unindexed values	`s = {1, 3, 5}`
`frozenset`	To store immutable version of the set	`f_set=frozenset({5,7})`
`range`	To generate a sequence of number	`numbers = range(10)`
`bytes`	To store bytes values	`b=bytes([5,10,15,11])`

2. How Would You Generate Random Numbers in Python?

1. Using `random.random()`:
   - This function generates a random floating-point number in the range [0.0, 1.0).

```
import random
random_number = random.random()
```

2. Using `random.randint(a, b)`:
   - This function generates a random integer between `a` and `b` (inclusive).

```
import random
random_integer = random.randint(1, 10)  # Generates a random integer between 1 and
10
```

3. Using `random.uniform(a, b)`:
   - This function generates a random floating-point number between `a` and `b` (inclusive).

```
import random
random_float = random.uniform(1.0, 10.0)  # Generates a random float between 1.0 and
10.0
```

4. Using `random.randrange(start, stop[, step])`:
   - This function generates a random integer from the range `start` (inclusive) to `stop` (exclusive), with an optional step parameter.

```
import random
random_number = random.randrange(0, 100, 2)  # Generates a random even number
between 0 and 100
```

5. Using `random.choice(seq)`:

- This function returns a random element from the non-empty sequence `seq`.

```
import random
my_list = [1, 2, 3, 4, 5]
random_element = random.choice(my_list)  # Selects a random element from the list
```

6. Using `random.shuffle(seq)`:
   - This function shuffles the elements of the sequence `seq` in place.

```
import random
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)  # Shuffles the list randomly
```

3. What Does the // Operator Do?

Ans. `//` Floor division. Returns the integer value. Integer quotient.
eg. `10//3 will return 3`. But 10/3 returns 3.3333.

4. What Does the 'is' Operator Do?

Ans. Returns `True` is both the objects are referring to the same object.

```
list1=[1,2,3]

list2=[1,2,3]

list3=list1

list1 == list2 → True

list1 is list2 → False

list1 is list3 → True
```

5. What Is the Purpose of the `Pass` Statement?

Ans. It is used as a placeholder for future code.

```
def add(a,b):
        return a+b

def power():
        pass # if you do not write pass here, it will throw a syntax error
```

```
def sub(a,b):
        return a-b
```

6. How Will You Merge Elements in a Sequence?

Ans. Concat the elements

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = l1+l2
print(l3)
#output [1,2,3,4,5,6]
```

`+` operator can be used to concat two or more lists, tuples ans strings.

7. How Will You Check If All the Characters in a String Are Alphanumeric?

Ans. Use built-in method called `isalnum()`.

```
str1 = 'abc123'
str1.isalnum()
#returns True

str2 = 'abc@123'
str2.isalnum()
#returns False because of special chracter
```

-- We can use Regular Expressions module for pattern matching

```
import re

# syntax (re.match('pattern', 'stringToBeChecked'))

print(bool(re.match('[A-Za-z0-9]+$', 'abcd123')))  # Output: True
print(bool(re.match('[A-Za-z0-9]+$', 'abcd@123')))  # Output: False
```

Or we can compile the pattern and check if it matches or not.

```
import re
# Compile the regular expression pattern
pattern = re.compile('[A-Za-z0-9]+[$,@]')

# Use the compiled pattern multiple times
print(bool(pattern.match('abcd123')))  # Output: False
print(bool(pattern.match('abcd$123')))  # Output: True
```

8. How Would You Remove All Leading Whitespace in a String?

Ans.

```
# Leading whitespaces
"     Python".lstrip()

Output: 'Python'

# Trailing spaces
"Python     ".rstrip()
Output: 'Python'

'''
Note: "      Python    ".strip()  removes both the leading and trailing whitespaces.
Output: 'Python'
'''
```

9. How Would You Replace All Occurrences of a Substring with a New String?

The `replace()` function is used with strings to replace all occurrences of a substring with a given string. Its syntax is as follows:

`str.replace(old, new, count)`

- `old`: The substring to be replaced.
- `new`: The new string that will replace the occurrences of `old`.
- `count` (optional): The maximum number of occurrences to replace. If omitted or -1, all occurrences are replaced.

The `replace()` method returns a new string without modifying the original string.

Example:

```
string = "Hey John. How are you, John?"
new_string = string.replace("John", "Jane")
print(new_string)

Output:
"Hey Jane. How are you, Jane?"
```

10. What Is the Difference Between Del and Remove() on Lists?

Ans.

del	remove()
- del removes all elements of a list within a given range - Syntax: del list[start:end]	- remove() removes the first occurrence of a particular character - Syntax: list.remove(element)

```
# Using slicing and del to delete elements from a list
lis = ['a', 'b', 'c', 'd']
del lis[1:3]
print(lis)  # Output: ['a', 'd']

# Using remove() to delete elements from a list by value
lis = ['a', 'b', 'b', 'd']
lis.remove('b')
print(lis)  # Output: ['a', 'b', 'd']
```

11. How Do You Display the Contents of a Text File in Reverse Order?

Ans. Let's first learn how to open and read a text file.

```
file = open('sample.txt', 'r')
content = file.readlines()
print(content)
file.close()  # It's good practice to close the file after reading its content


'''
But if you don't want to close it manually. Consider using with context manager
'''
with open('sample.txt', 'r') as file:
    content = file.readlines()
    print(content)
```

Now, how will you reverse the order.

```
with open('sample.txt', 'r') as file:
    content = file.readlines()

# Reverse the order of lines
content.reverse()

# Print the reversed content
for line in content:
    print(line.strip())  # Use strip() to remove leading and trailing whitespace
```

What are the different file processing modes supported by Python?

1. **Read-only (r) mode:**
   - Used to open a file in read-only mode.
   - Default mode if not specified explicitly.

```
with open('file.txt', 'r') as f:
    content = f.read()
```

2. **Write-only (w) mode:**
   - Opens a file for writing. If the file exists, its contents are truncated (i.e., deleted); otherwise, a new file is created.

```
with open('file.txt', 'w') as f:
    f.write('Hello, world!')
```

3. **Read-Write (rw) mode:**
   - Opens a file for both reading and writing.

```
with open('file.txt', 'r+') as f:
    content = f.read()
    f.write('Appending new content')
```

4. **Append (a) mode:**
   - Opens a file for writing and appends new data to the end of the file.

```
with open('file.txt', 'a') as f:
    f.write('New content appended')
```

12. Differentiate Between append() and extend().

append()	extend()
- append() adds an element to the end of the list - Example - >>lst=[1,2,3] >>lst.append(4) >>lst Output:[1,2,3,4]	- extend() adds elements from an iterable to the end of the list - Example - >>lst=[1,2,3] >>lst.extend([4,5,6]) >>lst Output:[1,2,3,4,5,6]

13 Differentiate Between List and Tuple.

Ans.

List	Tuple	Set	Dictionary
Comma-separated values within square brackets and brackets are necessary.	Comma-separated values within parentheses, but parentheses are not mandatory.	Comma-separated values within curly braces.	Key-value pairs enclosed within curly braces, separated by colons (":").
lst = [10, 20, 'a']	tup = (1, 2, 'a') tup = 1, 2, 'a'	s = {1, 2, 3}	d = {'name': 'Alice', 'age': 30, 'city': 'New York'}
`Mutable`. lst[0] = 100 will update the value at the 0th index.	`Immutable`. tup[0] = 10 is not possible.	`Mutable`. `Sets do not support indexing.` Cannot access elements by index.	Keys are unique within a dictionary, but values may not be. `Keys can be of any immutable type.`
Lists can contain duplicate elements and are `ordered`.	Tuples can contain duplicate elements and are `ordered`.	Sets contain unique elements and are `unordered`.	`Keys are unique. Values can be duplicated.`
Use lists when you need an ordered collection with duplicate elements.	Use tuples when you need an immutable, ordered collection with duplicate elements.	Use sets when you need a collection of unique elements.	Use dictionaries when you need to store key-value pairs for efficient lookup.

Set: A set in Python is an unordered collection of unique elements. Sets are mutable, meaning you can add or remove elements from them. Sets are commonly used for tasks that involve checking for the presence of an element or eliminating duplicates from a collection.

Example:

```
s = {1, 2, 3}  # Creating a set
s.add(4)       # Adding an element to the set
s.remove(2)    # Removing an element from the set
print(s)       # Output: {1, 3, 4}
```

Dictionary: A dictionary in Python is an unordered collection of key-value pairs. Each key in a dictionary must be unique and immutable, while values can be of any data type and may be duplicated. Dictionaries are often used to store and retrieve data efficiently based on keys.

Example:

```
d = {'name': 'Alice', 'age': 30, 'city': 'New York'}  # Creating a dictionary
print(d['name'])  # Accessing value associated with the key 'name'
d['age'] = 31     # Updating the value associated with the key 'age'
del d['city']     # Deleting the key-value pair with the key 'city'
print(d)          # Output: {'name': 'Alice', 'age': 31}
```

List

```python
# Sample list
tasks = ["Complete project", "Review presentation", "Send email"]

# Accessing tasks by index
print("Task at index 0:", tasks[0])
print("Task at index 1:", tasks[1])
print("Task at index 2:", tasks[2])

# Adding a new task
tasks.append("Submit report")
print("Updated tasks:", tasks)

# Removing a task
removed_task = tasks.pop(1)
print("Removed task:", removed_task)
print("Updated tasks:", tasks)

# Updating a task
tasks[0] = "Finish project"
print("Updated tasks:", tasks)
```

```
Task at index 0: Complete project
Task at index 1: Review presentation
Task at index 2: Send email
Updated tasks: ['Complete project', 'Review presentation', 'Send email', 'Submit report']
Removed task: Review presentation
Updated tasks: ['Complete project', 'Send email', 'Submit report']
Updated tasks: ['Finish project', 'Send email', 'Submit report']
```

Tuple

```python
city1 = ("New York", 75, "Sunny")

# Accessing elements of the tuple
print("City:", city1[0])
print("Temperature:", city1[1])
print("Weather condition:", city1[2])

# Unpacking the tuple into variables
city, temperature, weather = city1
print("City:", city)
print("Temperature:", temperature)
print("Weather condition:", weather)

# Concatenating tuples
city2 = ("London", 62, "Cloudy")
combined_cities = city1 + city2
print("Combined cities:", combined_cities)

# Length of a tuple
print("Length of the tuple:", len(city1))
```

```
City: New York
Temperature: 75
Weather condition: Sunny
City: New York
Temperature: 75
Weather condition: Sunny
Combined cities: ('New York', 75, 'Sunny', 'London', 62, 'Cloudy')
Length of the tuple: 3
```

14. What Is Docstring in Python?

Ans. Docstrings are used in providing documentation to various Python modules, classes, functions, and methods.
Example -

```
def add(a,b):
"""This function adds two numbers."""
        sum=a+b
        return sum

sum=add(10,20)

print("Accessing doctstring method 1:",add.__doc__)
```

15. How Do You Use Print() Without the Newline?

Ans.

```
print("Hi",end=" ")
print("How are you?") # Output: Hi. How are you?
```

16. How Do You Use the Split() Function in Python?

Ans. The `split()` function splits a string into a number of strings based on a specific delimiter.

Syntax:

```
string.split(delimiter, max)
```

Where:

- `delimiter` is the character based on which the string is split. By default, it is a space.
- `max` is the maximum number of splits.

Example:

```
var = "Red,Blue,Green,Orange"
lst = var.split(",", 2)
print(lst) # output: ['Red', 'Blue', 'Green,Orange']
```

17. Is Python Object-oriented or Functional Programming?

Python is considered a multi-paradigm language.

Python follows the object-oriented paradigm

- Python allows the creation of objects and their manipulation through specific methods
- It supports most of the features of OOPS such as inheritance and polymorphism

Python follows the functional programming paradigm

- Functions may be used as the first-class object
- Python supports Lambda functions which are characteristic of the functional paradigm

18. Write a Function Prototype That Takes a Variable Number of Arguments.

```
def add(*args):
    result = 0
    for num in args:
        result += num
    return result

# Example usage:
print(add(1, 2, 3))     # Output: 6
print(add(10, 20, 30))  # Output: 60
print(add(5, 10))       # Output: 15
print(add())            # Output: 0 (No arguments provided)
```

19. What Are *args and kwargs?

`*args`

- It is used in a function prototype to accept a varying number of arguments.
- It's an iterable object.
- Usage - def fun(*args)

`*kwargs`

- It is used in a function prototype to accept the varying number of keyword arguments.
- It's an iterable object
- Usage - def fun(**kwargs):

`fun(colour="red".units=2)`

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Example usage:
print_info(name="Alice", age=30, city="New York")

# Output:
```

```
name: Alice
age: 30
city: New York
```

20. "in Python, Functions Are First-class Objects." What Do You Infer from This?

It means that a function can be treated just like an object. You can assign them to variables, or pass them as arguments to other functions. You can even return them from other functions. Certainly! Here are some small code examples to illustrate the concept of functions as first-class objects in Python:

1. Assigning a function to a variable:

```
def greet(name):
    return f"Hello, {name}!"

hello_func = greet
print(hello_func("Alice"))  # Output: Hello, Alice!
```

2. Passing a function as an argument to another function:

```
def apply_func(func, arg):
    return func(arg)

def double(x):
    return x * 2

result = apply_func(double, 5)
print(result)  # Output: 10
```

3. Returning a function from another function:

```
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_5 = create_adder(5)
print(add_5(3))  # Output: 8
```

In all these examples, functions are treated as first-class objects, just like any other data type in Python.

21. How can you delete a file in Python?

Ans. We can delete the file in Python using the os module. The remove() function of the os module is used to delete a file in Python by passing the filename as a parameter. e.g.

```
import os
os.remove("txt1.txt")
```

22. Write a code snippet to reverse an array.

Ans. The two ways of reversing an array are as follows:

1. Using the flip() function

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.flip(arr1)
print(arr2)
```

Output:

[4, 3, 2, 1]

2. Without using any function

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = arr1[::-1]
print(arr2)
```

Output:

[4,3,2,1]

23. Write a code snippet to concatenate lists.

Ans. Suppose, given two lists are:

```
List1= ["W", "a", "w","b"]
List2 = ["e", "re","riting","log"]
```

And the output should be:

['We', 'are ', 'writing', 'blog']

This can concatenate the two lists using the zip() function, which iterates through both lists and combines them index-wise.

```
lst1 = ['W', 'a', 'w', 'b']
lst2 = ['e', 're', 'riting', 'log']
```

```
lst3 = [x + y for x, y in zip(lst1, lst2)]
print(lst3) # Output: [`We`, `are`, `writing`, `blog`]
```

Zip() function in python


```
fruits = ["apple", "banana", "orange", "mango"]
colors = ["red", "yellow", "orange", "yellow"]

fruit_color_pairs = zip(fruits, colors)
print(list(fruit_color_pairs))
```

Output:

```
[('apple', 'red'), ('banana', 'yellow'), ('orange', 'orange'), ('mango', 'yellow')]
```

24. Write a code snippet to generate the square of every element of a list.

Ans.

```
lst = [1, 2, 3, 4]
lst_final = []
for x in lst:
    lst_final.append(x * x)
print(lst_final) # Output: [1, 4, 9, 16]
```

25. What is pickling (serialization) and unpickling in Python?

Ans. Pickling is converting a Python object (list, dict, function, etc.) to a byte stream(0s and 1s), and unpickling is converting the byte stream back to a python object. It is used to transfer and store various Python objects. We can use pickle or dill Python packages for this.

we can use pickle module for this.

1. Pickling:

```
import pickle

# Define a Python object
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Pickle the object to a byte stream
with open('data.pkl', 'wb') as f:
    pickle.dump(data, f)
```

2. Unpickling:

```
import pickle

# Unpickle the byte stream back to a Python object
with open('data.pkl', 'rb') as f:
    restored_data = pickle.load(f)

print(restored_data)  # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

26. What is the difference between split and join?

Ans. Split and join are both functions of Python strings, but they are completely different when it comes to functioning.

The split function is used to create a list from strings based on some delimiter, for e.g., space.

```
a = 'This is a string'
li = a.split(' ')
print(li)
```

Output: ['This', 'is', 'a', 'string']

The join() method is a built-in function of Python's str class that concatenates a list of strings into a single string. It is called on a delimiter string and invoked with a list of strings to be joined. The delimiter string is inserted between each string in the list when the strings are concatenated.

Here is an example of how to use the join() method:

```
" ".join(li)
```

Output: This is a string

27. Explain the top 5 functions used for Python strings.

Ans. Here are the top 5 Python string functions:

- **len():** This function returns the length of a string.

```
s = 'Hello, World!'
print(len(s)) # output: 13
```

- **strip():** This function removes leading and trailing whitespace from a string.

```
s = '   Hello, World!   '
print(s.strip()) # 'Hello, World!'
```

- **replace():** This function replaces all occurrences of a specified string with another string.

```
s = 'Hello, World!'
print(s.replace('World', 'Universe')) # output: 'Hello, Universe!'
```

- **split():** This function splits a string into a list of substrings based on a delimiter.

```
s = 'Hello, World!'
print(s.split(',')) # output: ['Hello', ' World!']
```

- **upper() and lower():** These functions convert a string to uppercase or lowercase, respectively.

```
s = 'Hello, World!'
print(s.upper()) #output: 'HELLO, WORLD!'
```

```
s.lower() #output: 'hello, world!'
```

In addition to them, string also has capitalize, isalnum, isalpha, and other methods.

28. What Is the Difference Between Matrices and Arrays?

Matrices	Arrays
- A matrix comes from linear algebra and is a two-dimensional representation of data - It comes with a powerful set of mathematical operations that allow you to manipulate the data in interesting ways	- An array is a sequence of objects of similar data type - An array within another array forms a matrix

29. Name 2 mutable and 2 immutable data types in Python.

Ans. 2 **mutable** data types are **Dictionary** and **List**. You can change/edit the values in a Python dictionary and a list. It is not necessary to make a new list which means that it satisfies the

property of mutability.

2 immutable data types are Tuples and String. You cannot edit a string or a value in a tuple once it is created. You need to either assign the values to the tuple or make a new tuple.

30. What are Python functions, and how do they help in code optimization?

Ans. Functions in Python are blocks of reusable code that can be called by other parts of your program. They help in code optimization by enabling:

1. Code reuse: Functions encapsulate code in a single place, reducing redundancy and making code more concise and maintainable.
2. Improved readability: Dividing code into logical blocks enhances readability and comprehension, aiding in bug identification and code modification.
3. Easier testing: Functions enable testing of individual code blocks separately, facilitating bug detection and correction.
4. Enhanced performance: Functions allow utilization of optimized libraries and enable the Python interpreter to optimize code more effectively, contributing to improved performance.

31. Explain list comprehension and dict comprehension.

Ans. List comprehension and dict comprehension are both concise ways to create new lists or dictionaries from existing iterables.

List comprehension is a concise way to create a list. It consists of square brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result is a new list that evaluates the expression in the context of the for and if clauses.

1. Generate Squares:

```
#Regular Way
squares = []
for number in range(1, 6):
    squares.append(number ** 2)

# Program to generate a list of squares of numbers from 1 to 10.
   squares = [x**2 for x in range(1, 11)]
   print(squares)
```

2. Even Numbers:

```
# Regular Way
even_numbers = []
for number in range(1, 21):
    if number % 2 == 0:
```

```
        even_numbers.append(number)
print(even_numbers)
```

```
# Program to create a list of even numbers from 1 to 20.
even_numbers = [x for x in range(1, 21) if x % 2 == 0]
print(even_numbers)
```

3. String Lengths:

```
# Regular Way
words = ["apple", "banana", "orange"]
word_lengths = []
for word in words:
    word_lengths.append(len(word))
print(word_lengths)
```

```
# Program to find the lengths of words in a list.
words = ["apple", "banana", "orange"]
word_lengths = [len(word) for word in words]
print(word_lengths)
```

4. Prime Numbers:

```
# Regular Way
primes = []
for num in range(2, 21):
    is_prime = True
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        primes.append(num)
```

```
# Program to generate a list of prime numbers less than 20.
primes = [num for num in range(2, 21) if all(num % i != 0 for i in range(2,
int(num**0.5) + 1))]
print(primes)
```

Dict comprehension is a concise way to create a dictionary. It consists of curly braces containing a key-value pair, followed by a for clause, then zero or more for or if clauses. A result is a new dictionary that evaluates the key-value pair in the context of the for and if clauses.

```
# Creating a dictionary with squares of numbers from 1 to 5
squares = {num: num**2 for num in range(1, 6)}
```

```
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

32. What are global and local variables in Python?

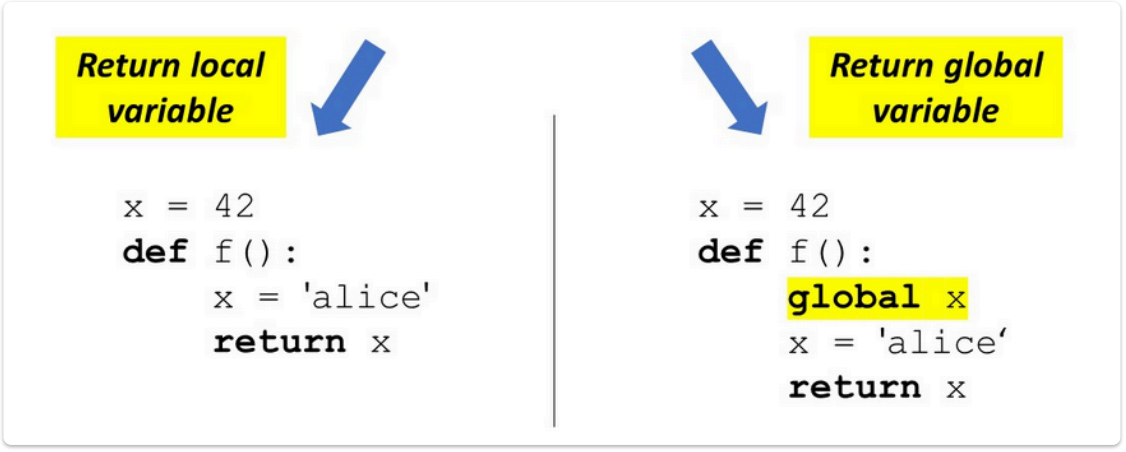
Ans. In Python, a variable that is defined outside of any function or class is a global variable, while a variable that is defined inside a function or class is a local variable.

A global variable can be accessed from anywhere in the program, including inside functions and classes. However, a local variable can only be accessed within the function or class in which it is defined.

It is important to note that you can use the same name for a global variable and a local variable, but the local variable will take precedence over the global variable within the function or class in which it is defined.

```
x = 10 # This is a global variable
def func():
    x = 5 # This is a local variable
    print(x)
func()
print(x)
```

Output: This will print 5 and then 10

> *using `global` keyword.*
>
> 
>

33. What is the difference between return and yield keywords?

Ans. `Return` is used to exit a function and return a value to the caller. When a return statement is encountered, the function terminates immediately, and the value of the expression following the return statement is returned to the caller.

1. Using `return`:

```
def square_numbers(nums):
    result = []
    for num in nums:
```

```
        result.append(num * num)
    return result

squares = square_numbers([1, 2, 3, 4, 5])
print(squares)  # Output: [1, 4, 9, 16, 25]
```

`Yield`, on the other hand, is used to define a generator function. A generator function is a special kind of function that produces a sequence of values one at a time instead of returning a single value. When a yield statement is encountered, the generator function produces a value and suspends its execution, saving its state for later.
2. Using `yield`:

```
def square_numbers(nums):
    for num in nums:
        yield num * num

squares_gen = square_numbers([1, 2, 3, 4, 5])
print(next(squares_gen))  # Output: 1
print(next(squares_gen))  # Output: 4
print(next(squares_gen))  # Output: 9
print(next(squares_gen))  # Output: 16
print(next(squares_gen))  # Output: 25
```

In this example, the `square_numbers` function is a generator function. It yields the squares of numbers one at a time as requested, saving its state between each yield. This allows for memory-efficient iteration over large sequences.

34. What are lambda functions in Python, and why are they important?

Ans. Lambda functions in Python are small, anonymous functions defined using the `lambda` keyword.
They're useful when you need a function temporarily and don't want to define a named function using `def`.

Lambda functions are important because:

1. They're concise: Lambda functions provide a compact way to define simple functions without the overhead of naming them.
2. They're used with higher-order functions: Lambda functions are often employed with higher-order functions like `map()`, `filter()`, and `reduce()` to perform operations on sequences.
3. They support functional programming: Lambda functions facilitate functional programming paradigms by allowing functions to be treated as first-class citizens.

Example:

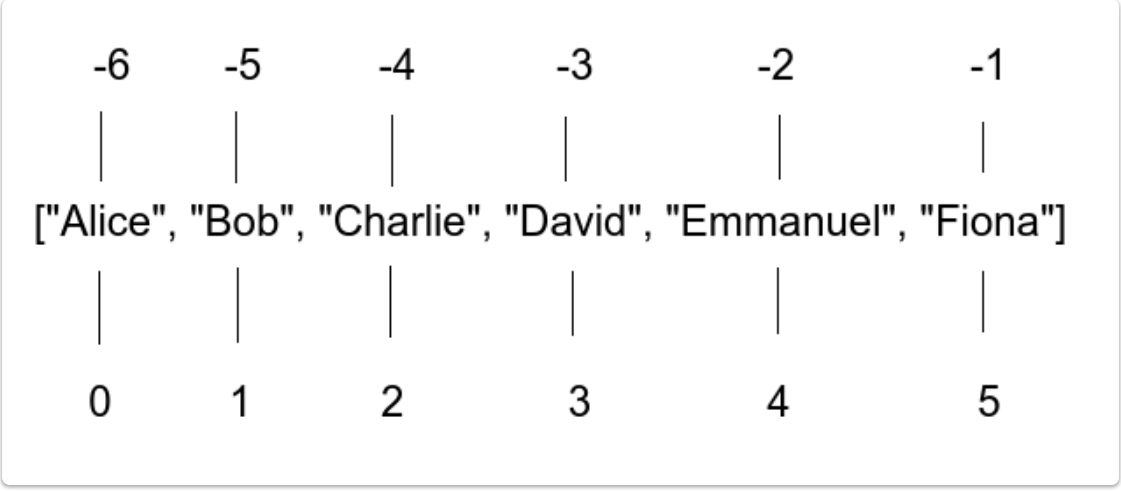
```
a = lambda x: x + 10
print(a(5))  # Output: 15
```

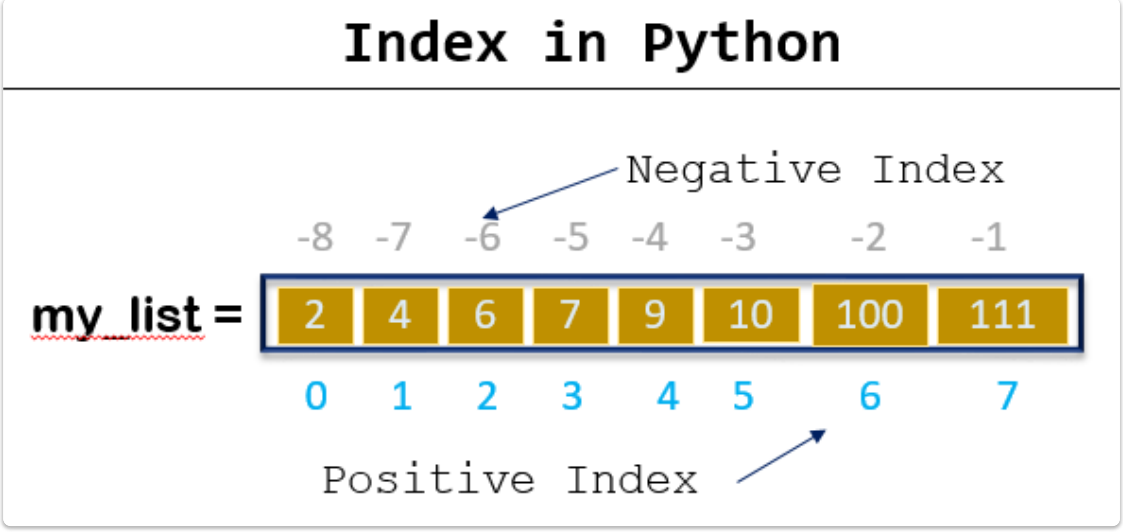
35. Where can we use a tuple instead of a list?

Ans. We can use tuples as dictionary keys as they are hashable. Since tuples are immutable, it is safer to use if we don't want values to change.
Tuples are faster and have less memory, so we can use tuples to access only the elements.

36. What is negative indexing?

Ans. Python sequence data types can be accessed with both positive and negative numbers. With negative indexing, -1 refers to the last element, -2 refers to the penultimate element, and so on.





37. What are the ways to swap the values of two elements?

Ans. One way is by using a third variable

```
temp = a
a = b
b = temp
```

In Python, we can also do it without using the third variable

```
a, b = b, a
```

38. Write a program in Python to return the factorial of a given number using recursion.

Ans.

```
def factorial(n):
    if n == 1:
        return n
    else:
        return n * factorial(n - 1)
```

39. Write a code to convert a list of characters to a string of characters separated by a comma.

Ans.

```
char_list = ['a', 'b', 'c', 'd']
result = ','.join(char_list)
print(result) # output: a,b,c,d
```

40. How can we find unique elements and value counts of a list using Numpy?

```
import numpy as np

# Example list
my_list = [1, 2, 3, 1, 2, 4, 5, 1, 3, 4, 1]

# Finding unique elements and their counts
unique_elements, counts = np.unique(my_list, return_counts=True)

print("Unique elements:", unique_elements)
print("Value counts:", counts)
```

Output:

```
Unique elements: [1 2 3 4 5]
Value counts: [4 2 2 2 1]
```

```
value_counts = [(x,y) for x, y in zip(unique_elements, counts)]
value_counts # output [(1, 4), (2, 2), (3, 2), (4, 2), (5, 1)]
```

In this example, `np.unique()` is used to find the unique elements and their corresponding counts in the list `my_list`. Setting `return_counts=True` ensures that both arrays of unique elements and their counts are returned.

41. What is the difference between indexing and slicing in Python?

Ans. Indexing creates an index, whereas slicing makes a shallow copy. Different types of indexing are possible, but there is no slicing category.

1. Indexing:

```
# Creating a Python list
my_list = [10, 20, 30, 40, 50]

# Indexing to access a single element
print(my_list[2])  # Output: 30
```

In this example, indexing is used to access the element at index 2 in the list `my_list`.

2. Slicing:

```
# Creating a Python list
my_list = [10, 20, 30, 40, 50]

# Slicing to create a shallow copy of a portion of the list
slice_list = my_list[1:4]
print(slice_list)  # Output: [20, 30, 40]
```

In this example, slicing is used to create a shallow copy of elements from index 1 to 3 in the list `my_list`. The `slice_list` variable holds this shallow copy.

42. What are deepcopy and shallowcopy?

Ans. Deepcopy copies the contents of the object to another location.Changes made in one object do not affect the other. In shallow copy, only the reference is copied. So, changes made in one affect the other object.

Here are examples demonstrating deepcopy and shallowcopy in Python:

1. Deepcopy:

```
import copy

# Original list
original_list = [[1, 2, 3], [4, 5, 6]]

# Performing deepcopy
copied_list = copy.deepcopy(original_list)
```

```
# Modifying the copied list
copied_list[0][0] = 100

# Original list remains unchanged
print(original_list)  # Output: [[1, 2, 3], [4, 5, 6]]
```

In this example, `copy.deepcopy()` creates a deep copy of the original list, so modifying the copied list does not affect the original list.

2. Shallowcopy:

```
import copy

# Original list
original_list = [[1, 2, 3], [4, 5, 6]]

# Performing shallow copy
copied_list = copy.copy(original_list)

# Modifying the copied list
copied_list[0][0] = 100

# Original list is affected
print(original_list)  # Output: [[100, 2, 3], [4, 5, 6]]
```

In this example, `copy.copy()` creates a shallow copy of the original list, so modifying the copied list affects the original list.

43. What is a callable object in Python?

Ans. An object which can invoke a process is a callable object. It uses the call method. Functions are examples of that. Callable objects have () at the end, while non-callable methods don't have () at the end.

44. How do you sort a dataframe by two columns?

Ans.

```
df.sort_values(by=['col1', 'col2'])
```

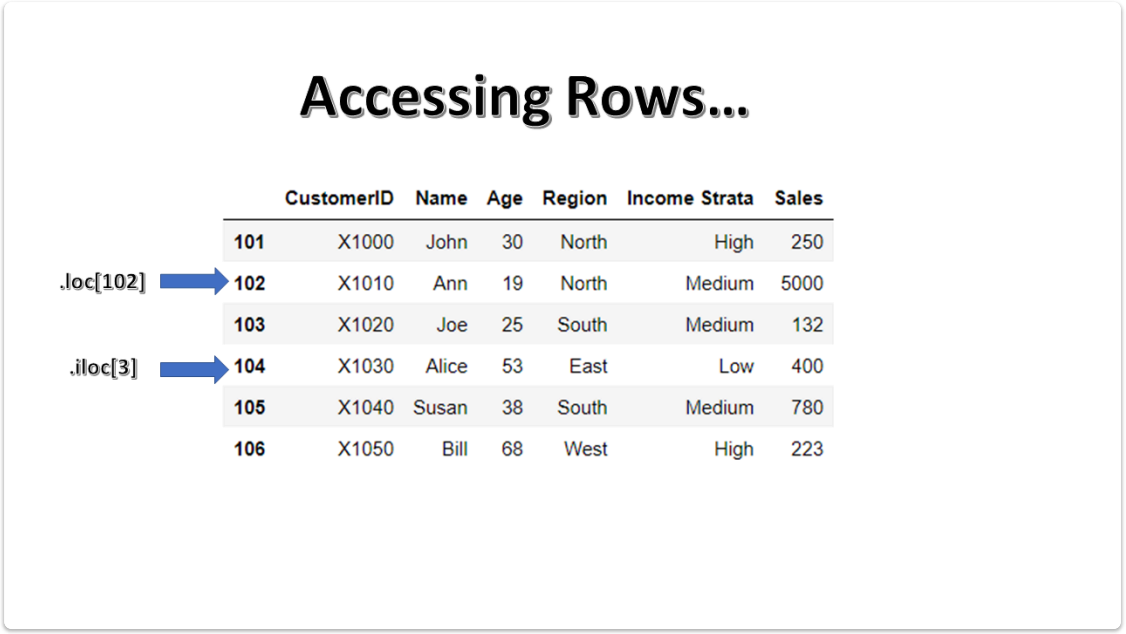
45. How can we transform a true/false value to 1/0 in a dataframe?

Ans.

```
df["column"] = df["column"].astype(int)
```

46. How are loc() and iloc() different?

Ans. `loc()` is used to access the rows or columns using labels, while `iloc()` is used to access using position-based indexing.




```
# Pandas.DataFrame.iloc[] usage
import pandas as pd
technologies = {
    'Courses':["Spark","PySpark","Hadoop","Python","pandas"],
    'Fee' :[20000,25000,26000,22000,24000],
    'Duration':['30day','40days','35days','40days','60days'],
    'Discount':[1000,2300,1200,2500,2000]
              }
index_labels=['r1','r2','r3','r4','r5']
df = pd.DataFrame(technologies,index=index_labels)

print(df)
```

	Courses	Fee	Duration	Discount
r1	Spark	20000	30day	1000
r2	PySpark	25000	40days	2300
r3	Hadoop	26000	35days	1200
r4	Python	22000	40days	2500
r5	pandas	24000	60days	2000

> *loc[] - when we want to fetch the data using the names/labels of the rows and columns*

```
print(df.loc['r5', 'Courses'])
```

```
pandas
```

> *iloc[] - when we want to fetch the data using the indexes*

```
print(df.iloc[4,0])
```

```
pandas
```

48. What is the use of the `break` statement?

Ans. The `break` statement is used to exit a loop prematurely, stopping further iterations even if the loop condition has not been met.
- early termination
- Exiting infinite loops

```
# Example using break statement
numbers = [1, 2, 3, 4, 5]

for number in numbers:
    if number == 3:
        break  # Exit the loop when number equals 3
    print(number)

# Output: 1
#         2
```

49. What is the use of the `continue` statement?

Ans. The `continue` statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration of the loop. It allows you to bypass certain iterations based on a condition without exiting the loop entirely.

```
# Example using continue statement
numbers = [1, 2, 3, 4, 5]

for number in numbers:
    if number == 3:
        continue  # Skip the iteration when number equals 3
    print(number)

# Output: 1
#         2
#         4
#         5
```

50) What is an operator in Python?

In Python, an operator is a symbol or a special character that performs an operation on one or more operands. Operators can perform arithmetic, comparison, logical, assignment, bitwise, and other operations depending on their type.

1. Unary Operators:
   Unary operators act on a single operand. They perform an operation on only one operand. Examples of unary operators in Python include:

   - Unary plus (+) and unary minus (-)
   - Logical NOT (! in Python, but ~ in some other languages)
   - Increment (++) and decrement (--) operators (Python does not have these specific operators)

   Example:

   ```
   x = 10
   y = -x  # Unary minus operator
   print(y)  # Output: -10
   ```

2. Binary Operators:
   Binary operators act on two operands. They perform an operation that involves two operands. Examples of binary operators in Python include:

   - Arithmetic operators (+, -, *, /, %)
   - Comparison operators (==, !=, <, >, <=, >=)
   - Logical operators (and, or)

- Bitwise operators (&, |, ^, <<, >>)
- Assignment operators (=, +=, -=, *=, /=, %=, etc.)

Example:

```
a = 10
b = 5
c = a + b  # Addition operator
print(c)   # Output: 15
```

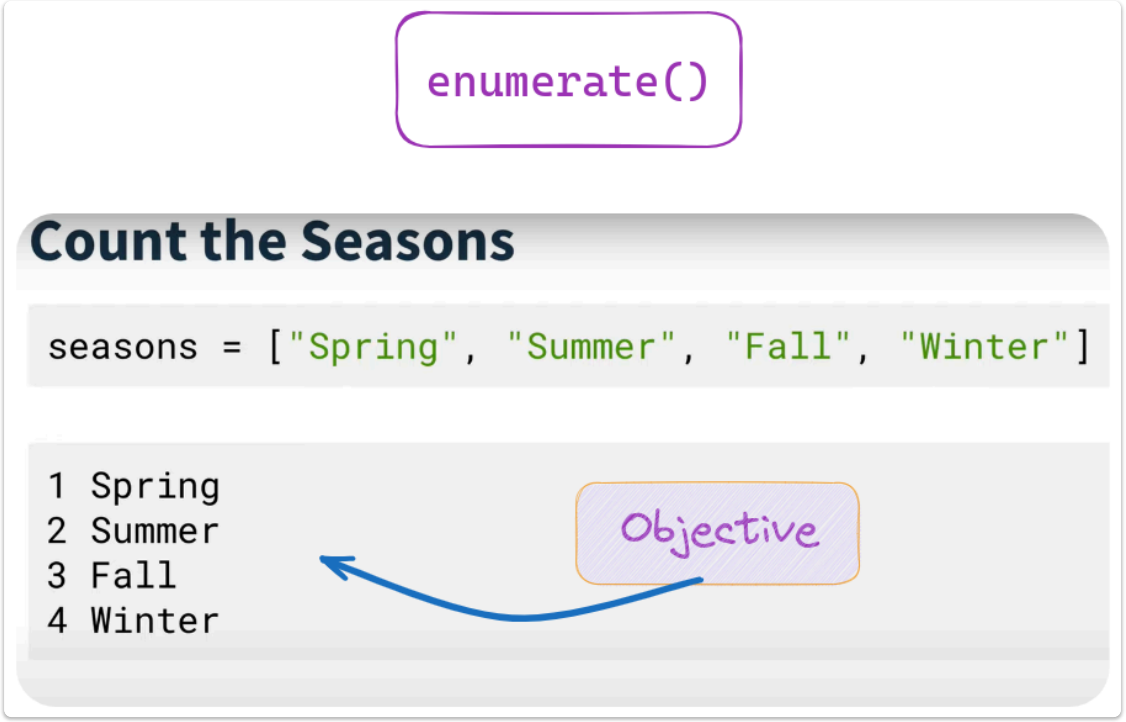
3. Ternary Operator (Conditional Operator):
Ternary operator is the only operator that takes three operands. It's often used as a shortcut for conditional expressions. In Python, the ternary operator is written as `x if condition else y`. If the condition evaluates to True, the expression returns `x`; otherwise, it returns `y`.

Example:

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)  # Output: "Adult"
```

In this example, if the `age` is greater than or equal to 18, the value of `status` will be "Adult"; otherwise, it will be "Minor".

51. `Enumerate()`




```
seasons = ['Spring', 'Summer', 'Autumn', 'Winter']
```

```
for index, season in enumerate(seasons, start=1):
    print(f"Season {index}: {season}")

# Output:
# Season 1: Spring
# Season 2: Summer
# Season 3: Autumn
# Season 4: Winter
```

52. What is type conversion in Python?

Type conversion refers to the conversion of one data type into another.

1. **int()**: converts any data type into integer type

```
x = int(5.5)
print(x)  # Output: 5
```

2. **float()**: converts any data type into float type

```
x = float("3.14")
print(x)  # Output: 3.14
```

3. **ord()**: converts characters into integer

```
x = ord('A')
print(x)  # Output: 65
```

4. **hex()**: converts integers to hexadecimal

```
x = hex(255)
print(x)  # Output: 0xff
```

5. **oct()**: converts integer to octal

```
x = oct(8)
print(x)  # Output: 0o10
```

6. **tuple()**: This function is used to convert to a tuple.

```
x = tuple([1, 2, 3])
print(x)  # Output: (1, 2, 3)
```

7. **set()**: This function returns the type after converting to set.

```
x = set([1, 2, 3])
print(x)  # Output: {1, 2, 3}
```

8. **list():** This function is used to convert any data type to a list type.

```
x = list((1, 2, 3))
print(x)  # Output: [1, 2, 3]
```

9. **dict():** This function is used to convert a tuple of order (key,value) into a dictionary.

```
x = dict([(1, 'one'), (2, 'two')])
print(x)  # Output: {1: 'one', 2: 'two'}
```

10. **str():** Used to convert integer into a string.

```
x = str(123)
print(x)  # Output: '123'
```

Object Oriented Programming

1. What is OOP, and why is it important in Python?

Answer: OOP, or Object-Oriented Programming, is a programming paradigm used in many languages, including Python.

- In OOP, you can model real-world concepts using classes and objects.
- You can break down complex problems into smaller, more manageable parts.
- his makes the code reusable, maintainable, and scalable.

2. What is a class in Python?

Answer: A class in Python is like a blueprint for creating objects. A class defines properties and methods that are common to all objects created from it.

Classes help organize code by grouping related attributes and functions, promoting reusability and modularity. For instance, if you're creating a program to manage a library, you might have classes for books, authors, and borrowers, each with its specific attributes and methods.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def display_info(self):
```

```
        print(f"Title: {self.title}\nAuthor: {self.author}")

# Creating an object of the Book class
book1 = Book("Python Programming", "John Smith")

# Calling the display_info method to display information about the book
book1.display_info()
```

> *The `__init__` method is a special method called the constructor, which initializes the object's attributes.*

3. What is an object in Python?

Answer: An object is an instance of a class. You can think of it as a specific realization of the blueprint provided by the class.

An object contains data in the form of attributes and code in the form of methods.
When you create an object, you are essentially creating a variable that has all the properties and behaviors defined in the class.
For example, if you have a class for cars, an object could represent a specific car, such as a Honda Civic, with attributes like color and speed, and methods like start and stop.

```
class Car:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

    def start(self):
        print(f"The {self.color} car starts moving at {self.speed} km/h.")

    def stop(self):
        print(f"The {self.color} car comes to a stop.")

# Creating an object of the Car class
car1 = Car("red", 60) # creating an instance of the class

# Calling methods on the car object
car1.start()
car1.stop()
```

5. What is the constructor in a Python class?

Answer: A constructor is a special method used to initialize objects. You can think of it as blueprint instructions for creating objects.
When you create an instance of a class, the constructor method, `___init___`, is automatically called. It allows you to set up properties or execute any setup code needed for the object.

```
class Person:
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}\nAge: {self.age}")

# Creating an object of the Person class using the constructor
person1 = Person("Alice", 30)

# Calling the display_info method to display information about the person
person1.display_info()
```

6. What is a decorator in Python?

A decorator in Python is a special type of function that is used to modify or extend the behavior of other functions or methods. It allows you to add functionality to an existing function without modifying its code directly.

Decorators are commonly used to:

1. Add logging, timing, or debugging functionality to functions.
2. Enforce access control or authentication checks.
3. Cache or memoize function results for optimization purposes.
4. Modify the behavior of functions dynamically based on certain conditions.

Decorators are implemented using the `@decorator_name` syntax, where `decorator_name` is the name of the decorator function. The decorator function typically takes another function as its argument, adds some functionality to it, and returns a new function.

Here's a simple example of a decorator in Python:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
```

```
Something is happening after the function is called.
```

In this example:

- `my_decorator` is a decorator function that takes another function (`func`) as its argument.
- Inside `my_decorator`, a new function `wrapper` is defined, which adds some functionality before and after calling the original function (`func`).
- The `@my_decorator` syntax is used to apply the `my_decorator` decorator to the `say_hello` function.
- When `say_hello()` is called, it is automatically wrapped by the `wrapper` function defined in the `my_decorator`, adding the extra functionality specified in the decorator.

7. What is the purpose of the `self`?

Answer:

- The self is used to represent the current instance of the class.
- It helps you access the attributes and methods of the class within the current object's context.
- When calling a method, you don't have to pass the *self* argument; Python automatically sends the current object reference to the method.

```
class food():
    # init method or constructor
    def __init__(self, fruit, color):
        self.fruit = fruit
        self.color = color

    def info(self):
        print(f" {self.fruit} is {self.color}.")

a = food("Apple", "Red") # craeting inscatnce / object of class `food`
o = food("Orange", "Orange") # craeting inscatnce / object of class `food`

a.info() # object is calling the class method
o.info()

#output
Apple is Red.
Orange is Orange.
```

8. How is inheritance implemented in Python?

Answer:

Inheritance allows a class to inherit attributes and methods from another class. You can create a new class – known as the child class – based on an existing class – known as the parent

class - and add new features or modify existing ones.

> *Allowing for code reusability and better organization.*

```
# Parent class
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        pass

# Child class inheriting from Animal
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Child class inheriting from Animal
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Creating objects of the child classes
dog = Dog("Canine")
cat = Cat("Feline")

# Calling methods from the parent class
print(dog.species)  # Output: Canine
print(cat.species)  # Output: Feline

# Calling overridden method from the child classes
print(dog.make_sound())  # Output: Woof!
print(cat.make_sound())  # Output: Meow!
```

9. What is the use of the *super()* function?

Answer:

Python's *super()* function is used within a class to call a method from a parent class, often within the context of method overriding.

If you have a method in a child class with the same name as a method in the parent class, you can use *super()* to call the parent method within the child method. This is particularly useful when you want to extend or modify the behavior of the parent method in the child class.

By using *super()*, you ensure that your code follows the inheritance hierarchy. It also makes the code more maintainable, as changes in the parent class can be easily propagated to child classes.

```
# Parent class
class Animal:
```

```
    def make_sound(self):
        return "Generic animal sound"

# Child class inheriting from Animal
class Dog(Animal):
    def make_sound(self):
        # Call the make_sound method from the parent class using super()
        parent_sound = super().make_sound()
        return f"{parent_sound} - Woof!"

# Creating an object of the child class
dog = Dog()

# Calling the overridden method from the child class
print(dog.make_sound())  # Output: Generic animal sound - Woof!
```

10. What is the purpose of the *@property* decorator?

Answer:

In Python, the *@property* decorator allows you to treat a method as a property of the class. By using this, you can create a "getter" method, which enables you to access a class method as though it's an attribute without needing to write parentheses when you call it. This means you can control how the attribute is accessed without directly exposing it.

The *@property* decorator provides a way to implement encapsulation. It helps in managing the attributes of a class and making the code more readable.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

    @property
    def area(self):
        return 3.14 * self._radius**2

# Creating an object of the Circle class
circle = Circle(5)

# Accessing the radius property without parentheses
print(circle.radius)  # Output: 5
```

```
# Accessing the area property without parentheses
print(circle.area)    # Output: 78.5

# Modifying the radius property using the setter
circle.radius = 7
print(circle.radius)  # Output: 7
print(circle.area)    # Output: 153.86

# Trying to set an invalid radius value
try:
    circle.radius = -1
except ValueError as e:
    print(e)  # Output: Radius must be positive
```

11. How can you achieve multiple inheritance in Python?

Answer:

Multiple inheritance in Python means that a class can inherit characteristics and features from more than one parent class. You can achieve this by defining a class with more than one parent class inside parentheses in the class definition. This lets the derived class access attributes and methods from all its parent classes.

While this can make code more flexible and powerful, you should use it cautiously. Multiple inheritance can lead to complex class structures.

```
# Parent class 1
class Bird:
    def fly(self):
        return "Flying"

# Parent class 2
class Fish:
    def swim(self):
        return "Swimming"

# Child class inheriting from Bird and Fish
class Duck(Bird, Fish):
    def quack(self):
        return "Quack!"

# Creating an object of the Duck class
duck = Duck()

# Calling methods from the parent classes
print(duck.fly())   # Output: Flying
print(duck.swim())  # Output: Swimming
```

```
# Calling method from the child class
print(duck.quack())  # Output: Quack!
```

12. What is a static method and how is it different from a class method?

Answer:

A static method in Python belongs to a class rather than an instance of the class. You can define it using the *@staticmethod* decorator. It doesn't require reference to the class or its instance and can be called on the class itself. Unlike regular instance methods, it doesn't take a *self* parameter.

A class method, on the other hand, is defined with the *@classmethod* decorator and takes a reference to the class itself as its first parameter – usually named "cls." It can access and modify class-level attributes, while a static method can't.

A class method is more versatile as subclasses can override it, whereas a static method remains unchanged.

```
class MyClass:
    class_variable = "Class variable"

    def __init__(self, value):
        self.instance_variable = value

    # Static method
    @staticmethod
    def static_method():
        return "This is a static method"

    # Class method
    @classmethod
    def class_method(cls):
        return f"This is a class method. Class variable: {cls.class_variable}"

    def show(self):
        return "Instance can call this"

# Creating an object of the MyClass class
obj = MyClass("Instance variable")

# Calling static method without creating an object
print(MyClass.static_method())  # Output: This is a static method

# Calling class method without creating an object
print(MyClass.class_method())   # Output: This is a class method. Class variable: Class
variable

# Accessing class variable through class method
MyClass.class_variable = "New class variable"
print(MyClass.class_method())   # Output: This is a class method. Class variable: New
```

```
class variable

# Accessing class variable through static method (not recommended)
print(MyClass.static_method())  # Output: This is a static method


print(obj.show())
# MyClass.show() # throws error
print(MyClass.show(obj)) # it works
```

```
#output
'''
This is a static method
This is a class method. Class variable: Class variable
This is a class method. Class variable: New class variable
This is a static method
Instance can call this
Instance can call this
'''
```

13. How do you implement abstract classes and methods in Python?

Answer:

In Python, you can implement abstract classes and methods using the *ABC* module. You would first import the module and then create a class that inherits from *ABC*, which stands for abstract base class.

Within this class, you can define abstract methods using the *@abstractmethod* decorator. These abstract methods serve as a blueprint for other classes, enforcing that the derived classes must provide a concrete implementation of these methods.

If a derived class does not implement the abstract methods, an instantiation error will occur.

```
from abc import ABC, abstractmethod

# Abstract class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Concrete subclass implementing the abstract method
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

```
# Creating an object of the concrete subclass
rectangle = Rectangle(5, 3)
print(rectangle.area())  # Output: 15
```

14. How can you prevent method overriding in Python?

Answer:

In Python, method overriding is a common feature that allows a subclass to provide a different implementation of a method defined in its superclass. However, if you want to prevent method overriding, you can do so by defining the method as private.

By prefixing the method's name with a double underscore (`__`), you make it private to the class, and it cannot be overridden in a subclass.

Though not strictly enforced, this convention signals that the method should not be accessed or overridden outside the class, and it helps maintain the integrity of the original method within the hierarchy.

```
class Parent:
    def __private_method(self):
        print("This is a private method in the Parent class.")

    def public_method(self):
        self.__private_method()

class Child(Parent):
    def __private_method(self):
        print("This is an attempt to override the private method in the Child class.")

# Creating objects of the classes
parent = Parent()
child = Child()

# Calling public method of the Parent class
parent.public_method()  # Output: This is a private method in the Parent class.

# Attempting to call private method of the Child class
child.public_method()
# Output: This is a private method in the Parent class.
# Although Child class defines a private method with the same name, it doesn't override
the Parent class's private method.
```

15. How does polymorphism work in Python?

Answer:

Polymorphism in Python allows different objects to be treated as instances of the same class, even if they belong to different classes. You can achieve this through inheritance, where a subclass can have methods with the same name as the methods in the superclass.

When you call a method on an object, Python will automatically use the correct method based on the object's class, even if the object is referred to by a variable of the superclass type.

This enables more flexible and reusable code, as you can write functions that work with different classes, provided they adhere to the same interface or method signature.

```
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Function demonstrating polymorphism
def make_sound(animal):
    return animal.speak()

# Creating objects of different classes
dog = Dog()
cat = Cat()

# Calling the function with different objects
print(make_sound(dog))  # Output: Woof!
print(make_sound(cat))  # Output: Meow!
```

Compile-time Polymorphism:

Compile-time polymorphism, also known as static polymorphism, is resolved during the compilation of the program. A common example is "method overloading".

```
class Employee1:
    def name(self):
        print("Minku is his name")
    def salary(self):
        print("3000 is his salary")
    def age(self):
        print("22 is his age")

class Employee2:
    def name(self):
        print("Minku2 is his name")
    def salary(self):
```

```
        print("4000 is his salary")
    def age(self):
        print("23 is his age")

def func(obj):  # Method Overloading
    obj.name()
    obj.salary()
    obj.age()

emp1 = Employee1()
emp2 = Employee2()

func(emp1)
func(emp2)
```

Output:

```
Harshit is his name
3000 is his salary
22 is his age
Rahul is his name
4000 is his salary
23 is his age
```

Run-time Polymorphism:

Run-time polymorphism, also called dynamic polymorphism, is resolved during runtime. A common example is "method overriding".

```
class Employee:
    def __init__(self, name, age, id, salary):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id

    def earn(self):
        pass

class ChildEmployee1(Employee):
    def earn(self):  # Run-time polymorphism
        print("no money")

class ChildEmployee2(Employee):
    def earn(self):
        print("has money")

c = ChildEmployee1
c.earn(Employee)
```

```
d = ChildEmployee2
d.earn(Employee)
```

Output:

```
no money
has money
```

Explanation:

- In compile-time polymorphism, we use method overloading where different methods have the same name but different parameters.
- In run-time polymorphism, we use method overriding where a method in the child class has the same name and signature as a method in the parent class, but with a different implementation. This allows different behaviors for objects of the same class hierarchy.

16. 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.

```
class employee(object):
    def __init__(self):
        self.name = 'Minku' # public
        self._age = 20 # protected
        self.__salary = '4cr' # private

object1 = employee()
print(object1.name)
print(object1._age)
print(object1.__salary) # Throws error
```

```
Minku
20
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[13], line 10
      8 print(object1.name)
      9 print(object1._age)
---> 10 print(object1.__salary) # Throws error

AttributeError: 'employee' object has no attribute '__salary'
```