**Q2)Python code execution flow explanation**

Ans) The flow of execution basically refers to the order in which statements are executed. That is to say, execution always starts at the first statement of the program. Moreover, statements execute one at a time. It happens in order, from top to bottom.

Further, functions definitions do not alter the flow of execution of the program. However, it remembers the statements inside the function do not execute until the functions is called.

Moreover, function calls are similar to a bypass in the flow of execution. Thus, instead of going to the next statement, the flow will jump to the first line of the called function. Then, it will execute all the statements there. After that, it will come back to pick up where it left off.Q.

Q3)List partition question.

```
l1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
partition_size = 4
result = [l1[i:i + partition_size] + [None] * (partition_size - len(l1[i:i + partition_size])) for i in range(0,
len(l1), partition_size)]
print(result)
```

Q4)Use Enum in python

```
from enum import Enum

class Role(Enum):
  MANAGER = "Manager"
  EMPLOYEE = "Employee"
  INTERN = "Intern"

class Employee:
  def __init__(self, name, role):
    self.name = name
    self.role = role
```

```
# Example usage:
emp1 = Employee("Alice", Role.MANAGER)
emp2 = Employee("Bob", Role.EMPLOYEE)
emp3 = Employee("Charlie", Role.INTERN)

print(emp1.name, "is a", emp1.role.value)
print(emp2.name, "is an", emp2.role.value)
print(emp3.name, "is an", emp3.role.value)
```

**Q5) Prepare notes for itertools functions which are commonly used.**

**Itertools** is a module in Python, it is used to iterate over data structures that can be stepped over using a for-loop. Such data structures are also known as iterables. This module works as a fast, memory-efficient tool that is used either by themselves or in combination to form iterator algebra.

**Different types of iterators provided by this module are:**

- Infinite iterators
- Combinatoric iterators
- Terminating iterators

**Infinite iterators**

Iterator in Python is any Python type that can be used with a 'for in loop'. Python lists, tuples, dictionaries, and sets are all examples of inbuilt iterators. But it is not necessary that an iterator object has to exhaust, sometimes it can be infinite. Such types of iterators are known as **Infinite iterators**.

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| count() | [start[, step]] | start, start+step, start+2*step, ... | count(10) → 10 11 12 13 14 ... |
| cycle() | p | p0, p1, ... plast, p0, p1, ... | cycle('ABCD') → A B C D A B C D ... |
| repeat() | elem [,n] | elem, elem, elem, ... endlessly or up to n times | repeat(10, 3) → 10 10 10 |

## Combinatoric iterators

The recursive generators that are used to simplify combinatorial constructs such as permutations, combinations, and Cartesian products are called combinatoric iterators.

| Iterator | Arguments | Results |
|---|---|---|
| product() | p, q, … [repeat=1] | cartesian product, equivalent to a nested for-loop |
| permutations() | p[, r] | r-length tuples, all possible orderings, no repeated elements |
| combinations() | p, r | r-length tuples, in sorted order, no repeated elements |
| combinations_with_replacement() | p, r | r-length tuples, in sorted order, with repeated elements |

## Terminating iterators

Terminating iterators are used to work on the short input sequences and produce the output based on the functionality of the method used.

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| accumulate() | p [,func] | p0, p0+p1, p0+p1+p2, … | accumulate([1,2,3,4,5]) → 1 3 6 10 15 |
| batched() | p, n | (p0, p1, …, p_n-1), … | batched('ABCDEFG', n=3) → ABC DEF G |
| chain() | p, q, … | p0, p1, … plast, q0, q1, … | chain('ABC', 'DEF') → A B C D E F |
| chain.from_iterable() | iterable | p0, p1, … plast, q0, q1, … | chain.from_iterable(['ABC', 'DEF']) → A B C D E F |
| compress() | data, selectors | (d[0] if s[0]), (d[1] if s[1]), … | compress('ABCDEF', [1,0,1,0,1,1]) → A C E F |
| dropwhile() | predicate, seq | seq[n], seq[n+1], starting when predicate fails | dropwhile(lambda x: x<5, [1,4,6,4,1]) → 6 4 1 |
| filterfalse() | predicate, seq | elements of seq where predicate(elem) fails | filterfalse(lambda x: x%2, range(10)) → 0 2 4 6 8 |
| groupby() | iterable[, key] | sub-iterators grouped by value of key(v) | |
| islice() | seq, [start,] stop [, step] | elements from seq[start:stop:step] | islice('ABCDEFG', 2, None) → C D E F G |

| | | | |
|---|---|---|---|
| pairwise() | iterable | (p[0], p[1]), (p[1], p[2]) | pairwise('ABCDEFG') → AB BC CD DE EF FG |
| starmap() | func, seq | func(*seq[0]), func(*seq[1]), ... | starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000 |
| takewhile() | predicate, seq | seq[0], seq[1], until predicate fails | takewhile(lambda x: x<5, [1,4,6,4,1]) → 1 4 |
| tee() | it, n | it1, it2, ... itn splits one iterator into n | |
| zip_longest() | p, q, ... | (p[0], q[0]), (p[1], q[1]), ... | zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D- |

**Q 6)Example of pass by reference and pass by value.**

**Ans)**

**Pass by value**: When passing immutable objects like integers, floats, strings, and tuples, a copy of the object is passed to the function. Any modifications made to the object within the function do not affect the original object.

```
def modify_value(x):
  x += 1
  print("Inside the function:", x)

value = 10
modify_value(value)
print("Outside the function:", value)
Inside the function: 11
Outside the function: 10
```

**Pass by reference**: When passing mutable objects like lists, dictionaries, and sets, the reference to the original object is passed. Modifications made to the object within the function affect the original object.

```
def modify_list(lst):
  lst.append(4)
  print("Inside the function:", lst)
```

```
my_list = [1, 2, 3]
modify_list(my_list)
print("Outside the function:", my_list)
Inside the function: [1, 2, 3, 4]
Outside the function: [1, 2, 3, 4]
```

**Q7)Demo of Union operator using typing library**

```
from typing import Union

def square_or_string(x: Union[int, float, str]) -> Union[int, str]:
  if isinstance(x, (int, float)):
    return x * x
  elif isinstance(x, str):
    return x.upper()

# Testing the function
print(square_or_string(5))    # Output: 25
print(square_or_string(2.5))  # Output: 6.25
print(square_or_string("hello"))  # Output: "HELLO"
```

Q8)How to check if variable is iterable python.

Ans) To check if a variable is iterable in Python, use the **iter()** function along with a try-except block.

```
def is_iterable(obj):
  try:
    iter(obj)
    return True
  except TypeError:
    return False

# Example usage:
print(is_iterable([1, 2, 3]))  # Output: True
print(is_iterable("hello"))   # Output: True
print(is_iterable(5))         # Output: False
```

**Q9)What are various ways to implement string formatting in python.**

**Ans)**

**1) Using the % Operator (Old Style Formatting):**

```
name = "Alice"
age = 30
formatted_string = "Name: %s, Age: %d" % (name, age)
print(formatted_string)
```

**2) Using the str.format() Method (str.format() Method):**

```
name = "Alice"
age = 30
formatted_string = "Name: {}, Age: {}".format(name, age)
```

**3) Using f-strings (Formatted String Literals):**

```
name = "Alice"
age = 30
formatted_string = f"Name: {name}, Age: {age}"
```

**4) Using Template Strings (string.Template):**

```
name = "Alice"
age = 30
template = Template("Name: $name, Age: $age")
formatted_string = template.substitute(name=name, age=age)
```

**Q10)Example of any() and all().**

**Ans)**

Any()-The **any()** function returns **True** if at least one element in the iterable is **True**, and **False** otherwise.

```
# Example 1
numbers = [1, 2, 3, 4, 0]
result = any(numbers)
print(result)  # Output: True

# Example 2
strings = ["hello", "", "world", ""]
result = any(strings)
print(result)  # Output: True
```

All()-The **all()** function returns **True** if all elements in the iterable are **True**, and **False** otherwise.

```
# Example 1
numbers = [1, 2, 3, 4, 0]
result = all(numbers)
print(result)  # Output: False

# Example 2
booleans = [True, True, True, True]
result = all(booleans)
print(result)  # Output: True
```