Problem Solving using Python

Unit 1: Introduction to Python and Functions

Prepared by:

Prof. Mohammed Siraj B

School of Engineering
St Aloysius (Deemed to be University), Mangalore

siraj_b@staloysius.edu.in

September 9, 2025

Contents

1. Python Concepts

1.1 Introduction to Python

Definition:

Python is a high-level, interpreted, interactive, and object-oriented scripting language.

Explanation:

- **High-level:** Python code is easy to read and write, resembling human language more than machine code. You don't need to worry about low-level details like memory management.
- **Interpreted:** Python code is executed line by line by an interpreter, rather than being compiled into machine code beforehand. This makes development faster as you can run code immediately after writing it.
- **Interactive:** You can interact with the Python interpreter directly, typing commands and seeing results instantly. This is great for testing small snippets of code.
- **Object-Oriented:** Python supports object-oriented programming (OOP) paradigms, allowing you to structure your programs using objects and classes, which promotes code reusability and organization.
- **Scripting Language:** Python is often used for scripting, automating tasks, and developing web applications, data analysis tools, and more.

General Syntax:

Python's syntax is known for its simplicity and readability. It uses indentation to define code blocks, unlike many other languages that use curly braces.

Examples:

Example: A Simple 'Hello, World!' Program

This is the classic first program in any language. It demonstrates how easy it is to print output to the console in Python.

```
print("Hello, World!")
```

Listing 1: Hello World

Output:

```
Hello, World!
```

Example 2: Basic Arithmetic Operations Python can be used as a powerful calculator. Here are some basic arithmetic operations.

```python
# Addition
result_add = 10 + 5
print(f"10 + 5 = {result_add}")

# Subtraction
result_sub = 10 - 5
print(f"10 - 5 = {result_sub}")

# Multiplication
result_mul = 10 * 5
print(f"10 * 5 = {result_mul}")

# Division (returns a float)
result_div = 10 / 5
print(f"10 / 5 = {result_div}")

# Floor Division (returns an integer, discards fractional part)
result_floor_div = 10 // 3
print(f"10 // 3 = {result_floor_div}")

# Modulus (remainder of division)
result_mod = 10 % 3
print(f"10 % 3 = {result_mod}")

# Exponentiation
result_exp = 2 ** 3
print(f"2 ** 3 = {result_exp}")
```

Listing 2: Basic Arithmetic

Output:

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2.0
10 // 3 = 3
10 % 3 = 1
2 ** 3 = 8
```

Exercises:

1. Write a Python program that prints your name and your favorite hobby on separate lines.
2. Calculate the area of a rectangle with length 15 units and width 8 units using Python. Print the result.
3. If you have 25 apples and you want to distribute them equally among 4 friends, how many apples will each friend get, and how many will be left over? Write a Python program to find this out.

1.2 Variables

Definition: Variables are named storage locations used to store data in a program. Think of them as containers that hold values.

Explanation: In Python, you don't need to declare the type of a variable explicitly. Python is dynamically typed, meaning the type of a variable is determined at runtime based on the value assigned to it. When you assign a new value to a variable, its type can change.

General Syntax:

```
variable_name = value
```

Examples:

Example 1: Storing Different Data Types

```python
# Integer variable
age = 25
print(f"Age: {age}, Type: {type(age)}")

# String variable
name = "Alice"
print(f"Name: {name}, Type: {type(name)}")

# Float variable
price = 19.99
print(f"Price: {price}, Type: {type(price)}")

# Boolean variable
is_student = True
print(f"Is student: {is_student}, Type: {type(is_student)}")
```

Listing 3: Storing Data Types

Output:

```
Age: 25, Type: <class 'int'>
Name: Alice, Type: <class 'str'>
```

```
Price: 19.99, Type: <class 'float'>
Is student: True, Type: <class 'bool'>
```

Example 2: Reassigning Variables Variables can be reassigned with new values, even of different types.

```python
x = 10
print(f"Initial value of x: {x}, Type: {type(x)}")

x = "Hello"
print(f"New value of x: {x}, Type: {type(x)}")
```

Listing 4: Reassigning Variables

Output:

```
Initial value of x: 10, Type: <class 'int'>
New value of x: Hello, Type: <class 'str'>
```

Exercises:

1. Create three variables: `city` (string), `population` (integer), and `area_sq_km` (float). Assign appropriate values and print them along with their types.
2. Swap the values of two variables, `a` and `b`, without using a third temporary variable. Print the values before and after swapping.

1.3 Keywords

Definition: Keywords (also known as reserved words) are special words that have a predefined meaning and purpose in Python. They cannot be used as variable names, function names, or any other identifiers.

Explanation: Python has a relatively small set of keywords, but they are fundamental to the language's structure and functionality. Understanding keywords is crucial for writing syntactically correct Python code.

General Syntax: Keywords are used as part of Python's syntax for control flow, defining functions, classes, and more. There is no specific syntax for keywords themselves, as they are part of the language's grammar.

Examples:

Example 1: Common Keywords in Use

```python
# if, elif, else (for conditional statements)
if True:
    print("This is true.")
elif False:
    print("This is false.")
else:
    print("Neither is true.")

# for, in (for loops)
for i in range(3):
    print(f"Loop iteration: {i}")

# def (to define a function)
def greet(name):
    return f"Hello, {name}!"
print(greet("World"))

# class (to define a class)
class MyClass:
    pass

# import (to import modules)
import math
print(math.pi)

# True, False, None (Boolean and NoneType values)
is_active = True
user_data = None
print(f"Is active: {is_active}, User data: {user_data}")
```

Listing 5: Keywords in Use

Output:

```
This is true.
Loop iteration: 0
Loop iteration: 1
Loop iteration: 2
Hello, World!
Is active: True, User data: None
```

Exercises:

1. Try to use a Python keyword (e.g., `if`, `for`, `def`) as a variable name. What error do you get? Explain why.
2. Research and list at least 5 more Python keywords not mentioned above and briefly describe their purpose.

1.4 Identifiers

Definition: Identifiers are names used to identify variables, functions, classes, modules, or other objects in Python.

Explanation: Identifiers are crucial for giving meaningful names to elements in your code, making it readable and understandable. There are specific rules for creating valid identifiers in Python.

Rules for Identifiers:

- Can contain letters (a-z, A-Z), digits (0-9), and underscores (_).
- Must start with a letter or an underscore.
- Cannot start with a digit.
- Are case-sensitive (e.g., `myVar` is different from `myvar`).
- Cannot be a Python keyword.

General Syntax:

```
valid_identifier = value
_another_identifier = value
myFunction = some_function
```

Examples:

Example 1: Valid and Invalid Identifiers

```python
# Valid identifiers
my_variable = 10
_private_var = "secret"
userName = "JohnDoe"
calculate_sum = lambda a, b: a + b

print(f"my_variable: {my_variable}")
print(f"_private_var: {_private_var}")
print(f"userName: {userName}")
print(f"Sum: {calculate_sum(5, 3)}")

# Invalid identifiers (will cause SyntaxError if uncommented)
# 1st_variable = 20 # Starts with a digit
# my-variable = 30 # Contains a hyphen
# class = 40 # Is a keyword
```

Listing 6: Valid and Invalid Identifiers

Output:

```
my_variable: 10
_private_var: secret
userName: JohnDoe
Sum: 8
```

Exercises:

1. Which of the following are valid Python identifiers? Explain why or why not for each:
   - `_total_count`
   - `user-age`
   - `def`
   - `item2price`
   - `global_variable`
2. Create a variable to store the number of students in a class. Choose a meaningful and valid identifier for it.

1.5 Literals

Definition: Literals are raw data values given in a variable or constant. They are the fixed values that a program uses.

Explanation: Literals are the actual data that you put into your code. Python supports various types of literals, each representing a different kind of data.

Types of Literals:

- **Numeric Literals:** Integers, floats, and complex numbers.
- **String Literals:** Sequences of characters enclosed in single, double, or triple quotes.
- **Boolean Literals:** `True` and `False`.
- **Special Literal:** `None`.
- **Collection Literals:** Lists, Tuples, Dictionaries, Sets (covered in Unit 2).

General Syntax:

```
integer_literal = 100
float_literal = 3.14
string_literal = "Hello"
boolean_literal = True
special_literal = None
```

Examples:

Example 1: Different Types of Literals

```python
# Integer Literals
num_int = 42
print(f"Integer Literal: {num_int}, Type: {type(num_int)}")

# Float Literals
```

```python
num_float = 3.14159
print(f"Float Literal: {num_float}, Type: {type(num_float)}")

# Complex Literal
num_complex = 3 + 4j
print(f"Complex Literal: {num_complex}, Type: {type(num_complex)}")

# String Literals
str_single = 'Python'
str_double = "Programming"
str_triple = """Multi-line
String"""
print(f"Single-quoted String: {str_single}")
print(f"Double-quoted String: {str_double}")
print(f"Triple-quoted String:\n{str_triple}")

# Boolean Literals
is_active = True
is_finished = False
print(f"Boolean True: {is_active}, Type: {type(is_active)}")
print(f"Boolean False: {is_finished}, Type: {type(is_finished)}")

# Special Literal
no_value = None
print(f"Special Literal None: {no_value}, Type: {type(no_value)}")
```

Listing 7: Different Types of Literals

Output:

```
Integer Literal: 42, Type: <class 'int'>
Float Literal: 3.14159, Type: <class 'float'>
Complex Literal: (3+4j), Type: <class 'complex'>
Single-quoted String: Python
Double-quoted String: Programming
Triple-quoted String:
Multi-line
String
Boolean True: True, Type: <class 'bool'>
Boolean False: False, Type: <class 'bool'>
Special Literal None: None, Type: <class 'NoneType'>
```

Exercises:

1. Identify the type of literal for each of the following values:
   - `"Hello, Python!"`
   - `123`

- `False`
- `5.0`
- `None`

2. Create a variable `pi_value` and assign it the literal value of Pi (3.14159). Print its value and type.

1.6 Operators

Definition: Operators are special symbols that perform operations on one or more operands (values or variables).

Explanation: Operators are the building blocks of any programming language, allowing you to perform calculations, comparisons, logical evaluations, and more. Python provides a rich set of operators.

Types of Operators:

- **Arithmetic Operators:** Perform mathematical calculations.
- **Comparison (Relational) Operators:** Compare two values and return a Boolean result.
- **Assignment Operators:** Assign values to variables.
- **Logical Operators:** Combine conditional statements.
- **Bitwise Operators:** Perform operations on individual bits (less common for beginners).
- **Identity Operators:** Check if two variables refer to the same object in memory.
- **Membership Operators:** Test if a sequence contains a specific value.

General Syntax: Varies depending on the operator type (e.g., `operand1 + operand2`, `variable = value`).

Examples:

Example 1: Arithmetic Operators

```python
a = 15
b = 4

print(f"a + b = {a + b}") # Addition
print(f"a - b = {a - b}") # Subtraction
print(f"a * b = {a * b}") # Multiplication
print(f"a / b = {a / b}") # Division
print(f"a % b = {a % b}") # Modulus
print(f"a // b = {a // b}") # Floor Division
print(f"a ** b = {a ** b}") # Exponentiation
```

Listing 8: Arithmetic Operators

Output:

```
a + b = 19
a - b = 11
a * b = 60
a / b = 3.75
a % b = 3
a // b = 3
a ** b = 50625
```

Example 2: Comparison Operators

```python
x = 10
y = 12

print(f"x == y: {x == y}") # Equal to
print(f"x != y: {x != y}") # Not equal to
print(f"x < y: {x < y}") # Less than
print(f"x > y: {x > y}") # Greater than
print(f"x <= y: {x <= y}") # Less than or equal to
print(f"x >= y: {x >= y}") # Greater than or equal to
```

Listing 9: Comparison Operators

Output:

```
x == y: False
x != y: True
x < y: True
x > y: False
x <= y: True
x >= y: False
```

Example 3: Assignment Operators

```python
c = 5
print(f"Initial c: {c}")

c += 3 # c = c + 3
print(f"c after c += 3: {c}")

c *= 2 # c = c * 2
print(f"c after c *= 2: {c}")

c /= 4 # c = c / 4
print(f"c after c /= 4: {c}")
```

Listing 10: Assignment Operators

Output:

```
Initial c: 5
c after c += 3: 8
c after c *= 2: 16
c after c /= 4: 4.0
```

Example 4: Logical Operators

```python
p = True
q = False

print(f"p and q: {p and q}") # Logical AND
print(f"p or q: {p or q}") # Logical OR
print(f"not p: {not p}") # Logical NOT
```

Listing 11: Logical Operators

Output:

```
p and q: False
p or q: True
not p: False
```

Example 5: Identity Operators (`is`, `is not`)

```python
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1

print(f"list1 is list2: {list1 is list2}") # False, different objects in memory
print(f"list1 == list2: {list1 == list2}") # True, values are equal
print(f"list1 is list3: {list1 is list3}") # True, refers to the same object
```

Listing 12: Identity Operators

Output:

```
list1 is list2: False
list1 == list2: True
list1 is list3: True
```

Example 6: Membership Operators (`in`, `not in`)

```python
my_string = "Python Programming"
my_list = [10, 20, 30, 40]

print(f"'Python' in my_string: {'Python' in my_string}")
print(f"'Java' in my_string: {'Java' in my_string}")
print(f"20 in my_list: {20 in my_list}")
print(f"50 not in my_list: {50 not in my_list}")
```

Listing 13: Membership Operators

Output:

```
'Python' in my_string: True
'Java' in my_string: False
20 in my_list: True
50 not in my_list: True
```

Exercises:

1. Given `x = 7` and `y = 2`:
   - Calculate `x` raised to the power of `y`.
   - Check if `x` is greater than or equal to `y`.
   - Use an assignment operator to multiply `x` by `y` and update `x`.
2. Explain the difference between `==` and `is` operators with an example.
3. Write a Python program that checks if the word "apple" is present in a given sentence.

1.7 Comments

Definition: Comments are explanatory notes within a program that are ignored by the Python interpreter. They are used to make the code more understandable for humans.

Explanation: Good commenting practices are essential for code readability and maintainability, especially in larger projects or when working in teams. They help explain *why* certain code exists, *how* it works, or *what* its purpose is.

Types of Comments:

- **Single-line comments:** Start with a hash symbol (`#`).
- **Multi-line comments (Docstrings):** Enclosed in triple single quotes (´´´) or triple double quotes (¨¨¨). While technically string literals, they are often used as multi-line comments or for documentation (docstrings).

General Syntax:

```
# This is a single-line comment

'''
This is a multi-line comment.
It can span multiple lines.
'''

"""
This is also a multi-line comment,
often used as a docstring for functions or classes.
"""
```

Examples:

Example 1: Using Single-line Comments

```python
# This program calculates the area of a circle
radius = 5 # Define the radius of the circle
PI = 3.14159 # Approximate value of PI

area = PI * radius ** 2 # Calculate the area
print(f"The area of the circle is: {area}") # Print the result
```

Listing 14: Single-line Comments

Output:

```
The area of the circle is: 78.53975
```

Example 2: Using Multi-line Comments (Docstrings)

```python
def add_numbers(a, b):
    """
    This function takes two numbers as input
    and returns their sum.
    """
    return a + b

print(add_numbers(10, 20))

'''
Another way to write multi-line comments.
This part of the code is for demonstration purposes only.
'''
```

Listing 15: Multi-line Comments (Docstrings)

Output:

```
30
```

Exercises:

1. Add comments to the following Python code to explain what each line does:

```python
name = "Python"
version = 3.9
message = f"Hello, {name} version {version}!"
print(message)
```

2. Write a short Python function that calculates the factorial of a number. Include a docstring that explains the function's purpose, arguments, and what it returns.

2. Control Statements

Control statements are fundamental to programming as they dictate the flow of execution in a program. They allow your code to make decisions, repeat actions, and handle different scenarios.

2.1 Conditional Statements: `if`, `if-else`, `elif`, `nested if`

Definition: Conditional statements allow a program to execute different blocks of code based on whether a specified condition evaluates to `True` or `False`.

Explanation: These statements are used for decision-making. Python uses `if`, `elif` (else if), and `else` keywords to define these conditions. Indentation is crucial to define the scope of each block.

General Syntax:

```
# if statement
if condition:
    # code to execute if condition is True

# if-else statement
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False

# if-elif-else statement
if condition1:
    # code to execute if condition1 is True
elif condition2:
    # code to execute if condition2 is True
elif condition3:
    # code to execute if condition3 is True
else:
    # code to execute if none of the above conditions are True

# Nested if statement
if condition_outer:
    # code for outer condition
    if condition_inner:
        # code for inner condition
    else:
        # code for inner else
else:
    # code for outer else
```

Examples:

Example 1: Simple `if` Statement Check if a number is positive.

```python
number = 10
if number > 0:
    print(f"{number} is a positive number.")

number = -5
if number > 0:
    print(f"{number} is a positive number.") # This line will not be executed
```

Listing 16: Simple if Statement

Output:

```
10 is a positive number.
```

Example 2: `if-else` Statement Determine if a number is even or odd.

```python
num = 7
if num % 2 == 0:
    print(f"{num} is an even number.")
else:
    print(f"{num} is an odd number.")

num = 12
if num % 2 == 0:
    print(f"{num} is an even number.")
else:
    print(f"{num} is an odd number.")
```

Listing 17: if-else Statement

Output:

```
7 is an odd number.
12 is an even number.
```

Example 3: `if-elif-else` Statement Assign a grade based on a score.

```python
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f"With a score of {score}, your grade is {grade}.")

score = 55
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"
print(f"With a score of {score}, your grade is {grade}.")
```

Listing 18: if-elif-else Statement

Output:

```
With a score of 85, your grade is B.
With a score of 55, your grade is F.
```

Example 4: `nested if` Statement Check eligibility for a discount based on age and membership status.

```python
age = 22
is_member = True

if age >= 18:
    print("You are old enough to participate.")
    if is_member:
        print("You are a member, so you get a 10% discount!")
    else:
        print("You are not a member, but you can still participate.")
else:
    print("You are too young to participate.")

age = 16
is_member = False

if age >= 18:
    print("You are old enough to participate.")
    if is_member:
        print("You are a member, so you get a 10% discount!")
    else:
        print("You are not a member, but you can still participate.")
else:
    print("You are too young to participate.")
```

Listing 19: Nested if Statement

Output:

```
You are old enough to participate.
You are a member, so you get a 10% discount!
You are too young to participate.
```

Exercises:

1. Write a Python program that asks the user to enter a number and then prints whether the number is positive, negative, or zero.
2. Create a program that determines if a given year is a leap year. A leap year is divisible by 4, but not by 100 unless it is also divisible by 400.
3. Simulate a simple login system. Ask the user for a username and password. If the username is "admin" and the password is "password123", print "Login Successful!". Otherwise, print "Invalid Credentials."

2.2 Looping Statements: `for loop`, `while loop`

Definition: Looping statements (or iteration statements) allow a block of code to be executed repeatedly until a certain condition is met.

Explanation: Loops are essential for automating repetitive tasks. Python provides `for` and `while` loops for different iteration scenarios.

General Syntax:

```python
# for loop
for item in iterable:
    # code to execute for each item

# while loop
while condition:
    # code to execute as long as condition is True
```

Examples:

Example 1: `for` loop with `range()` Print numbers from 0 to 4.

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

Listing 20: for loop with range()

Output:

```
0
1
2
3
4
```

Example 2: `for` loop with a List Iterate over elements in a list.

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(f"I like {fruit}.")
```

Listing 21: for loop with a List

Output:

```
I like apple.
I like banana.
I like cherry.
```

Example 3: `while` loop Print numbers from 1 to 5 using a `while` loop.

```python
count = 1
while count <= 5:
    print(count)
    count += 1
```

Listing 22: while loop

Output:

```
1
2
3
4
5
```

Example 4: Infinite `while` loop (and how to stop it) Be careful with `while` loops! If the condition never becomes `False`, you'll have an infinite loop. You can stop it by pressing `Ctrl+C` in your terminal.

```python
# This would be an infinite loop if uncommented
# while True:
# print("This will print forever!")
```

Listing 23: Infinite while loop

Exercises:

1. Use a `for` loop to calculate the sum of all numbers from 1 to 100.

2. Write a `while` loop that keeps asking the user to enter a password until they enter "secret".

3. Print all even numbers between 1 and 20 (inclusive) using both a `for` loop and a `while` loop.

2.3 Loop Control Statements: `break`, `continue`, `pass`

Definition: Loop control statements alter the normal execution flow of loops.

Explanation:

- **`break`:** Terminates the loop entirely and transfers control to the statement immediately following the loop.
- **`continue`:** Skips the rest of the current iteration of the loop and proceeds to the next iteration.
- **`pass`:** A null operation; nothing happens when it executes. It can be used as a placeholder where a statement is syntactically required but you don't want any code to execute.

General Syntax:

```python
# break
for item in iterable:
    if condition:
        break
    # code after break

# continue
for item in iterable:
    if condition:
        continue
    # code after continue

# pass
if condition:
    pass # Do nothing for now
```

Examples:

Example 1: `break` Statement Search for a number in a list and stop when found.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7

for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break # Exit the loop once target is found
    print(f"Checking {num}...")
print("Loop finished.")
```

Listing 24: break Statement

Output:

```
Checking 1...
Checking 2...
Checking 3...
Checking 4...
Checking 5...
Checking 6...
Found 7!
Loop finished.
```

Example 2: `continue` Statement Print only odd numbers from a list.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
    if num % 2 == 0: # If number is even
        continue # Skip the rest of the current iteration
    print(f"{num} is an odd number.")
```

Listing 25: continue Statement

Output:

```
1 is an odd number.
3 is an odd number.
5 is an odd number.
7 is an odd number.
9 is an odd number.
```

Example 3: `pass` Statement Placeholder for future code.

```python
def my_function():
    pass # TODO: Implement this function later

class MyClass:
    pass # TODO: Define methods and attributes later

if 10 > 5:
    pass # Condition is true, but do nothing
else:
    print("This will not be printed.")

print("Program continues after pass statements.")
```

Listing 26: pass Statement

Output:

```
Program continues after pass statements.
```

Exercises:

1. Write a `for` loop that iterates through numbers from 1 to 20. Use `break` to stop the loop if the number is greater than 15.
2. Write a `for` loop that iterates through a list of words. Use `continue` to skip words that have fewer than 5 characters.
3. Explain a scenario where the `pass` statement would be useful in your code.

3. Functions

Definition: A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Explanation: Functions are essential for breaking down complex problems into smaller, manageable pieces. They allow you to write code once and use it multiple times, avoiding repetition and making your programs more efficient and easier to maintain.

General Syntax:

```python
def function_name(parameter1, parameter2, ...):
    """Docstring: Explain what the function does."""
    # function body
    # ...
    return value # Optional: returns a value
```

3.1 Overview, Arguments and Return Values

Explanation:

- **Defining a Function:** Functions are defined using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The code block within the function must be indented.
- **Calling a Function:** To execute a function, you simply type its name followed by parentheses `()`, optionally passing arguments.
- **Arguments (Parameters):** These are values passed into the function when it is called. They act as inputs to the function.

- **Return Values:** Functions can optionally return a value using the `return` keyword. If no `return` statement is present, the function implicitly returns `None`.

Examples:

Example 1: Simple Function with No Arguments and No Return Value

```python
def say_hello():
    print("Hello from a function!")

say_hello() # Calling the function
```

Listing 27: Simple Function

Output:

```
Hello from a function!
```

Example 2: Function with Arguments and No Return Value

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

greet_user("Alice")
greet_user("Bob")
```

Listing 28: Function with Arguments

Output:

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

Example 3: Function with Arguments and a Return Value

```python
def add_two_numbers(num1, num2):
    sum_result = num1 + num2
    return sum_result

result = add_two_numbers(5, 7)
print(f"The sum is: {result}")

print(f"Another sum: {add_two_numbers(10, 20)}")
```

Listing 29: Function with Return Value

Output:

```
The sum is: 12
Another sum: 30
```

Example 4: Function Returning Multiple Values Python functions can return multiple values as a tuple.

```python
def calculate_stats(numbers):
    total = sum(numbers)
    average = total / len(numbers)
    return total, average

scores = [85, 90, 78, 92, 88]
sum_scores, avg_scores = calculate_stats(scores)
print(f"Total scores: {sum_scores}, Average score: {avg_scores}")
```

Listing 30: Function Returning Multiple Values

Output:

```
Total scores: 433, Average score: 86.6
```

Exercises:

1. Write a function `get_square(number)` that takes one argument and returns its square.
2. Create a function `is_prime(num)` that checks if a given number is prime and returns `True` or `False`.
3. Write a function `format_name(first, last)` that takes a first name and a last name, and returns a formatted full name (e.g., "Doe, John").

3.2 Formal vs Actual Arguments

Definition:

- **Formal Arguments (Parameters):** These are the names of the variables listed in the function definition.
- **Actual Arguments (Arguments):** These are the actual values passed to the function when it is called.

Explanation: When a function is called, the actual arguments are assigned to the formal arguments in the order they are provided. Understanding this distinction is key to how data flows into functions.

General Syntax:

```python
def function_name(formal_arg1, formal_arg2): # Formal Arguments
    # ...

function_name(actual_value1, actual_value2) # Actual Arguments
```

Examples:

Example 1: Illustrating Formal and Actual Arguments

```python
def multiply(x, y): # x and y are formal arguments
    result = x * y
    print(f"Inside function: {x} * {y} = {result}")

a = 10
b = 5
multiply(a, b) # a and b are actual arguments
multiply(20, 3) # 20 and 3 are actual arguments
```

Listing 31: Formal and Actual Arguments

Output:

```
Inside function: 10 * 5 = 50
Inside function: 20 * 3 = 60
```

Exercises:

1. Define a function `display_info(name, age, city)`. Call this function with your own name, age, and city. Clearly identify the formal and actual arguments.
2. What happens if you call a function with fewer actual arguments than formal arguments? Try it and explain the error.

3.3 Named Arguments (Keyword Arguments)

Definition: Named arguments (or keyword arguments) are arguments passed to a function by explicitly naming the corresponding parameter.

Explanation: This allows you to pass arguments in any order, as long as you specify the parameter name. It also makes function calls more readable, especially for functions with many parameters.

General Syntax:

```python
def function_name(param1, param2, param3):
    # ...

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

Examples:

Example 1: Using Named Arguments

```python
def describe_car(make, model, year, color):
    print(f"Make: {make}")
    print(f"Model: {model}")
    print(f"Year: {year}")
    print(f"Color: {color}")

# Calling with positional arguments (order matters)
describe_car("Toyota", "Camry", 2020, "Blue")
print("\n---")

# Calling with named arguments (order doesn't matter, readability improves)
describe_car(model="Civic", make="Honda", color="Red", year=2022)
print("\n---")

# Mixing positional and named arguments (positional must come first)
describe_car("Ford", "Mustang", year=2023, color="Black")
```

Listing 32: Using Named Arguments

Output:

```
Make: Toyota
Model: Camry
Year: 2020
Color: Blue

---
Make: Honda
Model: Civic
Year: 2022
Color: Red

---
Make: Ford
Model: Mustang
Year: 2023
Color: Black
```

Exercises:

1. Create a function `create_email(recipient, subject, body, sender="noreply@example.com")`. Call this function using named arguments for `subject` and `body`, and let `sender` use its default value.
2. What happens if you try to put a positional argument after a named argument in a function call? Provide an example.

3.4 Recursive Functions

Definition: A recursive function is a function that calls itself during its execution.

Explanation: Recursion is a powerful programming technique where a problem is solved by breaking it down into smaller, similar subproblems until a base case is reached. The base case is a condition that stops the recursion.

General Syntax:

```python
def recursive_function(parameters):
    if base_case_condition:
        return base_case_value
    else:
        # Recursive step: call the function itself with modified parameters
        return recursive_function(modified_parameters)
```

Examples:

Example 1: Factorial Calculation Calculating the factorial of a number (n!) is a classic example of recursion.

- Base case: `factorial(0) = 1`
- Recursive step: `factorial(n) = n * factorial(n-1)`

```python
def factorial(n):
    if n == 0:
        return 1 # Base case
    else:
        return n * factorial(n-1) # Recursive step

print(f"Factorial of 5: {factorial(5)}") # 5 * 4 * 3 * 2 * 1 = 120
print(f"Factorial of 0: {factorial(0)}")
```

Listing 33: Factorial Calculation

Output:

```
Factorial of 5: 120
Factorial of 0: 1
```

Example 2: Fibonacci Sequence Generating the nth Fibonacci number (where `fib(n) = fib(n-1) + fib(n-2)`).

- Base cases: `fib(0) = 0`, `fib(1) = 1`

```python
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

```python

print(f"Fibonacci(6): {fibonacci(6)}") # 0, 1, 1, 2, 3, 5, 8
print(f"Fibonacci(0): {fibonacci(0)}")
print(f"Fibonacci(1): {fibonacci(1)}")
```

Listing 34: Fibonacci Sequence

Output:

```
Fibonacci(6): 8
Fibonacci(0): 0
Fibonacci(1): 1
```

Exercises:

1. Write a recursive function to calculate the sum of numbers from 1 to `n`.
2. Implement a recursive function to reverse a given string.
3. Explain the concept of a "base case" in recursion and why it is crucial.

3.5 Lambda Functions (Anonymous Functions)

Definition: A lambda function is a small, anonymous (nameless) function defined with the `lambda` keyword. It can take any number of arguments but can only have one expression.

Explanation: Lambda functions are typically used for short, simple operations where a full `def` function definition would be overkill. They are often used as arguments to higher-order functions (functions that take other functions as arguments), like `map()`, `filter()`, and `sorted()`.

General Syntax:

```python
lambda arguments: expression
```

Examples:

Example 1: Simple Lambda Function

```python
# A lambda function to add two numbers
add = lambda a, b: a + b
print(f"Sum using lambda: {add(10, 15)}")

# A lambda function to check if a number is even
is_even = lambda num: num % 2 == 0
print(f"Is 4 even? {is_even(4)}")
print(f"Is 7 even? {is_even(7)}")
```

Listing 35: Simple Lambda Function

Output:

```
Sum using lambda: 25
Is 4 even? True
Is 7 even? False
```

Example 2: Lambda with `filter()` Filter out even numbers from a list.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Even numbers: {even_numbers}")
```

Listing 36: Lambda with filter()

Output:

```
Even numbers: [2, 4, 6, 8, 10]
```

Example 3: Lambda with `map()` Square each number in a list.

```python
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x * x, numbers))
print(f"Squared numbers: {squared_numbers}")
```

Listing 37: Lambda with map()

Output:

```
Squared numbers: [1, 4, 9, 16, 25]
```

Example 4: Lambda with `sorted()` Sort a list of tuples based on the second element.

```python
pairs = [(1, 'b'), (3, 'a'), (2, 'c')]
sorted_pairs = sorted(pairs, key=lambda pair: pair[1])
print(f"Sorted pairs by second element: {sorted_pairs}")
```

Listing 38: Lambda with sorted()

Output:

```
Sorted pairs by second element: [(3, 'a'), (1, 'b'), (2, 'c')]
```

Exercises:

1. Use a lambda function with `map()` to convert a list of temperatures from Celsius to Fahrenheit (Formula: `F = C * 9/5 + 32`).

2. Use a lambda function with `filter()` to select all strings from a list that start with the letter 'A'.

3. Explain when you would choose to use a `def` function over a `lambda` function.

3.6 Modules

Definition: A module is a file containing Python definitions and statements. It allows you to logically organize your Python code. Related code is grouped into a module, making the code easier to understand and use.

Explanation: Modules provide a way to reuse code. Instead of writing all your code in one large file, you can break it down into smaller, more manageable modules. This promotes modularity, reusability, and makes your projects easier to maintain. Python comes with a vast standard library, which is a collection of built-in modules.

General Syntax:

```python
# Importing an entire module
import module_name

# Importing specific objects from a module
from module_name import object_name1, object_name2

# Importing all objects from a module (generally discouraged)
from module_name import *

# Importing a module with an alias
import module_name as alias
```

Examples:

Example 1: Importing and Using Built-in Modules (`math`, `random`)

```python
# Import the entire math module
import math

print(f"Value of PI: {math.pi}")
print(f"Square root of 16: {math.sqrt(16)}")

# Import specific functions from the random module
from random import randint, choice

print(f"Random integer between 1 and 10: {randint(1, 10)}")
my_list = ["apple", "banana", "cherry"]
print(f"Random choice from list: {choice(my_list)}")

# Import with an alias
import datetime as dt
now = dt.datetime.now()
print(f"Current date and time: {now}")
```

Listing 39: Built-in Modules

Output:

```
Value of PI: 3.141592653589793
Square root of 16: 4.0
Random integer between 1 and 10: 7 # (This will vary)
Random choice from list: banana # (This will vary)
Current date and time: 2025-09-07 10:30:00.123456 # (This will vary)
```

Example 2: Creating and Using User-Defined Modules
First, let's create a file named `my_calculations.py` with some functions:
File: `my_calculations.py`

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

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

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Error: Cannot divide by zero!"
    return a / b
```

Now, let's use this module in another Python script (e.g., `main_program.py`):
File: `main_program.py`

```python
# Import the entire user-defined module
import my_calculations

print(f"Using my_calculations.add: {my_calculations.add(10, 5)}")
print(f"Using my_calculations.subtract: {my_calculations.subtract(10, 5)}")

# Import specific functions from the user-defined module
from my_calculations import multiply, divide

print(f"Using multiply: {multiply(4, 6)}")
print(f"Using divide: {divide(10, 2)}")
print(f"Using divide by zero: {divide(10, 0)}")
```

Listing 40: main$_p$*rogram.py*

Output (when `main_program.py` is run):

```
Using my_calculations.add: 15
Using my_calculations.subtract: 5
Using multiply: 24
Using divide: 5.0
Using divide by zero: Error: Cannot divide by zero!
```

Exercises:

1. Create a module named `string_utils.py` that contains two functions:

   - `reverse_string(s)`: Takes a string `s` and returns its reversed version.
   - `is_palindrome(s)`: Takes a string `s` and returns `True` if it's a palindrome (reads the same forwards and backward, ignoring case), `False` otherwise.

   Then, in a separate Python script, import `string_utils` and test both functions with various strings.

2. Explore the `os` built-in module. Find a function that lists the contents of a directory and use it to print the files and folders in your current working directory.

3. Explain the difference between `import module_name` and `from module_name import function_name` in terms of how you access the functions/objects.