Course Coordinator

Python Programming

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1

Course Outline

1.1 Course Overview

Credits: 2
Hours per Week: 2

Course Objectives:

1. To develop algorithmic thinking and problem-solving skills.
2. To familiarize students with computer systems, software, and language translators.
3. To justify the use of Python for programming and algorithmic design.
4. To introduce Python programming concepts, including data types, conditional statements, and loops.
5. To implement functions, string operations, and real-time applications using Python's data structures.
6. To enable students to apply their knowledge to solve practical problems through a micro project.

1.2 Key Ideas

1.2.1 Topic 1: Introduction to Algorithmic Thinking

Lesson Outcomes:

- Understand the concept of algorithmic thinking.
- Develop basic algorithms for simple problems.
- Recognize the importance of algorithms in problem-solving.

Content:

- What is Algorithmic Thinking?

  - Importance in problem-solving
  - Steps in designing an algorithm
- Example Algorithms:
  - Simple tasks (e.g., making a cup of tea)

Practical Situation:

- Create algorithms for everyday activities to illustrate the concept.

1.2.2 Topic 2: Familiarization with Computer Systems and Language Translators

Lesson Outcomes:

- Understand basic computer architecture and components.
- Identify different types of software and language translators.

Content:

- Introduction to Computer Architecture:
  - Block diagram of a computer
  - Hardware components (Input, Output devices)
  - Memory types
- Software Types:
  - High-level vs. Low-level languages
  - Assembly languages
- Language Translators:
  - Compilers, Interpreters, Assemblers

Practical Situation:

- Explore how different programming languages and translators affect the execution of a simple program.

1.2.3 Topic 3: Justification for Using Python

Lesson Outcomes:

- Understand the advantages of Python for algorithmic thinking and programming.
- Compare Python with other programming languages in terms of simplicity and effectiveness.

Content:

- Why Python?
  - Python's simplicity and readability
  - Comparison with other languages (e.g., C++, Java)
  - Python's role in modern software development and data science

Practical Situation:

- Demonstrate a basic Python script and compare it with an equivalent script in another language.

1.2.4 Topic 4: Developing Algorithms and Flow Charts

Lesson Outcomes:

- Develop and represent algorithms using flowcharts.
- Understand properties of good algorithms.

Content:

- Introduction to Algorithms:
  - Properties of good algorithms
- Flowchart Creation:
  - Basic flowchart symbols and conventions

Practical Situation:

- Design flowcharts for simple algorithms (e.g., sorting a list of numbers).

1.2.5 Topic 5: Data Types and Arithmetic Operations in Python

Lesson Outcomes:

- Understand and use basic data types and operators in Python.
- Perform arithmetic operations and handle expressions.

Content: - Introduction to Python Programming:

- Data types (int, float, str, etc.)
- Keywords and Variables
- Input and Output statements
- Operators and Arithmetic expressions

- Operator precedence and Evaluation of expressions

Practical Situation:

- Write a Python program that performs various arithmetic operations and displays results.

1.2.6 Topic 6: Conditional Statements in Python

Lesson Outcomes:

- Implement and use conditional statements to control the flow of programs.

Content: - Types of Conditional Statements:

- `if`, `if-else`, `elif`, nested `if-else`, `if-elif-else`
- Practical Examples:
  - Programs using conditional statements

Practical Situation:

- Create a Python program that determines if a number is positive, negative, or zero.

1.2.7 Topic 7: Loop Structures in Python

Lesson Outcomes:

- Use loop structures to repeat actions and iterate over data.

Content:

- Introduction to Looping:
  - `for`, `while`, nested loops
  - `break`, `continue`, `pass` statements
  - `range` function
- Sample Programs:
  - Implementing various loop constructs

Practical Situation:

- Write a Python program that calculates the factorial of a number using loops.

1.2.8 Topic 8: Functions in Python

Lesson Outcomes:

- Define and use functions for modular programming.
- Understand function concepts including parameter passing and return values.

Content:

- Concept of Functions:
  - Definition, Calling Functions
  - Passing Parameters and Return Values
  - Type Conversion and Coercion
- Advanced Function Concepts:
  - Lambda functions
  - Built-in Mathematical functions
- Sample Programs Using Functions

Practical Situation:

- Develop a Python program that uses functions to perform mathematical operations.

1.2.9 Topic 9: String Operations in Python

Lesson Outcomes:

- Manipulate and process strings using Python's string handling functions.

Content:

- Introduction to Strings:
  - String creation and manipulation
- String Handling Functions:
  - Commonly used functions (e.g., `split()`, `join()`, `replace()`)

Practical Situation:

- Write a Python program that processes and formats user input strings.

1.2.10 Topic 10: Real-time/Technical Applications Using Data Structures

Lesson Outcomes:

- Apply data structures (lists, tuples, dictionaries) to solve real-world problems.

Content:

- Lists, Tuples, Dictionaries:
  - Concepts, operations, and functions
  - Mutable vs Immutable data structures
- Applications:
  - Identifying use cases
  - Solving problems using lists, tuples, and dictionaries

Practical Situation:

- Create a Python program that manages a list of student records using lists and dictionaries.

1.2.11 Topic 11: Micro Project

Lesson Outcomes:

- Apply Python concepts to develop a project relevant to the student's field of study.

Content:

- Project Development:
  - Design and implementation of a simple project
  - Application of learned concepts to a practical problem

Practical Situation:

- Develop and present a micro project related to the student's branch of study.

1.3 Teaching Methodology

- Hands-on lab exercises
- Step-by-step problem-solving approach
- Regular assessments and feedback

1.4 Evaluation

- Lab exercises and practical implementation
- Micro Project

2

Basic Python Programming

Dynamic evaluation of user provided code and data visualisation

2.1 Introduction

This chapter introduce the `python` programming for beginers.

This book is created with Quarto and `pyodide` extension is cused for online python scripting to demonstrate the python programming in this course site itself.

See Knuth (1984) for additional discussion of literate programming.

```
a=10
print(a)
```

2.2 Examples

3

Sample Programs

3.1 Introduction to Python

In this tutorial, we will cover the basics of Python programming, including data types, keywords, variables, input/output statements, operators, arithmetic expressions, operator precedence, and evaluation of expressions.

3.2 Data Types

Python supports several built-in data types. Let's explore some of the most common ones:

- **Integer (`int`)**: Represents whole numbers.
- **Floating Point (`float`)**: Represents decimal numbers.
- **String (`str`)**: Represents sequences of characters.
- **Boolean (`bool`)**: Represents `True` or `False`.

Example 1

3.2.1 Static Template

```
# Demonstrating different data types

# Integer
a = 10
print("Integer:", a, type(a))

# Float
b = 3.14
print("Float:", b, type(b))
```

```
# String
c = "Hello, Python!"
print("String:", c, type(c))

# Boolean
d = True
print("Boolean:", d, type(d))
```

```
Integer: 10 <class 'int'>
Float: 3.14 <class 'float'>
String: Hello, Python! <class 'str'>
Boolean: True <class 'bool'>
```

3.2.2 Interactive Cell

```
# Demonstrating different data types

# Integer
a = 10
print("Integer:", a, type(a))

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


# String
c = "Hello, Python!"
print("String:", c, type(c))

# Boolean
d = True
print("Boolean:", d, type(d))
```

3.2.3 Variables

Variables are used to store data in memory. A variable is created when you assign a value to it using the = operator.

```
# Variable assignment

x = 5
y = 2.5
z = x + y
```

```
print("x =", x)
print("y =", y)
print("z =", z)
```

3.2.4 Input and Output Statements

Python provides the input() function to take user input and the print() function to display output.

```
# Input and Output
name="justin"
age=32
#name = input("Enter your name: ")
#age = int(input("Enter your age: "))

print(f"Hello, {name}! You are {age} years old.")
```

3.2.5 Operators

Operators are special symbols used to perform operations on variables and values. Python supports several types of operators:

Arithmetic Operators: +, -, *, /, //, %, ** Comparison Operators: ==, !=, >, <, >=, <= Logical Operators: and, or, not Assignment Operators: =, +=, -=, *=, /=, //=, %=, **=

```
# Arithmetic Operations

a = 15
b = 4

addition = a + b
subtraction = a - b
multiplication = a * b
division = a / b
floor_division = a // b
modulus = a % b
exponentiation = a ** b

print("Addition:", addition)
print("Subtraction:", subtraction)
print("Multiplication:", multiplication)
print("Division:", division)
```

```
print("Floor Division:", floor_division)
print("Modulus:", modulus)
print("Exponentiation:", exponentiation)
```

3.2.6 Arithmetic Expressions

An arithmetic expression is a combination of numbers, operators, and variables that evaluates to a value.

```
# Evaluating arithmetic expressions

expression = (5 + 2) * (10 - 3) / 2 ** 2
print("Expression Result:", expression)
```

3.2.7 Operator Precedence

Operator precedence determines the order in which operations are performed in an expression. The following list shows the precedence from highest to lowest:

** (Exponentiation) *, /, //, % (Multiplication, Division, Floor Division, Modulus) +, - (Addition, Subtraction)

```
# Operator precedence

result = 5 + 3 * 2 ** 2 - 1
print("Operator Precedence Result:", result)
```

3.2.8 Evaluation of Expressions

Python evaluates expressions from left to right, following the precedence rules.

```
# Evaluation of expressions

value = (10 + 5) * 2 - 3 / 3
print("Evaluation Result:", value)
```

3.2.9 Conditional Statements in Python

Conditional statements in Python allow the execution of specific code blocks based on whether a condition is true or false. Let's explore various types of conditional statements.

3.2.9.1 The `if` Statement

The `if` statement tests a specific condition. If the condition is true, the code block under the `if` statement is executed.

Example

```
# Example of an if statement

number = 10

if number > 0:
    print(f"{number} is a positive number.")
```

> **Explanation**
>
> The above program checks if number is greater than 0. Since 10 is greater than 0, the condition is true, and the message is printed.

3.2.9.2 The if-else Statement

The if-else statement allows you to execute one block of code if the condition is true and another block if it is false.

```
# Example of an if-else statement

number = -5

if number >= 0:
    print(f"{number} is a non-negative number.")
else:
    print(f"{number} is a negative number.")
```

> **Explanation**
>
> In this example, the program checks if number is greater than or equal to 0. If true, it prints that the number is non-negative. Otherwise, it prints that the number is negative.

Example 2

```
age = 20

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Explanation

This program checks if a person's age is greater than or equal to 18. If true, it prints that the person is eligible to vote. Otherwise, it states they are not eligible to vote.

3.2.10 The elif Statement

The elif statement, short for "else if," allows you to check multiple conditions sequentially. If one of the conditions is true, the corresponding block of code is executed.

```
# Example of an elif statement

number = 0

if number > 0:
    print(f"{number} is a positive number.")
elif number == 0:
    print(f"{number} is zero.")
else:
    print(f"{number} is a negative number.")
```

Explanation

Here, the program checks three conditions: whether the number is positive, zero, or negative. The elif statement handles the case where number is exactly 0.

```
# Another example of an elif statement

marks = 85

if marks >= 90:
    grade = 'A'
elif marks >= 80:
    grade = 'B'
elif marks >= 70:
    grade = 'C'
else:
    grade = 'F'

print(f"Your grade is {grade}.")
```

Explanation

This program assigns a grade based on the marks obtained. Depending on the range in which the marks fall, the corresponding grade is assigned and printed.

3.2.10.1 Nested if-else Statements

Nested if-else statements allow you to include an if-else statement inside another if-else block for handling more complex conditions.

```
# Example of nested if-else statements

number = 25

if number > 0:
    if number % 2 == 0:
        print(f"{number} is a positive even number.")
    else:
        print(f"{number} is a positive odd number.")
```

Explanation

This example checks if a number is positive and then further checks whether it is even or odd using nested if-else statements.

```
# Another example of nested if-else statements

score = 92

if score >= 50:
    if score >= 90:
        print("Excellent!")
    else:
        print("Good job!")
else:
    print("Better luck next time.")
```

Explanation

This program checks if a score is at least 50. If true, it further checks if the score is 90 or above, printing “Excellent!” if it is, and “Good job!” if it isn’t. If the score is below 50, it prints “Better luck next time.”

4

Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.