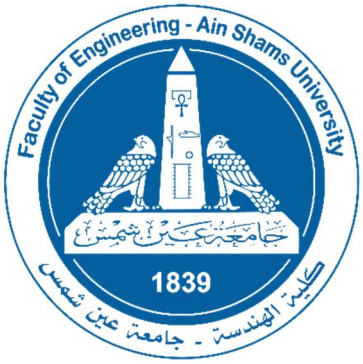


Python Specifications Document

CSE439: Design of Compilers
Team 10

MEMBERS

Adham Hisham Kandil

22P0217

Mohamed Yehia Zakria

22P0064

Jana Hany El Shafie

22P0235

Seif ElHusseiny

22P0215

Mohamed Ahmed Abdelhamid

22P0287

Moaz Mohamed Zakaria

22P0307

Represented to

Dr. Wafaa Samy
Professor, Computer and Systems Engineering

Eng. Mohammed Abdelmegeed
Teaching Assistant, Computer and Systems Engineering

Table of Contents

INTRODUCTION

This document presents a detailed specification of the Python programming language, covering its syntax, semantics, built-in libraries, and core functionalities. Renowned for its simplicity and readability, Python is widely utilized across various fields, including web development, data science, artificial intelligence, and automation. Serving as a reference for our project team, this specification outlines key language features, standard conventions, and best practices to ensure a thorough understanding for compiler development.

1 KEYWORDS & VARIABLE IDENTIFIERS

1.1 Keywords

The identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
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
```

Each keyword has a distinct role in Python programming, defining the structure and logic of programs.

Common Python Keywords and Their Uses:

1. **True, False** – Represent Boolean values.
2. **None** – Represents the absence of a value (null).
3. **and, or, not** – Logical operators for Boolean expressions.
4. **if, else, elif** – Implements conditional statements.
5. **except, raise, try, finally** – Used for exception handling.
6. **return** – Specifies the return value of a function.
7. **def** – Declares a function.
8. **for, while** – Implements loops for iteration.
9. **rom, import** – Used for importing modules.

There is description for the use of other keywords, I listed these because I will use them in the following Examples

1.2 Variables

A variable can have a simple short name (alphabetically like a, b, x , y) or a more descriptive one like (name , job , CalculateAVG, etc....)

Syntax when choosing a variable Name

- **Must start with a letter (A-Z, a-z) or an underscore (_)**
- **Can only contain letters, digits (0-9), and underscores (_)**
- **Cannot start with a number**
- **Cannot contain spaces** (Use underscores instead)
- **Cannot use special characters (@, $, %, etc.)**
- **Case-sensitive** (Python treats uppercase and lowercase letters differently)
- **Cannot be a Python keyword (e.g., if, for, while)**

Valid Statements ✅	***Invalid Statements*** ❌
my_var = 10	2name = "A"
_count = 5	user name = 3
price_2024 = 99	for = 10
UserAge = 30	total-price = 15
PI = 3.14159	@total = 100
num1 = 10	my-var = 50

2 FUNCTION IDENTIFIERS

Function identifiers are the names assigned to functions, allowing them to be defined, called, and referenced within code. Choosing appropriate function names is essential for ensuring clarity, readability, and maintainability. This documentation outlines the rules, best practices, and conventions for naming function identifiers in Python.

- **Must start with a letter (a-z, A-Z) or an underscore (_)**
- **Can contain letters, numbers (0-9), and underscores (_)**
- **Case-sensitive (Python treats uppercase and lowercase as different)**
- **Cannot use Python keywords (e.g., if, for, while)**

Examples:

valid

- def my_function(): #starts with a letter
- def _calculate_total(): #start with underscore

- def get_user_info():
- def sum_of_2_numbers(): # can contain numners

- def MYFUNCTION(): # python treates upper & lower case differently
- def myfunction()

Invalid

- def for(): # cannot use keywords
- def return():

2.1 Scope of Function Identifiers

The **scope** of a function identifier refers to the area of the program where the function name is accessible. In Python, function identifiers can have **local** or **global** scope depending on how and where they are defined.

2.1.1 Local Scope **(Function-Level Scope)**

A function identifier defined inside a function has **local scope**. It can only be accessed within the function where it is defined and cannot be used outside of it.

2.1.2 Global Scope

A function defined at the top level of a script or module has **global scope**. It can be called and accessed from anywhere in the program.

2.1.3 Modifying Global Variables Inside a Function

By default, functions cannot modify global variables unless explicitly declared using the global keyword

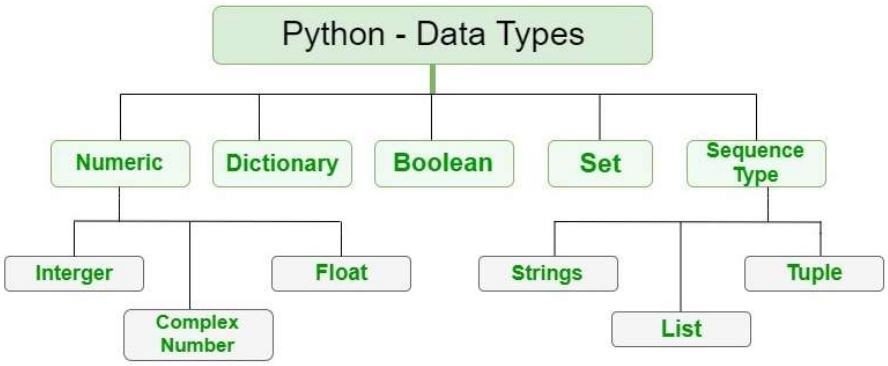
2.1.4 Nonlocal Scope (Nested Functions)

In nested functions, a variable declared in the **outer function** but not global can be modified using nonlocal. This allows the inner function to modify a variable from its enclosing function

```
# Global variable (accessible anywhere)
global_var = "I am global"

def outer_function():
    outer_var = "I am outer"  # Local to outer_function

    def inner_function():
        nonlocal outer_var  # Refers to outer_function's variable
        outer_var = "Modified by inner function"
        print("Inside inner_function:", outer_var)

        # Accessing global variable inside inner_function
        global global_var
        global_var = "Modified globally"
        print("Inside inner_function (global):", global_var)

    inner_function()
    print("Inside outer_function:", outer_var)

# Calling the functions
print("Before function calls:", global_var)
outer_function()
print("After function calls:", global_var)

```

3 DATA TYPES

Python provides various built-in data types that define the type of values a variable can hold. These data types can be categorized into the following groups:



3.1 Numeric Data Types

- int (Integer): Whole numbers (e.g., 10, -5).
- float (Floating-point): Numbers with decimals (e.g., 3.14, -0.99)
- complex: Numbers with real and imaginary parts (e.g., 2 + 3j).

```
a = 10      pi = 3.14159
b = -5      temperature = -10.5
c = 0
```

3.2.Boolean Data Type

Represents True or False values and is a subclass of int,
where True == 1 and False == 0.

```
is_valid = True
is_empty = False
```

3.3 Sequence Data Types

- **Strings (`str`)**: . Immutable sequences of Unicode characters used to store text Strings are indexed starting from 0 and -1 from the end. This allows us to retrieve specific characters from the string

```
name = "Alice"
print(name[0])  # Output: A
```

- **Lists (`list`)**: Ordered, mutable collections that can hold elements of different types.

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

- **Tuples (`tuple`)**: Ordered, immutable collections used for fixed collections of items.

```
coordinates = (10, 20)
print(coordinates[0])  # Output: 10
```

3.4 Set Data Types

- **Set (set)**: Unordered collection of unique elements, useful for eliminating duplicates and performing set operations.

Example:

```
unique_numbers = {1, 2, 3, 3}
print(unique_numbers)  # Output: {1, 2, 3}
```

- **Frozen Set (frozenset)**: An immutable version of a set.

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

3.5 Dictionary Data Type

Dictionary (dict): Collection of key-value pairs with fast lookups and dynamic size.

Example:

```
student = {"name": "Alice", "age": 25}
print(student["name"])  # Output: Alice
```

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

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

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 Functions

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

- 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.

4.1 Types of Functions in Python

4.1.1 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.

4.1.2 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

Example:
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.

```
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)
```

4.1.3 Lambda Functions (Anonymous Functions)

• Description:

- These are small, single-expression functions defined using the lambda keyword. They are often used for short, simple operations.

Example:
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.

```
add = lambda x, y: x + y
result = add(5, 5)
print(result) # Output: 10
```

4.1.4 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.

Example:
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)
```

5 Statements

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

5.1 Assignment Statements

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]**

5.1.1 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**

5.1.2 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)
- **%=** Modulus assignment (e.g., c %= 3 is equivalent to c = c % 3).
- ****=** Exponentiation assignment (e.g., d **= 2 is equivalent to d = d **

5.1.3 Sequence Unpacking

You can unpack the elements of a sequence (like a tuple or list) into individual variables.

- **a, b = (10, 20)**

5.2 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.

5.2.1 Declaring Variables via Assignment

```
city = "New York"
price = 19.99
```

5.2.2 String Initialization

- Using Single or Double Quotes:

```
message1 = 'Hello'
message2 = "World"
```

- Using Triple Quotes for Multi-line Strings:

```
long_text = '''Python supports multi-line strings.
They are useful for documentation.'''
```

5.2.3 Declaring Conditional Statements

- *Using an if Statement:*

```
score = 75
if score >= 50:
  print("Pass")
```

- *Using if-else for Decision Making:*

```
time = 18
if time < 12:
  print("Good morning")
else:
  print("Good evening")
```

5.2.4 Defining Functions

Functions encapsulate reusable logic.

```
def greet():
  print("Welcome to Python!")
greet()
```

5.2.5 Declaring Data Structures

- **Lists** (Ordered, mutable collection):
  numbers = [1, 2, 3, 4, 5]

- **Tuples** (Ordered, immutable collection):
  dimensions = (1920, 1080)

- **Sets** (Unordered collection of unique elements):
  unique_colors = {"red", "blue", "green"}

- **Dictionaries** (Key-value pairs):
  person = {"name": "Alice", "age": 30}

5.2.6 Declaring Classes and Objects

Classes define object blueprints.

```
class Animal:
   def __init__(self, species, sound):
      self.species = species
      self.sound = sound
dog = Animal("Dog", "Bark")
print(dog.sound)  # Output: Bark
```

5.3 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.

5.3.1 Basic Return Syntax

```
def square(n):
   return n * n
print(square(4))  # Output: 16
```

5.3.2 Returning Multiple Values

A function can return multiple values as a tuple.

```
def user_info():
   return "Alice", 25
name, age = user_info()
print(name, age)  # Output: Alice 25
```

5.3.3 Returning Lists and Dictionaries

```
def generate_sequence(n):
  return [n, n*2, n*3]
print(generate_sequence(5))  # Output: [5, 10, 15]
```

5.3.4 Returning a Function from Another Function

```
def multiply_by(factor):
  def multiplier(value):
    return value * factor
  return multiplier

double = multiply_by(2)
print(double(10))  # Output: 20
```

5.3.5 Using pass in Functions

The pass statement acts as a placeholder for future implementation.

```
def future_feature():
  pass  # Placeholder for code to be added later
```

5.3.6 Recursive Function Calls

Functions can call themselves for problems like factorial calculations.

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

print(factorial(6))  # Output: 720
```

5.4 Iterative Statements

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

5.4.1 for Loop

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

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

5.4.2 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

---

5.5.3 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}")
```

5.5 Conditional Statements

Conditional statements allow a program to make decisions based on given conditions. These statements enable logical branching, executing different code blocks depending on whether a condition evaluates to True or False.
Python provides structured decision-making through the following conditional constructs:

5.5.1 if Statement

The if statement executes a block of code only when a specified condition is True.
Example:

```
age = 20
if age >= 18:
   print("You are eligible to vote.")
```

Output:
You are eligible to vote.

5.5.2 if-else Statement

The if-else structure provides an alternative execution path when the condition evaluates to False.

Example:

```
temperature = 15
if temperature > 20:
  print("It's warm outside.")
else:
  print("It's cold outside.")
```

Output:

```
It's cold outside.
```

5.5.3 The if-elif-else Statement

The if-elif-else construct evaluates multiple conditions sequentially. The first True condition is executed, and the rest are ignored.

Example:

```
grade = 85
if grade >= 90:
  print("Grade: A")
elif grade >= 80:
  print("Grade: B")
elif grade >= 70:
  print("Grade: C")
elif grade >= 60:
  print("Grade: D")
else:
  print("Grade: F")
```

Output:

```
Grade: B
```

5.5.5 Nested if Statements

An if statement can be placed inside another if statement for hierarchical decision-making.

Example:

```
age = 19
if age >= 18:
  if age < 21:
    print("You are an adult but cannot drink alcohol in some countries.")
  else:
    print("You are fully an adult.")
```

Output:

```
You are an adult but cannot drink alcohol in some countries.
```

5.6 Function Call Statement

A function call statement executes a function by referencing its name followed by parentheses. If the function requires arguments, they are provided inside the parentheses.

8.1 Calling a Function Without Arguments

```
def greet():
  print("Hello, welcome to Python!")

greet()
```

Output:

Hello, welcome to Python!

8.2 Calling a Function With Arguments

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

result = add(5, 7)
print(result)
```

Output:

12

8.3 Calling a Function With Default Arguments

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

introduce()  # Uses default value
introduce("Alice")
```

Output:

Hello, Guest!
Hello, Alice!

8.4 Calling a Function With Keyword Arguments

```
def describe_pet(animal, name):
  print(f"I have a {animal} named {name}.")

describe_pet(animal="dog", name="Buddy")
describe_pet(name="Whiskers", animal="cat")
```

Output:

I have a dog named Buddy.
I have a cat named Whiskers.

*8.5 Calling a Function With Arbitrary Arguments (*args)*

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

print_numbers(1, 2, 3, 4, 5)
```

Output: 1 2 3 4 5

8.6 Calling a Function That Returns a Value

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

product = multiply(4, 3)
print(product)
```

Output:

12

6 Expressions

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

6.1 Arithmetic Expression

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

6.1.1 Basic Arithmetic Operators

Operator	Description	Example	Output
+	Addition	7 + 2	9
-	Subtraction	9 - 4	5
*	Multiplication	6 * 3	18
/	Division (Floating-point)	8 / 2	4.0

Example:

```
x = 12
y = 4
print(x + y) # Output: 16
print(x - y) # Output: 8
print(x * y) # Output: 48
print(x / y) # Output: 3.0
```

6.1.2 Floor Division (//)

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

Example:

```
print(15 // 4) # Output: 3
print(10.8 // 3) # Output: 3.0
```

6.1.3 Modulo (%)

- Returns the remainder of division.

Example:

```
print(17 % 5) # Output: 2
print(22 % 7) # Output: 1
```

6.1.4 Exponentiation (**)

- Raises a number to a power.

Example:
print(3 ** 3) # Output: 27
print(16 ** 0.5) # Output: 4.0 (square root)

6.1.5 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)

Example:
print(3 + 4 * 2) # Output: 11 (Multiplication first)
print((3 + 4) * 2) # Output: 14 (Parentheses first)
6.1.6 Unary Operations

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

Example:
print(+7) # Output: 7
print(-7) # Output: -7

6.1.7 Augmented Assignment Operators

These combine an arithmetic operation with assignment.
Mentioned before

6.2 Boolean Expressions

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

6.2.1 Boolean Values

Python provides two built-in Boolean values:

- True: Represents a true or affirmative state.
- False: Represents a false or negative state.

These values are case-sensitive, meaning true and false are invalid.

Example:

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

6.2.3 Boolean Operators

Python supports three logical operators that manipulate Boolean values:

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

Example:

```
x = 8
y = 12
print(x > 5 and y < 15)  # Output: True
print(x > 10 or y > 15)  # Output: False
print(not (x < y))      # Output: False
```

6.2.4 Comparison Operators

Comparison operators evaluate conditions and return True or False based on the result of the comparison.

Operator	Description	Example	Output
==	Equal to	10 == 10	True
!=	Not equal to	7 != 9	True
>	Greater than	12 > 8	True
<	Less than	3 < 1	False
>=	Greater than or equal	6 >= 6	True
<=	Less than or equal	4 <= 2	False

Example:

```
age = 20
height = 175
print(age >= 18)   # Output: True
print(height < 160)  # Output: False
print(age != 25)   # Output: True
```

6.2.5 Operator Precedence

Boolean expressions in Python follow a precedence order:

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

Example:

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

6.2.6 Boolean Short-Circuiting

Python optimizes Boolean expressions using 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 sample_function():
  print("Function executed")
  return True

print(False and sample_function())  # Output: False (Function is not called)
print(True or sample_function())    # Output: True (Function is not called)
```

6.2.7 Chained Comparisons

Python allows multiple comparisons in a single statement.

Example:

```
score = 85
print(50 < score < 100)  # Output: True (equivalent to 50 < score and score < 100)
```

7 Conclusion

The Python language specification outlines the core syntax, semantics, and features of the Python programming language, emphasizing readability, simplicity, and versatility. Python supports multiple programming paradigms, including object-oriented, structured, and functional programming, allowing developers to choose the style that best suits their needs.

In summary, the Python language specification provides a comprehensive framework that balances simplicity with powerful features, making it a popular choice for both beginners and experienced developers across various applications.

8 References

1 https://docs.python.org/3/tutorial/controlflow.html

2 https://www.geeksforgeeks.org/conditional-statements-in-python/

3 https://stackoverflow.com/questions/9195455/how-to-document-a-method-with-parameters

4 https://www.datacamp.com/tutorial/functions-python-tutorial

5 https://www.linode.com/docs/guides/if-statements-and-conditionals-in-python/

6 **Eric Matthes, Python Crash Course, No Starch Press, 2019**

7 **lecture Slides**