1. Basic Python Knowledge:

I) Explain the difference between Python 2 and Python 3.

- Python 3 is the current and recommended version and Python 2 support has officially ended.

- Major changes:

    - Print statement:

      ```
      print "Hello, World!"  # Python 2
      print("Hello, World!")  # Python 3
      ```

    - Division:

      ```
      result = 7 / 2  # result is 3
      result = 7 / 2  # result is 3.5
      ```

    - input() vs. raw_input()

      ```
      city = raw_input("Enter your city: ") # Python 2
      city = input("Enter your city: ") # Python 3
      ```

    - Exceptions:

      Python 2: Exceptions are raised using bare keywords like "raise"
      Python 3: Exceptions are raised using exception objects within parentheses.

    - Unicode:

      Python 3 has better support for Unicode by default. In Python 2, strings were represented as ASCII by default, causing issues with handling non-ASCII characters.

II) Describe Python's data types, such as integers, strings, lists, dictionaries, and Sets.

- Integers: Integers represent whole numbers, positive or negative, without any decimal points.

  ```
  number = 50
  ```

- Floats: Floats represent floating-point numbers, including numbers with decimal points.

  number = 50.5

- Strings: Strings are used to represent text.

  message = 'Hello, World!'

- Lists: Lists are ordered collections of items, which can be of different types. They are mutable, meaning their elements can be changed.

  list = [145, 23, 'car', 4.0]

- Dictionaries: Dictionaries store key-value pairs. They are unordered and mutable.

  dict = {'name': 'Pubudu', 'age': 25, 'city': 'Galle'}

- Tuples: Tuples are similar to lists, but they are immutable once created, meaning their elements cannot be changed.

  tuple = (23, 25, 'banana', 7.0)

- Sets (set): Sets are unordered collections of unique elements. They do not allow duplicate elements.

  set = {1, 2, 3, 4, 5}

III) Describe your understanding of variables, data assignment, and variable scope.

- Variables:

  - Variables are named containers used to store data in memory.
  - In Python, you can create a variable by choosing a name and using the assignment operator (=) to assign a value to it.
  - Variable names in Python can consist of letters (both uppercase and lowercase), digits, and underscores, but they cannot start with a digit.
  - E.g:
    age = 25
    name = "Pubudu"
    price = 150.69

- Data Assignment:

    - The assignment operator (=) assigns a value to a variable.
    - The value can be any valid data type like integers, strings, lists, dictionaries, etc.
    - Multiple variables can be assigned values simultaneously using comma-separated values.
    - E.g:

      ```
      x = 50   # integer assignment
      message = "Hello, World!"  # string assignment
      is_true = True  # boolean assignment
      ```

- Variable Scope:

    - Global Scope: Variables defined outside any function are accessible throughout the program.

      ```
      global_variable = "I am global"

      def another_function():
         print(global_variable)

      another_function()
      # Accessing global_variable outside the function works fine.
      ```

      The global keyword can be used to explicitly modify a global variable from within a function.

    - Local Scope: Variables defined inside a function are only accessible within that function.

      ```
      def my_function():
              local_variable = "I am local"
              print(local_variable)

      my_function()
      ```

2. Control Structures:

I) Write a simple if statement to check a condition.

```
x = 25
if x > 10:
   print("x is greater than 10")
```

II) Advice / write a code that uses a for loop to iterate over a list or range.

```
my_list = [1, 3, 5, 7, 9]

for number in my_list:
    print(number)
```

III) Tell us some example of using while loops.

- Performing Operations until a Condition is Met

  ```
  num = 1
  total = 0

  while num <= 10:
      total += num
      num += 1

  print("The sum of numbers from 1 to 10 is:", total)
  ```

- Handling Invalid Input

3. Functions:

I) Define a function that takes parameters and returns a value.

```
def add_numbers(a, b):
    result = a + b
    return result
```

II) Describe about the usage of keyword arguments and default parameter values.

- Keyword Arguments: It allow to pass arguments to a function by specifying their names explicitly.

  ```
  def greet(name, message="Hello!"):
      print(f"{message}, {name}.")

  # Call with keyword arguments
  greet(message="Good morning", name="John")
  ```

- Default Parameter Values: Default parameter values assign pre-defined values to function arguments if no value is explicitly provided during the call.

  ```
  def calculate_area(width, height=10):
      return width * height

  # Call with default value
  area = calculate_area(5)
  print("Area: {area}")
  ```

III) Request an example of a function that uses the return statement.

```
def calculate_rectangle_area(length, width):
    area = length * width
    return area
```

4. Data Structures:

I) Tell us about your knowledge of lists and their methods (e.g., append, pop, index).

- Append: append() adds an element to the end of the list.

  ```
  my_list = [1, 2, 3]
  my_list.append(4) # Result: [1, 2, 3, 4]
  ```

- Pop: pop() removes and returns the ast element

  ```
  my_list = [1, 2, 3, 4]
  popped_element = my_list.pop()  # Removes and returns 4
  ```

- Index: returns the index of the first occurrence of a specified value.

  ```
  my_list = [10, 20, 30, 20, 40, 20]
  index_of_20 = my_list.index(20) # Result: index_of_20 is 1 (the first occurrence of 20)
  ```

- Insert: inserts an element at a specified position in the list.

  ```
  my_list = [1, 2, 4, 5]
  my_list.insert(2, 3)  # Insert 3 at index 2
  # Result: [1, 2, 3, 4, 5]
  ```

- Remove: removes the first occurrence of a specified value from the list.

  ```
  my_list = [10, 20, 30, 20, 40, 20]
  my_list.remove(20)  # Removes the first occurrence of 20
  # Result: [10, 30, 20, 40, 20]
  ```

II) Advice about work with dictionaries, including adding, modifying, and accessing keys and values.

```
# Creating an empty dictionary
my_dict = {}

# Creating a dictionary with initial values
my_dict = {'name': 'Pubudu', 'age': 25, 'city': 'Colombo'}

# Adding a new key-value pair
my_dict['gender'] = 'Male'
# Result: {'name': 'Pubudu', 'age': 25, 'city': 'Colombo', 'gender': 'Female'}

# Adding multiple key-value pairs
my_dict.update({'occupation': 'Engineer', 'age': 26})
# Result: {'name': 'Pubudu', 'age': 26, 'city': 'Colombo', 'gender': 'Male', 'occupation': 'Engineer'}

# Accessing value by key
print(my_dict['name'])  # Output: Pubudu

# Using get() method to access value by key
print(my_dict.get('city'))  # Output: Colombo

# Accessing all keys
keys = my_dict.keys()
# Result: dict_keys(['name', 'age', 'city', 'gender', 'occupation'])

# Accessing all values
values = my_dict.values()
# Result: dict_values(['Pubudu', 26, 'Colombo', 'Male', 'Engineer'])
```

5. Exception Handling:

I) Write a code that handles exceptions using try and except blocks.

```
def divide(a, b):
    try:
        result = a / b
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed")
    except Exception as e:
        print("An error occurred:", e)
```

II) Tell us about the purpose of the finally block.

The finally block provides a mechanism to maintain code integrity and perform cleanup actions.

```
try:
    file = open('example.txt', 'r')
    # Perform operations on the file
    # ...
except FileNotFoundError:
    print("File not found!")
finally:
    if 'file' in locals():
        file.close()
```

6. File Handling:

I) Provide a code to read from and write to a text file.

Read:

```
with open('example.txt', 'r') as file:
    contents = file.read()
    print(contents)
```

Write:

```
with open('example.txt', 'w') as file:
    file.write("Hello, this is some text that we're writing to the file.\n")
    file.write("This is another line in the file.")
```

II) Explain the difference between reading modes ('r', 'w', 'a').

r: Opens the file for reading only, The file pointer is placed at the beginning of the file.
w: Opens the file for writing only. Creates a new file or truncates the existing file to zero length.
a: Opens the file for writing, appending data to the end of the file without truncating it.

7. Object-Oriented Programming (OOP):

I) Tell us about your understanding about the basics of classes and objects in Python.

Classes: Blueprints for Objects: Classes are essentially blueprints that define the properties (attributes) and methods of objects.

Objects: Objects have attributes and methods

II) Create a simple class with attributes and methods.

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        return f"Starting the engine of {self.make} {self.model}"

car1 = Car("Toyota", "Corolla")
car2 = Car("Tesla", "Model S")
```

8. Modules and Libraries:

I) Tell us about the importing and using external modules (e.g., math, random).

```
import math

print(math.sqrt(16))  # Output: 4.0
print(math.pi)  # Output: 3.141592653589793

from random import randint

print(randint(1, 10))  # Output: A random integer between 1 and 10 (inclusive)
```

II) Tell us about the purpose of commonly used libraries like os, sys, or Datetime.

os: Provides a way to interact with the operating system.
sys: Provides access to some variables used or maintained by the Python interpreter and functions that interact strongly with the interpreter.
Datetime: Provides classes for manipulating dates and times.

9. Basic Algorithms and Problem Solving:

I) Present a coding problem that involves iterating over data and performing a simple operation (e.g., finding the sum of all even numbers in a list).

```
result = sum_even_numbers([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(result)  # Output: 30 (Sum of even numbers: 2 + 4 + 6 + 8 + 10 = 30)
```

Solution:

```
def sum_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
    return even_sum
```

10. Coding Exercises:

I) Write a Python code that could solve a problem by include tasks like reversing a string, calculating Fibonacci numbers, or implementing a simple data structure.

Reversing a String:

```
def reverse_string(input_string):
    return input_string[::-1]
```

Calculating Fibonacci Numbers:

```
def fibonacci(n):
    fib_sequence = [0, 1]
    for i in range(2, n):
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
    return fib_sequence[:n]
```

Stack:

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Stack is empty"
```

11. Version Control:

I) Tell us about your understanding of basic Git commands.

- git init: Initializes a new Git repository in the current directory.
- git clone <repository URL>: Clones an existing repository from a remote URL to your local machine.
- git add <file>: Adds changes in a specific file to the staging area for the next commit.
- git branch: Lists all local branches in the repository.
- git commit -m "Commit message": Commits staged changes with a descriptive message.
- git push origin <branch_name>: Pushes committed changes from a local branch to a remote repository.
- git pull origin <branch_name>: Fetches changes from a remote repository and merges them into the current branch.

—--

Pubudu Wanigasekara

https://www.linkedin.com/in/pubuduarosha/