Functions - MAD 102 Week 6 Notes

Hia Al Saleh

October 9th, 2024

Contents

1 Introduction to Functions

1.1 Functions and Modules

Functions are sections of code that perform a specific task. They make a program more organized and clearer by breaking it down into smaller parts.

Key points:

- Functions are self-contained blocks of code.
- Functions help in clarity and reusability.
- Modules are sections of code that are used to perform specific tasks.

1.2 Top-down Design

Top-down design is a method of program design that outlines the tasks to be performed, with details being refined later. It makes the program structure clear by using a main module that can call other modules like `displayCurrentBalance()`, `withdraw()`, etc.

Listing 1: Example of Top-down Design

```
def main():
    displayCurrentBalance()
    withdraw()
    displayCurrentBalance()

def displayCurrentBalance():
    print("Current balance displayed.")

def withdraw():
    print("Amount withdrawn.")
```

2 Modularization and Function Naming

2.1 Modularizing a Program

Modularization helps in breaking down a large program into smaller, manageable sections (modules). Python recommends using lowercase function names with underscores between words, such as `calculate_interest`.

Listing 2: Example of Correct Function Naming

```
def calculate_interest():
    pass  # Function body goes here
```

2.2 Flowcharting Modules

When designing a program flow, modules are represented with a rectangle and the keyword `Call` followed by the function name.

3 Function Basics

3.1 Defining Functions

A function in Python is defined using the `def` keyword followed by the function name and parentheses to wrap any arguments. Functions can be called using their name followed by parentheses.

Listing 3: Defining and Calling a Function

```
def greet():
    print("Hello!")

greet()  # Calling the function
```

3.2 Passing Arguments

Functions can accept arguments (values) that are passed when the function is called. This helps in reducing coupling and improving flexibility. You can pass multiple arguments by separating them with commas.

Listing 4: Passing Arguments to a Function

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

greet("Alice")  # Calling the function with an argument
```

4 Scope of Variables in Functions

4.1 Local and Global Variables

Variables declared inside a function are local to that function, meaning they cannot be accessed outside of it. Global variables are accessible by all functions, but using them increases the coupling between modules.

Listing 5: Using Local and Global Variables

```
x = "global variable"

def my_function():
    x = "local variable"
    print(x)  # Prints local variable
```

```
my_function()
print(x)  # Prints global variable
```

4.2 Why Use Local Variables?

Local variables prevent name duplication and allow for efficient program execution by limiting the scope of variables. When needed, data can be passed to a function using arguments instead of relying on global variables.

5 Returning Values from Functions

Functions can return values using the `return` keyword. Functions that do not return anything are known as void functions. If multiple values need to be returned, a tuple can be used.

Listing 6: Returning Values from a Function

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

result = add(5, 3)
print(result)  # Output: 8
```

6 Advanced Function Topics

6.1 Polymorphism

Functions can accept different types of arguments, a feature known as polymorphism. This allows the same function to work with strings, numbers, or other data types.

Listing 7: Polymorphic Function Example

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

print(add(5, 3))  # Works with numbers
print(add("hello", "world"))  # Works with strings
```

6.2 Nested Functions

Functions can be defined inside other functions. These are known as nested functions, and they are hidden from code outside the enclosing function.

Listing 8: Nested Functions Example

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

    inner_function()

outer_function()
```

6.3 Incremental Development and Function Stubs

Incremental development involves writing and testing small pieces of code gradually. Function stubs, using the `pass` keyword or placeholders, can be useful for setting up a program before all functions are fully implemented.

Listing 9: Function Stub Example

```
def my_function():
    pass  # Placeholder for function implementation
```

6.4 Functions as Objects

Functions in Python are objects, which means they have identity, type, and value. Functions can be assigned to variables and passed as arguments to other functions.

Listing 10: Assigning Functions to Variables

```
def greet():
    print("Hello!")

g = greet  # Assign function to a variable
g()  # Call the function using the variable
```

6.5 Namespaces

A namespace maps names to objects. Python provides the `locals()` and `globals()` functions to access local and global namespaces, respectively.

7 Keyword and Default Arguments

7.1 Keyword Arguments

Keyword arguments map arguments to parameters by their names rather than their order, making function calls clearer when dealing with multiple arguments.

Listing 11: Using Keyword Arguments

```
def greet(name, message):
    print(f"{message}, {name}!")

greet(name="Alice", message="Good morning")
```

7.2 Default Parameter Values

Functions can have default parameter values that can be overridden by providing new values when calling the function.

Listing 12: Using Default Parameter Values

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Alice")  # Uses default value for message
greet("Bob", "Good evening")  # Overrides default value
```

7.3 Arbitrary Arguments

Functions can accept arbitrary numbers of arguments using the `*args` parameter, and keyword arguments can be captured using `**kwargs`, which stores them in a dictionary.

Listing 13: Arbitrary Arguments Example

```
def print_numbers(*args):
    for num in args:
        print(num)

print_numbers(1, 2, 3, 4, 5)  # Prints multiple numbers
```

8 Documenting Functions with Docstrings

A `docstring` is a literal string placed in the first line of a function's body to document its purpose. This helps both the original programmer and others understand the function's use and behavior.

Listing 14: Docstring Example

```
def greet():
    """This function prints a greeting message."""
    print("Hello!")
```

9 Week 7: Functions in Class Demo

9.1 Basic Function Example

This section covers basic function examples demonstrated in class.

Listing 15: Basic Addition Function

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

# Calling the function
add(3, 4)  # Outputs: 7
add('x', 'y')  # Outputs: xy
```

9.2 Scope of Variables

The following example demonstrates how global and local variables work in Python.

Listing 16: Variable Scope Example

```
# Global variable
c = 15

# Function to perform addition
def add(a, b):
    c = 20
    output = a + b + c
    print(output)

# Calling the function
add(2, 3)  # Outputs: 25
```

9.3 Nested Functions and Function Stubs

This example demonstrates nested functions and the use of function stubs in Python.

Listing 17: Billing System with Nested Functions

```
def billing_system():
    cart = []

    def add_item(item_name, price):
        cart.append((item_name, price))

    def calculate_total():
        total = sum(price for _, price in cart)
        return total
```

```
    def apply_discount(total, discount_percentage):
        discount_amount = (discount_percentage / 100) *
            total
        total = total - discount_amount
        return total, discount_amount

    def generate_bill():
        print("Shopping Cart:")
        for item_name, price in cart:
            print(f"{item_name}: ${price:.2f}")
        total = calculate_total()

        discount_percentage = 10  # Example discount rate
        if len(cart) > 2:
            total_with_discount, discount_amount =
                apply_discount(total, discount_percentage)
            print(f"Total before discount: ${total:.2f}")
            print(f"Discount ({discount_percentage}%): ${
                discount_amount:.2f}")
            print(f"Total after discount: ${
                total_with_discount:.2f}")
        else:
            print(f"Total: ${total:.2f} (No discount
                applied)")

    while True:
        print("Options:")
        print("1. Add item to cart")
        print("2. Generate bill")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            item_name = input("Enter item name: ")
            price = float(input("Enter item price: "))
            add_item(item_name, price)
        elif choice == "2":
            generate_bill()
        elif choice == "3":
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    billing_system()
```

9.4 Arguments: Keyword, Default, and Positional

This section covers different types of arguments.

Listing 18: Keyword and Default Arguments

```
def add(a, b, c=1):  # c is the default argument
    print(a + b + c)

add(a=3, b=4, c=5)  # Outputs: 12
```

Listing 19: Positional Arguments

```
def sub(a, b):
    print(a - b)

sub(3, 4)  # Outputs: -1
sub(4, 3)  # Outputs: 1
```

9.5 Arbitrary Arguments

This section demonstrates how to work with variable-length arguments.

Listing 20: Arbitrary Positional Arguments (*args)

```
def arbitrary_demo_args(*args):
    list_created = []
    for arg in args:
        list_created.append(arg)
    print(list_created)

arbitrary_demo_args('1', '6', '8')  # Outputs: ['1', '6',
    '8']
```

Listing 21: Arbitrary Keyword Arguments (**kwargs)

```
def arbitrary_demo_kwargs(**a): # Variable length argument
     with Keyword
    for key, val in kwargs.items():
        print(f"Keyword: {key}, Value: {val}")

arbitrary_demo_kwargs(firstName='Aishwarya', secondName='
    Raj')
# Outputs:
# Keyword: firstName, Value: Aishwarya
# Keyword: secondName, Value: Raj
```

10 Class Exercises

10.1 Exercise 1: Comparing Scores and Finding Differences

Requirements:

1. Ask for two player scores.
2. Find the highest and lowest score using a *min_max* function.
3. Calculate the difference using a *score_difference* function.

```
# Function to return the minimum and maximum of two values
def min_max(score1, score2):
    if score1 > score2:
        return score1, score2
    elif score1 < score2:
        return score2, score1
    else:
        return score1, score2  # Both are equal, returning
            either order doesn't matter

# Function to return the absolute difference between the
    two scores
def score_difference(score1, score2):
    return abs(score1 - score2)

# Main program for Exercise 1
def main_exercise1():
    # Get the scores from the user
    score1 = int(input("Enter the first player's score: ")
        )
    score2 = int(input("Enter the second player's score: "
        ))

    # Get the highest and lowest scores
    high, low = min_max(score1, score2)

    # Calculate the score difference
    difference = score_difference(high, low)

    # Display the results
    print(f"The highest score is: {high}")
    print(f"The lowest score is: {low}")
    print(f"The difference between the scores is: {
        difference}")

# Run Exercise 1
main_exercise1()
```

10.2 Exercise 2: Number Guessing Game

Requirements:

- A number between 1 and 10 is randomly generated.
- The user has 3 chances to guess it.
- Provide feedback if the guess is too high or too low.

```
import random

# Main program for Exercise 2
def main_exercise2():
    # Generate a random number between 1 and 10
    number_to_guess = random.randint(1, 10)

    # Allow the user 3 attempts to guess
    attempts = 3
    for i in range(attempts):
        guess = int(input(f"Attempt {i+1}/{attempts} -
            Guess the number (1-10): "))

        if guess == number_to_guess:
            print("Congratulations! You've guessed the
                correct number.")
            break
        elif guess < number_to_guess:
            print("Too low! Try again.")
        else:
            print("Too high! Try again.")

    # If the user does not guess correctly after 3
        attempts
    else:
        print(f"Sorry, you've used all attempts. The
            correct number was {number_to_guess}.")

# Run Exercise 2
main_exercise2()
```