Python Specifications Document

Yousif Salah Mohammed
ID: 22P0232

Philopater Mina Adel
ID: 22P0250

Hams Hassan
ID: 22P0253

Jana Sameh Samir
ID: 22P0237

Ahmed Mohamed Lotfy
ID: 22P0251

Ahmed Mohamed Alamin
ID: 22P0137

Supervisor: Dr. Wafaa Samy
Professor, Computer and Systems Engineering

Co-supervisor: Eng. Mohammed Abdelmegeed
Teaching Assistant, Computer and Systems Engineering

Ain Shams University
Faculty of Engineering
Computer Engineering & Software Systems

CSE439: Design of Compilers
14 March 2025

Contents

List of Figures

List of Tables

1

Introduction

This document provides a comprehensive overview of the specifications of the Python programming language, detailing its syntax, semantics, built-in libraries, and core functionalities. Python, known for its simplicity and readability, is widely used in various domains, including web development, data science, artificial intelligence, and automation. This specification document serves as a reference for our project team, outlining the language's key features, standard conventions, and best practices to ensure complete knowledge in building the compiler.

2

Keywords & Variable Identifiers

2.1 Keywords

Keywords are reserved words in Python and cannot be used as variable names, function names, or any other identifiers.

Keywords in Python				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Table 2.1: *List of Keywords in Python*

2.2 Description of Keywords in Python

2.2.1 True, False

`True` and `False` are truth values in Python. They are the results of comparison operations or logical (Boolean) operations in Python.

2.2.2 None

`None` is a special constant in Python that represents the absence of a value or a null value. It is an object of its own datatype, the `NoneType`. We cannot create multiple `None` objects, but we can assign it to variables. These variables will be equal to one another.
We must take special care that `None` does not imply `False`, `0`, or any empty list, dictionary, string, etc.

2.2.3 and, or, and not

`and`, `or`, and `not` are the logical operators in Python, and will result in `True` only if both the operands are `True`.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Table 2.2: *Truth Table for AND*

The `or` operator will result in `True` if any of the operands is `True`. The truth table for `or` is given below:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Table 2.3: *Truth Table for OR*

The `not` operator is used to invert the truth value. The truth table for `not` is given below:

A	not A
True	False
False	True

Table 2.4: *Truth Table for NOT*

2.2.4 as

`as` is used to create an alias while importing a module. It means giving a different name (user-defined) to a module while importing it.

2.2.5 assert

`assert` is used for debugging purposes. If the condition is true, nothing happens. But if the condition is false, an `AssertionError` is raised.

2.2.6 async, await

The `async` and `await` keywords are provided by the `asyncio` library in Python. They are used to write concurrent code in Python.

2.2.7 break, continue

`break` and `continue` are used inside `for` and `while` loops to alter their normal behavior.

`break` will end the smallest loop it is in and control flows to the statement immediately below the loop. `continue` causes the current iteration of the loop to end, but not the whole loop.

2.2.8 class

`class` is used to define a new user-defined class in Python. Classes can be defined anywhere in a program.

2.2.9 def

`def` is used to define a user-defined function.

2.2.10 del

`del` is used to delete the reference to an object. Everything is an object in Python. We can delete a variable reference using `del`.

2.2.11 if, else, elif

`if`, `else`, and `elif` are used for conditional branching or decision-making. `if` and `elif` are used when we want to test a condition and execute a block only if the condition is true. `elif` is short for ”else if”. `else` is executed when all previous conditions are false.

2.2.12 except, raise, try

`except`, `raise`, and `try` are used with exceptions in Python. We can raise an exception explicitly with the `raise` keyword.

2.2.13 finally

`finally` is used with the `try...except` block to close up resources or file streams. Using `finally` ensures that the block of code inside it gets executed even if there is an unhandled exception.

2.2.14 for

`for` is used for looping. In Python, we can use it with any type of sequence, such as a list or a string.

2.2.15 from, import

The `import` keyword is used to import modules into the current namespace. The `from...import` syntax is used to import specific attributes or functions into the current namespace.

2.2.16 global

`global` is used to declare that a variable inside a function is global (exists outside the function).

2.2.17 in

`in` is used to test if a sequence (list, tuple, string, etc.) contains a value. It returns `True` if the value is present, else it returns `False`.

2.2.18 is

`is` is used in Python for testing object identity. The == operator checks if two variables are equal. The `is` operator checks if the two variables refer to the same object. It returns `True` if the objects are identical and `False` otherwise.

2.2.19 lambda

`lambda` is used to create an anonymous function (function with no name). It is an inline function that does not contain a return statement. It consists of an expression that is evaluated and returned.

2.2.20 nonlocal

The use of `nonlocal` is similar to `global`. `nonlocal` is used to declare that a variable inside a nested function (function inside a function) is not local to it, meaning it belongs to the enclosing function.

2.2.21 pass

`pass` is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

For example, if we have a function that is not implemented yet, simply writing it will cause an `IndentationError`. Instead, we use the `pass` statement.

2.2.22 return

`return` is used inside a function to exit it and return a value. If we do not explicitly return a value, `None` is returned automatically.

2.2.23 while

`while` is used for looping in Python. The statements inside a `while` loop continue to execute until the condition evaluates to `False` or a `break` statement is encountered.

2.2.24 with

The `with` statement is used to wrap the execution of a block of code within methods defined by the context manager.

A context manager is a class that implements `__enter__` and `__exit__` methods. Using `with` ensures that the `__exit__` method is called at the end of the block, similar to using a `try...finally` block.

2.2.25 yield

`yield` is used inside a function like a `return` statement. However, `yield` returns a generator.

A generator is an iterator that generates one item at a time. A large list of values would consume a lot of memory, whereas a generator produces only one value at a time instead of storing everything in memory.

2.3 Python Variable Names

A variable can have a short name (like `x` and `y`) or a more descriptive name (e.g., `age`, `carname`, `total_volume`).

2.3.1 Rules for Python Variables:

- A variable name must start with a letter or the underscore (`_`) character.
- A variable name cannot start with a number.
- A variable name can only contain alphanumeric characters and underscores (`A-Z`, `0-9`, and `_`).
- Variable names are case-sensitive (`age`, `Age`, and `AGE` are three different variables).

```
#Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

#Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"

```

3

Function Identifiers

Function identifiers are the names given to function. They are used to define, call, and reference functions in your code. Proper naming of functions is crucial for writing clean, readable, and maintainable code. This documentation explains the rules, best practices, and conventions for naming function identifiers in Python.

3.1 Rules for Naming Function Identifiers

3.1.1 Syntax Rules

- Function names must start with a letter (`a-z`, `A-Z`) or an underscore (`_`).
- The rest of the name can contain letters, numbers (`0-9`), and underscores.
- Function names are case-sensitive (`myFunction` and `myfunction` are different).
- Function names cannot be the same as Python keywords (e.g., `if`, `for`, `while`).

3.1.2 Examples

```
1  def my_function():  # Valid
2      pass
3
4  def _my_function():  # Valid (starts with an underscore)
5      pass
6
7  def myFunction():  # Valid (but not recommended; use snake_case)
8      pass
9
10 def 123_function():  # Invalid (starts with a number)
11     pass
12
13 def my-function():  # Invalid (contains a hyphen)
14     pass
```

3.2 Scope of Function Identifiers

3.2.1 Local Scope

- Function identifiers are local to the function in which they are defined.
- They cannot be accessed outside the function.

3.2.2 Global Scope

- If a function needs to modify a global variable, use the `global` keyword.
- Avoid overusing global variables, as they can lead to code that is hard to debug.

3.3 Function Identifier Naming Conventions

3.3.1 Standard Conventions

- Use **snake_case** for function names (e.g., `calculate_area`).
- Use verbs or verb phrases to describe actions (e.g., `get_user_data`, `validate_input`).
- Avoid using abbreviations unless they are widely understood (e.g., `calc` for `calculate`).

3.3.2 Special Cases

- For private functions (intended for internal use), prefix the name with an underscore (`_`).
- For magic methods (e.g., `__init__`, `__str__`), use double underscores (`__`).

4

Data Types

4.1 Data Types in Python

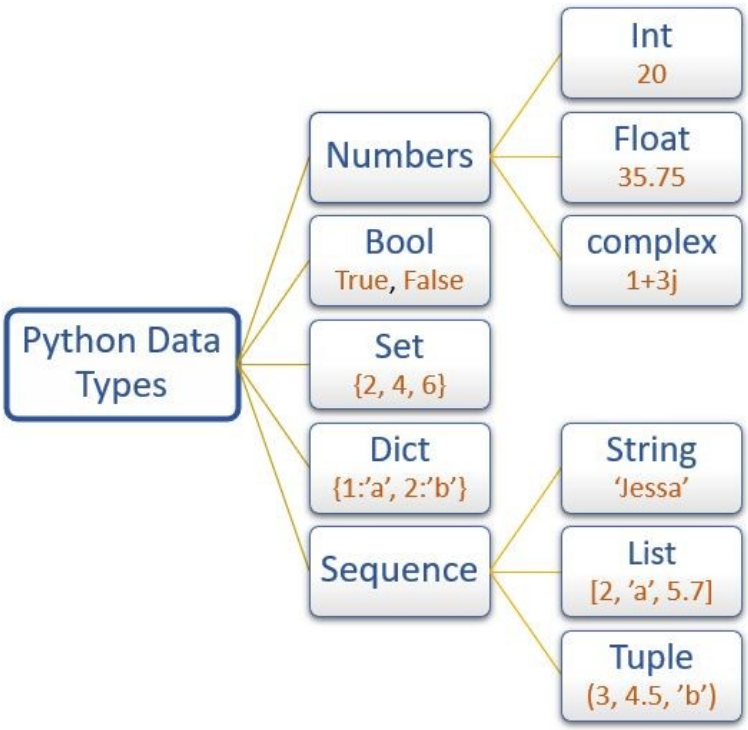

Figure 4.1: *Python Data Types*

4.2 Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python `int`, Python `float` and Python `complex` classes in Python.

4.2.1 int (Integer)

- **Description**: Represents whole numbers (positive, negative, or zero).
- **Use Case**: Counting, indexing, and mathematical operations.
- **Characteristics**:
  1. Unlimited precision (can handle very large numbers).
  2. Immutable (cannot be changed after creation).
- **Example**:

```
a = 10
b = -5
c = 0
```

4.2.2 float (Floating-Point Number)

- **Description**: Represents real numbers with decimal points.
- **Use Case**: Scientific calculations, measurements, and financial computations.
- **Characteristics**:
  1. Stored as double-precision (64-bit) floating-point numbers.
  2. Immutable.
- **Example**:

```
pi = 3.14159
temperature = -10.5
```

4.2.3 complex (Complex Number)

- **Description**: Represents numbers with real and imaginary parts.
- **Use Case**: Advanced mathematics, engineering, and physics.
- **Characteristics**:
  1. Immutable.
  2. Written as $a + bj$, where a is the real part and b is the imaginary part.
- **Example**:

```
z = 2 + 3j
```

4.3 bool (Boolean)

4.3.1 Description

- The `bool` data type represents one of two possible values: `True` or `False`.

- Booleans are a subclass of integers in Python, where `True` is equivalent to 1 and `False` is equivalent to 0.

4.3.2 Use Cases

- **Conditional Statements**: Used in `if`, `elif`, and `else` statements to control program flow.
- **Logical Operations**: Used with logical operators (`and`, `or`, `not`) to evaluate expressions.
- **Loop Control**: Used in `while` loops to determine whether to continue iterating.
- **Data Filtering**: Used to filter data in lists, dictionaries, or other collections.

4.3.3 Characteristics

1. **Immutability:** Booleans are immutable, meaning their value cannot be changed after creation.
2. **Subclass of int:** `True` and `False` are instances of `int` in Python.
3. **Truthy and Falsy Values:** In Python, non-Boolean values can be evaluated in a Boolean context. Values like `0`, `""`, `None`, `[]`, and `{}` are considered falsy, while most other values are truthy.

4.3.4 Example

```
1 is_valid = True
2 is_empty = False
3 print(bool(0))      # False
4 print(bool(1))      # True
```

4.4 set (Set)

A set is a collection of unique data, meaning that elements within a set cannot be duplicated.

4.4.1 Description

- A set is an unordered collection of unique elements.
- Sets are mutable, meaning you can add or remove elements after creation.
- Sets are commonly used for:
    - Removing duplicates from a collection.
    - Performing mathematical set operations like union, intersection, and difference.
    - Checking membership efficiently.

4.4.2 Characteristics

1. **Uniqueness:** Sets automatically eliminate duplicate elements.
2. **Unordered:** Sets do not maintain the order of elements.
3. **Mutability:** Sets are mutable, so you can add or remove elements.
4. **Hashable Elements:** Set elements must be hashable (immutable types like `int`, `str`, `tuple`). Lists and dictionaries are not allowed as set elements.

4.4.3 Creating a Set in Python

- In Python, the most basic and efficient method for creating a set is using curly braces.

4.4.4 Adding Elements to a Set in Python

- We can add items to a set using the `add()` method and `update()` method. The `add()` method can be used to add only a single item. To add multiple items we use the `update()` method.

4.4.5 Removing Elements from the Set in Python

- We can remove an element from a set in Python using several methods: `remove()`, `discard()` and `pop()`.

4.4.6 Accessing a Set in Python

- We can loop through a set to access set items as a set is unindexed and does not support accessing elements by indexing.
- We can also use the `in` keyword, which is a membership operator, to check if an item exists in a set.

4.4.7 Frozen Sets in Python

A `frozenset` in Python is a built-in data type that is similar to a set but with one key difference: immutability. This means that once a `frozenset` is created, we cannot modify its elements—that is, we cannot add, remove, or change any items in it. Like regular sets, a `frozenset` cannot contain duplicate elements.

If no parameters are passed, it returns an empty `frozenset`.

4.5 dict (Dictionary)

4.5.1 Description

- A dictionary is an unordered collection of key-value pairs.

- Keys must be unique and immutable (e.g., `int`, `str`, `tuple`), while values can be of any type (e.g., `int`, `list`, `dict`).
- Dictionaries are mutable, meaning you can add, modify, or remove key-value pairs after creation.

4.5.2 Characteristics

1. **Unordered:** Dictionaries do not maintain the order of elements (prior to Python 3.7). Starting from Python 3.7, dictionaries maintain insertion order as an implementation detail, and this became part of the language specification in Python 3.8.
2. **Mutable:** You can add, modify, or remove key-value pairs after creation.
3. **Fast Lookups:** Dictionaries are optimized for fast lookups using keys (average time complexity of $O(1)$ for lookups, insertions, and deletions).
4. **Dynamic:** Dictionaries can grow or shrink in size as needed.

4.5.3 Create a Dictionary

- A dictionary can be created by placing a sequence of elements within curly {} braces, separated by a comma.

4.5.4 Accessing Dictionary Items

- We can access a value from a dictionary by using the key within square brackets or `get()` method.

4.5.5 Adding and Updating Dictionary Items

- We can add new key-value pairs or update existing keys by using assignment.

4.5.6 Removing Dictionary Items

- We can remove items from a dictionary using the following methods:
  - `del`: Removes an item by key.
  - `pop()`: Removes an item by key and returns its value.
  - `clear()`: Empties the dictionary.
  - `popitem()`: Removes and returns the last key-value pair.

4.5.7 Iterating Through a Dictionary

- We can iterate over keys [using `keys()` method], values [using `values()` method] or both [using `items()` method] with a `for` loop.

4.6 Sequence Data Types

4.6.1 Strings (str)

Description

- A string is an immutable sequence of Unicode characters.
- Strings are used to represent text data.

Characteristics

1. **Immutable:** Once created, a string cannot be modified.
2. **Indexing and Slicing:** Strings support indexing and slicing.
3. **Concatenation:** Strings can be concatenated using the + operator.
4. **Repetition:** Strings can be repeated using the * operator.

Creating a String

- Strings can be created using either single (') or double (") quotes.

Accessing characters in Python String

- Strings in Python are sequences of characters, so we can access individual characters using indexing. Strings are indexed starting from 0 and -1 from the end. This allows us to retrieve specific characters from the string.

4.6.2 Lists (list)

Description

- A list is a mutable sequence of elements.
- Lists can contain elements of different data types.

Characteristics

1. **Mutable:** Lists can be modified after creation.
2. **Dynamic:** Lists can grow or shrink in size.
3. **Heterogeneous:** Lists can contain elements of different types.

Creating a List

- Lists can be created using square brackets.

Accessing List Elements

- Elements in a list can be accessed using indexing. Python indexes start at 0, so `a[0]` will access the first element, while negative indexing allows us to access elements from the end of the list. Like index -1 represents the last element of the list.

Adding Elements into List

- We can add elements to a list using the following methods:
  - `append()`: Adds an element at the end of the list.
  - `extend()`: Adds multiple elements to the end of the list.
  - `insert()`: Adds an element at a specific position.

Removing Elements from List

- We can remove elements from a list using:
  - `remove()`: Removes the first occurrence of an element.
  - `pop()`: Removes the element at a specific index or the last element if no index is specified.
  - `del` statement: Deletes an element at a specified index.

4.6.3 Tuples (tuple)

Description

- A tuple is an immutable sequence of elements.
- Tuples are often used for fixed collections of items.

Characteristics

1. **Immutable:** Tuples cannot be modified after creation.
2. **Ordered:** Elements are ordered and can be accessed using indices.
3. **Heterogeneous:** Tuples can contain elements of different types.

5

Functions

In Python, functions are fundamental building blocks that allow you to organize and reuse code.

5.1 What are Python Functions

- Functions are blocks of code designed to perform specific tasks. Instead of writing the same code repeatedly, you can encapsulate it within a function and call it whenever needed.
- They promote modularity by breaking down complex programs into smaller, manageable units. This makes code easier to read, understand, and maintain.
- Functions can accept input values (arguments or parameters) and return output values.

5.2 Types of Functions in Python

5.2.1 Built-in Functions

- **Description**:
  - These are functions that are readily available in Python without requiring any additional imports.
- **Examples**:
  - `print()`: Outputs the specified object(s) to the standard output device (the screen).
  - `len()`: Returns the length (number of items) of an object (string, list, tuple, etc.).
  - `type()`: Returns the type of an object.
  - `range()`: Generates a sequence of numbers.
  - `abs()`: Returns the absolute value of a number.
  - `sum()`: Returns the sum of the items in an iterable (list, tuple, etc.).
  - `max()`: Returns the largest item in an iterable.

- `min()`: Returns the smallest item in an iterable.
- `input()`: Reads a line from standard input (the keyboard) and returns it as a string.

5.2.2 User-defined Functions

- **Description**:
  - These are functions that you create to perform specific tasks tailored to your program's needs, you define them using the `def` keyword.
- **Examples**:
  - `is_even()`: This function takes a number as input, It uses the modulo operator (%) to check if the number is divisible by 2, It returns `True` if it's even, and `False` if it's odd, This demonstrates a function that returns a boolean value.

```python
def is_even(number):
    """This function checks if a number is even."""
    if number % 2 == 0:
        return True
    else:
        return False

# Calling the function and using the result
num = 7
if is_even(num):
    print(num, "is even.")
else:
    print(num, "is odd.")
```

  - `calculate_rectangle_area()`: takes two parameters, length and width, It calculates the area by multiplying them, The `return` statement sends the calculated area back to where the function was called, This example shows a function that returns a value.

```python
def calculate_rectangle_area(length, width):
    """This function calculates the area of a rectangle."""
    area = length * width
    return area

# Calling the function and storing the result
rectangle_area = calculate_rectangle_area(5, 10)
print("The area of the rectangle is:", rectangle_area)
```

  - `greet()`: This function takes one parameter `name`, It then prints a greeting message that includes the provided name, This demonstrates a basic function that performs a simple action.

```python
# Function to greet a person
def greet(name):
    """This function greets the person passed in as a parameter."""
    print("Hello, " + name + "!")

# Calling the function
greet("Wafaa")
```

5.2.3 Lambda Functions (Anonymous Functions)

- **Description**:
  - These are small, single-expression functions defined using the `lambda` keyword. They are often used for short, simple operations.
- **Examples**:
  - **`Adding Two Numbers`**: This lambda function takes two arguments, `x` and `y`, and returns their sum, It's assigned to the variable `add`, which can then be called like a regular function.

```python
add = lambda x, y: x + y
result = add(5, 3)
print(result)  # Output: 8
```

  - **`Squaring a Number`**: This lambda function takes one argument, `x`, and returns its square.

```python
square = lambda x: x * x
result = square(4)
print(result)  # Output: 16
```

  - **`filter()`**: This function constructs an iterator from elements of an iterable for which a function returns true, Here, we use a lambda function to filter out only the even numbers from the `numbers` list.

```python
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

  - **`map()`**: This function applies a given function to each item of an iterable (like a list), Here, we use a lambda function to square each number in the `numbers` list.

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

```
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

5.2.4 Recursive Functions

- **Description**:
  - These are functions that call themselves within their own definition. This is very useful for problems that can be broken down into smaller, self similar problems.
- **Examples**:
  - **Factorial Calculation**: The factorial of a number `n` is the product of all positive integers less than or equal to `n`, The function checks if `n` is 0 (base case). If it is, it returns 1 (because 0! = 1), Otherwise, it returns `n` multiplied by the factorial of `n - 1`. This is the recursive step, Each recursive call reduces `n` until it reaches the base case.

```
def factorial(n):
    "Calculates the factorial of a non-negative integer."
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)  # Output: 120 (5! = 5 * 4 * 3 * 2 * 1)
```

  - **Fibonacci Sequence**: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, The function checks if `n` is 0 or 1 (base cases). If it is, it returns `n`, Otherwise, it returns the sum of the (`n-1`)th and (`n-2`)th Fibonacci numbers, Again, this breaks the problem into smaller self similar problems.

```
def fibonacci(n):
    "Calculates the nth Fibonacci number."
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

result = fibonacci(6)
print(result)  # Output: 8 (0, 1, 1, 2, 3, 5, 8)
```

  - **Sum of List Elements**: The function checks if the list is empty (base case). If it is, it returns 0, Otherwise, it returns the first element of the list plus the

sum of the rest of the list (obtained by recursively calling the function with the sublist `my_list[1:]`.)

```
def sum_list(my_list):
    "Calculates the sum of elements in a list."
    if not my_list:
        return 0
    else:
        return my_list[0] + sum_list(my_list[1:])

numbers = [1, 2, 3, 4, 5]
result = sum_list(numbers)
print(result)  # Output: 15
```

5.3 Key Concepts

- **def Keyword**: Used to define a function.
- **Function Name**: A unique identifier for the function.
- **Parameters (Arguments)**: Input values passed to the function.
- **return Statement**: Used to return a value from the function. If no return statement is present, the function returns `None`.

6

Statements

6.1 Assignment Statement

In Python, assignment statements are fundamental for binding names (variables) to objects. They're how you store and manipulate data.

6.1.1 Basic Assignment

- The most common form uses the equals sign (=) to assign a value to a variable, this creates a reference from the variable name to the object in memory.
  - `x = 10`
  - `name = "Python"`
  - `my_list = [1, 2, 3]`

6.1.2 Key Characteristics

- **Object References**: Python assignments create references to objects, not copies of them (in most cases). This is a crucial concept to understand.
- **Dynamic Typing**: Python is dynamically typed, meaning you don't have to declare the variable's type beforehand. The type is determined at runtime based on the assigned value.
- **Variable Creation**: If a variable doesn't exist, the assignment statement creates it. If it does exist, the assignment rebinds it to the new object.

6.1.3 Common Assignment Forms

Multiple Assignment

- You can assign the same value to multiple variables simultaneously.
  - `x = y = 5`
- You can assign multiple values to multiple variables at the same time.
  - `x, y, z = 1, 2, 3`

Augmented Assignment

- These operators combine an arithmetic operation with an assignment.
  - `+=`: Addition assignment (e.g., `x += 5` is equivalent to `x = x + 5`)
  - `-=`: Subtraction assignment (e.g., `y -= 2` is equivalent to `y = y - 2`)
  - `*=`: Multiplication assignment (e.g., `z *= 3` is equivalent to `z = z * 3`)
  - `/=`: Division assignment (e.g., `a /= 4` is equivalent to `a = a / 4`)
  - `//=`: Floor division assignment (e.g., `b //= 2` is equivalent to `b = b // 2`) - Integer division.
  - `%=`: Modulus assignment (e.g., `c %= 3` is equivalent to `c = c % 3`) - Remainder.
  - `**=`: Exponentiation assignment (e.g., `d **= 2` is equivalent to `d = d ** 2`)

Sequence Unpacking

- You can unpack the elements of a sequence (like a tuple or list) into individual variables.
  - `a, b = (10, 20)`

6.2 Declarations

In Python, declaration statements are not explicitly defined as a separate concept like in some other programming languages. Instead, variable declarations are done implicitly during variable assignments. Python is dynamically typed, meaning variables don't need to be declared with a type upfront, and the type is inferred from the assigned value.

6.2.1 Variable Declaration through Assignment

A variable in Python is declared by simply assigning a value to it. There is no need for a specific declaration statement like `int x;` in languages like C++ or Java.

- **Basic Variable Declaration**

```
x = 10  # Here, x is declared and assigned the value 10 (an integer)
name = "John"  # name is declared and assigned the value "John" (a
↪ string)
```

- **Declaring Multiple Variables at Once**

```
a, b, c = 5, 10, 15
print(a, b, c)  # Output: 5 10 15
```

- **No Need for Explicit Data Type Declaration**
  Since Python is dynamically typed, the type of variable is automatically inferred based on the assigned value. There's no need to specify types during declaration.

```
x = 42  # x is an integer
x = 3.14  # x is now a float
x = "Hello"  # x is now a string
```

6.2.2 String declaration

- Using Single or Double Quotes
  You can use either single (`' '`) or double (`" "`) quotes to define a string.

```
string1 = 'Hello'
string2 = "World"
print(string1, string2)  # Output: Hello World
```

- Using Triple Quotes (`'''` or `"""`)
  Triple quotes are used for multi-line strings.

```
multiline_string = '''This is
a multi-line
string.'''
print(multiline_string)
```

6.2.3 Declaring conditional statements

- Basic if Statement
  - If the condition is True, the indented code runs.
  - If False, nothing happens.

```
age = 18

if age >= 18:
    print("You are an adult.")
```

- If else statements

```
age = 16

if age >= 18:
    print("You are an adult.")
```

```
else:
    print("You are a minor.")
```

- if-elif-else Statement

```
age = 70

if age < 18:
    print("You are a minor.")
elif age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

Indentation is very important in Python because it defines the structure of the code. Unlike other languages that use `{}` or `;` to separate code blocks, Python relies on indentation.

6.2.4 Declaring functions

In Python, functions are declared using the `def` keyword.

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

greet()  # Calling the function
```

6.2.5 Lists

Lists are used to store multiple items in a single variable. Use square brackets to create lists.

```
my_list = [1, 2, 3, 4, 5]
print(my_list)  # Output: [1, 2, 3, 4, 5]
```

6.2.6 Tuples

Tuples are like lists but cannot be changed (immutable). Use normal brackets to create tuples.

```
my_tuple = (10, 20, 30)
print(my_tuple)  # Output: (10, 20, 30)
```

6.2.7 Sets

Sets store unique values without duplicates. Use curly brackets to create sets.

```
my_set = {1, 2, 3, 4, 4, 5}
print(my_set)  # Output: {1, 2, 3, 4, 5}
```

6.2.8 Classes and Objects

Use the `class` keyword then a colon.

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

p1 = Person("Alice", 25)
print(p1.name)  # Output: Alice
```

6.3 Return Statement

A return statement is used to end the execution of the function call and it "returns" the value of the expression following the return keyword to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value `None` is returned. A return statement is overall used to invoke a function so that the passed statements can be executed.

6.3.1 Syntax

```
def function_name(parameters):
    # Function body
    return value
```

When the return statement is executed, the function terminates and the specified value is returned to the caller. If no value is specified, the function returns `None` by default.

6.3.2 Returning Multiple Values

Python allows you to return multiple values from a function by returning them as a tuple:

```python
def fun():
    name = "Alice"
    age = 30
    return name, age

name, age = fun()
print(name) # Output: Alice
print(age)  # Output: 30
```

6.3.3 Returning List

We can also return more complex data structures such as lists or dictionaries from a function:

```python
def fun(n):
    return [n**2, n**3] # Output: [9, 27]
```

6.3.4 Function returning another function

In Python, functions are first-class citizens, meaning you can return a function from another function. This is useful for creating higher-order functions.

```python
def fun1(msg):
    def fun2():
        # Using the outer function's message
        return f"Message: {msg}"
    return fun2

# Getting the inner function
fun3 = fun1("Hello, World!")

# Calling the inner function
print(fun3()) # Output: Message: Hello, World!
```

6.4 Conditional Statements

Conditional statements in Python allow for decision-making by enabling the execution of specific code blocks based on evaluated conditions. Without conditional statements, every line of code would execute sequentially without logical branching, making it difficult to handle decision-making scenarios efficiently. Python provides a structured approach to conditional logic through compound statements. The primary conditional constructs in Python include:

6.4.1 if Statement

Executes a designated block of code if a specified condition evaluates to `True`. The `if` statement serves as the foundation of decision-making structures in Python. It is one of the simplest forms of conditional execution.

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

6.4.2 if-else Statement

Provides an alternative execution branch when the condition evaluates to `False`.

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

6.4.3 if-elif-else Statement

Evaluates multiple conditions sequentially and executes the first block where the condition is `True`. This prevents unnecessary evaluations by stopping once a valid condition is found, improving efficiency in decision-making.

```
x = 10
if x < 5:
    print("x is less than 5")
elif x == 10:
    print("x is exactly 10")
else:
    print("x is greater than 5 but not 10")
```

6.4.4 Nested if Statements

`if` statements can be nested within other `if` statements to allow for hierarchical decision-making.

```
x = 10
if x > 5:
    if x < 15:
        print("x is between 5 and 15")
```

6.4.5 Ternary Operator (Conditional Expression)

A conditional operator with compact syntax for simple `if-else` statements, enhancing readability and making assignments more compact. It is important to note that conditional expressions have the lowest priority of all Python operations.

```
x = 10
result = "Positive" if x > 0 else "Negative"
print(result)
```

6.4.6 Match-Case Statement

`match` statements provide a structured approach to pattern matching, it checks a given value against a set of predefined patterns and executes code based on the first matching pattern. Python also allows for a default pattern called the wildcard pattern, represented using `'_'`. It will always succeed by matching anything and binds no name.

```
def process_command(command):
    match command:
        case "start":
            print("Starting the system...")
        case "stop":
            print("Stopping the system...")
        case "restart":
            print("Restarting the system...")
        case _:
            print("Unknown command")

process_command("start")
```

6.5 Iterative Statements

Iterative statements, also known as loops, facilitate repeated execution of a specific code block as long as a given condition holds true.

6.5.1 for Loop

Iterates over iterable data structures such as lists, tuples, dictionaries, and strings. `for` loops are mainly used for sequential traversal

Basic Iteration

A simple `for` loop iterating over a range of numbers:

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

Iterating Over a List

The `for` loop can iterate over elements in a list:

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

Iterating Over a Tuple

Tuples, being immutable, can also be iterated using a `for` loop:

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)
```

Iterating Over a Dictionary

To iterate through keys, values, or key-value pairs in a dictionary:

```
person = {"name": "Alice", "age": 25, "city": "New York"}

# Iterating over keys
for key in person:
    print(key, "->", person[key])

```

```
# Iterating over key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
```

Iterating Over a String

Strings are iterable, meaning each character can be accessed sequentially:

```
word = "Python"
for letter in word:
    print(letter)
```

Using the `enumerate()` Function

The `enumerate()` function allows iteration while keeping track of the index:

```
languages = ["Python", "Java", "C++"]
for index, language in enumerate(languages):
    print(f"Index {index}: {language}")
```

6.5.2 while Loop

Continuously executes a block of code as long as the specified condition remains `True`.

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

`while` loops can also have an else statement which executes after the condition becomes `False`

```
x = 0
while x < 5:
    print(x)
    x += 1
else:
    print("x is now equal to 5")
```

6.5.3 Nested Loops

Loops can be nested to allow for extended behavior. In this example, the nested for loop allows iteration across multiple dimensions or layers.

```
for i in range(3):
    for j in range(2):
        print(f"i: {i}, j: {j}")
```

6.5.4 List Comprehensions with Loops

Python allows compact iterations using list comprehensions, which provide a more efficient way to generate lists dynamically.

```
# Creating a list of squares using a for loop
squares = [x**2 for x in range(6)]
print(squares)
# Output: [0, 1, 4, 9, 16, 25]
```

6.6 Function Call Statements

In Python, a function is called using its name followed by parentheses. If the function accepts parameters, they are provided inside the parentheses.

6.6.1 Passing Arguments to Functions

Arguments are values passed to a function when calling it. Multiple arguments are separated by commas.

```
def greet(name):
    print("Hello,", name)

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

6.6.2 Number of Arguments

By default, a function must be called with the exact number of arguments it expects; otherwise, an error occurs.

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

```

print(add(2, 3))     # Correct
print(add(2))        # Error: missing argument
print(add(2, 3, 4)) # Error: too many arguments
```

6.6.3 Arbitrary Arguments (*args)

Using an asterisk (*) before a parameter name allows the function to accept any number of positional arguments as a tuple.

```
def sum_all(*numbers):
    return sum(numbers)

print(sum_all(1, 2, 3, 4))  # Output: 10
```

6.6.4 Keyword Arguments

Arguments can be passed using key=value syntax, which allows changing the order of parameters.

```
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduce(age=25, name="Alice")  # Order doesn't matter
```

6.6.5 Arbitrary Keyword Arguments (**kwargs)

Using double asterisks (**) allows passing named arguments as a dictionary.

```
def person_info(**info):
    print(info)

person_info(name="Alice", age=25, city="NY")
# Output: {'name': 'Alice', 'age': 25, 'city': 'NY'}
```

6.6.6 Default Parameter Values

If an argument is not provided, the function uses a default value.

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

```
3
4 greet()           # Output: Hello, Guest!
5 greet("Alice")    # Output: Hello, Alice!
```

6.6.7 Passing Lists and Other Data Types

A list (or any data type) passed to a function remains the same type inside the function.

```
1 def print_items(items):
2     for item in items:
3         print(item)
4
5 print_items(["apple", "banana", "cherry"])
```

6.6.8 Returning Values

A function returns a value using the return statement.

```
1 def square(n):
2     return n * n
3
4 result = square(4)
5 print(result)  # Output: 16
```

6.6.9 The pass Statement

If a function is defined but not yet implemented, `pass` can be used as a placeholder.

```
1 def future_function():
2     pass  # Does nothing
```

6.6.10 Positional-Only Arguments (/,)

Adding / in the function definition means all preceding arguments must be passed positionally.

```
1 def divide(a, b, /):
2     return a / b
3
4 divide(10, 2)    # Correct
5 divide(a=10, b=2)  # Error: keyword arguments not allowed
```

6.6.11 Keyword-Only Arguments (*)

Adding * before arguments means all following arguments must be passed using keywords.

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

greet(name="Alice")  # Correct
greet("Alice")       # Error: positional argument not allowed
```

6.6.12 Combining Positional-Only and Keyword-Only Arguments

Both can be used together.

```
def example(a, /, b, *, c):
    print(a, b, c)

example(1, 2, c=3)  # Correct
example(a=1, b=2, c=3)  # Error: a must be positional
```

6.6.13 Recursion

A function can call itself (recursion). This is useful for problems like factorial or Fibonacci sequences.

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

7

Expressions

7.1 Arithmetic Expressions

Python supports various arithmetic operations, including basic mathematical operations, exponentiation, and modulo. These operations work with integers, floating-point numbers, and complex numbers.

7.1.1 Basic Arithmetic Operators

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division (Floating-point)	5 / 2	2.5

Table 7.1: *Basic Arithmetic Operators in Python*

Example:

```
a = 10
b = 5
print(a + b)  # Output: 15
print(a - b)  # Output: 5
print(a * b)  # Output: 50
print(a / b)  # Output: 2.0
```

7.1.2 Floor Division (//)

- Returns the quotient rounded down to the nearest integer.
- Works with both integers and floats.

Example:

```
print(10 // 3)  # Output: 3
print(10.5 // 2)  # Output: 5.0
```

7.1.3 Modulo (%)

- Returns the remainder of division.

Example:

```
print(10 % 3)  # Output: 1
```

7.1.4 Exponentiation (**)

- Raises a number to a power.

Example:

```
print(2 ** 3)  # Output: 8
print(9 ** 0.5)  # Output: 3.0 (square root)
```

7.1.5 Operator Precedence (Order of Execution)

Python follows PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction):

Precedence	Operators
1 (Highest)	() (Parentheses)
2	** (Exponentiation)
3	*, /, //, % (Multiplication, Division, Modulo)
4 (Lowest)	+, - (Addition, Subtraction)

Table 7.2: *Operator Precedence in Python*

Example:

```
print(2 + 3 * 4)   # Output: 14 (Multiplication first)
print((2 + 3) * 4) # Output: 20 (Parentheses first)
```

7.1.6 Unary Operations

- `+x`: Positive sign (no effect)
- `-x`: Negation (changes sign)

Example:

```
print(+5)  # Output: 5
print(-5)  # Output: -5
```

7.1.7 Augmented Assignment Operators

These combine an arithmetic operation with assignment.

Operator	Equivalent To	Example
+=	a = a + b	a += 5
-=	a = a - b	a -= 3
*=	a = a * b	a *= 2
/=	a = a / b	a /= 4
//=	a = a // b	a //= 3
%=	a = a % b	a %= 2
**=	a = a ** b	a **= 3

Table 7.3: *Augmented Assignment Operators*

Example:

```
x = 10
x += 5  # x = x + 5
print(x)  # Output: 15
```

7.1.8 Working with Different Data Types

- Integers (`int`): Whole numbers (5, -10).
- Floating-point (`float`): Decimal numbers (5.2, -3.14).
- Complex Numbers (`complex`): Includes imaginary numbers (2 + 3j).

Operations with mixed types follow type conversion rules:

```
print(5 + 2.5)  # Output: 7.5 (int + float → float)
print(2 * 3j)   # Output: 6j (int * complex → complex)
```

7.1.9 Handling Division by Zero

- `/` and `//` raise a `ZeroDivisionError` when dividing by zero.
- `%` also raises a `ZeroDivisionError` if the divisor is zero.

Example:

```
print(5 / 0)  # Raises ZeroDivisionError
print(5 % 0)  # Raises ZeroDivisionError
```

7.2 Boolean Expressions

A Boolean expression evaluates to either `True` or `False`. These expressions are commonly used in conditional statements, loops, and logical operations.

7.2.1 Boolean Values

Python provides two built-in Boolean values:

```
True  # Boolean value representing truth
False  # Boolean value representing falsehood
```

These are case-sensitive (`true` and `false` are not valid in Python).

Example:

```
print(True)   # Output: True
print(False)  # Output: False
```

7.2.2 Boolean Operators

Python supports three logical operators:

Operator	Description	Example	Output
and	Logical AND	True and False	False
or	Logical OR	True or False	True
not	Logical NOT	not True	False

Table 7.4: *Boolean Operators in Python*

7.2.3 Comparison Operators

Comparison operators return `True` or `False` based on conditions.

Operator	Description	Example	Output
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	5 <= 4	False

Table 7.5: *Comparison Operators in Python*

Example:

```
a = 10
b = 5

print(a > b)  # Output: True
print(a == b) # Output: False
print(a != b) # Output: True
```

7.2.4 Operator Precedence in Boolean Expressions

Python evaluates Boolean expressions using precedence rules:

Precedence	Operators
1 (Highest)	`not`
2	`and`
3 (Lowest)	`or`

Table 7.6: *Operator Precedence in Boolean Expressions*

Example:

```
print(True or False and False)  # Output: True (AND before OR)
print(not False and True)       # Output: True (NOT before AND)

# To override precedence, use parentheses:
print((True or False) and False)  # Output: False
```

7.2.5 Boolean Short-Circuiting

Python uses short-circuit evaluation:

- In `x and y`, if `x` is `False`, `y` is not evaluated.
- In `x or y`, if `x` is `True`, `y` is not evaluated.

Example:

```
def test():
    print("Function executed")
    return True

print(False and test())  # Output: False (test() is never called)
print(True or test())    # Output: True (test() is never called)
```

7.2.6 Chained Comparisons

Python allows multiple comparisons in one statement.

Example:

```
x = 5
print(1 < x < 10)  # Output: True (equivalent to 1 < x and x < 10)
```

This works similarly for other comparison operators.

7.2.7 Combining Comparisons with Boolean Operators

Example:

```
age = 25
is_adult = age >= 18 and age < 65

print(is_adult)  # Output: True
```

7.2.8 Identity Operators (is, is not)

Used to compare whether two objects refer to the same memory location.

Operator	Description	Example	Output
`is`	Identity check	`x is y`	True if x and y refer to the same object
`is not`	Not identical	`x is not y`	True if x and y are different objects

Table 7.7: *Identity Operators in Python*

Example:

```
a = [1, 2, 3]
b = a  # `b` points to the same object as `a`
c = [1, 2, 3]  # New object

print(a is b)   # Output: True
print(a is c)   # Output: False
```

7.2.9 Membership Operators (in, not in)

Used to check if a value is inside a collection.

Operator	Description	Example	Output
`in`	Value exists	`"a" in "apple"`	True
`not in`	Value does not exist	`"b" in "apple"`	False

Table 7.8: *Membership Operators in Python*

Example:

```
print(3 in [1, 2, 3])    # Output: True
print("x" not in "text") # Output: False
```

8

Conclusion

Python's versatility, ease of use, and extensive ecosystem make it a powerful tool for developers across different industries. By adhering to the specifications outlined in this document, we can write efficient, scalable, and high-quality code that aligns with best practices. As Python continues to evolve, maintaining a clear understanding of its core specifications will be essential for leveraging its full potential in software development.