Basic Programming with Python

Prepared By:
Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

2

Introduction To Python Programming

CONTENTS

2.3. Elements of Python Program 1
2.3.1. Variables in Python 7
2.3.2. Built-in Data Types (Data Structures) 13
2.3.3. Input Function 17
2.3.4. Type Conversion 20
2.3.5. Operators in Python 22
2.3.6. Statements and Expressions 25
2.3.7 Keywords in Python 28

2.3. ELEMENTS OF PYTHON PROGRAM

The elements of a Python program are the building blocks that are used to create a complete and functional program. Some of the key elements of a Python program include:

1. **Data Types:** These are the different types of values that can be stored in variables. Some of the basic data types in Python include integers, floats, strings, and booleans. For example:

```
x = 10          # integer
```

```
y = 3.14         # float
z = "Hello"      # string
is_valid = True  # Boolean
```

2. **Operators:** Operators are special symbols used to perform operations on variables, such as addition, subtraction, multiplication, and division. These are used to perform operations on values, such as arithmetic or comparison operations. Some examples of operators in Python include +, -, *, /, ==, !=, <, >, <=, >=. For example:

```
x = 10
y = 5
print(x + y)   # 15
print(x * y)   # 50
print(x == y)  # False
```

3. **Expressions:** An expression is a combination of variables, operators, and data types that evaluates to a value. In Python, an expression is a combination of values, variables, operators, and function calls that are evaluated to produce a result. Some examples of expressions in Python:

   a. **Arithmetic expressions:** These are expressions that involve arithmetic operators such as addition, subtraction, multiplication, and division. For example:

```
x = 10
y = 5
print(x + y)      # 15
print(x - y)      # 5
print(x * y)      # 50
print(x / y)      # 2.0
```

   b. **Comparison expressions:** These are expressions that involve comparison operators such as ==, !=, <, >, <=, and >=. For example:

```
x = 10
y = 5
print(x == y)     # False
print(x != y)     # True
print(x > y)      # True
print(x <= y)     # False
```

   c. **Boolean expressions:** These are expressions that evaluate to either True or False. They are often used in if-else statements and loops. For example:

```
x = 10
y = 5
print(x > y and x < 20)    # True
print(x == y or x > 20)    # False
print(not(x == y))         # True
```

   d. **String expressions:** These are expressions that involve strings and string manipulation. For example:

```
name = "John"
age = 30
print("My name is " + name)                  # "My name is John"
print("I am " + str(age) + " years old")     # "I am 30 years old"
print(name.lower())                          # "john"
print(name.upper())                          # "JOHN"
print(len(name))                             # 4
```

e. **Function call expressions:** These are expressions that involve calling a function and passing in arguments. For example:

```
import math
x = 2.5
print(math.floor(x))     # 2
print(math.ceil(x))      # 3
print(math.sqrt(x))      # 1.5811388300841898
```

4. **Conditional Statements:** Conditional statements are used to control the flow of a program based on conditions, such as if a value is greater than another value. Conditional statements in Python are used to execute certain parts of code based on certain conditions. The most commonly used conditional statements in Python are if, elif, and else.
The following are some examples of conditional statements in Python:

a. if statement:

```
x = 10
if x > 5:
   print("x is greater than 5")
```

b. if-else statement:

```
x = 3
if x > 5:
   print("x is greater than 5")
else:
   print("x is less than or equal to 5")
```

c. if-elif-else statement:

```
x = 7
if x < 5:
   print("x is less than 5")
elif x < 10:
   print("x is between 5 and 10")
else:
   print("x is greater than or equal to 10")
```

d. Nested if statement:

```
x = 10
if x > 5:
   if x < 15:
```

```
        print("x is between 5 and 15")
```

5. **Switch statement in Python:** Python does not have a switch statement like some other programming languages. However, the same functionality can be achieved using a dictionary or if-elif-else statements.

   a. Using a dictionary:

```
def switch_case(case):
    switcher = {
        1: "Case 1",
        2: "Case 2",
        3: "Case 3",
    }
    return switcher.get(case, "Invalid case")

print(switch_case(1))    # "Case 1"
print(switch_case(4))    # "Invalid case"
```

   a. Using if-elif-else statements:

```
def switch_case(case):
    if case == 1:
        return "Case 1"
    elif case == 2:
        return "Case 2"
    elif case == 3:
        return "Case 3"
    else:
        return "Invalid case"

print(switch_case(1))    # "Case 1"
print(switch_case(4))    # "Invalid case"
```

6. **Loops:** Loops allow you to repeat a section of code multiple times, either until a condition is met or for a specified number of iterations. The two main types of loops in Python are for loops and while loops. Some examples of loops in Python include:

   a. for loop:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

   b. while loop:

```
x = 0
while x < 5:
    print(x)
    x += 1
```

   c. Loop control statements:

```
fruits = ["apple", "banana", "cherry"]
```

```
for fruit in fruits:
  if fruit == "banana":
    continue
  print(fruit)
```

d. Nested loops:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for a in adj:
  for f in fruits:
    print(a, f)
```

7. **Functions:** A function is a block of code that can be called from other parts of a program, allowing you to reuse code and make your program more modular. In Python, functions are a way to group related code and reuse it throughout a program. The following is an example of a simple function in Python:

```
def square(x):
  s = x ** 2
  return s
```

In this example, the square function takes a number as an argument and returns its square. The results are printed using the print function.

```
print(square(3))    # 9
print(square(5))    # 25
```

8. **Classes:** Classes are the building blocks of object-oriented programming in Python and allow you to create objects that can have properties and methods. In Python, a class is a blueprint for creating objects. Objects are instances of a class that can have their unique attributes and methods.
The following is an example of a simple class in Python:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def greet(self):
    print("Hello, my name is " + self.name + " and I am " + str(self.age) + " years old.")

person1 = Person("Afaz", 25)
person2 = Person("Rahman", 30)

person1.greet()
person2.greet()
```

In this example, the Person class has a constructor method __init__ that takes two arguments name and age, and initializes the name and age attributes of the object. The class also has a greet method that prints a personalized greeting using the name and age

attributes. Two Person objects are created with different names and ages, and the greet method is called on each object to print a personalized greeting.

9. **Exceptions:** Exceptions are events that occur during the execution of a program that can cause it to stop working correctly. Python provides a way to handle exceptions, allowing you to write robust code that can handle errors gracefully. Exceptions can be handled using try and except blocks.
   a. An example of **using try and except to handle a ZeroDivisionError:**

```
try:
  x = 1 / 0
except ZeroDivisionError:
  print("Cannot divide by zero.")
```

   In this example, the try block contains the code that might raise an exception, which in this case is dividing by zero. The except block catches the ZeroDivisionError exception and prints an error message.

   b. An example of handling an exception is **using try and except to handle a FileNotFoundError:**

```
try:
  with open("myfile.txt", "r") as f:
    data = f.read()
except FileNotFoundError:
  print("File not found.")
```

   In this example, the try block contains code that reads from a file named "myfile.txt". The except block catches the FileNotFoundError exception that might occur if the file does not exist and prints an error message.

   c. Exceptions can also be raised manually using the raise statement. The following is an example of **raising a ValueError exception:**

```
def divide(x, y):
  if y == 0:
    raise ValueError("Cannot divide by zero.")
  return x / y

try:
  result = divide(10, 0)
except ValueError as e:
  print(e)
```

   In this example, the divide function raises a ValueError exception if the y argument is zero. The try block calls the divide function with y = 0, which raises the ValueError exception. The except block catches the exception and prints the error message that was passed to the ValueError constructor.

2.3.1. Variables in Python

In Python, variables are used to store data values. They act as symbolic names that reference a location in memory where the data is stored. Variables allow us to manipulate and work with data within a program. A basic terminology of variables in Python is summaried below:

- **Variable Naming:** Variable names can contain letters, numbers, and underscores. They cannot start with a number. Variable names are case-sensitive. It's good practice to use descriptive names that indicate the purpose of the variable.
- **Variable Assignment:** Variables are created when they are first assigned a value. We can assign a value to a variable using the assignment operator, =.
- **Data Types:** Python is dynamically typed, meaning we don't need to declare the type of a variable. The type of the variable is determined by the value it holds. Common data types in Python include **integers, floats, strings, lists, tuples, dictionaries**, etc.
- **Variable Reassignment:** We can reassign a variable to a new value of a different type. The variable will simply reference the new value.
- **Scope:** Variables can have either local or global scope. **Local variables** are defined within a function and can only be accessed within that function. **Global variables** are defined outside of any function and can be accessed throughout the program.

Example 2.8: Demonstrating variable usage in Python.

```
# Variable assignment
x = 5
y = "Hello"

# Printing variables
print(x)
print(y)

# Variable reassignment
x = 10
y = "World"

# Printing variables after reassignment
print(x)
print(y)
```

In this example, x and y are variables assigned to the integer 5 and the string "Hello", respectively. Later, their values are reassigned to 10 and "World".

Output:

Types of Variable Assignment in Python:

In Python, there are various ways to assign values to variables, each with its own specific use case. The different types of variable assignments in Python:

- **Single Variable Assignment:** This is the most common type of assignment where a single value is assigned to a single variable.
  ```
  x = 10
  ```
- **Multiple Assignment:** Multiple variables can be assigned values in a single line.
  ```
  a, b, c = 1, 2, 3
  ```
- **Chained Assignment:** Values can be assigned to multiple variables in a single line using the same value.
  ```
  x = y = z = 0
  ```
- **Augmented Assignment:** This type of assignment is used to perform an operation on a variable and then assign the result back to the same variable.
  ```
  x = 5
  x += 2  # equivalent to x = x + 2
  ```
- **Unpacking Assignment:** Unpacking assignment is used to unpack iterables (like lists, tuples) into individual variables.
  ```
  values = (1, 2, 3)
  x, y, z = values
  ```
- **Extended Unpacking Assignment:** It allows unpacking of iterables with a variable number of elements, using the * operator.
  ```
  a, *b, c = [1, 2, 3, 4, 5]
  ```
- **Assignment with * (Starred Expression):** The * operator can also be used in assignment to collect multiple values into a list.
  ```
  x, *rest = range(5)
  ```
- **Assignment with ** (Double Starred Expression):** It's used to collect multiple keyword arguments into a dictionary.
  ```
  kwargs = {'a': 1, 'b': 2, 'c': 3}
  ```

Example 2.9: Demonstrating different types of variable assignments in Python.

```
# Single Variable Assignment
x = 10
print("Single Variable Assignment:", x)

# Multiple Assignment
a, b, c = 1, 2, 3
print("Multiple Assignment:", a, b, c)
```

```
# Chained Assignment
x = y = z = 0
print("Chained Assignment:", x, y, z)

# Augmented Assignment
x = 5
x += 2  # equivalent to x = x + 2
print("Augmented Assignment:", x)

# Unpacking Assignment
values = (1, 2, 3)
x, y, z = values
print("Unpacking Assignment:", x, y, z)

# Extended Unpacking Assignment
a, *b, c = [1, 2, 3, 4, 5]
print("Extended Unpacking Assignment:", a, b, c)

# Assignment with *
x, *rest = range(5)
print("Assignment with *:", x, rest)

# Assignment with **
kwargs = {'a': 1, 'b': 2, 'c': 3}
print("Assignment with **:", kwargs)
```

This code demonstrates each type of variable assignment in Python along with sample outputs.

Output:

Terminal

```
Single Variable Assignment: 10
Multiple Assignment: 1 2 3
Chained Assignment: 0 0 0
Augmented Assignment: 7
Unpacking Assignment: 1 2 3
Extended Unpacking Assignment: 1 [2, 3, 4] 5
Assignment with *: 0 [1, 2, 3, 4]
Assignment with **: {'a': 1, 'b': 2, 'c': 3}
```

Python Variable Types:

In Python, variables can have either local or global scope, determining where they can be accessed within a program.

- **Local Variables:** Local variables are defined within a function and can only be accessed from within that function. They are created when the function is called and destroyed when

the function exits. Local variables cannot be accessed from outside of the function in which they are defined.

```
def my_function():
    x = 10  # Local variable
    print("Inside the function:", x)

my_function()
# Accessing x outside the function will raise an error
```

- **Global Variables:** Global variables are defined outside of any function and can be accessed throughout the program. They can be accessed from within functions, but if we want to modify a global variable within a function, we need to explicitly declare it using the global keyword. Global variables are initialized before any function execution.

```
x = 10  # Global variable
def my_function():
    global x  # Declare x as global if you want to modify it
    x = 20
    print("Inside the function:", x)

my_function()
print("Outside the function:", x)  # Accessing x outside the function
```

- **Nonlocal Variables:** Nonlocal variables are used in nested functions where a variable in the outer (enclosing) function is redefined in the inner (nested) function. The **nonlocal keyword** is used to indicate that the variable is not local to the current function but is defined in an outer scope.

```
def outer_function():
    x = 10  # Local variable in outer_function

    def inner_function():
        nonlocal x  # Accessing the variable from the outer function
        x = 20
        print("Inside the inner function:", x)

    inner_function()
    print("Inside the outer function:", x)

outer_function()
```

Example 2.10: Demonstrating different types of variables usage in Python.

```
# Global variable
global_var = 1000

def my_function():
    # Local variable
    local_var = 200
```

```
    # Accessing local variable
    print("Local variable inside function:", local_var)

    # Accessing global variable inside function
    print("Global variable inside function:", global_var)

# Accessing global variable outside function
print("Global variable outside function:", global_var)

# Trying to access local variable outside function (will raise an error)
# print("Local variable outside function:", local_var)

# Calling the function
my_function()
```

In this example, global_var is a global variable defined outside of any function, and local_var is a local variable defined inside the my_function() function. Inside the function, we can access both local and global variables. Outside the function, we can only access the global variable.

Output:

```
Global variable outside function: 1000
Local variable inside function: 200
Global variable inside function: 1000
```

Example 2.11: Calculating the area and perimeter of a circle in Python.

```
# Global variable for pi
pi = 3.14159

def circle(radius):
    # Local variable for area and perimeter
    area = pi * radius ** 2
    peri = 2 * pi * radius
    print("Area: ", area)
    print("Perimeter: ", peri)

# Test the function
radius = float(input("Enter the value of radius: "))
circle(radius)
```

In this example, pi is a global variable, and area and peri are local variables inside the function circle. The circle function calculates the area and perimeter of a circle with a radius given by the user.

Output:

```
Terminal
Enter the value of radius: 5
Area:  78.53975
Perimeter:  31.4159
```

Variable Deletion in Python:

In Python, we can delete a variable using the del keyword. This removes the reference to the variable, allowing Python's garbage collector to reclaim the memory associated with the variable's value if there are no other references to it.

The following code illustrates how we can delete a variable:

```
x = 10
print(x)  # Output: 10

# Deleting the variable 'x'
del x

# Trying to access 'x' after deletion will result in an error
# print(x)  # This line will raise a NameError: name 'x' is not defined
```

In this example, the variable x is deleted using del x, and attempting to access x after deletion raises a NameError because the variable no longer exists.

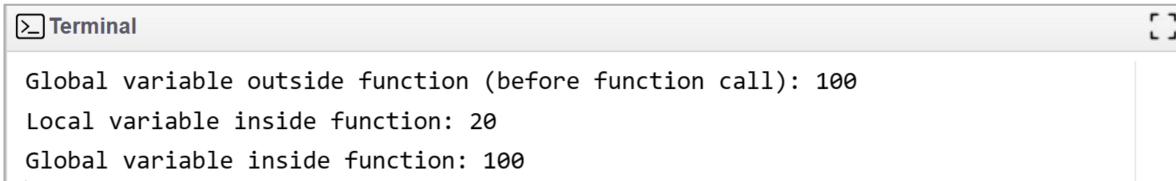
Example 2.12: Demonstrating scope of variable deletion in Python.

```
# Global variable
global_var = 100

def my_function():
   # Local variable
   local_var = 20

   # Accessing local variable
   print("Local variable inside function:", local_var)

   # Accessing global variable inside function
   print("Global variable inside function:", global_var)

   # Deleting local variable
   del local_var
   # print("Local variable inside function after deletion:", local_var)

# Example Usage
# Accessing global variable outside function
print("Global variable outside function (before function call):", global_var)
```

```
# Calling the function
my_function()
```

In this example, global_var is a global variable defined outside of any function, and local_var is a local variable defined inside the my_function() function. Inside the function, we can access both local and global variables. After deleting the local variable using **del local_var**, we cannot access it anymore, even within the same function or outside the function.

Output:



2.3.2. Built-in Data Types (Data Structures)

In Python, data types represent the type of data that can be stored and manipulated within a program. Some of the common data types in Python:

Numeric Types:

- **int:** Integer type represents whole numbers, positive or negative, without any decimal point.
- **float:** Float type represents floating-point numbers, which include decimal points.

```
x = 10  # int
y = 3.14  # float
```

String Type:

- **str:** String type represents a sequence of characters, enclosed within single, double, or triple quotes.

```
name = "Alice"  # str
```

Boolean Type:

- **bool:** Boolean type represents a binary value indicating True or False.

```
is_valid = True  # bool
```

Sequence Types:

- **list:** List represents an ordered collection of items. Lists are mutable.
- **tuple:** Tuple represents an ordered collection of items, similar to lists, but tuples are immutable.
- **range:** Range represents a sequence of numbers. It's commonly used for looping a specific number of times.

```
numbers = [1, 2, 3, 4, 5]  # list
point = (10, 20)  # tuple
count = range(1, 10)  # range
```

Mapping Type:

- **dict:** Dictionary represents a collection of key-value pairs. Keys must be unique within a dictionary, but values can be duplicated.

```
person = {"name": "Alice", "age": 30}  # dict
```

Set Types:

- **set:** Set represents an unordered collection of unique items.
- **frozenset:** Frozenset is an immutable version of set.

```
unique_numbers = {1, 2, 3, 4, 5}  # set
frozen_numbers = frozenset({1, 2, 3, 4, 5})  # frozenset
```

None Type:

- **None:** Represents the absence of a value.

```
result = None  # NoneType
```

Example 2.13: Demonstrating built-in data types in Python.

```
# Integer
x = 10
print("Integer:", x, type(x))

# Float
y = 3.14
print("Float:", y, type(y))

# String
name = "Alice"
print("String:", name, type(name))

# Boolean
is_valid = True
print("Boolean:", is_valid, type(is_valid))

# List
numbers = [1, 2, 3, 4, 5]
print("List:", numbers, type(numbers))

# Tuple
point = (10, 20)
print("Tuple:", point, type(point))

# Range
count = range(1, 10)
print("Range:", count, type(count))

# Dictionary
person = {"name": "Alice", "age": 30}
print("Dictionary:", person, type(person))
```

```
# Set
unique_numbers = {1, 2, 3, 4, 5}
print("Set:", unique_numbers, type(unique_numbers))

# Frozenset
frozen_numbers = frozenset({1, 2, 3, 4, 5})
print("Frozenset:", frozen_numbers, type(frozen_numbers))

# NoneType
result = None
print("NoneType:", result, type(result))
```

Output:

```
Terminal
Integer: 10 <class 'int'>
Float: 3.14 <class 'float'>
String: Alice <class 'str'>
Boolean: True <class 'bool'>
List: [1, 2, 3, 4, 5] <class 'list'>
Tuple: (10, 20) <class 'tuple'>
Range: range(1, 10) <class 'range'>
Dictionary: {'name': 'Alice', 'age': 30} <class 'dict'>
Set: {1, 2, 3, 4, 5} <class 'set'>
Frozenset: frozenset({1, 2, 3, 4, 5}) <class 'frozenset'>
NoneType: None <class 'NoneType'>
```

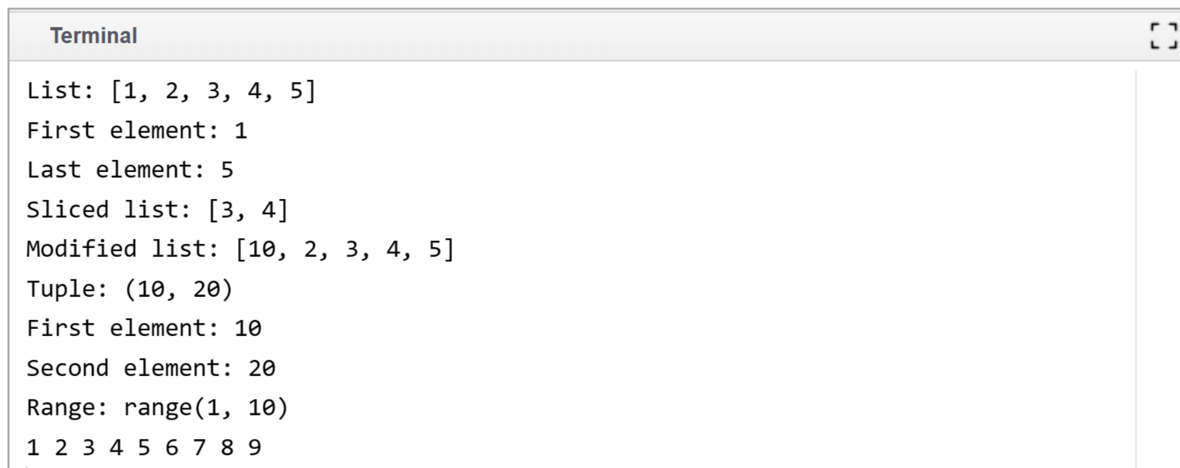
Example 2.14: Demonstrating the usage of sequence types, namely lists, tuples, and ranges.

```
# List
numbers = [1, 2, 3, 4, 5]
print("List:", numbers)

# Accessing elements of a list
print("First element:", numbers[0])
print("Last element:", numbers[-1])

# Slicing a list
print("Sliced list:", numbers[2:4])

# Modifying elements of a list
numbers[0] = 10
print("Modified list:", numbers)

# Tuple
```

```
point = (10, 20)
print("Tuple:", point)

# Accessing elements of a tuple
print("First element:", point[0])
print("Second element:", point[1])

# Range
count = range(1, 10)
print("Range:", count)

# Iterating over a range
for num in count:
   print(num, end=' ')
print()  # newline
```

This script demonstrates the creation, accessing, slicing, modification, and iteration over elements of lists, tuples, and ranges.

Output:



```
Terminal
List: [1, 2, 3, 4, 5]
First element: 1
Last element: 5
Sliced list: [3, 4]
Modified list: [10, 2, 3, 4, 5]
Tuple: (10, 20)
First element: 10
Second element: 20
Range: range(1, 10)
1 2 3 4 5 6 7 8 9
```

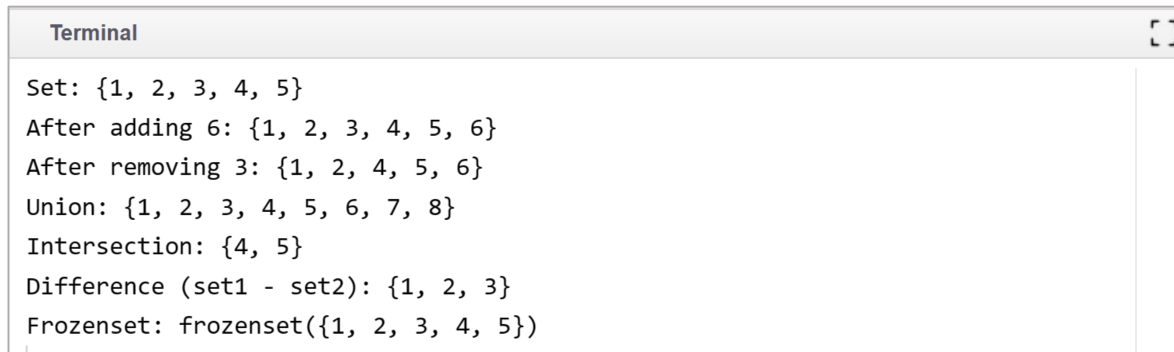
Example 2.15: Demonstrating the usage of set types in Python.

```
# Set
unique_numbers = {1, 2, 3, 4, 5}
print("Set:", unique_numbers)

# Adding elements to a set
unique_numbers.add(6)
print("After adding 6:", unique_numbers)

# Removing elements from a set
unique_numbers.remove(3)
print("After removing 3:", unique_numbers)
```

```
# Set operations
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# Union
union_set = set1.union(set2)
print("Union:", union_set)

# Intersection
intersection_set = set1.intersection(set2)
print("Intersection:", intersection_set)

# Difference
difference_set = set1.difference(set2)
print("Difference (set1 - set2):", difference_set)

# Frozenset
frozen_numbers = frozenset({1, 2, 3, 4, 5})
print("Frozenset:", frozen_numbers)
```

This script demonstrates the creation, addition, removal, and various operations (union, intersection, difference) on set types. Additionally, it shows the usage of a frozenset, which is an immutable version of a set.

Output:



2.3.3. Input Function

In Python, the **input()** function is used to accept user input from the keyboard as a string. It allows a program to pause and wait for the user to enter data, which can then be processed by the program. This is how the **input()** function works:

```
user_input = input("Enter something: ")
```

The **input()** function takes an optional prompt argument, which is a string that is displayed to the user before waiting for input. This prompt is optional.

When the **input()** function is called, the program stops executing and waits for the user to enter something from the keyboard. After the user enters input and presses the Enter key, the input is read by the **input()** function and returned as a string. The returned string can be assigned to a variable, as shown in the example above.

Note: The input() function always returns a string, even if the user enters a number or any other type of input. If you need the input to be interpreted as a different data type, such as an integer or a float, you need to explicitly convert it using functions like int() or float(). For example:

```
# Accepting an integer input
age = int(input("Enter your age: "))

# Accepting a float input
salary = float(input("Enter your salary: "))
```

If the user enters a non-numeric value when expecting a numeric input (e.g., entering "abc" instead of a number), attempting to convert it to an integer or float will raise a ValueError. Therefore, it's a good practice to handle potential errors when working with user input.

The following are several examples demonstrating different uses of the **input()** function in Python:

- **Simple input with prompt:**

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

- **Accepting numeric input and performing calculations:**

```
radius = float(input("Enter the radius of a circle: "))
area = 3.14 * radius ** 2
print("The area of the circle is:", area)
```

- **Accepting multiple inputs and storing them in variables:**

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello, " + name + "! You are " + str(age) + " years old.")
```

- **Using input in a loop to accept multiple inputs:**

```
numbers = []
for i in range(3):
  num = int(input("Enter number " + str(i+1) + ": "))
  numbers.append(num)
print("The numbers you entered are:", numbers)
```

- **Using input for conditional statements:**

```
grade = int(input("Enter your exam grade: "))
if grade >= 90:
  print("Your grade is A.")
elif grade >= 80:
  print("Your grade is B.")
elif grade >= 70:
  print("Your grade is C.")
else:
  print("Your grade is F.")
```

- **Using input in a while loop with error handling:**

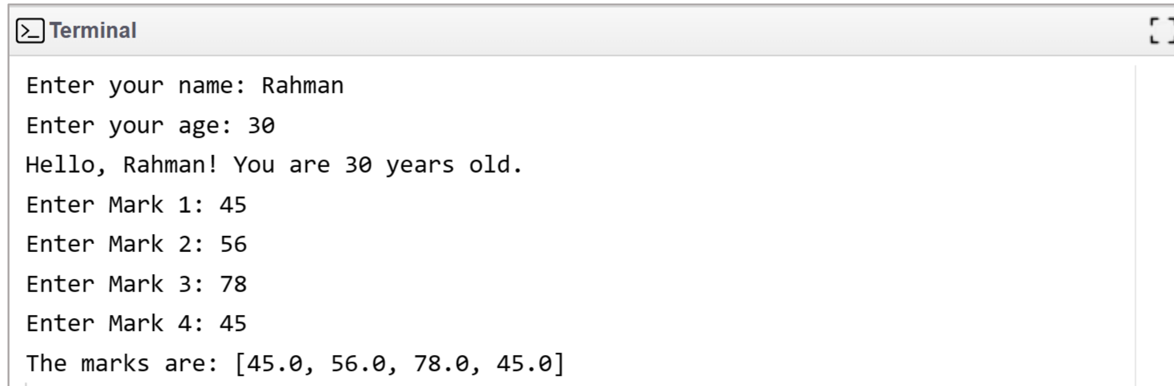
```
while True:
    try:
        num = int(input("Enter a number: "))
        print("You entered:", num)
        break
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```

Example 2.16: Demonstrating various uses of input function in Python.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello, " + name + "! You are " + str(age) + " years old.")

marks = []
for i in range(4):
    num = float(input("Enter Mark " + str(i+1) + ": "))
    marks.append(num)
print("The marks are:", marks)
```

These examples demonstrate various ways to use the input() function to interact with users, accept input, and perform different operations based on that input.

Output:



Example 2.17: Demonstrating input function with error handling in Python.

```
while True:
    try:
        num = int(input("Enter a number: "))
        print("You entered:", num)
        break
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```

This example demonstrates a way to use the input() function in Python for accepting user input, and handling errors.

Output:

```
Terminal
Enter a number: a
Invalid input. Please enter a valid number.
Enter a number: 12
You entered: 12
```

2.3.4. Type Conversion

Type conversion in Python refers to the process of converting one data type into another. Python provides built-in functions for performing type conversion, allowing us to convert data from one type to another as needed. This is particularly useful when we need to perform operations that require operands of the same type or when we need to format data for output.

In Python, there are primarily two types of type conversions:

1. **Implicit Type Conversion (Automatic Type Conversion):**
   - Implicit type conversion, also known as automatic type conversion, is performed by Python automatically during expressions evaluation.
   - Python automatically converts the data type of one operand to match the data type of the other operand in an expression.
   - Implicit type conversion occurs when the data types of the operands are compatible with each other.
   - For example:

```
# Implicit type conversion (int to float)
x = 10  # integer
y = 3.5  # float
z = x + y  # x is implicitly converted to float before addition
print(z)  # Output: 13.5
```

2. **Explicit Type Conversion (Type Casting):**
   - Explicit type conversion, also known as type casting, is performed explicitly by the programmer using built-in functions to convert data from one type to another.
   - It allows the programmer to manually convert data from one data type to another, even if the data types are not compatible.
   - Python provides built-in functions for explicit type conversion such as int(), float(), str(), bool(), etc.
   - For example:

```
# Explicit type conversion (str to int)
x = "10"  # string
y = int(x)  # x is explicitly converted to int
print(y)  # Output: 10

# Explicit type conversion (float to int)
x = 3.7  # float
y = int(x)  # x is explicitly converted to int
print(y)  # Output: 3
```

```
# Explicit type conversion (int to str)
x = 123  # integer
y = str(x)  # x is explicitly converted to string
print(y)  # Output: "123"

# Explicit type conversion (str to float)
x = "3.14"  # string
y = float(x)  # x is explicitly converted to float
print(y)  # Output: 3.14
```

Explicit type conversion allows the programmer to handle data effectively and perform operations that require operands of compatible types. It gives more control over data manipulation and transformation in Python.

The following are the commonly used type conversion functions in Python:

- **int(x):** Converts x to an integer. If x is a float, it truncates towards zero. If x is a string, it should represent an integer value.

```
x = 10.5
int_x = int(x)  # int_x will be 10
```

- **float(x):** Converts x to a floating-point number. If x is an integer or a string representing a number, it converts it to a float.

```
x = "3.14"
float_x = float(x)  # float_x will be 3.14
```

- **str(x):** Converts x to a string. It returns a string representation of the object.

```
x = 123
str_x = str(x)  # str_x will be "123"
```

- **bool(x):** Converts x to a Boolean value. Returns True if x is true, False otherwise. x can be any Python object.

```
x = 0
bool_x = bool(x)  # bool_x will be False
```

- **list(x):** Converts x to a list. It can convert sequences (like tuples and strings) or iterables into lists.

```
x = (1, 2, 3)
list_x = list(x)  # list_x will be [1, 2, 3]
```

- **tuple(x):** Converts x to a tuple. It can convert sequences (like lists and strings) or iterables into tuples.

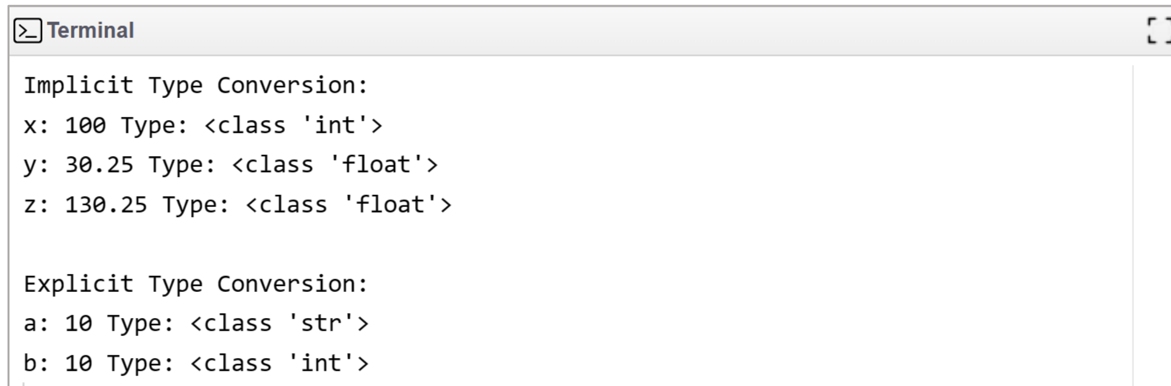
```
x = [1, 2, 3]
tuple_x = tuple(x)  # tuple_x will be (1, 2, 3)
```

Example 2.18: Demonstrating implicit and explicit type conversion in Python.

```
# Implicit Type Conversion (Automatic Type Conversion)
x = 100  # integer
y = 30.25  # float
z = x + y  # x is implicitly converted to float before addition

print("Implicit Type Conversion:")
print("x:", x, "Type:", type(x))
print("y:", y, "Type:", type(y))
print("z:", z, "Type:", type(z))
print()

# Explicit Type Conversion (Type Casting)
a = "10"  # string
b = int(a)  # a is explicitly converted to int
print("Explicit Type Conversion:")
print("a:", a, "Type:", type(a))
print("b:", b, "Type:", type(b))
```

In the implicit type conversion section, Python automatically converts the integer x to a float before performing the addition with the float y. In the explicit type conversion section, the string a is explicitly converted to an integer using the int() function. We print the types of variables before and after the conversion to demonstrate the type changes.

Output:



2.3.5. Operators in Python

In Python, operators are special symbols or keywords that perform operations on operands (variables or values). Python supports various types of operators, including arithmetic operators, assignment operators, comparison operators, logical operators, bitwise operators, and membership operators.

1. **Arithmetic Operators:** Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, modulus, and exponentiation.

   Examples: +, -, *, /, %, **

2. **Assignment Operators:** Assignment operators are used to assign values to variables. They also combine assignment with arithmetic operations.

   Examples: =, +=, -=, *=, /=, %= etc.

3. **Comparison Operators:** Comparison operators are used to compare two values. They return a Boolean value (True or False) based on the comparison result.

   Examples: == (equal), != (not equal), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to)

4. **Logical Operators:** Logical operators are used to combine conditional statements. They return a Boolean value (True or False) based on the logical relationship between the operands.

   Examples: and (logical AND), or (logical OR), not (logical NOT)

5. **Bitwise Operators:** Bitwise operators perform operations on individual bits of binary numbers.

   Examples: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift)

6. **Membership Operators:** Membership operators are used to test whether a value is present in a sequence (such as a string, list, or tuple).

   Examples: in (returns True if a value is present in the sequence), not in (returns True if a value is not present in the sequence)

7. **Identity Operators:** Identity operators are used to compare the memory location of two objects.

   Examples: is (returns True if both operands refer to the same object), is not (returns True if both operands do not refer to the same object)

Operators are fundamental building blocks in Python programming, allowing us to perform various operations and manipulate data effectively. Understanding and mastering operators is essential for writing efficient and concise Python code.

Example 2.19: Demonstrating various operators in Python.

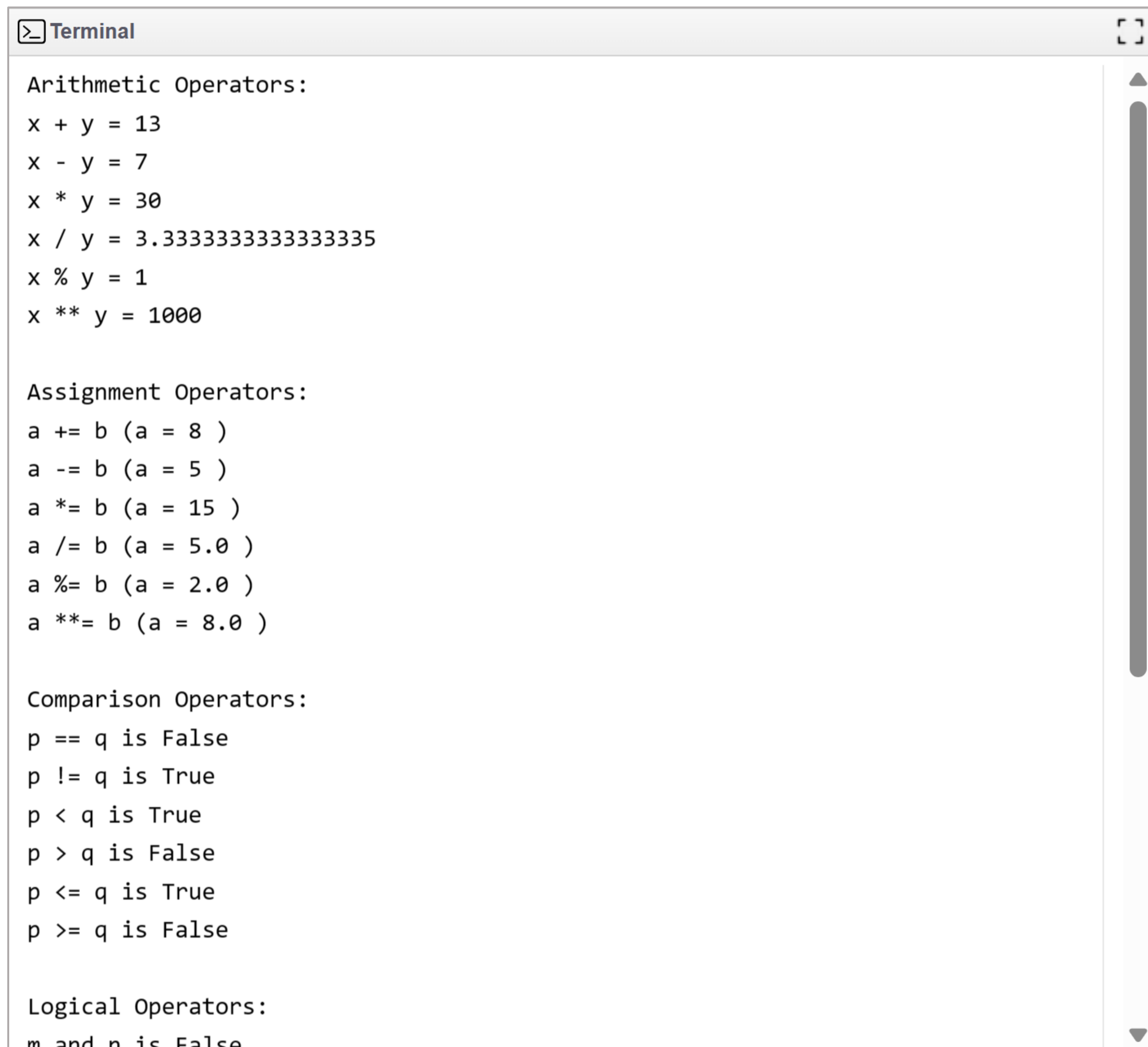
```
# Arithmetic operators
x = 10
y = 3

print("Arithmetic Operators:")
print("x + y =", x + y)  # Addition
print("x - y =", x - y)  # Subtraction
print("x * y =", x * y)  # Multiplication
print("x / y =", x / y)  # Division
print("x % y =", x % y)  # Modulus
print("x ** y =", x ** y)  # Exponentiation
print()

# Assignment operators
a = 5
b = 3

print("Assignment Operators:")
a += b  # Equivalent to: a = a + b
```

```
print("a += b (a =", a, ")")
a -= b  # Equivalent to: a = a - b
print("a -= b (a =", a, ")")
a *= b  # Equivalent to: a = a * b
print("a *= b (a =", a, ")")
a /= b  # Equivalent to: a = a / b
print("a /= b (a =", a, ")")
a %= b  # Equivalent to: a = a % b
print("a %= b (a =", a, ")")
a **= b  # Equivalent to: a = a ** b
print("a **= b (a =", a, ")")
print()

# Comparison operators
p = 10
q = 20

print("Comparison Operators:")
print("p == q is", p == q)  # Equal to
print("p != q is", p != q)  # Not equal to
print("p < q is", p < q)  # Less than
print("p > q is", p > q)  # Greater than
print("p <= q is", p <= q)  # Less than or equal to
print("p >= q is", p >= q)  # Greater than or equal to
print()

# Logical operators
m = True
n = False

print("Logical Operators:")
print("m and n is", m and n)  # Logical AND
print("m or n is", m or n)  # Logical OR
print("not m is", not m)  # Logical NOT
print()

# Bitwise operators
num1 = 10  # Binary: 1010
num2 = 5   # Binary: 0101

print("Bitwise Operators:")
print("num1 & num2 is", num1 & num2)  # Bitwise AND
print("num1 | num2 is", num1 | num2)  # Bitwise OR
print("num1 ^ num2 is", num1 ^ num2)  # Bitwise XOR
print("~num1 is", ~num1)  # Bitwise NOT
print("num1 << 2 is", num1 << 2)  # Left shift by 2 bits
print("num1 >> 2 is", num1 >> 2)  # Right shift by 2 bits
print()
```

```
# Membership operators
lst = [1, 2, 3, 4, 5]

print("Membership Operators:")
print("1 in lst is", 1 in lst)  # Check if 1 is in the list
print("6 not in lst is", 6 not in lst)  # Check if 6 is not in the list
```

This program demonstrates various arithmetic, assignment, comparison, logical, bitwise, and membership operators in Python. It performs operations and prints the results for each operator.

Output:



2.3.6. Statements and Expressions

In Python, statements and expressions are two fundamental concepts used in programming, but they serve different purposes:

1. **Statements:**

- A statement is a complete unit of execution that performs some action. It can consist of one or more expressions, variables, keywords, and operators.
- Statements in Python typically end with a newline character, but you can also use a semicolon (;) to write multiple statements on a single line.
- Examples of statements in Python include assignment statements, conditional statements (if-else), loop statements (for, while), import statements, function definitions, class definitions, etc.
- Statements can have side effects (e.g., changing the value of variables, printing output to the console, modifying data structures), and they contribute to the overall behavior of the program.
- Example of a statement:

```
x = 10
```

2. **Expressions:**
   - An expression is a combination of values, variables, operators, and function calls that evaluates to a single value. It represents a computation or calculation that Python can evaluate and produce a result.
   - Expressions can be simple, like a single variable or constant, or complex, involving multiple operands and operators.
   - Examples of expressions in Python include arithmetic expressions, logical expressions, string expressions, list comprehensions, lambda expressions, etc.
   - Expressions do not have side effects; they only produce a value when evaluated.
   - Example of an expression:

```
x + 5
```

In summary, statements are units of action that contribute to the program's logic and behavior, while expressions are units of computation that produce a single value when evaluated.

The following is an example of writing multiple statements on a single line:

```
x = 5; y = 10; z = x + y; print(z)
```

In this example, four statements are written on a single line. The first three statements assign values to variables x, y, and z. The fourth statement prints the value of z.

While this syntax is valid, it's not commonly used in Python code. Instead, it's more typical to write each statement on its own line:

```
x = 5
y = 10
z = x + y
print(z)
```

This makes the code easier to read and understand, especially for other developers who may work with the code in the future.

Example 2.20: Demonstrating statement and expression in Python.

```
# Assignment statement
```

```
x = 5

# Conditional statement (if-else)
if x > 0:
   print("x is positive")
else:
   print("x is non-positive")

# Loop statement (for loop)
for i in range(x):
   print("Iteration", i+1)

# Function definition statement
def greet(name):
   return "Hello, " + name + "!"

# Function call (expression)
message = greet("Rahman")
print(message)

# Arithmetic expression
result = 20 * (30 + 40)
print("Result of arithmetic expression:", result)

# Boolean expression
a = 100
b = 50
is_greater = a > b
print("Is a greater than b?", is_greater)
```

In this program, we have various types of statements, including assignment statements, conditional statements (if-else), loop statements (for loop), and function definition statements. We also have expressions, such as function calls, arithmetic expressions, and boolean expressions. Each statement performs a specific action, while expressions produce values when evaluated.

Output:

```
Terminal
x is positive
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Hello, Rahman!
Result of arithmetic expression: 1400
Is a greater than b? True
```

2.3.7 Keywords in Python

In Python, keywords are reserved words that have predefined meanings and purposes in the language. These keywords cannot be used as identifiers (such as variable names or function names) because they are part of the language's syntax and serve specific roles in defining the structure and behavior of Python programs. Some key points about keywords in Python are:

1. **Reserved Words:** Keywords are reserved words that are predefined in the Python language and cannot be redefined or used as identifiers.
2. **Predefined Functionality:** Each keyword has a predefined functionality or behavior associated with it. For example, the if keyword is used for conditional statements, for is used for loops, def is used for defining functions, and so on.
3. **Case Sensitivity:** Keywords in Python are case-sensitive, meaning that they must be written exactly as specified. For example, if is a keyword, but If or IF are not.
4. **Examples of Keywords:** Some common keywords in Python include if, else, elif, for, while, def, class, return, import, from, as, True, False, None, and, or, not, in, is, try, except, finally, raise, with, yield, global, nonlocal, etc.
5. **Help Documentation:** You can view the list of all keywords in Python using the keyword module. The **keyword.kwlist** attribute provides a list of all the keywords in Python. An example of how to list all the keywords in Python:

```
import keyword
print(keyword.kwlist)
```

6. **Avoiding Confusion:** Since keywords have predefined meanings and functionalities, it's important to avoid using them as identifiers (variable names, function names, etc.) to prevent confusion and ensure the clarity and readability of the code.

How many keywords in Python?

As of Python 3.10, there are 35 keywords in Python. These keywords are reserved for special purposes and cannot be used as identifiers such as variable names or function names. We can get the list of keywords in Python using the keyword module, as shown in the following example.

Example 2.21: Displaying a list of keywords in Python.

```
import keyword
print("Number of Keywords: ")
print(len(keyword.kwlist))  # Prints the number of keywords
print("List of Keywords:")
print(keyword.kwlist)  # Prints the list of keywords
```

When we run this code, we'll get the total number of keywords and a list of all the keywords in Python.

Output:

Terminal

```
Number of Keywords:
35
List of Keywords:
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
    'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
    'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
    'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

⌘⌘⌘