1. What is the relationship between def statements and lambda expressions ?

Answer: Both def and lambda create functions, they both have the same capability, with the only difference being lambda is a single line expression, and def creates a function encompassing multiple lines of code.

Lambda does not require a function name, however a function can be passed into lambda just like def function.

Def function contains a function name.

Lambda Function contains only one expression which returns a functional object that can be assigned to a value outside lambda.

Def function can return a value, or print a value as per users requirement.
Def function is for larger functions which requires more variables, objects.

2. What is the benefit of lambda?

Answer: The benefit of lambda are as follows :-

Readability,

Shorter Code,

Conciseness,

Elimination of variables,

Functional Programming,

Iterative Syntax,

Simplified variable scope, etc

3. Compare and contrast map, filter, and reduce.

Answer:

- The map function iterates through an iterable and executes a function we passed as an argument to it.

Example:

Map Function syntax >>> **map(function, iterable)**

```
fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
map_object = map(lambda s: s[0] == 'A',fruit)

print(list(map_object))
 >>> [True, False, False, True, False]
```

- The filter function liek map, iterates through an iterable and executes a function passed to it and creates a list.

  Example: filter expression >>> filter(function, iterable)

```
fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
filter_object = filter(lambda s: s[0] == 'A',fruit)

print(list(filter_object))

>>> ['Apple', 'Apricot']
```

- The reduce function works differently than map and reduce, it does not return a list based on the passed function but returns a single value. Reduce will execute the function on the first two values in the list and return the answer, and pass that answer with the next variable to the function in the next call, and so on.

  Example: reduce expression >>> reduce(function, sequence[,initial])

```
from functools import reduce
list1 = [1,2,3,4,5,6]

print(reduce(lambda x,y: y+x,list1))


OUTPUT >>> 21
```

4. What are function annotations, and how are they used?

Answer: Function annotations are arbitrary functions that are associated with various parts of a function. These expressions are evaluated at compile time but have no life in python run time. Python does not attach any meaning to these annotations. The benefits of python annotations can be reaped by third parties

For instance, the annotation
[def foo(a:'int', b:'float'=5.0) → 'int'] will keep track of input parameter types and return type of the function to keep track of the type change occurring in the function.

Some syntaxes of annotations

1) nested functions → def foo ((a:expression,b:expression),(c:expression,d:expression))
2) excess parameters → def foo(*args: expression, **kwargs:expression)
3) simple parameters → def foo(a: expression, b:expression)

5. What are recursive functions, and how are they used?

Answer: Recursive functions are functions that call itself from within the function. Similar to functions which call other functions, the only difference being recursive functions calls itself.

Recursive functions are used in a construct wherein the function is called from within itself as below

```
def factorial(x):
   while x > 1:
      return (x * factorial(x-1))
   else:
      return 1

result=factorial(4)
result

OUTPUT >> 24
```

6. What are some general design guidelines for coding functions?

Answer: Here are some of the design guidelines for coding functions :-

1) Try not to use GOTO statements
2) Limited use of globals
3) Standard header for different modules
4) Meaningful Naming convention for variables, functions, classes, global and local variables
5) Error return and exception handling convention such as error should return 0 or 1 to simplify the debugging

7. Name three or more ways that functions can communicate results to a caller.

Answer: Here are three ways a function can send results to the caller :-

1) A return statement can be used to end the execution and return a value to the caller which can be assigned to a variable at the caller end.

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

   sum=foo(2,3)
   ```

   Here sum will store the value 5

2) We can use multiple return statements to return multiple values, and they can be assigned to multiple variables at the caller end

   ```
   def foo(a,b):
       return a*a,b*b

   a_sqaure,b_square = foo(2,3)
   ```

   Here a_square will hold the first return variable and b_square will hold the second

3) Using the yield function, a generator function saves the current state of the function and the yield variable, which can be called using the next function. The yield does not end the execution of the function, rather it saves the current state of the function, returns the value when th function object is passed into next().

```
def fibo():
   a,b=0,1
   while True:
      yield a
      a,b=b,a+b

series = fibo()

print(', '.join(str(next(series)) for _ in range(10)))

OUTPUT  >>> 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```