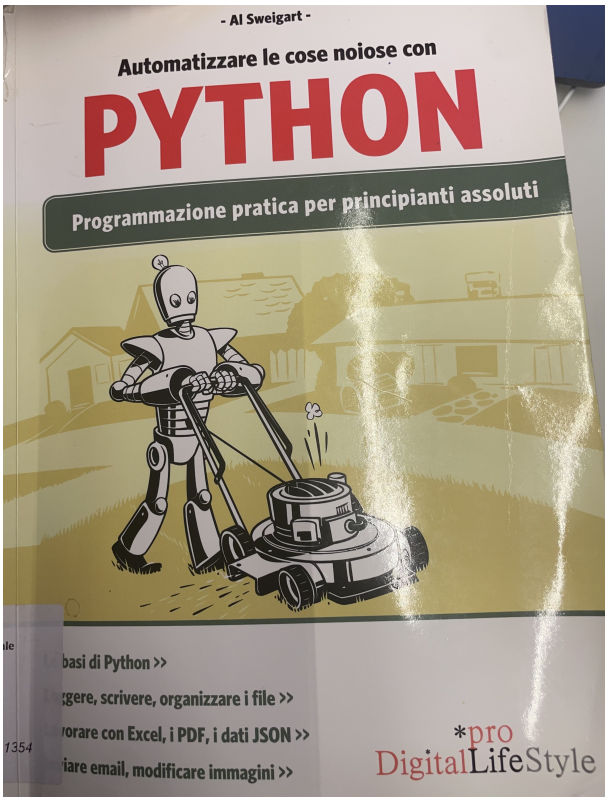
Python 1

A Self-Learning Guide

Part 2: Core Concepts

Prepared by Simone Capodivento



March 9, 2025

Abstract

This guide explores fundamental Python programming concepts including functions, lists, dictionaries, and control structures. It provides practical examples and projects to reinforce learning through hands-on practice.

Contents

1 Introduction

Python has emerged as one of the most versatile and accessible programming languages for automation. This guide aims to provide a solid foundation in Python programming while focusing on practical applications that can save time and effort in everyday tasks.

2 Python Fundamentals

2.1 Data Types

In Python, different types of data are used to represent numbers, text, and more complex structures.

2.1.1 Numeric Types

- **Integer (int)**: Whole numbers without decimal points (e.g., `42`, `-10`)
- **Floating-point (float)**: Numbers with decimal points (e.g., `3.14`, `-0.001`)

2.1.2 String Notation

Strings in Python can be defined using several methods:

- **Single quotes**: `'Hello world'`
- **Double quotes**: `"Hello world"`
- **Triple quotes**: For multi-line strings `'''Hello nworld'''` or `"""Hello nworld"""`

2.1.3 String Operations

```
# Concatenation
>>> 'Alice' + 'Bob'
'AliceBob'

# Replication
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

Listing 1: Basic string operations

2.1.4 Type Conversion

Python allows conversion between different data types:

```
# String to other types
>>> int('42')
42
>>> float('3.14')
3.14

# Numbers to string
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'

# Integer to float and vice versa
>>> int(1.99)
1
>>> float(10)
```

```
10.0
```

Listing 2: Examples of type conversion

2.1.5 Type Equivalence

Even when the string representation of a number matches its numeric value, they are considered different types:

```
# String vs integer comparison
>>> 42 == "42"
False

# Integer vs float comparison
>>> 42 == 42.0
True

# Float formatting doesn't affect values
>>> 42.0 == 0042.000
True
```

Listing 3: Equality comparisons between data types

2.2 Variables

A variable serves as a container in computer memory where a single value can be stored and retrieved later.

2.2.1 Assignment Statement

Variables are assigned using the assignment operator `=`:

```
>>> spam = 40
>>> spam
40
>>> eggs = 2
>>> spam = 1 / eggs
>>> spam
0.5
>>> spam + eggs + spam
3.0
>>> spam = spam + 2
>>> spam
2.5
```

Listing 4: Variable assignment examples

2.2.2 Variable Naming Rules

Python has specific rules for valid variable names:

2.3 Operators

2.3.1 Arithmetic Operators

Python follows the PEMDAS order of operations:

Valid Variable Names	Invalid Variable Names
`balance`	`balance-current` (hyphens are not allowed)
`balanceCurrent`	`balance current` (spaces are not allowed)
`_private`	`4account` (cannot start with a number)
`SPAM`	`42` (cannot be just a number)
`account4`	`total_$` (special characters like $ are not allowed)
`snake_case`	`''` (empty strings are not allowed)

Table 1: Examples of valid and invalid variable names in Python.

1. **P**arentheses - Expressions inside `(...)` are evaluated first
2. **E**xponents - Powers (e.g., `x**y`) are evaluated next
3. **M**ultiplication and **D**ivision - `*` and `/` are evaluated from left to right
4. **A**ddition and **S**ubtraction - `+` and `-` are evaluated from left to right

Operator	Operation	Example	Result
`**`	Exponentiation	`2 ** 3`	8
`%`	Modulus (Remainder)	`22 % 8`	6
`//`	Floor Division	`22 // 8`	2
`/`	Division	`22 / 8`	2.75
`*`	Multiplication	`3 * 5`	15
`-`	Subtraction	`5 - 2`	3
`+`	Addition	`2 + 2`	4

Table 2: Python arithmetic operators listed in order of precedence.

3 Flow Control

Flow control statements allow for conditional execution and looping in code.

3.1 Boolean Expressions

Boolean expressions evaluate to either `True` or `False` and form the basis for decision-making in code.

3.1.1 Comparison Operators

3.1.2 Boolean Logic Operators

Boolean logic operators combine multiple conditions:

- `and`: Returns `True` if both conditions are `True`
- `or`: Returns `True` if at least one condition is `True`
- `not`: Inverts the value, turning `True` into `False` and vice versa

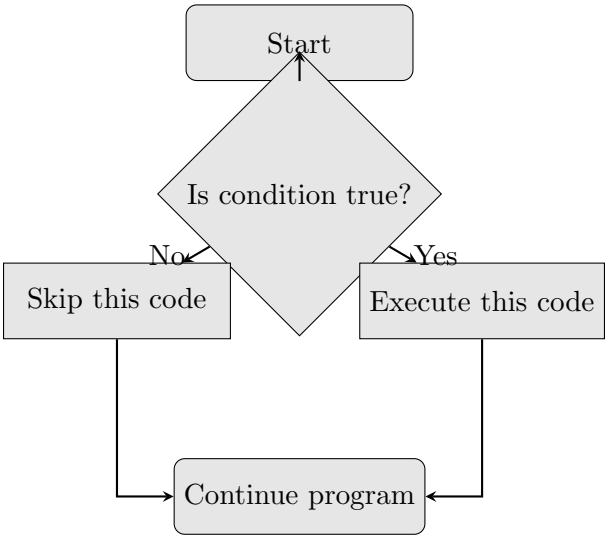


Figure 1: Flowchart demonstrating a basic decision-making process in programming.

Operator	**Meaning**
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Table 3: Python comparison operators.

```
>>> True and True
True
>>> True and False
False
>>> False or False
False
>>> True or False
True
>>> not True
False
>>> not not not not True
True
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Listing 5: Examples of Boolean operators

3.2 Conditional Statements

3.2.1 If Statements

Conditional execution is performed using `if`, `elif`, and `else` statements.

```
if name == 'Mary':
    print('Hello␣Mary')
    if password == 'swordfish':
        print('Access␣granted.')
    else:
        print('Incorrect␣password.')
```

Listing 6: Basic if-else statement

3.3 Blocks in Python

In Python, code blocks are defined by indentation rather than braces or keywords:

1. Blocks begin when indentation increases
2. Blocks can contain other blocks (nested blocks)
3. A block ends when indentation decreases to match an outer block or the file's base level

4 Loops and Iteration

Loops allow repetitive execution of code blocks.

4.1 While Loops

A `while` loop executes as long as its condition remains `True`:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Listing 7: Basic while loop example

4.2 For Loops and range()

The `for` loop iterates over a sequence, often generated by the `range()` function:

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

Listing 8: For loop with range() function

range() Call	Equivalent Sequence	Description
`range(5)`	`[0, 1, 2, 3, 4]`	Start at 0, end before 5
`range(1, 5)`	`[1, 2, 3, 4]`	Start at 1, end before 5
`range(0, 10, 2)`	`[0, 2, 4, 6, 8]`	Start at 0, increment by 2, end before 10
`range(5, 0, -1)`	`[5, 4, 3, 2, 1]`	Count down from 5 to 1

Table 4: Common uses of the range() function.

4.3 Loop Control Statements

Python provides several ways to control loop execution:

4.3.1 Break Statement

The `break` statement exits a loop immediately:

```
while True:
    print('Please enter your name.')
    name = input()
    if name == 'your name':
        break
    print('Thank you')
```

Listing 9: Using break to exit a loop early

4.3.2 Continue Statement

The `continue` statement skips the rest of the current iteration and jumps to the next iteration:

```
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
```

```
    print('Hello, Joe. What is your password? (Hint: It\'s a type of
        fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted.')
```

Listing 10: Using continue to skip iterations

4.3.3 Terminating a Program

The `sys.exit()` function terminates a program completely:

```
import sys
while True:
    user_input = input("Enter 'exit' to stop the loop: ")
    if user_input.lower() == "exit":
        print("Exiting the program...")
        sys.exit()  # Terminates the script immediately
    print(f"You entered: {user_input}")
```

Listing 11: Using sys.exit() to terminate a program

5 Python Built-in Functions

Python provides many built-in functions for common operations.

5.1 Input and Output

- `print()`: Displays text or variables on the screen
- `input()`: Accepts user input as a string

5.2 String Manipulation

- `len()`: Returns the length of a string
- `str()`: Converts a value to its string representation

6 Comments in Python

Comments are used to document code and make it more understandable:

```
# Different ways to use comments in Python
print("Hello, World")  # This is an inline comment

# This is a full line comment

# print("This line won't run because it's commented out")

"""
This is a multi-line comment (technically a string literal)
It can span multiple lines
And is often used for docstrings
"""

```

```
# Comments can explain what code does
name = "Alice"  # Store user name
print(f"Hello, {name}")  # Greet the user

# Comments can explain why we do something
x = x + 1  # Increment to account for zero-indexing
```

Listing 12: Different ways to use comments in Python

7 Development Environment

Understanding your development environment is crucial for efficient Python programming.

Program Editor

```
# Python Program
def hello():
  print("Hello world")
name = input("Enter
name: ")
print(f"Hello, {name}")
for i in range(5):
  print(i)
```

Python Shell (REPL)

```
>>> name = "Python"
>>> print(f"Hello,
{name}")
Hello, Python
>>> 2 + 2
4
>>> for i in range(3):
... print(i)
...
0
1
2
>>>
```

Figure 2: Python development environment showing shell (REPL) on the right and program editor on left.

8 Advanced Topics

8.1 Exception Handling

Python provides mechanisms to handle errors gracefully:

```
try:
    age = int(input("Enter your age: "))
    years_to_100 = 100 - age
    print(f"You will be 100 in {years_to_100} years")
except ValueError:
    print("That's not a valid age")
except:
    print("An unexpected error occurred")
finally:
    print("Thank you for using this program")
```

Listing 13: Basic exception handling

8.2 File Operations

Python makes working with files straightforward:

```
# Reading a file
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)

# Writing to a file
with open('output.txt', 'w') as file:
    file.write('Hello, world!\n')
    file.write('This is a test file.')
```

Listing 14: Basic file operations