TAPC2411 Professional Course on Python Programming

TechZ Academy

July 31, 2025

Contents

Chapter 1

Introduction to Python Programming

1.1 Introduction

1.1.1 Programming Language

A programming language is a formal language comprising a set of instructions that produce various kinds of output. It allows programmers to communicate instructions to a computer system, enabling it to perform specific tasks or functions. Programming languages are used to create software, websites, applications, and other computational solutions.

Examples of programming languages include `Python`, `Java`, `C++`, `JavaScript`, `Ruby`, and many more. Each language has its own syntax, semantics, and rules that govern how instructions are written and executed.

1.1.2 Algorithms

An **algorithm** is a step-by-step procedure or set of rules designed to solve a specific problem or perform a particular task. It is a finite sequence of well-defined, unambiguous instructions that, when followed, lead to a desired outcome or solution. Algorithms can be expressed in various forms, including natural language, pseudocode, flowcharts, or programming languages.

In computer science, algorithms are fundamental to solving computational problems efficiently. They provide a blueprint for writing computer programs and are essential for tasks such as sorting data, searching for information, performing mathematical operations, and optimizing processes.

Key characteristics of algorithms:

1. **Input:** The algorithm receives data or input values on which it operates.

2. **Output:** The algorithm produces a result or output based on the input and the operations performed.

3. **Determinism:** Each step of the algorithm is precisely defined and produces the same result when executed with the same input.

4. **Finiteness:** The algorithm has a finite number of steps, meaning it eventually terminates.

5. **Effectiveness:** Each step of the algorithm must be clear and executable within finite time and space constraints.

Algorithms are used extensively in various fields, including computer science, mathematics, engineering, and everyday problem-solving tasks. They serve as the foundation for designing efficient software systems, algorithms for data analysis, artificial intelligence, and more.

1.1.3 Types of Programming Language

Programming languages can be categorized into several types based on various criteria such as their level of abstraction, paradigm, purpose, and domain of application. Here are some common types:

1. **Low-level languages:**

   - **Machine Language:** The lowest-level programming language consisting of binary code understood directly by the computer's hardware.
   - **Assembly Language:** Uses mnemonic codes to represent machine instructions, making it more readable than machine language but still closely tied to the hardware architecture.

2. **High-level languages:**

   - **Procedural Languages:** Focus on describing a sequence of steps to solve a problem. Examples: `C`, `Pascal`, `BASIC`.
   - **Object-Oriented Languages:** Organize code into objects that interact with each other, emphasizing encapsulation, inheritance, and polymorphism. Examples: `Java`, `C++`, `Python`.
   - **Functional Languages:** Treat computation as the evaluation of mathematical functions and emphasize immutability and declarative programming. Examples: `Haskell`, `Lisp`, `Scala`.

- **Scripting Languages:** Designed for quick and easy development of small to medium-sized programs. Examples: `Python`, `Ruby`, `JavaScript`.

3. **Domain-specific languages (DSLs):**

   - **Markup Languages:** Used to annotate text for formatting or semantic information. Examples: `HTML`, `XML`, `Markdown`.
   - **Query Languages:** For querying and manipulating databases or data sources. Examples: `SQL`, `XPath`.
   - **Statistical Languages:** For statistical analysis and data manipulation. Examples: `R`, `MATLAB`.

4. **Functional purpose:**

   - **General-purpose languages:** Versatile and used for a wide range of applications. Examples: `Python`, `Java`, `C++`.
   - **Domain-specific languages:** Tailored to a specific application domain. Examples: `SQL`, `HTML/CSS`, `VHDL`.

5. **Compiled vs. interpreted languages:**

   - **Compiled Languages:** Code is translated into machine code before execution. Examples: `C`, `C++`, `Rust`.
   - **Interpreted Languages:** Code is executed line by line by an interpreter. Examples: `Python`, `Ruby`, `JavaScript`.

6. **Imperative vs. Declarative languages:**

   - **Imperative Languages:** Programs are composed of statements that change a program's state. Examples: `C`, `Java`, `Python`.
   - **Declarative Languages:** Programs describe the desired result without specifying the exact steps. Examples: `SQL`, `HTML/CSS`, `Prolog`.

1.2 Compilers

1. **Lexical Analysis (Scanning):**

   - Breaks the source code into tokens or lexemes.
   - The lexer (scanner) reads the source code character by character and groups them into tokens.
   - Comments and whitespace are usually discarded.

2. **Syntax Analysis (Parsing):**
    - Analyzes the structure of the source code to ensure it conforms to syntax rules.
    - The parser uses a grammar specification to check if the tokens form valid statements and expressions.
    - Generates a parse tree or abstract syntax tree (AST).
3. **Semantic Analysis:**
    - Verifies the meaning of the source code beyond its syntax.
    - Checks for semantic errors, such as type mismatches or undefined variables.
    - Involves symbol table management.
4. **Intermediate Code Generation:**
    - Translates the source code into an intermediate representation (IR).
    - May involve translating the AST into a lower-level representation.
5. **Optimization:**
    - Applies optimization techniques to improve code efficiency.
    - Examples: constant folding, dead code elimination, loop optimization.
6. **Code Generation:**
    - Translates the optimized intermediate code into machine code.
    - Register allocation and instruction scheduling may occur.
7. **Linking (for multi-file programs):**
    - Combines object files into a single executable or library.
    - Resolves external references and performs address binding.

Each stage of the compiler transforms the source code into executable code efficiently. The resulting executable can then be run on the target platform.

1.3 Interpreter Language

1. **Lexical Analysis:**
    - Reads the source code character by character and groups them into tokens.
    - Tokens represent the smallest units, such as keywords, identifiers, operators, and literals.

- Comments and whitespace are typically ignored.

2. **Parsing:**

   - Analyzes the structure of the source code to ensure it conforms to syntax rules.
   - Checks if the tokens form valid statements and expressions.
   - May generate an abstract syntax tree (AST).

3. **Semantic Analysis:**

   - Verifies the meaning of the source code beyond its syntax.
   - Checks for semantic errors.
   - Involves symbol table management.

4. **Code Execution:**

   - Executes the parsed code directly without a separate code generation stage.
   - Interprets the AST or intermediate representation.
   - May use strategies like bytecode interpretation or just-in-time (JIT) compilation.

Unlike compilers, interpreters do not generate machine code but execute the code directly.

Table 1.1: Difference Between Compiler and Interpreter

Compiler	**Interpreter**
Translates entire program to machine code before execution.	Translates code line by line and executes it simultaneously.
Produces intermediate object code or executable file.	Does not produce an intermediate object code or executable file.
Typically generates faster-executing code.	Generally slower as it executes code directly.
Detects all syntax and semantic errors before execution.	May detect errors at runtime during interpretation.
Requires a separate compilation stage before execution.	Does not require a separate compilation stage.
Examples include GCC (GNU Compiler Collection), Clang.	Examples include Python interpreter, JavaScript interpreter.

Table 1.2: Differences Between Procedural, Object-Oriented, and Functional Programming

Procedural Programming	**Object-Oriented Programming**	**Functional Programming**
Focuses on procedures or functions that operate on data.	Organizes code into objects that encapsulate data and behavior.	Emphasizes mathematical functions and immutable data.
Data and procedures are separate entities.	Data and behavior are encapsulated within objects.	Functions are first-class citizens and can be passed as arguments or returned.
Languages: C, Pascal.	Languages: Java, C++, Python.	Languages: Haskell, Lisp, Clojure.
Follows a top-down approach to problem-solving.	Follows a bottom-up approach with focus on reusability and modularity.	Follows a declarative approach; functions describe what should be done.
Less emphasis on code reusability and scalability.	Promotes code reusability through inheritance, polymorphism, encapsulation.	Encourages code reusability through higher-order functions and immutability.

Chapter 2

Conditional Statements in Python

Conditional statements let your Python programs make decisions and control the flow of execution based on whether certain conditions are **true** or **false**. They are a fundamental part of programming and real-world problem solving.

2.1 Introduction to Conditional Statements

Python provides several types of conditional statements:

- `if`
- `if-else`
- `if-elif-else`
- Nested `if-else`

Note: Indentation in Python

Indentation in Python is critical. It shows which lines of code belong to which code blocks.

- **Default indentation**: 1 tab *or* 4 spaces (may vary by system).
- **Rule**: Use consistent indentation in the same block.

Example:

```
if condition:
    statement1
    statement2
```

Here, `statement1` and `statement2` will be executed only if the condition is true.

2.2 The `if` Statement

The simplest decision-making statement in Python is the `if` statement.

Syntax

```
if <relational expression>:
    # Block of code (executed if condition is True)
    Line 1
    Line 2
    ...
```

Example 1: Checking Zero

```
number = input("Enter the number: ")
# The input function returns a string.
number = int(number)     # Convert to integer

if number == 0:
    print("Entered Number is ZERO")
```

If the user enters 0, "Entered Number is ZERO" is printed.

2.3 Truthy and Falsy Values

In Python, certain values are considered "true" or "false" in conditions (e.g., 0 → False, nonzero → True).

Example 2: Truthy/Falsy Check

```
number = input("Enter the number: ")
number = int(number)

if number:
    print("Entered Number is not ZERO")

if not number:
    print("Entered Number is ZERO")
```

When `number` is 0, only the second block executes. Otherwise, the first block executes.

2.4 The `if-else` Statement

The `if-else` statement allows one block (if) when true, another (`else`) when false.

Syntax

```
if <condition>:
    # true block
else:
    # false block
```

Example 3: Zero or Not

```
number = input("Enter the number: ")
number = int(number)
if number == 0:
    print("Entered Number is ZERO")
else:
    print("Entered Number is not ZERO")
```

2.5 The `if-elif-else` Statement

For multiple conditions, use `if-elif-else`.

Syntax

```
if <condition1>:
    # block 1
elif <condition2>:
    # block 2
else:
    # block 3
```

Example 4: Zero, One, or Others

```
number = input("Enter the number: ")
number = int(number)

if number == 0:
    print("Entered Number is ZERO")
elif number == 1:
```

```
    print("Entered Number is ONE")
else:
    print("Others")
```

2.6 Practical Examples

Example 5: Multiple Divisibility Checks

```
number = input("Enter the number: ")
number = int(number)

if number % 2 == 0:
    print("Twilight")
elif number % 3 == 0:
    print("SQL")
else:
    print("PYTHON")
```

Checks for divisibility by 2 (Twilight), 3 (SQL), or otherwise outputs PYTHON.

2.7 Nested `if-else` Statements

Statements can be nested for more complex logic.

Syntax

```
if <condition1>:
    if <condition2>:
        # Block A
    else:
        # Block B
else:
    # Block C
```

Example 6: Multiple Divisibility with Nesting

```
number = input("Enter the number: ")
number = int(number)

if number % 2 == 0:
```

```
    if number % 3 == 0:
        print("The number is divisible by 6")
    elif number % 5 == 0:
        print("The number is divisible by 10")
```

Checks for divisibility by 6 or 10, only if the number is even.

2.8 Summary

- `if`: For single conditions.
- `if-else`: For two-way branching.
- `if-elif-else`: For multiple branches.
- **Indentation** is essential in Python for defining code blocks.
- Use nesting for complex decision logic.

Practice writing conditional statements to master this essential concept!

Exercises

1. Write a program that accepts marks from the user and prints "Pass" if marks are greater than or equal to 35, otherwise prints "Fail".
2. Write a program to check if a number is positive, negative, or zero.
3. Write a program to check if a number is odd, even, or zero.

Conditional statements give Python programs the ability to make decisions and adapt to different situations. Master them to become a better Python programmer!