```python
# Q1)
'''
In Python, a lambda function is a small, anonymous function defined
using the lambda keyword.
It is a way to create functions on the fly without formally defining
them using the def keyword.
Lambda functions are often used for short, simple operations.

Lambda Function:

Anonymous function using lambda.
Single expression.
No explicit return statement.

Regular Function:

Named function using def.
Can have multiple expressions and statements.
Uses return statement for explicit return.
'''
```

```
'\nIn Python, a lambda function is a small, anonymous function defined
using the lambda keyword. \nIt is a way to create functions on the fly
without formally defining them using the def keyword. \nLambda
functions are often used for short, simple operations.\n'
```

```python
square = lambda x: x**2
print(square(5))
```

```
25
```

```python
# Q2)
'''
Yes, a lambda function in Python can have multiple arguments.
We can define and use multiple arguments in a lambda function in the
following way:
'''

multiply = lambda x, y: x * y
print(multiply(3, 4))
```

```
12
```

```python
# Q3)
'''
Lambda functions in Python are typically used for short-lived
operations where a
full function definition might be overly verbose. They are often
employed in situations
where a small, anonymous function is needed, especially when functions
are used as arguments
```

```python
to higher-order functions like map(), filter(), or sorted().
'''

numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

```python
# Q4)
'''
Advantage:
Conciseness: Lambda functions are concise and useful for short, simple
operations.
Readability: They can enhance readability for straightforward logic.
Inline Usage: Often used inline as arguments for functions like map()
and filter().

Limitations:
Limited Expressiveness: Can only contain a single expression.
No Statements: Cannot include statements, limiting complexity.
Lack of Name: Anonymous, making code less self-documenting.
Reduced Debugging: Debugging can be more challenging than with named
functions.
'''

# Q5)
'''
Yes, lambda functions in Python can access variables defined outside
of their own scope.
This behavior is known as lexical scoping or closure.
'''

outside_variable = 10

# Lambda function accessing a variable from the outer scope
lambda_function = lambda x: x + outside_variable

result = lambda_function(5)
print(result)
```

```
15
```

```python
# Q6)
square = lambda x: x**2
result = square(5)
print(result)
```

```
25
```

```python
# Q7)
find_max = lambda lst: max(lst)
```

```python
numbers = [10, 5, 8, 20, 15]
max_value = find_max(numbers)
print(max_value)
```

```
20
```

```python
# Q8)
filter_even = lambda lst: list(filter(lambda x: x % 2 != 0, lst))
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
filtered_list = filter_even(numbers)
print(filtered_list)
```

```
[1, 3, 5, 7, 9]
```

```python
# Q9)
sort_by_length = lambda lst: sorted(lst, key=lambda x: len(x))
strings = ["apple", "banana", "kiwi", "orange"]
sorted_strings = sort_by_length(strings)
print(sorted_strings)
```

```
['kiwi', 'apple', 'banana', 'orange']
```

```python
# Q10)
find_common_elements = lambda list1, list2: list(filter(lambda x: x in
list1, list2))

# Q11)
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
result = factorial(5)
print(result)
```

```
120
```

```python
# Q12)
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
result = fibonacci(6)
print(result)
```

```
8
```

```python
# Q13)
def list_sum(lst):
    if not lst:
        return 0
    else:
        return lst[0] + list_sum(lst[1:])
numbers = [1, 2, 3, 4, 5]
result = list_sum(numbers)
print(result)
```

```
15
```

```
# Q14)
def is_palindrome(s):
    s = s.lower()
    if len(s) <= 1:
        return True
    elif s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])

result1 = is_palindrome("radar")
print(result1)

result2 = is_palindrome("hello")
print(result2)
```

```
True
False
```

```
# Q15)
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
result = gcd(48, 18)
print(result)
```

```
6
```