NAME	
ROLL NUMBER	
SEMESTER	
COURSE CODE	DCA_3104
COURSE NAME	Python Programming

SET - I

Q.1) Write a notes on types of operators in python with appropriate example

Answer.:- I'd be glad to provide notes on the different types of operators in Python, incorporating examples within a 450-word limit:

Operators in Python

Operators are symbols that perform operations on values (operands) in your Python programs. They manipulate data, control the flow of your code, and combine values to produce new results. Here are the primary categories of operators in Python:

1. Arithmetic Operators:

- Perform mathematical calculations on numbers.
  - Examples:
    - `+` (addition): `5 + 3 = 8`
    - `-` (subtraction): `10 - 2 = 8`
    - `*` (multiplication): `4 * 5 = 20`
    - `/` (division): `12 / 3 = 4.0` (floating-point result)
    - `//` (floor division): `12 // 3 = 4` (integer quotient)
    - `%` (modulo): `10 % 3 = 1` (remainder)
    - `**` (exponentiation): `2 ** 3 = 8` (2 raised to the power of 3)

2. Comparison Operators:

- Compare operands and return Boolean values (True or False) based on the comparison.
  - Examples:
    - `==` (equal to): `x == 5` (checks if x is equal to 5)
    - `!=` (not equal to): `y != 10` (checks if y is not equal to 10)
    - `<` (less than): `a < b` (checks if a is less than b)
    - `>` (greater than): `c > d` (checks if c is greater than d)
    - `<=` (less than or equal to): `e <= f` (checks if e is less than or equal to f)
    - `>=` (greater than or equal to): `g >= h` (checks if g is greater than or equal to h)

3. Assignment Operators:

- Assign values to variables.
  - Example:
    - `=` (assignment): `age = 25` (assigns the value 25 to the variable `age`)
  - Combined assignment operators (perform operation and assignment in one step):
    - `+=` (add and assign): `x += 3` (equivalent to `x = x + 3`)
    - `-=` (subtract and assign): `y -= 2` (equivalent to `y = y - 2`)
    - `*=` (multiply and assign): `z *= 4` (equivalent to `z = z * 4`)
    - `/=` (divide and assign): `a /= 5` (equivalent to `a = a / 5`)
    - `//=` (floor divide and assign): `b //= 3` (equivalent to `b = b // 3`)
    - `%=` (modulo and assign): `c %= 2` (equivalent to `c = c % 2`)
    - `**=` (exponentiate and assign): `d **= 3` (equivalent to `d = d ** 3`)

4. Logical Operators:

- Combine conditional expressions and return Boolean values.

- Examples:
  - `and` (returns True if both operands are True): `(x > 0) and (y < 10)`
  - `or` (returns True if at least one operand is True): `(a == 5) or (b != 7)`
  - `not` (inverts the truth value of the operand): `not (name is None)`

5. Identity Operators:

- Check if two objects are the same object in memory.
  - Examples:
    - `is` (returns True if both operands are the same object): `x is y`
    - `is not` (returns True if both operands are not the same object): `a is not b`

6. Membership Operators:

- Check if a value is present in a sequence (like a list, tuple, or string).
  - Examples:
    - `in` (returns True if the value is present): `name in ["Alice", "Bob"]`
    - `not in` (returns True if the value is not present): `number not in [1, 3, 5]`

7. Bitwise Operators (less common but useful for low-level operations):

- Perform operations on bits within integers.
  - We'll cover these in a separate explanation if you

Q.2) Explain about string slicing with examples .

Answer.:- **String Slicing in Python: Extracting Substrings with Ease**

String slicing is a powerful technique in Python that allows you to extract specific portions of a string. It's achieved using square brackets (`[]`) and optional parameters within them. Here's a breakdown of how it works, along with examples to illustrate its versatility:

Basic Slicing:

- Specify the starting and ending indices (positions) of the characters you want to extract, separated by a colon (`:`).
  - **Note:** Indices in Python start at 0, so the first character has index 0, the second has index 1, and so on.
- The ending index is exclusive, meaning the character at that index is not included in the extracted substring.

Examples:

Python

```
string = "Hello, World!"

# Extract characters from index 0 (inclusive) to index 5 (exclusive):
substring = string[0:5]  # "Hello"

# Extract everything from the beginning (index 0) to the end:
substring = string[:]  # "Hello, World!"

# Extract everything from index 7 (inclusive) to the end:
```

```
substring = string[7:]  # "World!"
```

Slicing Without Starting or Ending Index:

- If you omit the starting index, the slice starts from the beginning (index 0).
- If you omit the ending index, the slice goes all the way to the end of the string.

Examples:

Python

```
string = "Good Morning"

# Extract from the beginning to index 5 (exclusive):
substring = string[:5]  # "Good "

# Extract from index 3 (inclusive) to the end:
substring = string[3:]  # "Morning"
```

Negative Indexing:

- Use negative indices to start from the end of the string.
    - Index -1 refers to the last character, -2 to the second-last character, and so on.

Examples:

Python

```
string = "Welcome Back"

# Extract the last character:
substring = string[-1]  # "k"

# Extract the last three characters:
substring = string[-3:]  # "ack"

# Extract from the second-to-last character (inclusive) to the
beginning (exclusive):
substring = string[:-2]  # "Welcome Ba"
```

Step Size (Optional):

- Include a third value after the colon, separated by a space, to specify a step size.
- This determines which characters are included in the substring, skipping characters based on the step value.

Example:

Python

```
string = "Python Programming"

# Extract every other character, starting from the beginning:
substring = string[::2]  # "Pto rman"
```

Remember:

- Slicing a string does not modify the original string. It creates a new string containing the extracted portion.
- Attempting to access an index outside the string's bounds will result in an `IndexError`.

Q.3) Differentiate between local and global variable with example .

Answer .:- Local vs. Global Variables: Understanding Scope in Python

In Python, variables have a concept called **scope**, which determines where they can be accessed and modified within your program. Here's a breakdown of the key differences between local and global variables:

Local Variables:

- **Scope:** Local variables are confined to the function or block (like an if statement or loop) in which they are declared.
- **Lifetime:** They are created when the function or block starts executing and are destroyed when the execution finishes. This ensures they don't interfere with variables in other parts of your code.
- **Modification:** Changes made to local variables within their scope do not affect variables with the same name outside that scope.

Example:

Python

```
def calculate_area(length, width):
  # 'area' is a local variable here
  area = length * width
  return area

# 'length' and 'width' are local variables here (outside the function)
length = 5
width = 10

# Calling the function to calculate area
result = calculate_area(length, width)
print(result)  # Output: 50

# 'area' is not accessible outside the function
# Trying to access it here would result in a NameError
# area = 100
```

Global Variables:

- **Scope:** Global variables can be accessed and modified from anywhere in your program, as long as they are declared before use.
- **Lifetime:** They exist throughout the program's execution, from the beginning to the end.
- **Modification:** Changes made to a global variable anywhere in the program affect its value everywhere else.

Example (Use with caution):

Python

```
# 'global_count' is declared as a global variable (outside any function)
global_count = 0

def increment_count():
  global global_count  # We need to declare 'global_count' again to modify it
  global_count += 1

increment_count()
print(global_count)  # Output: 1 (modified inside the function)

increment_count()
```

```
print(global_count)  # Output: 2 (global value reflects the changes)
```

Why Use Local Variables (Preferably):

- **Encapsulation:** Local variables promote modularity and code organization. Functions have their own private variables, preventing accidental modification from other parts of the code.
- **Namespace Management:** With local variables, you can reuse variable names within different functions without conflicts. This makes your code more readable and less prone to errors.
- **Debugging:** Local variables are easier to track and debug because their scope is limited.

When Might Global Variables Be Useful (Used Sparingly):

- **Program-Wide Constants:** If a value truly needs to be constant throughout the program and accessed from various places, a global constant can be considered. However, explore using constants defined outside functions but within modules for better organization.
- **Configuration Settings:** In some rare cases, global variables might be used for configuration settings that need to be accessed from multiple parts of the program. However, consider using a dedicated configuration file or module for better maintainability.

Q.4) What kinds of errors can one encounter when writing in Python ? Give a through explanation using example .

Answer .:- Encountering errors is a natural part of the Python learning process. Here's a breakdown of some common errors you might face, along with examples and tips for handling them:

1. Syntax Errors:

- These are the most basic errors, occurring when your code violates Python's grammar rules.
  - Examples:
    - Missing colon (:) after an `if` statement:
      Python
      ```
      if x > 5  # Missing colon here
          print("X is greater than 5")
      ```
    - Mismatched indentation (Python relies on indentation for blocks):
      Python
      ```
      if x > 5
      print("X is greater than 5")  # Incorrect indentation
      ```
  - **Solution:** Carefully review your code for typos, missing punctuation, and correct indentation. Python's error messages usually pinpoint the exact line where the syntax error occurs.

2. Name Errors:

- These arise when you try to use a variable that hasn't been defined yet, or when a function or module name is misspelled.
  - Example:
    Python
    ```
    print(name)  # 'name' is not defined yet
    ```
  - **Solution:** Make sure you define variables before using them, and double-check the spelling of function and module names. Use a code editor with syntax highlighting to catch these errors more easily.

3. Type Errors:

- These occur when you perform an operation on values of incompatible types.
  - Example:
    Python
    ```
    number = "10"
    result = number + 5  # Trying to add a string and an integer
    ```
  - **Solution:** Ensure your operations are valid for the data types involved. You can convert data types using functions like `int()` or `str()` if necessary.

4. Index Errors:

- These happen when you try to access an element in a sequence (like a list or string) using an invalid index.
  - Example:
    Python
    ```
    my_list = [1, 2, 3]
    print(my_list[4])  # Index 4 is out of range (list only has 3 elements)
    ```

- **Solution:** Check the valid range of indices for the sequence you're accessing. You can use the `len()` function to find the length of the sequence.

5. Attribute Errors:

- These occur when you try to access an attribute (like a property or method) of an object that doesn't exist.
  - Example:
    Python
    ```
    name = "Alice"
    name.upper()  # 'name' (string) doesn't have a 'upper' method
    ```
  - **Solution:** Verify that the object you're working with has the attribute you're trying to access. Use `dir(object_name)` to see a list of available attributes and methods.

6. Logical Errors:

- These are the trickiest errors to catch, as they might not cause syntax errors but produce incorrect results due to flaws in your program's logic.
  - Example:
    Python
    ```
    def calculate_average(grades):
      total = 0
      for grade in grades:
        total += grade  # Incorrectly adding strings instead of converting to numbers
      return total / len(grades)

    # Calling the function with a list of strings (grades)
    ```
  - **Solution:** Carefully analyze your code's logic and test it with different inputs to identify unintended behavior. Use print statements or a debugger to step through your code and inspect values at different points.
- Error messages often provide valuable clues about the type of error and its location in your code.
- Use a code editor with syntax highlighting and error checking to catch basic errors early on.
- Practice good coding habits like indentation, meaningful variable names, and commenting your code to make it easier to debug.
- Leverage Python's built-in debugging tools like `print()` statements or a debugger to step through your code and identify the root cause of errors.

Q.5) Define python tuples ? With example explain the concept of Accessing Values in tuples , updating tuples and deleting tuple elements .

Answer .:- **Python Tuples: Ordered and Immutable Collections**

Tuples in Python are ordered, immutable collections of elements. They are similar to lists in terms of storing multiple items, but with a key difference: tuples cannot be modified after creation. This makes them ideal for situations where you need data integrity and want to prevent accidental changes.

1. Creating Tuples:

Tuples are enclosed in parentheses `()`, with elements separated by commas.

Python

```
# Empty tuple
my_tuple = ()

# Tuple with different data types
my_tuple = ("apple", 42, True)

# Single-element tuple (requires a trailing comma)
name = ("Alice",)
```

2. Accessing Values:

- Elements in a tuple are accessed using zero-based indexing, similar to lists.
- The index starts from 0 for the first element, 1 for the second, and so on.

Python

```
fruits = ("mango", "banana", "orange")
first_fruit = fruits[0]  # first_fruit will be "mango"
last_fruit = fruits[2]  # last_fruit will be "orange"

# Negative indexing to access from the end
second_last_fruit = fruits[-2]  # second_last_fruit will be "banana"
```

3. Immutability (Cannot Update Tuples):

- Once a tuple is created, you cannot modify its elements. Trying to do so will result in a `TypeError`.

Python

```
numbers = (1, 2, 3)

# Attempting to modify an element will raise a TypeError
# numbers[1] = 5  # This will cause an error
```

4. Working with Tuples:

- Tuples support various operations like concatenation (`+`), repetition (`*`), membership (`in`, `not in`), and length (`len()`).
- You can create a new tuple with modifications based on the original one.

Python

```
fruits = ("apple", "banana")
more_fruits = ("mango", "orange")

# Concatenation
all_fruits = fruits + more_fruits

# Repetition
repeated_fruits = fruits * 3

# Membership check
```

```
is_mango_present = "mango" in all_fruits

# Finding the length
num_fruits = len(fruits)
```

5. When to Use Tuples:

- Use tuples when you need an ordered collection of elements that should not be changed after creation.
- They are useful for representing data structures that have a fixed definition, like coordinates (x, y) or representing configuration settings.

Q.6) With suitable example explain the concept of using else statement with loops related to while loop .

Answer .:- **The `else` Statement with `while` Loops in Python**

The `else` statement combined with a `while` loop in Python provides a way to execute code only when the loop terminates normally (i.e., the loop condition becomes `False`). This allows for cleaner and more readable code, especially when you want to perform an action **after** the loop has completed all its iterations.

Here's how it works:

1. **The `while` Loop:**
    - The `while` loop continues to execute its block of code as long as the specified condition remains `True`.
2. **The `else` Block:**
    - The `else` block is indented after the `while` loop.
    - It only gets executed **once** when the loop's condition eventually becomes `False`, and the loop exits normally (without using `break`).

Example:

Python

```
count = 1
while count <= 5:
  print(count)
  count += 1

# The else block executes after the loop finishes iterating 5 times
else:
  print("Loop completed! Reached count", count)
```

Output:

```
1
2
3
4
5
Loop completed! Reached count 6
```

Explanation:

- The `while` loop iterates 5 times, printing the `count` value from 1 to 5.
- Since the loop condition (`count <= 5`) becomes `False` after the fifth iteration, the loop exits normally.
- The `else` block then executes, printing the message "Loop completed!" along with the final value of `count` (which is now 6 as it was incremented in the last iteration).

Benefits of Using `else` with `while` Loops:

- **Clarity and Readability:** It separates the code that executes after the loop from the loop's main logic, making your code easier to understand and maintain.
- **Logical Code Organization:** You can place code that relies on the loop's completion within the `else` block, ensuring it only runs when the loop finishes iterating as expected.

Important Points:

- If the loop exits prematurely using `break`, the `else` block will not be executed.
- You can use `else` with `for` loops as well, following the same principles.

Additional Example:

Python

```
search_term = "python"
found = False

# Loop to search for the term in a list
while search_term in my_list:
  print("Found", search_term)
  found = True
  break  # Exit the loop after finding the first occurrence

# The else block will execute if the loop doesn't find the term
else:
  print(search_term, "not found in the list")
```

In this example, the `else` block will only print "python not found in the list" if the `search_term` is not present in the `my_list` at all. If it's found even once, the loop exits using `break`, and the `else` block is skipped.