Inner functions

In Python, inner functions, also known as nested functions, are functions defined inside another function. Inner functions have access to the variables of their outer (enclosing) function, allowing for more modular and organized code. They are often used to encapsulate functionality that is only relevant within a specific context. Here's an explanation with examples:

Basic Structure of Inner Functions:

```python
def outer_function(outer_parameter):
    def inner_function(inner_parameter):
        # Inner function code
        return inner_parameter + 1

    # Outer function code
    result = inner_function(outer_parameter)
    return result

# Calling the outer function
output = outer_function(5)
print(output)  # Output: 6
```

In this example, `inner_function` is an inner function defined within `outer_function`. It has access to the `outer_parameter` from its enclosing function.

Accessing Outer Function Variables:

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

    return inner_function

# Creating a closure
closure = outer_function(10)

# Calling the closure
result = closure(5)
print(result)  # Output: 15
```

Here, `inner_function` forms a closure over the variable `x` from its enclosing function `outer_function`. The closure is then assigned to the variable `closure` and called with an argument.

Use Case: Function Factories:

Inner functions are commonly used in function factories, where a function returns another function with some specific behavior.

```python
```

```
def exponent_factory(power):
    def power_function(x):
        return x ** power

    return power_function

# Creating specific power functions
square = exponent_factory(2)
cube = exponent_factory(3)

print(square(4))  # Output: 16
print(cube(3))    # Output: 27
```

Here, `exponent_factory` is a function factory that returns a specific power function based on the desired exponent.

Closure with Mutable Objects:

When using mutable objects like lists in closures, be cautious about unexpected behavior due to shared references.

```python
def outer_function():
    counter = [0]
```

```
    def inner_function():
        counter[0] += 1
        return counter[0]

    return inner_function

# Creating a closure with a mutable object
counter_closure = outer_function()

# Calling the closure
print(counter_closure())  # Output: 1
print(counter_closure())  # Output: 2
```

In this example, the inner function `inner_function` modifies the contents of the list `counter` even though it is defined outside of it. This is because the list is a mutable object, and the closure shares a reference to it.

Inner functions provide a way to structure code by encapsulating functionality within a specific context. They are particularly useful when you want to create closures or function factories. However, it's important to be aware of scoping and potential issues with mutable objects in closures.