Defining and Calling Functions

Introduction

- Functions are essential building blocks of a program.
- They help organize code, promote reusability, and simplify complex tasks.

What are Functions?

- Functions are blocks of code that perform a specific task.
- They take input (arguments), process it, and produce output (return value).

Significance of Functions

- Functions improve code readability and maintainability.
- They break down large tasks into smaller, manageable parts.
- Reusing functions saves time and effort in writing repetitive code.

Benefits of Using Functions

- **Modularization**: Dividing code into smaller, logical units.
- **Reusability**: Using functions in multiple parts of the program.
- **Abstraction**: Focusing on what a function does, not how it does it.

Function Anatomy

- **Function definition:** def function_name(parameters):
- **Function body**: Code block that performs the task.
- **Function call:** Invoking the function with arguments.

Functions and Python Libraries

- Python provides built-in functions (e.g., print(), len()).
- Libraries offer predefined functions for specific tasks (e.g., math.sqrt(), random.choice()).

Demonstrating Functions

- We'll now delve into defining and calling functions in Python.
- Understanding the syntax and concepts will set the foundation for practical implementation.

Function Definition

- In Python, functions are defined using the '**def**' keyword followed by the function name and parentheses.
- Function name should be descriptive and follow naming conventions.
- Parameters are enclosed in parentheses (optional).

Function Syntax

```
def function_name(parameters):

    # Function body - code block

    # Perform tasks and computations

    # Optional return statement
```

Example: Print Function

```
def greet(name):

    print("Hello, " + name + "!")



greet("Alice")

greet("Bob")
```

Example: Simple Addition Function

```
def add_numbers(a, b):

    return a + b



result = add_numbers(5, 10)

print(result) # Output: 15
```

Default Parameters

- Functions can have default parameter values.
- If a parameter is not provided during the function call, the default value is used.

Example: Function with Default Parameter

```
def greet(name="Guest"):

    print("Hello, " + name + "!")



greet("Alice") # Output: Hello, Alice!

greet()        # Output: Hello, Guest! (Default value used)
```

Keyword Arguments

- Arguments can be passed by explicitly mentioning the parameter name.
- This allows for a flexible and clear function call.

Example: Function with Keyword Arguments

```
def create_email(name, domain="example.com"):

return name + "@" + domain

email1 = create_email("john", domain="company.com")

email2 = create_email(name="alice", domain="mycompany.com")
```

Variable Number of Arguments

Functions can accept a variable number of arguments using

*args and **kwargs.

Example: Function with Variable Arguments

```
def add_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total


result1 = add_numbers(1, 2, 3)
result2 = add_numbers(10, 20, 30, 40, 50)
```

Return Statement

- Functions can return values using the return statement.
- If no return statement is used, the function returns None by default.

Calling Functions

Syntax

function_name(arguments)

Passing Arguments to Functions

1. **Positional Arguments:** These are arguments passed to a function based on their position in the function call. The order matters, and the function uses them in the same order as the parameters in its definition.

   function_name(arg1, arg2, arg3)

2. **Keyword Arguments**: Instead of relying on the order, we can explicitly mention the parameter names and their corresponding values when calling the function.

   function_name(param1=value1, param2=value2, param3=value3)

Examples of Calling Functions

Example 1:

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


# Calling function with positional argument
print(greet("Alice"))  # Output: Hello, Alice!


# Calling function with keyword argument
print(greet(name="Bob"))  # Output: Hello, Bob!
```

Example 2

```
def calculate_total(price, quantity):
    return price * quantity


# Calling function with positional arguments
print(calculate_total(10, 5))  # Output: 50


# Calling function with keyword arguments
print(calculate_total(price=8, quantity=3))  # Output: 24
```

Understanding Function Return Values

- After a function performs its task, it may return a value back to the caller.
- We use the return statement to specify the value that the function should return.
- If no return statement is used, the function will return None by default.

Function Return Values and Their Usage

Function return values can be used in various ways:

- Assigning the Return Value: We can store the return value of a function in a variable for further use.

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


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

- Using Return Values in Expressions: We can use function return values directly in expressions.

```
def square(number):
    return number ** 2


area = 3 * square(2)
print(area)  # Output: 12
```

Modular Programming

Modular programming is an approach in which we divide a complex problem into smaller, more manageable pieces, called functions. Each function performs a specific sub-task, making the code easier to understand and maintain. By using this technique, we can build larger programs by combining these smaller functions.

Advantages of Modular Programming

- **Code Reusability:** Reusing functions in different parts of the program.
- **Maintainability:** Easier to update and maintain smaller functions.
- **Readability:** Makes the code more organized and easy to understand.
- **Collaboration:** Multiple developers can work on different modules simultaneously.

Writing Modular Code

Guidelines for writing modular code:

- Define clear and specific functions.
- Avoid functions with too many responsibilities.
- Name functions descriptively to indicate their purpose.
- Use function arguments and return values effectively.

Example: Modular Approach

```
# Non-modular approach
def calculate_area(length, width):
    area = length * width
    return area

def calculate_perimeter(length, width):
    perimeter = 2 * (length + width)
    return perimeter

# Modular approach
def calculate_area(length, width):
    return length * width

def calculate_perimeter(length, width):
    return 2 * (length + width)
```

Introduction to Exception Handling

- What is exception handling in Python?
- Why do we need to handle exceptions in programs?
- The role of try-except blocks in handling errors gracefully.

What is Exception Handling?

- Explanation of what exceptions are.
- How exceptions can disrupt program flow.
- The need for exception handling to prevent program crashes.

Using try-except Blocks

- Syntax of a try-except block.
- Working principle of try-except blocks.
- Code inside try block vs. code inside except block.

Handling Exceptions with try-except Block

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example: Exception Handling

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ValueError:
    print("Invalid input! Please enter valid numbers.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print("Result:", result)
```

Handling Multiple Exceptions

Explain how to handle multiple exceptions using multiple except blocks or a single except block with multiple exception types.

The Finally Block

Introduce the finally block, which allows executing code regardless of whether an exception occurred.

Example: finally Block

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
finally:
    # Code that will always run, exception or not
```

Handling Common Exceptions in Functions

- Identifying and handling specific exceptions in functions.
- Examples of common exceptions: ZeroDivisionError, ValueError, etc.
- Writing custom exception handlers for different scenarios.

Example: Handling a ZeroDivisionError

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    else:
        print("Result:", result)
```

Advantages of Exception Handling

- Prevents program crashes and abrupt termination.
- Provides meaningful error messages for debugging.
- Allows for controlled error handling and graceful degradation.

Practical Examples

Implementing Simple Functions for Arithmetic Operations

- Write a function called "add_numbers" that takes two arguments, "num1" and "num2," and returns their sum.
- Create a function named "multiply_numbers" that multiplies three numbers, "num1," "num2," and "num3," and returns the result.
- Define a function called "calculate_average" that calculates the average of a list of numbers and returns it.

Answers

```
# Exercise 1
def add_numbers(num1, num2):
    return num1 + num2

def multiply_numbers(num1, num2, num3):
    return num1 * num2 * num3

def calculate_average(numbers):
    return sum(numbers) / len(numbers)
```

Exercise 2: Creating Custom Functions to Solve Specific Problems

- Write a function called "is_even" that takes an integer as input and returns True if it is even, and False otherwise.
- Implement a function named "convert_to_uppercase" that takes a string as input and returns the same string in uppercase.
- Create a function called "find_max_length" that takes a list of strings and returns the length of the longest string in the list.

Answers

```
# Exercise 2
def is_even(number):
    return number % 2 == 0


def convert_to_uppercase(input_string):
    return input_string.upper()


def find_max_length(string_list):
    return max(len(s) for s in string_list)
```

Exercise 3: Applying Functions to Real-World Scenarios

1. Write a function named "calculate_total_price" that takes a price and quantity as inputs and returns the total cost.
2. Implement a function called "calculate_discounted_price" that takes a price and a discount percentage as inputs and returns the discounted price.
3. Create a function called "calculate_gpa" that takes a list of grades and their respective credits as inputs and returns the GPA.

Answers

```
# Exercise 3
def calculate_total_price(price, quantity):
    return price * quantity


def calculate_discounted_price(price, discount_percent):
    return price - (price * discount_percent / 100)


def calculate_gpa(grades, credits):
    total_credits = sum(credits)
    weighted_sum = sum(grade * credit for grade, credit in zip(grades, credits))
    return weighted_sum / total_credits
```

Exercise 4: Applying Functions to Real-World Scenarios in Finance Domain

1. **Financial Investment:**

   Create a function called "calculate_future_value" that takes the present value, interest rate, and number of years as inputs and returns the future value of an investment using the compound interest formula.

Answer:

```
def calculate_future_value(present_value, interest_rate, years):
    future_value = present_value * (1 + interest_rate / 100) ** years
    return future_value
```

2. Loan Repayment:

Implement a function named "calculate_monthly_payment" that takes the loan amount, annual interest rate, and loan period (in years) as inputs and returns the monthly payment for a fixed-rate loan using the formula for monthly loan payment.

Answer:

```
def calculate_monthly_payment(loan_amount, annual_interest_rate, loan_period):
    monthly_interest_rate = annual_interest_rate / 12 / 100
    months = loan_period * 12
    monthly_payment = loan_amount * monthly_interest_rate * ((1 + monthly_interest_rate) ** months) / (((1 + monthly_interest_rate) ** months) - 1)
    return monthly_payment
```

3. Portfolio Management:

Create a function called "calculate_portfolio_return" that takes a list of asset returns and their respective weights in a portfolio as inputs and returns the portfolio's overall return.

Answer:

```
def calculate_portfolio_return(returns, weights):
    portfolio_return = sum(return_i * weight_i for return_i, weight_i in zip(returns, weights))
    return portfolio_return
```

4. Risk Assessment:

Implement a function named "calculate_portfolio_variance" that takes a list of asset returns, their respective weights, and the covariance matrix of asset returns as inputs and returns the variance of a portfolio.

Answer:

```
def calculate_portfolio_variance(returns, weights, covariance_matrix):
    weighted_variance = sum(weight_i ** 2 * covariance_matrix[i][i] for i, weight_i in enumerate(weights))
    for i in range(len(returns)):
        for j in range(i + 1, len(returns)):
            weighted_variance += 2 * weights[i] * weights[j] * covariance_matrix[i][j]
    return weighted_variance
```

Interactive Coding Session: Defining and Calling Functions with Exception Handling

Hands-on Project: Creating a Simple Calculator

Q&A