Functions in Python: A deep dive

1. What is a Function and Why Use a Function?

A **function** in Python is a reusable block of code that performs a specific task. Functions help break down large problems into smaller, manageable parts and enhance code modularity and reusability. They reduce redundancy, make code more readable, and facilitate debugging.

Why Use Functions?

- **Code Reusability:** Functions allow you to write code once and use it multiple times.
- **Modularity:** Helps divide the code into logical sections.
- **Improves Maintainability:** Changes need to be made in one place only.
- **Enhances Debugging:** Smaller functions make it easier to isolate and fix bugs.

Example: Basic Function

```
#define the function
def greet(name):
  """This function greets the person passed in as a parameter."""
  print(f"Hello, {name}!")
```

```
#call the function
x = greet("World")
print(x)
```

```
Hello, World!
None
```

2. The Anatomy of a Function Definition

A Python function is defined using the `def` keyword followed by the function name, parentheses, and a colon. Inside the parentheses, you can define parameters. The body of the function contains the code that executes when the function is called.

Function Components:

1. **Function Name**: Descriptive name for the function.
2. **Parameters**: Input variables passed to the function.
3. **Colon (:)**: Indicates the start of the function body.
4. **Docstring (Optional)**: Describes the purpose and usage of the function.
5. **Function Body**: The set of statements that perform the function's tasks.

6. **Return Statement (Optional)**: Sends a result back to the caller.

Example:

```
def calculate_area(length, width):
    """Returns the area of a rectangle."""
    area = length * width
    return area

x = calculate_area(10,20)
print(x)  # Output: 50
```

```
200
```

3. Scope in Functions (Local, Global, and `nonlocal`)

Scope refers to the region where a variable can be accessed. In Python, variables inside functions have their own local scope, while variables defined outside functions have a global scope.

Types of Scope:

1. **Local Scope**: Variables declared inside a function are accessible only within that function.
2. **Global Scope**: Variables declared outside any function are accessible throughout the program.
3. **`nonlocal` Scope**: Refers to variables in the nearest enclosing scope (excluding global).

Example 1: Local and Global Scope

```
x = 10  # Global variable

def show_value():
    x = 15
    print(f"Local x: {x}")

show_value()  # Output: Local x: 5
print(f"Global x: {x}")  # Output: Global x: 10
```

```
Local x: 15
Global x: 10
```

Example 2: Using global Keyword

```
y = 20

def modify_global():

    y = 15

modify_global()
```

```
print(f"Modified global y: {y}")  # Output: Modified global y: 15
```

```
Modified global y: 20
```

Example 3: Using nonlocal Keyword

```
def outer_function():
    z = 100

    def inner_function():
        nonlocal z
        z = 200

    inner_function()
    print(f"Value of z after modification: {z}")

outer_function()  # Output: Value of z after modification: 200
```

4. Docstrings and `__doc__`

A **docstring** provides documentation for a function and appears as the first statement in the function body. Docstrings follow a specific convention:

Structure of a Docstring:

1. **Single-line Docstring:** One sentence describing the function's purpose.
2. **Multi-line Docstring:** Detailed information, including description, arguments, and return values.

Example:

```
def describe():
    """
    Prints a description message.

    This function demonstrates the use of a single-line and multi-line docstring format.

    Returns:
        None
    """
    print("This is a sample description.")

print(help(describe))
print(describe.__doc__)  # Output: Prints a description message.
```

```
Help on function describe in module __main__:

describe()
    Prints a description message.
```

```
        This function demonstrates the use of a single-line and multi-line docstring format.

        Returns:
            None

    None

        Prints a description message.

        This function demonstrates the use of a single-line and multi-line docstring format.

        Returns:
            None
```

Expanded Example:

```
def find_average(a: float, b: float, c: float) -> float:
    """
    Calculates the average of three numbers.

    Args:
        a (float): The first number.
        b (float): The second number.
        c (float): The third number.

    Returns:
        float: The average of the three numbers.
    """
    return (a+b+c)/3

#print(find_average.__doc__)  # Output: Multi-line docstring with argument details.

print(find_average(1,2,3))  # Output:
```

```
2.0
```

5. Functions as First-Class Objects

In Python, functions are **first-class objects**, meaning they can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned from other functions

Example 1: Assigning a Function to a Variable

```
def multiply(x, y):
    return x * y

# Assign function to a variable
```

```
operation = multiply
print(operation(5, 6))  # Output: 30
```

```
30
```

Example 2: Passing Function as an Argument

```
def execute_operation(func, a, b):
    return func(a, b)

print(execute_operation(multiply, 2, 3))  # Output: 6
```

```
6
```

Example 3: Returning a Function from Another Function

```
def power_function(exponent):
    def power(base):
        return base ** exponent
    return power



square = power_function(2)
print(type(square))
print(square(4))  # Output: 16
```

```
<class 'function'>
16
```

Example 4: functions as elements in a collection

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

operations = {
    "add": add,
    "subtract": subtract
}

print(operations["add"](10, 5))
print(operations["subtract"](10, 5))
```

6. What are Arguments and Parameters?

- **Parameters**: Variables listed in the function definition.
- **Arguments**: Values passed to the function during the call.

Example:

```
def greet_user(username):  # username is a parameter
    print(f"Hello, {username}!")

# "Alice" is an argument
greet_user("Alice")  # Output: Hello, Alice!
```

Expanded Examples:

```
def calculate_sum(a, b, c):
    return a + b + c

print(calculate_sum(1, 2, 3))  # Output: 6
```

7. `return` Statement

The `return` statement sends a value back to the caller. If no `return` is used, the function returns `None`.

Example:

```
def get_square(num):
    return num ** 2

result = get_square(5)
print(result)  # Output: 25
```

```
25
```

Returning Multiple Values:

```
def get_coordinates():
    x, y = 10, 20
    return x, y

x_coord, y_coord = get_coordinates()
print(f"X: {x_coord}, Y: {y_coord}")  # Output: X: 10, Y: 20
```

8. Optional Parameters (Defaults)

Functions can have default parameter values, which are used if arguments are not provided.

Example 1: Greeting Function with Default

```
def greet(name, greeting="Hi"):
    print(f"{greeting}, {name}!")

greet("Alice")  # Output: Hi, Alice!
greet("Bob", "Hello")  # Output: Hello, Bob!
```

```
Hi, Alice!
Hello, Bob!
```

Example 2: Computing the Total Price with Optional Discount

```
def total_price(price, quantity=1, discount=5):
    """
    Calculates the total price after applying a discount.

    Args:
        price (float): Price of a single item.
        quantity (int, optional): Number of items. Defaults to 1.
        discount (float, optional): Discount to apply (percentage as decimal). Defaults to 0.0.

    Returns:
        float: The total cost after discount.
    """
    subtotal = price * quantity
    discounted_price = subtotal * (1 - discount)
    return discounted_price

print(total_price(price= 100))  # Output: 100.0
print(total_price(price=100, discount=3))  # Output: 300.0
print(total_price(100, 3, 0.1))  # Output: 270.0 (10% discount)
```

```
-400
-200
270.0
```

9. Required Positional Arguments (`/`)

The `/` in a function definition indicates that arguments before it must be passed positionally. This ensures that the function is called with arguments in the correct order and not by keyword.

Benefits of Positional-Only Arguments:

- Prevents accidental misuse of argument names.
- Enforces a strict function call pattern.

Example 1: Simple Positional-Only Function

```
def add(a, b, / ):
    return a + b

print(add(3, 5))  # Output: 8
#print(add(a=3, b=5))  # Error: a and b must be passed positionally.
```

```
8
```

Example 2: Mixing Positional-Only and Keyword Arguments

```
def describe(a, b, /, c, d):
    return f"{a}, {b}, {c}, and {d}"

print(describe(1, 2, c="Hello", d="World"))  # Output: "1, 2, Hello, and World"
```

```
1, 2, Hello, and World
```

10. `*args` and `**kwargs`

- `args` : Collects extra positional arguments and stores them as a tuple.
- `*kwargs` : Collects extra keyword arguments and stores them as a dictionary.

Example 1: Basic Use of `*args` and **`kwargs`

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

display_info(1, 2, 3, name="John", age=25)
```

Example 2: Passing Tuples/Lists and Dicts Directly

```
def show_data(a, b, c):
    print(f"a: {a}, b: {b}, c: {c}")

args = (1, 2, 3)
kwargs = {"a": 10, "b": 20, "c": 30}

show_data(*args)  # Output: a: 1, b: 2, c: 3
show_data(**kwargs)  # Output: a: 10, b: 20, c: 30
```

11. Lambda Functions

A **lambda function** is an anonymous function defined with the `lambda` keyword. These are useful for short, single-use operations.

Structure of a Lambda Function:

```
lambda arguments: expression
```

Examples:

1. **Doubling a Number:**

```
double = lambda x: x * 2
print(double(4))  # Output: 8
```

```
8
```

2. **Sorting Example:**

```
def my_sort(x):
  return x[1]

items = [("apple", 3), ("banana", 1), ("cherry", 2)]
items.sort(key=lambda x: x[1])
#items.sort(key=my_sort)
print(items)  # Output: [('banana', 1), ('cherry', 2), ('apple', 3)]
```

```
[('banana', 1), ('cherry', 2), ('apple', 3)]
```

Where Lambda Functions Should Be Used:

- **Short, simple operations:** When the function is small and doesn't require naming.

- **Key functions for sorting, filtering, and mapping:** Lambda functions are often passed to `sorted()`, `filter()`, and `map()`.
- **Inline and throwaway use cases:** When you don't want to define a full function for quick use.

Example: Filtering even numbers using `filter()`

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

Where Lambda Functions Should **NOT** Be Used:

- **Complex logic:** If the logic requires more than one line or is difficult to read, use `def` to define a named function.
- **When documentation is needed:** Lambda functions cannot include docstrings.
- **When the function is reused multiple times:** Instead of rewriting the lambda, define a named function to improve readability and maintainability.

Counterexample:

```
# Avoid using lambda for multi-step operations
# Instead of:
multi_step = lambda x: (x + 2) * (x - 1) // 2

# Use a named function:
def multi_step_fn(x):
    y = x + 2
    z = y * (x - 1)
    result = z // 2
    return result
```

Type hints indicate the expected types of arguments and return values.

Example:

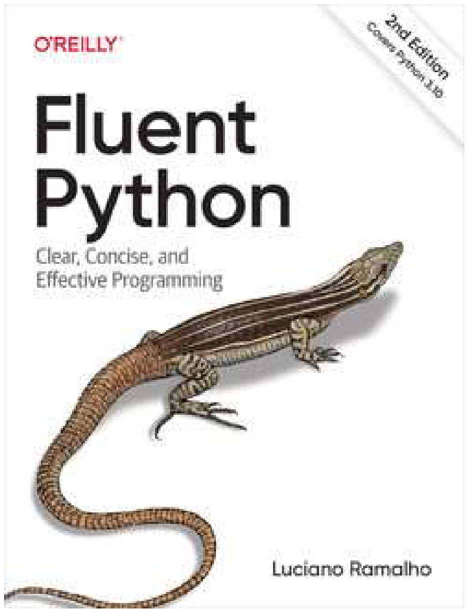
12. Type Hints

```
from typing import List

def add(a: int, b: int) -> int:
    return a + b

def process_list(values: List[int]) -> List[int]:
    return [v * 2 for v in values]

print(add(3, 4))  # Output: 7
print(process_list([1, 2, 3]))  # Output: [2, 4, 6]
```

```
7
[2, 4, 6]
```

By using functions effectively, you can write clear, maintainable, and reusable code. Advanced features like `*args`, `**kwargs`, `lambda` functions, and type hints make Python functions powerful tools for any project.

Book Recommendation to dig deep into Python:



Python For Loops: A Comprehensive Guide

1. **What is a For loop**

A `for` loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, string, or range) and perform a block of code for each element in the sequence. It allows you to automate repetitive tasks without manually writing the same code multiple times.

Key Reasons to Use `for` Loops:

- To process elements in a collection sequentially.
- To avoid redundancy by automating iterations.
- To handle dynamic-sized data structures gracefully.
- To make code concise and maintainable.

2. **The Anatomy of a For Loop**

The structure of a `for` loop in Python is simple:

```
for variable in iterable:
    # Code block to execute for each item
    print(variable)
```

- **`variable`**: A temporary placeholder that stores the current element of the iterable.
- **`in`**: A keyword that signifies iteration over the iterable.
- **`iterable`**: The collection (e.g., list, tuple, range, or string) you want to iterate over.
- **Code Block**: The indented code that runs during each iteration.

Example:

```
name = "Python"
for char in name:
    print(char)
```

```
P
y
t
h
```

```
o
n
```

Example Use Case: Suppose you have a list of grades for students and want to calculate the average grade.

```
grades = [85, 90, 78, 92, 88]

# Calculate the average grade
sum_of_grades = 0
for grade in grades:
    sum_of_grades = sum_of_grades +  grade

average = sum_of_grades / len(grades)
print(f"Average grade: {average}")
```

3. **Loop Control Statements**

Loop control statements modify the flow of the loop. Python provides three main control statements:

- **`break`**: Exits the loop prematurely when a condition is met.
- **`continue`**: Skips the current iteration and moves to the next.
- **`else`**: Executes after the loop finishes, but only if `break` is not encountered.

Example with `break`:

```
for num in [1,2,3,4,5,6,7]:
    if num == 5:
        break  # Stops when num equals 5
    print(num)
```

```
1
2
3
4
```

Example with `continue`:

```
for num in [0,1,2,3,4,5]:
    if num == 2:
        continue  # Skips the iteration when num equals 2
    print(num)
```

Example with `else`:

```
for num in [0,1,2,3,4,5]:
    print(num)
    if num == 2:
      break
else:
    print("Loop completed without a break.")
```

```
0
1
2
```

4. The `range()` Function

The `range()` function generates a sequence of numbers, often used with `for` loops to control iteration.

Syntax:

```
range(start, stop, step)
```

- **`start`**: The starting number (inclusive). Defaults to 0.
- **`stop`**: The stopping number (exclusive).
- **`step`**: The increment (defaults to 1).

Example 1: Iterate from 0 to 4:

```
for i in range(5):
    print(i)
```

Example 2: Iterate from 2 to 10 with step 2:

```
for i in range(2, 11, 2):
    print(i)
```

Example 3: Countdown using negative steps:

```
for i in range(10, 0, -2):
    print(i)
```

5. **Looping Through Collections (Lists and Tuples)**

Lists and tuples are among the most common data structures used with `for` loops.

List Example:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(f"Fruit: {fruit}")
```

Tuple Example:

```
coordinates = (10, 20, 30)
for coord in coordinates:
    print(f"Coordinate: {coord}")
```

Nested Loops with Lists of Tuples:

```
points = [(1, 2), (3, 4), (5, 6)]
for x, y in points:
    print(f"Point: ({x}, {y})")
```

```
Point: (1, 2)
Point: (3, 4)
Point: (5, 6)
```

6. **Looping Through Sets**

A `set` is an unordered collection of unique elements. You can iterate over sets just like lists and tuples:

Set Example:

```
unique_numbers = {1, 2, 3, 4, 5}
for num in unique_numbers:
    print(num)
```

Since sets are unordered, the output order may vary.

7. **Looping Through Dictionaries (Keys, Values, and Items)**

Dictionaries store key-value pairs, and `for` loops allow you to access keys, values, or both.

Iterating Through Keys:

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
for key in person:
    print(f"Key: {key}")
```

Iterating Through Values:

```
for value in person.values():
    print(f"Value: {value}")
```

Iterating Through Key-Value Pairs:

```
for key, value in person.items():
    print(f"{key}: {value}")
```

8. **List Comprehensions**

List comprehensions provide a concise way to create lists using `for` loops.

Syntax:

```
[expression for item in iterable if condition]
```

Example: Create a list of squares of even numbers from 1 to 10:

```
squares = [x ** 2 for x in range(1, 11) if x % 2 == 0]
print(squares)  # Outputs: [4, 16, 36, 64, 100]
```

List comprehensions make the code shorter and more readable.

9. **Using `enumerate()`**

The `enumerate()` function adds a counter to an iterable and returns it as an `enumerate` object.

Example:

```
languages = ['Python', 'JavaScript', 'C++']
for index, language in enumerate(languages, start=1):
    print(f"{index}: {language}")
```

```
1: Python
2: JavaScript
3: C++
```

10. **Nested For Loops**

You can use `for` loops inside each other for operations like matrix traversal.

Example:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for item in row:
        print(item, end=' ')
```

Nested loops can be useful for tasks like comparing all pairs in a collection or working with multidimensional data.

11. **For Loop with `iter()` and `next()`**

The `iter()` function returns an iterator object, and the `next()` function retrieves the next item from the iterator.

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruit_iterator = iter(fruits)

print(next(fruit_iterator))  # Outputs: apple
print(next(fruit_iterator))  # Outputs: banana
print(next(fruit_iterator))  # Outputs: cherry
fruit_iterator= iter(fruits)
print(next(fruit_iterator))  # Outputs: cherry
```

```
apple
banana
cherry
apple
```

12. **Generators and Defining a Generator Function**

Generators yield items one at a time using the `yield` keyword. They are memory-efficient when dealing with large datasets.

Example:

```
def fibonacci_sequence(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for num in fibonacci_sequence(5):
    print(num)  # Outputs: 0, 1, 1, 2, 3
```

For loops in Python provide an elegant and powerful way to traverse collections, whether they are lists, tuples, dictionaries, or custom iterables. With the help of functions like range(), enumerate(), and iter(), you can perform complex iterations with minimal code. Additionally, understanding how generators work and using comprehensions allows you to write efficient, readable, and memory-friendly code for a wide range of tasks.

Part A Collab Link:
https://colab.research.google.com/drive/1C3z6r3Yy76xZTUiJJZh_KENN1NlCoBdM?usp=sharing

Part B Collab Link:
https://colab.research.google.com/drive/1DoZl7Kr7eWFZR9PhHPpem4yRGy018iXc?usp=sharing