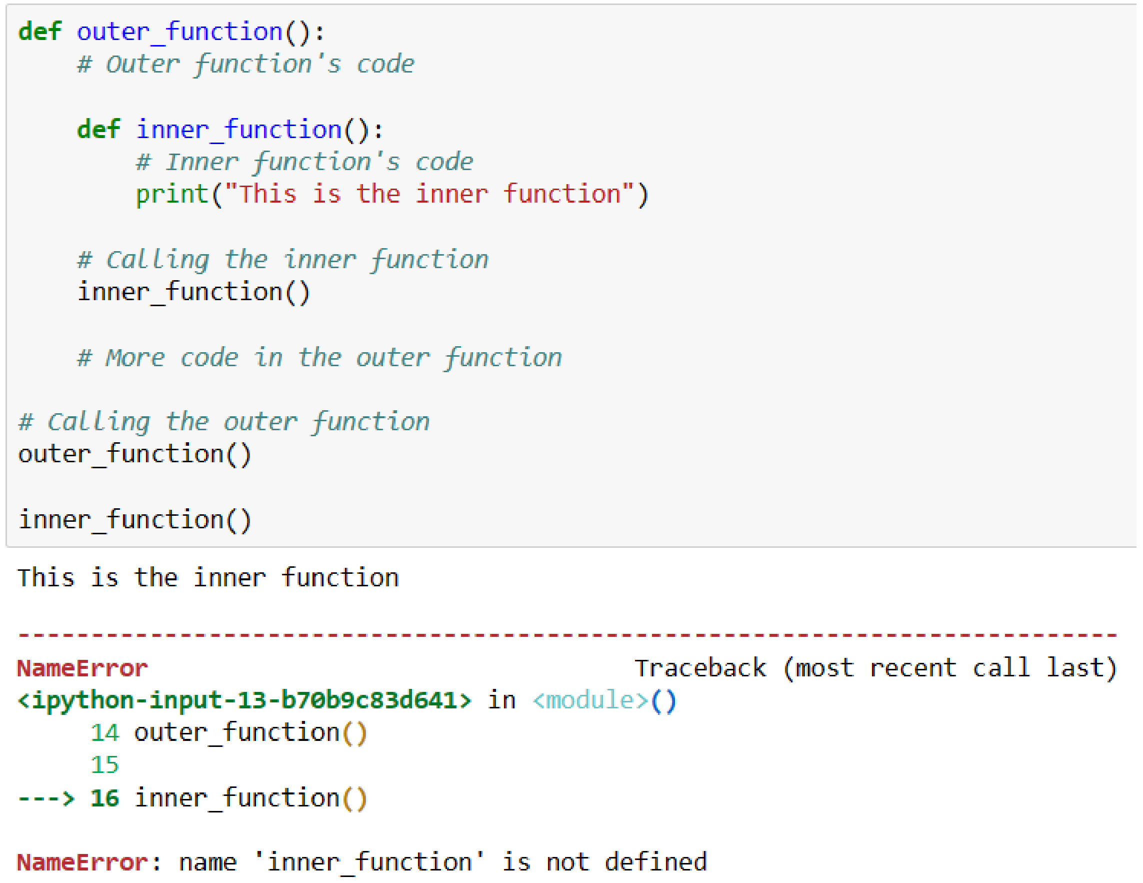
Nested Functions, Closure, Generators

Anab Batool Kazmi

Nested Functions

- A nested function is a function that is defined inside of another function.
- These inner functions are also known as "nested functions" or "inner functions."
- The inner function can access variables and parameters of the outer function.
- Nested functions can be used to encapsulate functionality and to create more modular code.
- You can call the inner function from within the outer function, but you can’t call outside of the outer function

Nested Functions

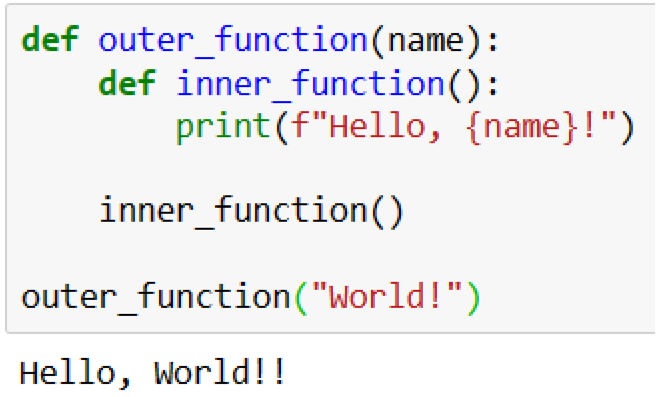


```
def outer_function():
    # Outer function's code

    def inner_function():
        # Inner function's code
        print("This is the inner function")

    # Calling the inner function
    inner_function()

    # More code in the outer function

# Calling the outer function
outer_function()

inner_function()
```

```
This is the inner function

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-b70b9c83d641> in <module>()
     14 outer_function()
     15
---> 16 inner_function()

NameError: name 'inner_function' is not defined
```

Nested Functions

The inner function can access variables and parameters of the outer function.

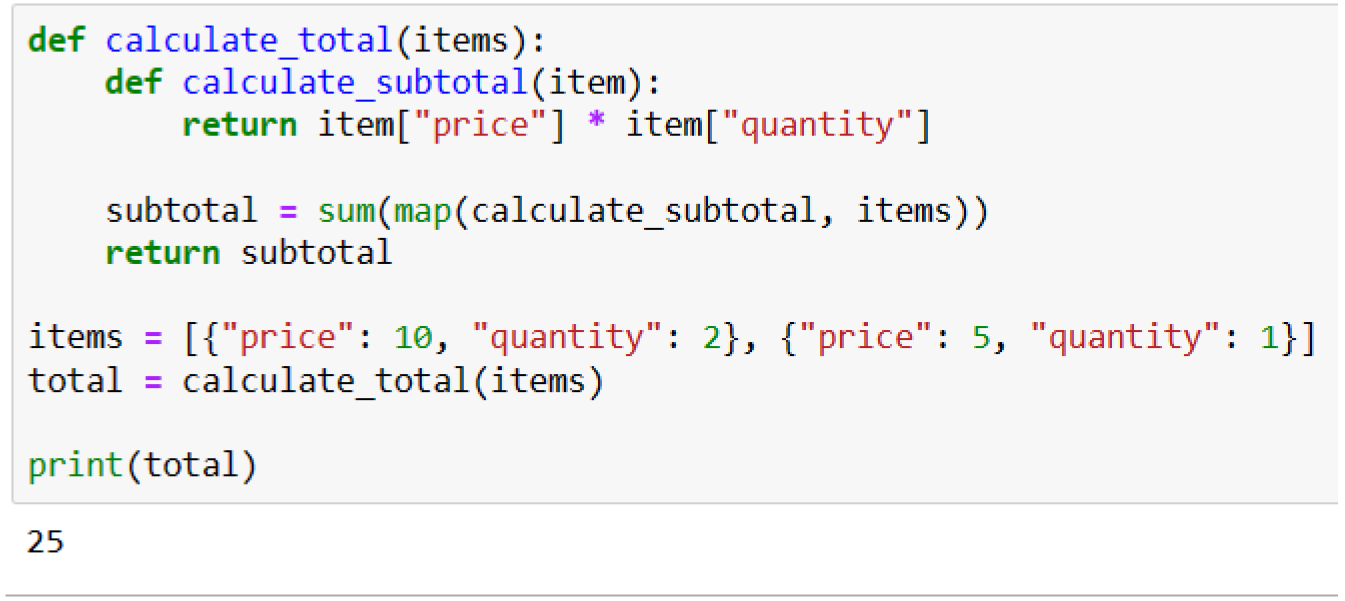


```
def outer_function(name):
    def inner_function():
        print(f"Hello, {name}!")

    inner_function()

outer_function("World!")
```

```
Hello, World!!
```

Nested Functions

Nested functions can be very useful for encapsulating functionality and creating more modular code. For example, you could use a nested function to implement a helper function that is only needed by a specific outer function.



```python
def calculate_total(items):
    def calculate_subtotal(item):
        return item["price"] * item["quantity"]

    subtotal = sum(map(calculate_subtotal, items))
    return subtotal

items = [{"price": 10, "quantity": 2}, {"price": 5, "quantity": 1}]
total = calculate_total(items)

print(total)
```

```
25
```

Generator in Python

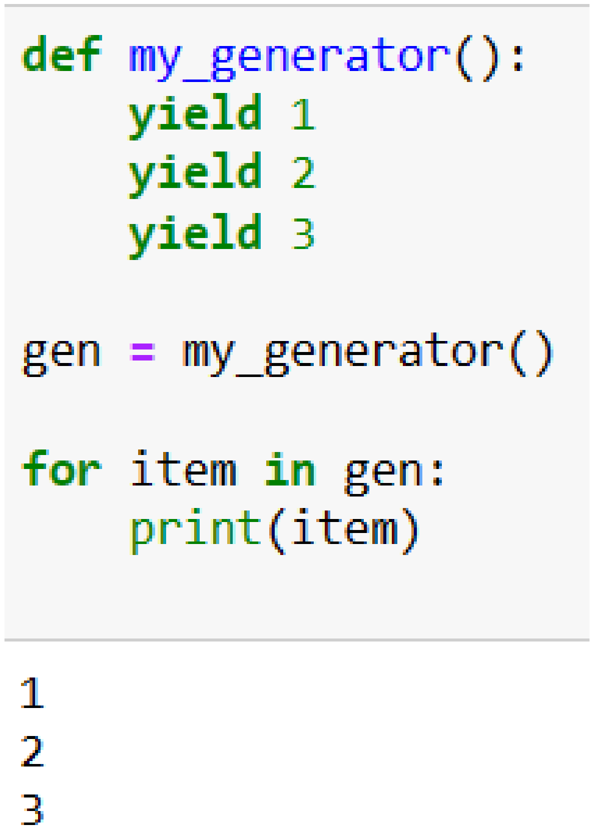
- A generator in Python is a function **that returns an iterator**.
- An iterator is an object that can be iterated over, meaning that you can loop through its values.
- Generators are useful for creating iterators over large or infinite sequences of data, because they can produce the data on demand instead of having to store it all in memory at once.
- To create a generator, you use the **yield keyword** instead of the return keyword in your function.
- This tells Python to **return the current value of the function and then pause execution.** When the iterator is called again, Python will resume execution from the point where it paused.

Generator in Python

Generators can be used in a variety of ways, such as:

- Creating iterators over large or infinite sequences of data.
- Implementing lazy evaluation.
- Creating pipelines of data processing operations.

Generator in Python

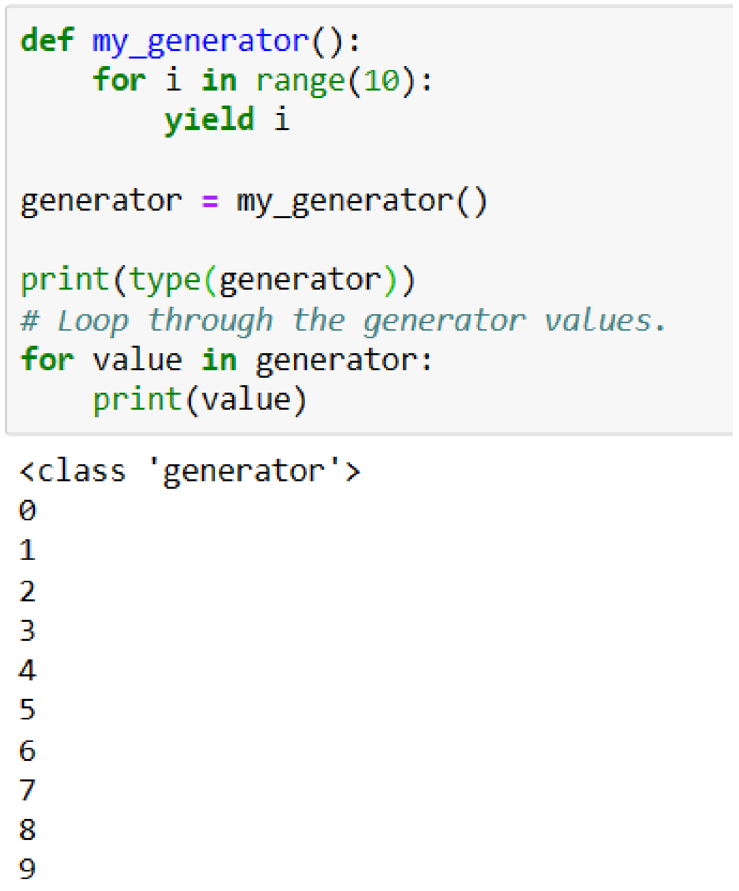


```python
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

for item in gen:
    print(item)
```

```
1
2
3
```

- **yield Statement**: The yield statement is used to produce a value from the generator and pause the generator's execution until the next value is requested.
- **Lazy Evaluation:** Generators use lazy evaluation, meaning they generate values on-the-fly as you request them, rather than generating all values at once.

Generator in Python

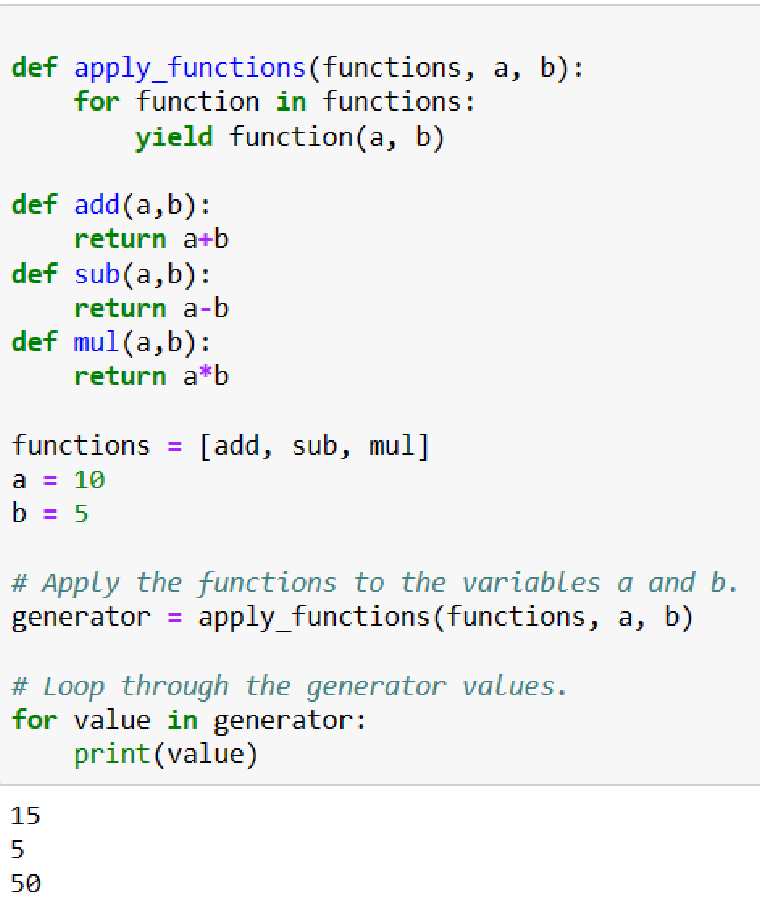


```
def my_generator():
    for i in range(10):
        yield i

generator = my_generator()

print(type(generator))
# Loop through the generator values.
for value in generator:
    print(value)
```

```
<class 'generator'>
0
1
2
3
4
5
6
7
8
9
```

- **State Preservation:** Generator functions maintain their state between calls, allowing them to remember where they left off when you iterate over them.
- **Memory Efficiency:** Generators are memory-efficient because they don't store the entire sequence in memory, which is beneficial for large datasets.

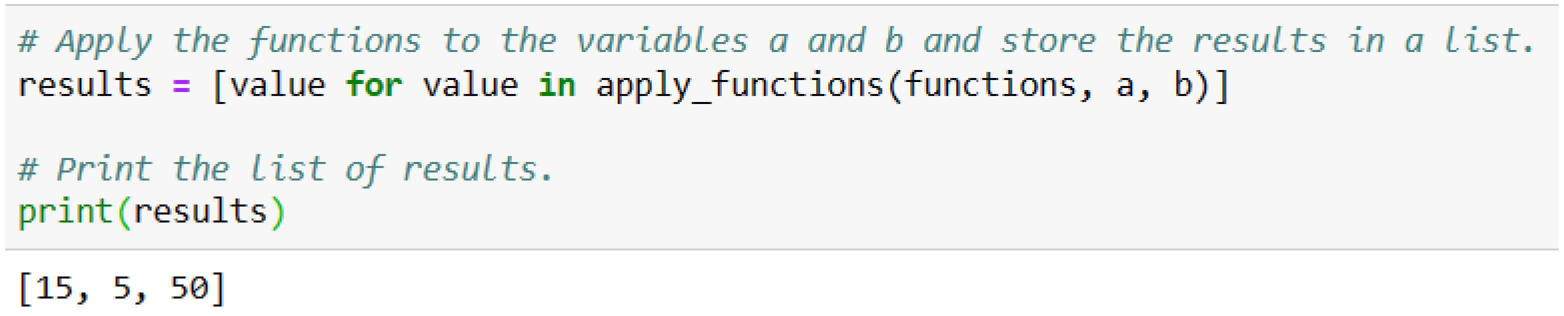
Generator in Python



```
def apply_functions(functions, a, b):
    for function in functions:
        yield function(a, b)

def add(a,b):
    return a+b
def sub(a,b):
    return a-b
def mul(a,b):
    return a*b

functions = [add, sub, mul]
a = 10
b = 5

# Apply the functions to the variables a and b.
generator = apply_functions(functions, a, b)

# Loop through the generator values.
for value in generator:
    print(value)
```

```
15
5
50
```

The generator expression is the same as comprehension, evaluated lazily, meaning that Python only evaluates the expression when it is needed. This can be useful for avoiding unnecessary computation.



```
# Apply the functions to the variables a and b and store the results in a list.
results = [value for value in apply_functions(functions, a, b)]

# Print the list of results.
print(results)
```

```
[15, 5, 50]
```