Q2. Explain Python code execution flow?

Ans: The execution starts from top to bottom as program written. Program statements are executed one at a time. When you define a function in a program, it doesn't change how the program runs. Instead, it's like stores the statements and executes them when that function is used later.

When you call a function, it's like taking a bypassing the program's flow. Instead of following the next line of code in order, the program jumps to the beginning of the function you called. It then runs through all the steps inside that function. Once it's done with the function, it goes back to where it was in the main program and continues from there. It's a bit like pausing your main task to do something else, and then coming back to where you left off.

When a nested function is called, control jumps to that function. It returns control back to the calling function when it finishes.

Q3. Perform list partitioning

```
from itertools import zip_longest

def split_list(lst, chunk_size):
  return list(zip_longest(*[iter(lst)] * chunk_size, fillvalue=None))

numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
chunk_size = 3
chunks = split_list(numbers, chunk_size)
print("Chunks:", chunks)
```

Q4. Perform a use of enum in python

Ans:

```
from enum import Enum
class Color(Enum):
  RED = 1
  GREEN = 2
  BLUE = 3

print(Color.RED.value)
print(Color.GREEN)
```

```
print("\nIterating on enum:")
for color in Color:
    print(color)
```

Q5. Utilize itertools functions which are commonly used.

Ans:

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:

1. Infinite iterators
2. Combinatoric iterators
3. Terminating iterators

1. 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 must 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

2. Combinatoric iterators

The recursive generators 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

3. 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-

Q6. Give example of pass by reference and pass by value
Ans:

Pass by Value:
- If we change the value of the object inside the function, the changes will not reflect outside the function as well.

```
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 Value:
- When we are passing an argument to a function, we pass the reference to the object.
- The below example shows, when we make changes to the list passed to modify_list function then the changes will also reflect to the original list.

```
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. Give the demo of Union operator using typing library
Ans:

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

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

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

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

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

4) Using Template Strings (string.Template):

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

Q10. Give example of any() and all()
Ans:

1) any():
- any() function returns 'True' if at least one element in an iterable is 'True', and 'False' otherwise.
- If the iterable is empty then it returns 'False'.
- For dictionaries, any() checks whether any key evaluates to True.

```
lst = [False, False, True, False]
result1 = any(lst)
print(result1)

tup = (0, '', None, False)
result2 = any(tup)
print(result2)
```

2) all():
- all() function returns 'True' if all the elements in an iterable is 'True', and 'False' otherwise.
- If the iterable is empty then it returns 'True'.

- For dictionaries, all() checks whether all keys evaluate to True.

```
my_list = [True, True, True, True]
result = all(my_list)
print(result)

my_tuple = (0, '', None, False, True)
result = all(my_tuple)
print(result)
```