Introduction to Programming and Problem Solving

Week 7: Functions

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

Recap of Chapter 6

- In our last week lecture, we delved into the realm of loops in Python, a pivotal concept for automating repetitive tasks and efficiently processing data.
- Types of Loops Covered:
- for Loops:
- for loops were introduced as a means to iterate over sequences such as lists and strings.
- They empower the execution of specific actions for each item within the sequence.

Recap of Chapter 6+

- while Loops: while loops were explored as a tool for code repetition based on a specified condition.
- Strategies to prevent infinite loops were discussed.
- Loop Control Statements: We introduced loop control statements like break and continue to modify loop behavior and enhance control over program flow.

Recap of Chapter 6++

- Advanced Techniques:
- Nested Loops:
- The concept of nested loops, where one loop is enclosed within another, was elucidated.
- Nested loops were showcased for handling intricate repetitive tasks and multi-dimensional data structures.
- Looping with Range:

Recap of Chapter 6+++

- We examined the utility of the range() function, which simplifies the management of loop iterations by generating sequences of numbers.
- In Conclusion:
- Mastery of loops is indispensable for any Python programmer, as loops constitute the foundation of diverse programming tasks, spanning from data processing to algorithm implementation.

Recap of Chapter 6++++

- Loops are pivotal for automation, efficient data processing, and effective problem-solving, establishing them as a cornerstone of Python programming.
- Equipped with this knowledge, students are well-prepared to tackle a wide range of programming challenges and develop sophisticated Python applications.

Content

- Introduction to functions
- Types of functions
- Parameters and arguments
- Types of arguments
- Uses of functions
- Local and global arguments
- Lambda function
- Recursive function

Introduction to Function

- A function is a self-contained block of one or more statements that performs a special task when called[1]
- A function in python is a reusable block of code that performs a specific task or a set of related tasks
- **Defining a Function:** To define a function in Python, you use the **def** keyword, followed by the function name, a pair of parentheses, and a colon.
- The function name should follow Python's variable naming rules.
- The parentheses can contain zero or more input parameters, known as parameters, that the function may accept

Types of Functions

- Functions can be categorized into several types based on their characteristics and purposes.
- The are three major common types of functions which are;
- Built-in Functions: Python provides a wide range of built-in functions that are available without the need to define them. Examples include print(), len(), type(), max(), min(), sum(), and many others.
- User-Defined Functions: These are functions created by the programmer to perform specific tasks. User-defined functions are defined using the def keyword. They can take parameters and return values. Examples are functions created by the programmer to perform custom operations.
- This lecture will focused on the user designed function

Types of Functions+

- Anonymous Functions (Lambda Functions): Lambda functions are small, unnamed functions defined using the lambda keyword. They are typically used for simple operations and are often employed within other functions like map(), filter(), and reduce(). Lambda functions have no function name.
- Others are Recursive Function, high order functions, generators, decorators and closures
- Details of some of these functions will be discussed in the following slides

Basics of Function/Syntax

- **Function Syntax:** In Python, you define a function using the **def** keyword followed by the function name, parentheses for parameters (if any), a colon, and an indented block for the function's body. Here's the general syntax:

```
1.def function_name(parameters):
2.    #Function body
3.    #Code to perform the desired task
4.    #return result
```

Basics of Function/Syntax +

- Now, let's delve into the essential elements and their explanations:
- **def**: The **def** keyword is used to declare that you are defining a function.
- **function_name**: This is the name you give to your function. It should follow Python's variable naming conventions.
- **Parameters**: Inside the parentheses, you can specify zero or more parameters (also called arguments) that the function can accept. Parameters are placeholders for values that will be passed when you call the function.

Basics of Function/Syntax ++

- **Colon (:)**: The colon signifies the start of the function's body and is a required part of the syntax.
- **Function Body**: The indented block following the colon contains the code that specifies what the function does. This is where you write the actual instructions or computations.
- **return (Optional)**: If your function is designed to produce a result, you can use the **return** statement to send that result back to the caller. This statement is optional, and you can have functions that don't return any value.

Function example

- Here's an example of a simple function that calculates the square of a number:

```
1.def square(num):
2.      result = num * num
3.       return result
```

- In this example, the function **square** takes one parameter, **num**, and returns the square of that number.

Calling a Function

- To use a function, you simply call it by its name and provide the required arguments:

```
1. result = square(5)
2. print(result) # This will print 25
```

- In this code, **square(5)** is a function call that returns the square of 5, which is then assigned to the variable **result**.

Uses of a Function

- Functions are a fundamental building block in programming, and they serve various essential purposes in any programming language, including Python.

Common use cases and benefits of using functions:

- **Modularity**: Functions allow you to break your code into smaller, more manageable pieces. Each function can focus on a specific task or operation. This promotes code modularity and makes it easier to understand, test, and maintain your codebase.

Uses of a Function+

```
1. # Define modular functions to perform specific tasks
2. # Function to calculate the square of a number
3. def square(num):
4.     return num ** 2
5. # Function to calculate the cube of a number
6. def cube(num):
7.     return num ** 3
8. # Function to calculate the sum of two numbers
9. def add(a, b):
10.    return a + b
```

Uses of a Function++

```
11.# Function to display a result with a message
12.def display_result(message, result):# Main program8
13.    print(message, result)
14.# Calculate the square and cube of a number
15.num = 5
16.square_result = square(num)
17.cube_result = cube(num)
```

Uses of a Function+++

```
18.# Calculate the sum and difference of two numbers
19.num1 = 10
20.num2 = 3
21.sum_result = add(num1, num2)
22.# Display the results
23.display_result(f"Square of{num} is ", square_result)
24.display_result(f"Cube of {num} is ", cube_result)
25.display_result(f"The sum of {num1} and {num2} is ",
   sum_result)
```

Uses of a Function++++

- **Reusability**: You can reuse functions throughout your code. Once you've defined a function to perform a particular task, you can call it whenever needed without rewriting the same code. This reduces redundancy and saves time.

- Let us say that we have been asked to find the products of the ranges 1-11, 24-37 and 52-65, if we use a loop to implement this, we will have to write three different loops as seen below.

```
1.pd=0
2.for p in range(1, 11):
3.    pd*=p
4.print(pd)
```

```
1.pd2=0
2.for y in range(52,65):
3.    pd2*=y
4.print(pd2)
```

```
1. pd1=0
2. for x in range(24, 37):
3.     pd1*=x
4. print(pd1)
```

Uses of a Function+++++

Now let us implement this using function

```
1.def rang_sum(x, y):
2.     sm=0
3.     for p in range(x, y):
4.         sm+=p
5.     print("Results from a function")
6.     print(sm)
7.rang_sum(1, 11)
8.rang_sum(24, 37)
9.rang_sum(52, 65)
```

Uses of a Function++++++

- **Abstraction**: Functions can abstract complex operations. You can create functions with clear and meaningful names that hide the underlying complexity, making your code more readable and accessible.
- **Parameterization**: Functions can accept parameters, allowing you to customize their behavior by passing different values. This parameterization makes your code more flexible and adaptable.
- **Encapsulation**: Functions can encapsulate a series of related operations into a single unit. For example, a function to calculate the total price of items in a shopping cart can encapsulate the logic for adding up all the prices.

Uses of a Function+++++++

- **Readability**: Well-named functions enhance code readability. They act as self-documenting units that describe what a particular part of your code does.
- **Organization**: Functions help organize your code logically. You can group related code together in functions, making it easier to locate and understand specific parts of your program.
- **Testing**: Functions can be tested in isolation, which simplifies the debugging process. By testing individual functions, you can identify and fix issues more easily.

.

Parameters and Arguments in a Function

- In the context of functions, the terms "parameters" and "arguments" refer to the values that are passed into and received by a function. However, they have distinct meanings

Parameters:

- Parameters are the variables or placeholders listed in the function definition.
- They act as local variables within the function and represent the values that the function expects to receive when it's called.
- Parameters are defined in the function's header inside the parentheses.

Parameters+

- Parameters are essential for defining the function's signature, which specifies the number and order of values the function accepts.
- They have local scope within the function, meaning they are only accessible within the function.
- Example:

```
1.  def greet(name, age):
2.      print(f"Hello, {name}! You are {age} years old.")
```

- In this example, **name** and **age** are parameters of the **greet** function.

Arguments

- Arguments are the actual values or expressions provided when calling a function.
- They are passed to the function to be used in place of the parameters defined in the function's header.
- Arguments can be of different data types (e.g., numbers, strings, lists) and can be expressions or variables.
- The order of the arguments should match the order of the parameters in the function definition.

Arguments+

- The number of arguments should match the number of parameters unless default values are provided for some parameters.
- Example:
- greet("Alice", 30)
- In this function call, "Alice" and 30 are arguments provided for the **name** and **age** parameters.

Relationship between parameters and arguments in a function:

- Parameters are defined in the function's signature and act as placeholders for values the function expects.
- Arguments are actual values or expressions provided when calling the function.
- When you call a function, the arguments are assigned to the parameters in the order they appear in the function's definition.
- The number of arguments must match the number of parameters, unless default values are provided for some parameters.
- The values of the arguments are accessible as local variables within the function using the parameter names.

Types of Arguments

- You can pass arguments when calling a function. Arguments are values that you provide to a function to perform specific tasks or computations.
- There are different types of arguments that be use in Python functions
- **Positional Arguments:** Positional arguments are the most common type of arguments in Python functions. They are matched to function parameters based on their position or order. The first argument passed corresponds to the first parameter, the second argument to the second parameter, and so on.

```
1. def greet(name, message):
2.     print(f"Hello, {name}! {message}")
3. greet("Alice", "Good morning!")
```

- # "Alice" matches with "name," and "Good morning!" matches with "message."

Keyword Arguments

- Keyword arguments are passed to a function using parameter names.
- This allows you to pass arguments out of order, making the code more readable and self-explanatory.
- You specify the parameter name, followed by the value.

```
1.  def greet(name, message):
2.      print(f"Hello, {name}! {message}")
3.  greet(message="Good morning!", name="Alice")
```

Default Arguments

Default arguments have predefined values, which are used if an argument is not explicitly provided when calling the function. You specify default values in the function definition.

```
def greet(name, message="Hello!"):
    print(f"{message} {name}")
greet("Bob")  # "Hello! Bob" because "message" has a default value
greet("Alice", "Good morning!")  # "Good morning! Alice"
```

Variable-Length Positional Arguments (args)

- You can use the *args syntax to allow a function to accept a variable number of positional arguments. These arguments are collected into a tuple.

```
1. def add(*args):
2.     result = 0
3.     for num in args:
4.         result += num
5.     return result
6. sum_result = add(1, 2, 3, 4)
# sum_result will be 10
```

Variable-Length Keyword Arguments (kwargs)

- Similar to *args, you can use **kwargs to allow a function to accept a variable number of keyword arguments.
- These arguments are collected into a dictionary.

```
1.  def print_info(**kwargs):
2.      for key, value in kwargs.items():
3.          print(f"{key}: {value}")
4.  print_info(name="Alice", age=30, city="New York")
```

Output:

name: Alice

age: 30

city: New York

Unpacking Arguments

- You can unpack a sequence (e.g., a list or a tuple) or a dictionary and pass its elements as arguments to a function using the * and ** operators.

```
1. def greet(name, message):
2.     print(f"{message} {name}")
3. args = ("Alice", "Good morning!")
4. greet(*args)  # Unpacking a tuple
5. kwargs = {"name": "Bob", "message": "Hello!"}
6. greet(**kwargs)  # Unpacking a dictionary
```

Recursive Functions

- Recursive functions are functions that call themselves, either directly or indirectly. They are commonly used to solve problems that can be broken down into smaller, similar subproblems. Examples include calculating factorials, Fibonacci sequences, and traversing tree structures.

```
1.  def factorial(n):
2.      if n == 0:
3.          return 1
4.      else:
5.          return n * factorial(n - 1)
```

```
def factor(num):
    fact=1
    for i in range(1, num+1):
        fact*=i
        print(fact)
    print(f"The factorial of {num} is", fact)
numb=int(input("Please enter a number to factorise\n"))
factor(numb)
```

Lambda functions

- **Lambda functions** often referred to as anonymous functions, are a way to create small, unnamed, and single-expression functions.
- They are defined using the lambda keyword and are typically used for simple operations.
- Lambda functions are particularly handy when you need a quick, throwaway function without the need to define a full named function.
- The syntax of a lambda function is as follows:
- lambda arguments: expression

Lambda functions +

Some key points to note about lambda functions:

- No Function Name: Lambda functions do not have a name; they are anonymous.
- Single Expression: A lambda function consists of a single expression, and the result of this expression is automatically returned.
- Limited to One Line: Lambda functions are limited to a single line of code.
- Typically Used with Higher-Order Functions: Lambda functions are often used as arguments in higher-order functions such as map(), filter(), and reduce().

Lambda functions ++

```
1. # A lambda function that squares its argument
2. square = lambda x: x**2
3. # Using a lambda function with the map() function to double
   each element in a list
4. numbers = [1, 2, 3, 4, 5]
5. doubled = map(lambda x: x * 2, numbers)
6. # Using a lambda function with the filter() function to get
   even numbers from a list
7. even_numbers = filter(lambda x: x % 2 == 0, numbers)
8. # Using a lambda function to sort a list of tuples based on
   the second element
9. pairs = [(1, 4), (3, 2), (2, 8)]
10.sorted_pairs = sorted(pairs, key=lambda x: x[1])
```

The Local and Global Scope of a Variable

- In Python, variables have different scopes, which determine where in your code a variable can be accessed or modified. The two primary scopes are local and global:

- **Local Scope**:
  - Variables defined inside a function have a local scope. They are called local variables.
  - Local variables are accessible only within the function in which they are defined.
  - They are created when the function is called and destroyed when the function exits.

Local Scope of a Variable

Example of local scope:

```
1. def my_function():
2.         x = 10 # x is a local variable
3.         print(x)
4. my_function()
```

- # Call the function print(x)
- # This will result in an error because x is not defined in the global scope
- In this example, **x** is a local variable and can only be used within the **my_function** function.

Global Scope:

- Variables defined outside of any function, typically at the top level of a script or module, have a global scope. They are called global variables.
- Global variables are accessible from any part of your code, both within and outside functions.
- They are created when they are defined and persist as long as the program is running.
- Example of global scope:

Global Scope+

```
1. y = 20 # y is a global variable
2. def my_function():
3.     print(y)
4. my_function()
5. print(y)
```

- # The function can access the global variable y print(y)
- # You can also access y outside the function

The return Statement

- The **return** statement is used within a function to specify the value that the function should return when it is called. The **return** statement is crucial for functions that are designed to produce a result that you want to use or work with outside the function. Here are the key points to understand about the **return** statement:
- **Syntax**:
- def function_name(parameters): # Function body # ... return expression

- **function_name**: The name of the function.

The return Statement+

- **Parameters**: Optional, but if the function accepts input, you specify the parameters here.
- **Expression**: The value you want to return from the function. This can be a variable, an expression, or a literal value.

➢ **Usage**:

➢ **Returning a Value**: You use the **return** statement to specify a value that the function will return when it is called.

The return Statement++

This value can be of any data type, including numbers, strings, lists, or more complex data structures.

- Example:

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

- In this example, the **add** function returns the sum of the values **x** and **y**.

The return Statement++

Exiting the Function: When a **return** statement is encountered, the function immediately exits, and the specified value is passed back to the caller. This means that any code after the **return** statement within the function is not executed.

Example:

```
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False
    print("This line is never reached")
```

The return Statement++++

- In this example, the **print** statement after the **return** statements is never executed because the function exits when a **return** statement is reached.

- **Multiple Return Statements**: A function can have multiple **return** statements. The one that gets executed is determined by the flow of the program. For example, in an **if** condition, different **return** statements can be used to return different values.

The return Statement++++

```
1.def get_grade(score):
2.    if score >= 90:
3.    return "A"
4.    elif score >= 80:
5.    return "B"
6.    elif score >= 70:
7.    return "C"
8.    else:
9.    return "F"
```

The return Statement+++++

- In this example, the function returns different grades based on the value of the **score** parameter.
- **Using the Returned Value**: When you call a function with a **return** statement, you can assign the returned value to a variable or use it directly in your code.
- Example:
- result = add(3, 4) # Call the function and store the result in the variable "result" print(result) # This will print 7
- The **return** statement is essential for functions that need to provide meaningful output or results to the calling code.

The return Statement++++++

- It allows you to encapsulate functionality and data processing within functions and then use the results as needed in the rest of your program.

Compound Interest Calculator

```
def calculate_compound_interest(principal, rate, time, compounding_frequency):
    # Convert the annual rate to decimal
    rate = rate / 100
    # Calculate the number of times interest is compounded per year
    n = compounding_frequency
    # Calculate the total number of compounding periods
    nt = n * time
    # Calculate the compound interest
    amount = principal * (1 + rate / n) ** nt
    # Calculate the interest earned
    interest = amount - principal
    return amount, interest
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the annual interest rate (%): "))
time = float(input("Enter the number of years: "))
compounding_frequency = int(input("Enter the compounding frequency per year: "))
# Calculate compound interest
result, interest_earned = calculate_compound_interest(principal, rate, time, compounding_frequency)
# Display the result
print(f"Total amount after {time} years: {result:.2f}")
print(f"Interest earned: {interest_earned:.2f}")
```

Higher Order functions

- A function is called Higher Order Function if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions.[2]
- In other words, a higher-order function treats functions as first-class citizens, allowing you to work with functions in a flexible and modular way.
- Higher-order functions are a fundamental concept in functional programming and can lead to more concise and expressive code
- Some of the common higher order functions include map, filters, reduce etc

map in Python

- map is a built-in function that is used to apply a given function to each item in an iterable (such as a list, tuple, or other iterable objects) and returns an iterator that yields the results.
- The primary purpose of map is to transform and process data efficiently without the need for explicit loops.
- The basic syntax of the map function
- map(function, iterable, ...)

map in Python+

- function: This is the function that you want to apply to each item in the iterable. It can be a regular function or a lambda function.
- iterable: This is the iterable (e.g., a list, tuple, or other iterable objects) on which the function will be applied.
- The map function can take multiple iterable arguments if the provided function expects multiple arguments.
- map returns a map object, which is an iterator. To see the results as a list, tuple, or another iterable, you need to convert the map object into the desired type, such as a list.

map in Python+

➢ Example of how to use the map function:

```
1. # Define a function to square a number
2. def square(x):
3.     return x ** 2
4. # Create a list of numbers
5. numbers = [1, 2, 3, 4, 5]
6. # Use map to apply the square function to each number in the list
7. squared_numbers = map(square, numbers)
8. # Convert the map object to a list to see the results
9. squared_numbers_list = list(squared_numbers)
10.print(squared_numbers_list)
```

map in Python++

- In this example, the square function is applied to each element in the numbers list, and the result is stored in the squared_numbers_list. The output will be [1, 4, 9, 16, 25].
- You can also use a lambda function with map to achieve the same result more concisely:

```
1. numbers = [1, 2, 3, 4, 5]
2. squared_numbers = map(lambda x: x ** 2, numbers)
3. squared_numbers_list = list(squared_numbers)
4. print(squared_numbers_list)
```

The filter function

- The filter function in Python is a built-in higher-order function that is used to filter elements from an iterable based on a specified filtering function.
- The filter function creates a new iterable containing only the elements for which the filtering function returns True.
- Basic syntax of filter function
- filter(function, iterable)
- function: This is the filtering function. It should take a single argument (an element from the iterable) and return True or False to indicate whether the element should be included in the result.

The filter function+

- iterable: This is the iterable (e.g., a list) from which elements are to be filtered.
- The filter function returns a filter object, which is an iterator. To see the results as a list, tuple, or another iterable, you need to convert the filter object into the desired type.
- Here's an example of how to use the filter function:

The filter function++

```
1. # Define a filtering function to check if a number is even
2. def odd_nums(p):
3.     return p % 2 != 0
4. # Create a list of numbers
   lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
5. # Use filter to include only the even numbers
6. odd_numbers = filter(odd_nums, lst)
7. # Convert the filter object to a list to see the results
8. odd_numbers_list = list(odd_numbers)
9. print(odd_numbers_list)
```

The filter function++

- In this example, the odd_nums function is used as the filtering function.
- The filter function iterates over each element in the numbers list and includes only the elements for which the odd_nums function returns True.
- You can also use lambda functions with the filter function to achieve the same result more concisely:

```
1. lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
2. old_numbers = filter(lambda x: x % 2 != 0, lst)
3. odd_numbers_list = list(odd_numbers)
4. print(odd_numbers_list)
```

reduce function

- reduce() is a function in Python that is part of the functools module (Python 3)
- It's a higher-order function used to reduce a sequence of values to a single cumulative result by applying a specified function successively to the elements of the sequence.
- It's especially useful for operations that involve cumulative or aggregative calculations.
- Here's the basic syntax of the reduce() function:
- functools.reduce(function, iterable[, initializer])

reduce function +

- function: This is the function that you want to apply cumulatively to the items of the iterable. It should take two arguments, for example, function(accumulator, element).
- iterable: This is the sequence of values (e.g., a list or tuple) to which the function will be applied.
- initializer (optional): This is an optional value that can be used as the initial value of the accumulator.
- If provided, the function is first applied to the initializer and the first element of the iterable.
- Here's an example of how to use the reduce() function to calculate the cumulative sum of a list of numbers:

reduce function++

```
1. from functools import reduce
2. # Define a function to add two numbers
3. def add(x, y):
4.     return x + y
5. numbers = [1, 2, 3, 4, 5]
6. # Use reduce to calculate the cumulative sum of the
   numbers
7. cumulative_sum = reduce(add, numbers)
8. print(cumulative_sum)
```

Output: 25

reduce function++

- In the above example, the cumulative sum starts with an initial value of 10, and the elements of the numbers list are added to it, resulting in 25.
- reduce() is particularly useful for scenarios where you need to perform operations that accumulate or aggregate data, such as calculating the product of all elements in a list, finding the maximum or minimum value, or any other operation that can be expressed as a cumulative process.

Summary

- Throughout this week, our exploration centered on the dynamic world of functions in Python. We embarked on a journey to comprehend the fundamental concepts that form the backbone of Python's functional capabilities.
- First and foremost, we categorically distinguished between the two major types of functions: custom or user-defined functions, which we create to suit our specific needs, and Python's built-in functions, a treasure trove of pre-existing tools at our disposal. Armed with this knowledge, we dove into the art of creating and invoking functions, a foundational skill that empowers us to streamline and organize our code effectively.

Summary+

- To deepen our understanding, we unraveled the difference between parameters and arguments, crucial elements in function usage. We explored the various categories of arguments and honed our ability to harness their versatility to our advantage.
- We delved into the intriguing world of variables, distinguishing between local variables, whose scope is confined to within a function, and global variables, which transcend the confines of a single function and are visible throughout the entire application.

Summary++

- Our journey didn't stop there; we embraced the elegance of lambda functions and the power of recursion in Python, equipping ourselves with these advanced tools for more sophisticated problem-solving and elegant code design.
- Our exploration was comprehensive, touching upon a plethora of other captivating topics, forming a robust foundation for our continued journey into the world of Python programming.

Reference

[1] Kamthane, A. N., & Kamthane , A. A. (2018). *PROGRAMMING AND PROBLEM SOLVING WITH PYTHON* (p. 139). McGraw Hill Education (India) Private Limited.

[2] (2020, January 30). *Higher Order Functions in Python*. GeeksforGeeks. Retrieved October 16, 2023, from https://www.geeksforgeeks.org/higher-order-functions-in-python/