Advance Function Concepts

Python Anonymous (Lambda) Functions

- A lambda function in Python is a small, anonymous function that can be defined without name.
- Lamdba functions are used to write functions consisting of a single statement.
- A lambda function can take any number of arguments, but can only have one expression.
- It is created using the 'lambda' keyword, and it is often used as an argument to a higher-order function (a function that takes another function as an argument).

Syntax:

The syntax of a lambda is

```
lambda arguments:express
```

Example #1:

Following code is used to write the function to add 10 in given number.

```
def add_ten(x)
    return x + 10
```

above function can be written by the lamdba function in python.

```
add_ten = lamdba x: x + 10
print(add_ten(5) # 15
```

Example #2: multiple two numbers

use of lambda function to multiple two numbers

```
mul = lambda a, b : a * b
print(mul(2,4)) # 8
```

Example #3:

```
lambda x, y : x + y
```

```
_(6,8) # 14
```

Note: In the interactive interpreter, the single underscore**(_)** is bound to the last expression evalued.

Example #4: Immediately invoked function expression

```
(lambda x, y : x + y)(6,8) # 14
```

The lambda function above is defined and then immediately called with two arguments (6,8). it retuns the value **14**, which is the sum of the arguments.

Example #5:

```
def multiply(lambda(x,y):
    retun x*y

result = (lambda x,y : multiply(x,y))(5,3)
print(result) # Output: 15

mult = lambda x,y : multiply(x,y)
result = mult(6,2)
print(result) # Output: 12
```

- Video: How to: Use of Lambda function
- Video: How to: Use of Lambda function

2. **Higher-Order Functions**

- A higher-order function is one that takes another function as an argument or returns a function as its result.
- Common examples in Python include `map`, `filter`, and `reduce`.

Map()

- In Python, the 'map()' function is used to apply a certain function to each element of an iterable (e.g. list, tuple, etc.) and return an iterable containing the results.
- The 'map()' function takes two arguments: a function and an iterable. The function is applied to each element of the iterable, and the result of the function is included in the resulting iterable.

Example #6: use of lambda in map() function

```
numbers = [1,2,3,4,5,6,7,8,9,10]

squared_numbers = map(lambda x : x **2 ,numbers)
```

```
print(list(squared_numbers))
```

- `map` applies a function to all items in an iterable:

```
numbers = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)  # Output: [1, 4, 9, 16]
```

Filter()

- In Python, the 'filter()' function is used to filter a sequence (e.g. list, tuple, etc.) by applying a certain test to each eleent of the sequence and returning only the elements that pass the test.
- The 'filter()' function takes two arguments: function and an iterable. The function is applied to each element of the iterable, and if the function retuns 'True' for that element, the element is included in the resulting iterable.

Example #5 use of lambda in filter() function

```
numbers = [1,2,3,4,5,6,7,8,9,10] # list

even_numbers= list(filter(lambda x : x % 2 == 0,numbers))

print(even_numbers)
```

- `filter` filters items in an iterable based on a function that returns `True` or `False`:

```
numbers = [1, 2, 3, 4]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

See also:

- Video: How to: use a LAMBDA function as an argument in filter() and map()

More on Defining Functions

`*args` and `**kwargs`

Arbitrary Positional Arguments (`*args`)

These allow a function to take any number of positional arguments. Inside the function, `*args` collects all the positional arguments as a tuple.

- Video: How to Use *args in Python Functions
- Video: Understanding *args in Functions - How to Add Any Number of Arguments with *args

Example:

```
def greet(*names):
    for name in names:
        print(f"Hello, {name}!")

greet("Ali", "Hamza", "Ahmad")
```

Output:

```
Hello, Ali!
Hello, Hamza!
Hello, Ahmad!
```

In this example, the `greet` function can take any number of names. The `*names` collects them into a tuple (`names`), which can be iterated over.

1. Arbitrary Keyword Arguments (`**kwargs`)

These allow a function to accept any number of keyword arguments (arguments passed as key-value pairs). Inside the function, `**kwargs` collects these as a dictionary.

- Video: How to use **kwargs in Python

Example:

```
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Ali", age=25, city="Multan")
```

Output:

```
name: Ali
age: 25
city: Multan
```

In this case, the function accepts any number of keyword arguments and collects them into a dictionary (`info`), which you can then work with inside the function.

Combined Use

You can also use both `*args` and `**kwargs` in the same function to handle a combination of positional and keyword arguments.

Example:

```
def display_data(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

display_data(1, 2, 3, name="Ali", age=25)
```

Output:

```
Positional arguments: (1, 2, 3)
Keyword arguments: {'name': 'Ali', 'age': 25}
```

Key Points:

- `*args` collects all positional arguments into a tuple.
- `**kwargs` collects all keyword arguments into a dictionary.
- You can use both `*args` and `**kwargs` together to handle any type of arguments passed to a function.
-

'nonlocal' keyword

In Python, the `nonlocal` keyword is used to declare that a variable inside a nested function refers to a variable in the nearest enclosing scope that is not global. This allows you to modify a variable from an outer (but not global) scope within a nested function.

Here's an example to illustrate how `nonlocal` works:

```
def outer_function():
    x = 10  # This is the enclosing variable

    def inner_function():
        nonlocal x  # Declare that we want to use the outer variable x
        x += 5      # Modify the outer variable
        print("Inner x:", x)

    inner_function()
    print("Outer x:", x)

outer_function()
```

Output:

```
Inner x: 15
Outer x: 15
```

Explanation:

1. `outer_function` defines a variable `x`.
2. `inner_function` modifies `x` using the `nonlocal` keyword.
3. When `inner_function` is called, it updates `x`, and both the inner and outer prints show the updated value.

When to Use `nonlocal`:

- When you have nested functions and you want to modify a variable from the outer function.
- When you need to avoid using global variables and want to keep your code cleaner and more modular.

If you don't use `nonlocal`, Python will treat the variable as a new local variable in the inner function, which can lead to unexpected behavior or errors.

Key Terms

Fix the Errors

True/False (Mark T for True and F for False)

Answer Key (True/False):

Multiple Choice (Select the best answer)

41. **What is the output of the following code?** [Python Code #92]

```
def my_func():
  global x
  x = 10

x = 5
my_func()
print(x)
```

```
- A) `5`
- B) `10`
- C) `None`
- D) `Error`
```

```
def outer():
    x = 1
    def inner():
        print(x)
    return inner

func = outer()
func()
```

- A) None
- B) Error
- C) 1
- D) Function object

27. **What will be the output of the following code?** [Python Quiz #95]

```
def outer():
    x = 5
    def inner():
        nonlocal x
        x = 10
    inner()
    return x
print(outer())
```

- A) 5
- B) 10
- C) None
- D) Error

What is the output of the following code? [Python Quiz #2]

```
def foo(x):
  if x == 1:
    return 1
  else:
    return x * foo(x - 1)

print(foo(5))
```

- A) 5
- B) 15
- C) 120
- D) None

Watch this video for answer: https://www.youtube.com/shorts/k50czTu7vao

For more details, see Appendix A

1. **What is the output of the following code?** [Python Quiz #30]

```
def calculate_sum(n):
    if n == 0:
        return 0
    else:
        return n + calculate_sum(n-1)

print(calculate_sum(4))
```

- A) 4
- B) 6
- C) 10
- D) 15

Watch the video for the answer: https://youtube.com/shorts/LQEfGgJYlT4?si=MDvSvVHiBc6hCJ0W

4. **What is the output of the following expression?** [Python Quiz #13]

```
def add(a,b,*parm):
    total = 0
    print(a+b)
    for n in parm:
        total += n
    return total

print(add(1, 2))
```

- A) 3 0
- B) 3
- C) 0
- D) Error

Watch this video for answer: https://youtube.com/shorts/k4KVCxU5oMg

5. **What is the output of the following code?** [Python Quiz #14]

```
def add(*args):
     print(type(args))

add(1, 2,8,9)
```

- A) set
- B) tuple

- C) list
- D) None

Watch this video for answer: https://youtube.com/shorts/VQT4Cllpf9M

30. **What is the output of the following code?** [Python Quiz #97]

```
def f(a, b, *args):
    return len(args)
print(f(1, 2, 3, 4, 5))
```

- A) 2
- B) 3
- C) 5
- D) None

1. **What is the output of the following code?** [#41 Python Quiz]

```
def display_data(**kwargs):
    print(type(kwargs))

display_data(name="Ali", age=25)
```

- A) <class 'set'>
- B) <class 'tuple'>
- C) <class 'list'>
- D) <class 'dict'>

Watch this video for answer: https://youtu.be/5IWmz7iWqUE?si=Wx0OeTwME3XEiL-h

What is the output of the following code? [Python Quiz #98]

```
def outer_function(message):
    def inner_function():
        print(message)
    return inner_function

my_function = outer_function("Hello, world!")
my_function()
```

- A) Hello, world!
- B) Error
- C) None
- D) outer_function

1. **What is the output of the following code?** [Python Quiz #99]

```
def apply_function(func, x):
    return func(x)

def square(x):
    return x * x

result = apply_function(square, 5)
print(result)
```

- A) 25
- B) 5
- C) 10
- D) Error

Function Composition

21. **What is function composition in Python?**

- A. Combining multiple functions into a single function
- B. Applying a function multiple times
- C. Creating a new function from existing functions
- D. All of the above

22. **What is the output of the following code?**

```
def square(x):
    return x * x

def add_one(x):
    return x + 1

def compose(f, g):
    def composed_function(x):
        return f(g(x))
    return composed_function

result = compose(add_one, square)(5)
print(result)
```

- A. 26
- B. 36
- C. 25
- D. 11

Partial Application

23. **What is partial application in Python?**

- A. Applying a function to some of its arguments
- B. Creating a new function with fewer arguments
- C. Applying a function multiple times
- D. All of the above

24. **What is the output of the following code?**

```
from functools import partial

def add(x, y):
    return x + y

add_5 = partial(add, 5)
result = add_5(3)
print(result)
```

- A. 8
- B. 5
- C. 3
- D. Error
-

21. **What is a function in Python?** [#42 Python Quiz]

- A) A built-in tool that performs a specific operation.
- B) A block of code that only executes when it is called.
- C) A variable used to store data.
- D) A loop structure for repetitive tasks.

1. **Which function would you use to determine the type of a variable in Python?**
    - A) id()
    - B) type()
    - C) str()
    - D) isinstance()

Watch this video for the answer:

#10 Python supports the creation of anonymous functions at runtime, using a construct called

Python YouTube Playlist: https://www.youtube.com/playlist?list=PLKYRx0Ibk7Vi-CC7ik98qT0VKK0F7ikja

a) pi b) anonymous c) lambda d) none of the mentioned

What is the syntax for defining a lambda function in Python? A) lambda x: x + 1 B) def x(lambda): return x + 1 C) func x = lambda: x + 1 D) x lambda x + 1 Answer: A) lambda x: x + 1

Related video https://youtu.be/Z8Zeen4WwJQ

What is the output of the following code? f = lambda x, y: x * y print(f(3, 4))

related video: https://youtu.be/Z8Zeen4WwJQ

A) 12 B) 7 C) '34' D) None Answer: A) 12

Related video: https://youtu.be/Z8Zeen4WwJQ

What is the output of the following code? g = lambda x: x ** 2 print(g(5)) A) 25 B) 5 C) '5' D) None Answer: A) 25 related video https://youtu.be/Z8Zeen4WwJQ

Answer key (Mutiple Choice):

Fill in the Blanks

Answer Key (Fill in the Blanks):

Exercises

4. **Problem Statement:**

Write a function `add(*args)` that takes a variable number of arguments and returns the sum of all the arguments. The function should handle any number of arguments, including zero arguments. If no arguments are passed, the function should return `0`.

Function Signature:

```
def add(*args):
```

Input:

- The function accepts a variable number of integer arguments. These integers can be positive, negative, or zero. The number of arguments can range from 0 to any positive integer.

Output:

- The function returns an integer, which is the sum of all the arguments passed to it. If no arguments are passed, the function should return `0`.

Sample Input:

```
add(1, 2, 3)
```

Sample Output:

```
6
```

- **Watch the Solution Now ✨**

Review Questions

References and Bibliography

Appendices

Appendix A: Recursive program

- A recursive program is one that calls itself in order to solve a problem. In Python, this usually happens within a function where the function continues to call itself with a modified argument until a base condition is met.

In the example, the function `foo(x)` is a recursive function that calculates the factorial of `x`.

The code:

```
def foo(x):
    if x == 1:
        return 1
    else:
        return x * foo(x - 1)

print(foo(5))
```

Step-by-Step Explanation:

1. **Base Case**:
    - The function has a base case `if x == 1: return 1`. This stops the recursion. Without this base case, the function would keep calling itself indefinitely, leading to a "stack overflow" or "maximum recursion depth exceeded" error.
2. **Recursive Case**:
    - If `x` is not equal to `1`, the function returns `x * foo(x - 1)`. This is the recursive step, which calls `foo` again with `x - 1`.
3. **Example with `foo(5)`**: Let's break down the flow when you call `foo(5)`:
    - `foo(5)` checks if `x == 1`. Since `x = 5`, the base case is not satisfied, so the function returns `5 * foo(4)`.
    - Now, the function evaluates `foo(4)`. Again, `x == 1` is false, so the function returns `4 * foo(3)`.
    - Next, `foo(3)` is evaluated. It returns `3 * foo(2)`.
    - Then, `foo(2)` returns `2 * foo(1)`.
    - Finally, `foo(1)` hits the base case and returns `1`.

    Now, the recursive calls start to resolve from the deepest level:

    - `foo(2)` returns `2 * 1 = 2`
    - `foo(3)` returns `3 * 2 = 6`

- `foo(4)` returns `4 * 6 = 24`
- `foo(5)` returns `5 * 24 = 120`

4. **Output**: The result of `foo(5)` is `120`, which is the factorial of 5. Hence, `print(foo(5))` will output `120`.

Conclusion:

This is a classic example of recursion being used to calculate the factorial of a number. The function continues to break down the problem (finding factorial of smaller numbers) until it hits the simplest case (`x == 1`), after which it multiplies the results together to get the final answer.