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

Ans: Both lambda and def create function objects to be called later. Because lambda is an expression, though, it returns a function object instead of assigning it to a name, and it can be used to nest a function definition in places where a def will not work syntactically. A lambda allows for only a single implicit return value expression, though; because it does not support a block of statements, it is not ideal for larger functions.

2. What is the benefit of lambda?

Ans: lambdas allow us to "inline" small units of executable code, defer its execution, and provide it with state in the form of default arguments and enclosing scope variables. Using a lambda is never required; you can always code a def instead and reference the function by name. lambdas come in handy, though, to embed small pieces of deferred code that are unlikely to be used elsewhere in a program. They commonly appear in callback-based programs such as GUIs, and they have a natural affinity with functional tools like map and filter that expect a processing function.

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

Ans: These three built-in functions all apply another function to items in a sequence (or other iterable) object and collect results. map passes each item to the function and collects all results, filter collects items for which the function returns a True value, and reduce computes a single value by applying the function to an accumulator and successive items. Unlike the other two, reduce is available in the functools module in 3.X, not the built-in scope; reduce is a built-in in 2.X.

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

Ans: Function annotations, available in 3.X (3.0 and later), are syntactic embellishments of a function's arguments and result, which are collected into a dictionary assigned to the function's __annotations__ attribute. Python

places no semantic meaning on these annotations, but simply packages them for potential use by other tools.

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

Ans: Recursive functions call themselves either directly or indirectly in order to loop. They may be used to traverse arbitrarily shaped structures, but they can also be used for iteration in general (though the latter role is often more simply and efficiently coded with looping statements). Recursion can often be simulated or replaced by code that uses explicit stacks or queues to have more control over traversals.

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

Ans: Functions should generally be small and as self-contained as possible, have a single unified purpose, and communicate with other components through input arguments and return values. They may use mutable arguments to communicate results too if changes are expected, and some types of programs imply other communication mechanisms

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

Ans: Functions can send back results with return statements, by changing passed-in mutable arguments, and by setting global variables. Globals are generally frowned upon (except for very special cases, like multithreaded programs) because they can make code more difficult to understand and use. return statements are usually best, but changing mutables is fine (and even useful), if expected. Functions may also communicate results with system devices such as files and sockets, but these are beyond our scope here.