Nested Functions

Nested functions, as the name suggests, are functions defined within another function. These nested functions have access to variables in the outer (enclosing) function's scope. They provide a way to encapsulate functionality that is only relevant to the outer function and help in avoiding namespace pollution.

Here's an explanation of nested functions in Python:

Syntax:

- Nested functions are defined inside the body of another function.
- They follow the same syntax as regular functions, using the `def` keyword.

```
def outer_function():
 def inner_function():
 # Function body
 pass
```

Accessing Enclosing Scope:

- Nested functions have access to variables in the outer function's scope.
- They can read and modify variables defined in the enclosing function.

```
def outer_function():
 x = 10
 def inner_function():
 print(x) # Access outer function's variable
 inner_function()

outer_function() # Output: 10
```

Encapsulation:

- Nested functions are useful for encapsulating functionality that is only relevant to the outer function.
- They help in keeping related code together and improve code organization and readability.

```
def calculate_total(prices):
    def apply_discount(price):
        return price * 0.9 #give 10% off
    total = sum(apply_discount(price) for price in prices)
    return total

result = calculate_total([10, 20, 30])
print(result)
```

Returning Nested Functions:

- Nested functions can also be returned from the outer function, allowing for dynamic function creation.

```
def outer_function(x):
 def inner_function(y):
 return x + y
 return inner_function

add_five = outer_function(5)
result = add_five(3) # Output: 8
```

Scope Resolution:

- Python follows the LEGB (Local, Enclosing, Global, Built-in) rule for variable resolution.
- Nested functions first search for variables in their local scope, then in enclosing functions' scopes, followed by the global and built-in scopes.

Nested functions are a powerful feature in Python that allow for better code organization, encapsulation of functionality, and dynamic function creation. They are commonly used in scenarios where you need to define helper functions that are only relevant to a specific part of your code.

Exercises and Answers for Nested Functions

Exercise 1:

Create a function called `outer_function` that defines a nested function called `inner_function`. The `inner_function` should print "Inner function called" when invoked. Test the `outer_function` by calling it, and then calling the `inner_function` from within the `outer_function`.

Answer 1:

```
def outer_function():
 def inner_function():
 print("Inner function called")

 print("Outer function called")
 inner_function()

# Test the function
outer_function()
```

Exercise 2:

Define a function named `calculate_total` that calculates the total price of items in a shopping cart. The function should accept a list of item prices as its argument. Inside the `calculate_total` function, define a nested function called `apply_discount` that applies a 10% discount to the total price. Call the `apply_discount` function from within `calculate_total` and return the discounted total. Test the `calculate_total` function with different lists of item prices.

Answer 2:

```
def calculate_total(items):
 def apply_discount(total):
 return total * 0.9 # 10% discount

 total_price = sum(items)
 discounted_price = apply_discount(total_price)
 return discounted_price

# Test the function
cart_items = [10, 20, 30, 40]
print("Total cost without discount:", sum(cart_items))
print("Total cost with 10% discount:", calculate_total(cart_items))
```

Exercise 3:

Write a function called `parent_function` that defines two nested functions: `child_function1` and `child_function2`. The `child_function1` should print "Child function 1 called", and `child_function2` should print "Child function 2 called". Test the `parent_function` by calling it, and then calling both `child_function1` and `child_function2` from within the `parent_function`.

Answer 3:

```
def parent_function():
 def child_function1():
 print("Child function 1 called")

 def child_function2():
 print("Child function 2 called")

 print("Parent function called")
 child_function1()
 child_function2()

# Test the function
parent_function()
```

Lambda Functions

Lambda functions, also known as anonymous functions, are a concise way of defining small, single-expression functions in Python. Unlike regular functions defined using the `def` keyword, lambda functions are defined using the `lambda` keyword. Lambda functions are particularly useful when you need to create short, throwaway functions without the need for a formal function definition.

Here's a more detailed explanation of lambda functions:

Syntax:

- Lambda functions have the following syntax: `lambda arguments: expression`
- The `lambda` keyword is followed by a comma-separated list of parameters (arguments), followed by a colon `:` and the expression that defines the function's behavior.

```
add = lambda x, y: x + y
```

Single Expression:

- Lambda functions are limited to a single expression. This means you can't include multiple statements or use control flow structures like `if`, `else`, or `for` loops within a lambda function.
- The expression's result is automatically returned by the lambda function.

Implicit Return:

- Lambda functions automatically return the result of evaluating the expression.
- There's no need to use the `return` keyword explicitly.

```
double = lambda x: x * 2
print(double(3)) # Output: 6
```

Anonymous Functions:

- Lambda functions are anonymous, meaning they don't have a name associated with them.
- They are typically used in situations where a function is needed temporarily or as an argument to higher-order functions like `map()`, `filter()`, and `sorted()`.

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
print(list(squared)) # Output: [1, 4, 9, 16, 25]
```

Usage:

- Lambda functions are often used in scenarios where defining a separate named function would be overkill or when passing a simple function as an argument to another function.

```
numbers = [1, 2, 3, 4, 5]
filtered = filter(lambda x: x % 2 == 0, numbers)
print(list(filtered)) # Output: [2, 4]
```

Lambda functions provide a compact and expressive way to define simple functions in Python. While they are limited in their capabilities compared to regular functions, lambda functions are a valuable tool for writing clean and concise code, especially in situations where brevity and simplicity are paramount.

The pros and cons of lambda functions:

Pros:

Concise Syntax: Lambda functions allow you to define functions in a single line of code, making them ideal for situations where brevity is important.
Readability: For simple operations, lambda functions can enhance code readability by keeping the function definition close to where it's used.
Avoiding Named Function Overhead: In scenarios where a function is needed only once or as an argument to higher-order functions like `map()`, `filter()`, and `sorted()`, lambda functions eliminate the need to define a separate named function.
Functional Programming: Lambda functions align well with the principles of functional programming, where functions are treated as first-class citizens and can be passed around as arguments or returned from other functions.

Cons:

Limited Expressiveness: Lambda functions are restricted to a single expression, which limits their ability to handle more complex logic compared to regular functions defined using `def`.

Lack of Name: Lambda functions are anonymous, meaning they don't have a name associated with them. This can make debugging and error messages less informative, as the function name won't appear in tracebacks.
Reduced Readability for Complex Logic: While lambda functions can improve readability for simple operations, they can make code harder to understand when used for more complex logic, especially if the lambda expression becomes too long or convoluted.
Difficulty in Testing: Lambda functions can make unit testing more challenging, as they can't be easily referred to by name. Testing code that relies heavily on lambda functions may require additional effort to maintain readability and test coverage.

Overall, lambda functions are a valuable tool in Python for writing clean and concise code, especially for simple operations and in functional programming paradigms. However, they should be used judiciously and with consideration for readability and maintainability, especially in scenarios where their limitations may outweigh their benefits.

Exercises & Answers for Lambda Functions

Exercise 1:
Create a lambda function called `double` that doubles the value of its input. Test the lambda function by calling it with different numeric values.

Answer 1:

```
double = lambda x: x * 2

# Test the lambda function
print(double(3)) # Output: 6
print(double(5)) # Output: 10
```

Exercise 2:
Write a lambda function called `is_even` that returns True if a given number is even, and False otherwise. Test the lambda function with different numeric values.

Answer 2:

```
is_even = lambda x: x % 2 == 0

# Test the lambda function
print(is_even(4)) # Output: True
print(is_even(7)) # Output: False
```

Exercise 3:
Define a lambda function called `average` that calculates the average of three numbers. Test the lambda function with different sets of numeric values.

Answer 3:

```
average = lambda x, y, z: (x + y + z) / 3

# Test the lambda function
print(average(3, 6, 9)) # Output: 6.0
print(average(10, 20, 30)) # Output: 20.0
```