4 Programming Languages

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand the importance of programming languages.
- Learn to read Python-written programs.
- Use LLMs to assist in writing and debugging Python programs.

4.1 Overview of Programming Languages

A programming language is a formal system for specifying computation. It provides a way for humans to write instructions that a computer can understand and execute. Over the decades, many programming languages have been developed, each designed with different goals.

In general, there are two types of programming languages.

- **Low-Level Languages**: They provide direct control over hardware but are difficult to write and maintain. Examples are assembly languages.
- **High-Level Languages**: They are closer to human language, easier to write, read, and maintain, and abstract away hardware details. Examples include Python, Java, and Rust.

Writing software directly in assembly code is inefficient and error-prone. It is rarely used today, except when low-level control is required, such as in device drivers or operating system development. In contrast, high-level languages are widely used in the software industry. Programs written in high-level languages are executed after being translated into assembly (or machine) code, either ahead of time by a compiler or at runtime by an interpreter or virtual machine.

A high-level language must not be ambiguous, whereas natural language is especially prone to it. For example: *A programmer' s wife asks him to go to the grocery store. She says: "Get a gallon of milk. If they have eggs, get 12." The programmer returns with 12 gallons of milk.* This humorous story shows how natural language can be interpreted in multiple ways. That is also a reason why we could not use natural language to code in the past. However, with the evolution of large language models, it has become possible to write programs using natural language to some extent. Still, developers should be aware of two risks: ambiguity in prompts, and under-specification, where the prompt is incomplete and lacks sufficient detail, causing the system to make unintended assumptions.

This course uses Python as the main programming language, because it offers a gentle learning curve while being powerful enough for serious software development.

4.2 Python

Python is an interpreted, high-level programming language, invented by Dutch programmer Guido van Rossum in 1991. It is widely used in areas such as artificial intelligence, data science, and automation, and is famous for its clean, simple syntax. A Python program consists of a sequence of statements executed by the Python interpreter. In this section, we will learn the basic components of Python programs. Students are not required to write programs manually, but they should be able to read them and make modifications.

4.2.1 Variables

A variable in Python is like a label you stick on a box: it allows you to store a value and retrieve it later. Variables provide a way to refer to data by name instead of using the data directly, which makes programs easier to read and manage. When naming variables, there are a few important rules. A variable name must begin with either a letter or an underscore, and the rest of the name can include letters or numbers. For example, valid variable names include `abc123`, `_abc`, and `ifabc`, while `123abc` is invalid because it begins with a number. Certain words, called Python keywords (such as `def`, `for`, or `if`), cannot be used as variable names because they have special meanings in the language. Besides, Python variable names are case-sensitive, which means that `abc123` and `Abc123` would be considered different identifiers.

Python programmers also follow naming conventions to make code clearer. The most common style is snake case, where all letters are lowercase and words are separated by underscores, such as `fudan_university` or `total_score`. In Python, snake case is typically used for variable names and function names. Another style is camel case, where each word begins with a capital letter, such as `FudanUniversity` or `TotalScore`. Camel case is mainly used for naming classes.

```python
x = 25                  # integer type
y = 3.14                # float type
name: str = "Alice"     # string type, characters enclosed within double quotes.
age = "25"              # string type, characters enclosed within double quotes.
is_student = True       # boolean type
```

Code 4.1: Example of variables

Each variable in Python has a type, such as integer, float, or string. Code 4.1 defines five variables of different types. Unlike some programming languages, Python does not require you to declare the type explicitly; instead, the type is inferred automatically based on the value you assign. For example, `x = 25` creates an integer, `y = 3.14` creates a float, and `name = "Alice"` creates a string. Starting from Python 3.5, you can optionally add type annotations to variables and functions to make the intended types clearer, `name: str = "Alice"`.

4.2.2 Type Checking

In a valid Python program, the types of variables or values must match the requirements of the operators being applied. Otherwise, Python raises an error. As shown in Code 4.2, the first two statements are valid, while the third is invalid because it attempts to add a number to a string, which has no logical meaning. In such cases, Python raises a `TypeError` to prevent the program from executing incorrectly. Besides, Python requires variables must be defined before use. Therefore, statement four is also invalid.

```python
a = 5 + 3         # Valid: add two integers
b = "5" + "3"     # Valid: '+' means concatenating two strings; result: "53"
c = 5 + "3"       # Type Error: add an integer to a string is invalid.
x = a + d;        # Invalid: d is not defined.
x = 0.5 + 3       # Type coercion: the result is float 3.5
x = 5 + True      # Type coercion: the result is integer 6
```

Code 4.2: Example of type checking

In some cases, Python performs type coercion, also called implicit conversion. When two different numeric types are combined in an expression, Python automatically converts the narrower type to the wider type so that the operation can be carried out safely. For example, in Code 4.2, the fifth statement adds a floating-point number to an integer: Python first converts the integer to a float, and the result is a float. In the last statement, the boolean value is converted to an integer before the addition. This behavior makes Python both safe and flexible when handling mixed numeric operations.

4.2.3 Control Flow

Control flow statements allow a program to execute different code blocks depending on certain conditions or to repeat code multiple times. Python provides several mechanisms for controlling the flow of execution, including conditional statements and loops.

Conditional Statements

Conditional statements enable a program to make decisions by evaluating boolean expressions.
The most commonly used conditional statements are if statements, which can take three main forms.

- **if statement:** Executes a block of code if a condition is true.

```
if condition:
    print("Code executed if condition is true")
```

Code 4.3: Example of an if statement

- **if-else statement:** Executes one block if a condition is true, another if it is false.

```
if condition:
    print("Code executed if condition is true")
else:
    print("Code executed if condition is false")
```

Code 4.4: Example of an if-else statement

- **if-elif-else statement:** Handles multiple conditions.

```
if condition1:
    print("Code executed if condition1 is true")
elif condition2:
    print("Code executed if condition2 is true")
else:
    print("Code executed if all conditions are false")
```

Code 4.5: Example of an if-elseif-if statement

The match-case statement in Python provides a way to perform pattern matching, which is useful for handling complex conditions more clearly than multiple `if-elif-else` statements. It allows a variable or expression to be compared against multiple patterns, executing the corresponding block of code for the first matching pattern. A wildcard pattern "_" can be used to define a default case when no other patterns match.

```python
match variable:
    case pattern1:
        print("Code executed if the variable matches pattern1")
    case pattern2:
        print("Code executed if the variable matches pattern2")
    case _:
        print("Code executed if the variable does not matches any pattern")
```

Code 4.6: Example of a match-case statement

Loops

Loops allow a program to repeatedly execute a block of code.

The while loop repeats a block of code as long as a given condition is true. It is often used when the number of iterations is not known in advance and depends on some dynamic condition. Before each iteration, Python evaluates the condition; if it is true, the loop body executes, and the condition is checked again. This process continues until the condition becomes false.

```python
while condition:
    print("code executed repeatedly if the condition is true")
```

Code 4.7: Example of a while statement

The for loop, on the other hand, repeats a block of code for each item in a sequence of items. You do not need to manually manage loop counters, as Python automatically iterates over each element in the sequence. For loops are particularly convenient when the number of iterations is determined by the length of a collection.

```python
for item in items:
    print("code executed for each item")
```

Code 4.8: Example of a for statement

In addition, both while and for loops can include the `break` and `continue` statements within their bodies. The `break` statement immediately exits the loop, allowing the program to skip the remaining iterations. The `continue` statement skips the rest of the current iteration and moves on to the next one. These statements provide finer control over loop execution and make it easier to handle special conditions within loops.

4.2.4 Functions

A function in Python is an abstraction that maps a set of input values to a set of output values. It is a named block of code designed to perform a specific task. Functions can take inputs, called parameters, and optionally produce an output using the return statement. By defining a function once, you can call it multiple times throughout your program, which makes it a reusable unit of work. Functions help organize code, reduce repetition, and make programs easier to read, maintain, and test.

```python
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

Code 4.9: Example function: Factorial in iterative mode

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

Code 4.10: Example function: Factorial in recursive mode

Code 4.9 and Code 4.10 both implement the factorial function. Code 4.9 uses an iterative approach based on a `for` loop, repeatedly multiplying values to compute the result. Code 4.10, on the other hand, uses a recursive approach, where the function calls itself to compute the factorial.

4.2.5 Built-in Data Structures

Besides basic data types such as integers and strings, Python provides several other powerful built-in data structures.

Tuple

A tuple is an ordered collection of elements, which can hold heterogeneous data, including numbers, strings, lists, or even other tuples. Tuples are immutable, so once a tuple is created, its elements cannot be modified, added, or removed.

```
t1 = (1, 2, 3)              # Tuple with three integers
t2 = (1, "hello", 3.14)     # Tuple with mixed types
t3 = (1, (2, 3), [4, 5])    # Nested tuple
print(t1[0])                # 1, access the first element of t1
print(t1[0:2])              # 1,2, access the slice of t1
for item in t1:             # Iterate over all elements of the tuple
    print(item)
```

Code 4.11: Example of tuples

List

A list in Python is similar to a tuple in that it can contain elements of different types, including numbers, strings, or even other lists. Unlike tuples, however, lists are mutable, which means you can add, remove, or modify their elements. This flexibility makes lists a versatile and commonly used data structure in Python programs.

```
a = []                  # Empty list
b = [1, 2, 3]           # List with elements
c = [1, "hello", 3.14]  # List with mixed types
print(b[0])             # access the first element of b
b.append(4)             # Add at the end
b.insert(2, 100)        # Insert 100 at index 1
b.remove(2)             # Remove first occurrence of 2
for n in nums:          # Iterate over all elements of the List
    print(n)
```

Code 4.12: Example of lists

Dictionary

A dictionary in Python is a built-in data structure that stores data as key-value pairs. Each key in a dictionary must be unique, while values can be of any type and may be duplicated. Dictionaries are very efficient for lookups, allowing you to quickly retrieve or modify a value using its key. We will explain the details of the implementation in the next lecture.

```python
students = {
    "Alice": 20,
    "Bob": 21,
    "Charlie": 19
}
print(students["Bob"])  # Output: 21
```

Code 4.13: Example of a dictionary

4.2.6 Class

Python also supports classes, which are a programming abstraction that defines a new type of object. A class specifies the attributes (data) and methods (behavior) that its instances will have. Similar to a function acting as a reusable unit of work, a class is defined once, and you can create multiple instances. Each instance shares the same structure and behavior defined by the class but can hold different data, making classes a powerful tool for organizing and reusing code.

```python
import math # Import the math library

class Ellipse:  # define a class named Ellipse
    def __init__(self, f1, f2, radius_sum): # Constructor
        self.f1 = f1
        self.f2 = f2
        self.radius_sum = radius_sum
        dist_foci = math.dist(f1, f2)
        if radius_sum <= dist_foci:
            raise ValueError("Invalid ellipse")

    def contains_point(self, x, y):         # Is (x,y) inside the ellipse?
        dist_to_f1 = math.dist((x, y), self.f1)
        dist_to_f2 = math.dist((x, y), self.f2)
        return dist_to_f1 + dist_to_f2 <= self.radius_sum

ellipse = Ellipse((0, 0), (4, 0), 6)
print(e.contains_point(3, 0))
```

Code 4.14: Class in Python

Code 4.14 defines a class `Ellipse`, which models an ellipse using its two foci and the sum of distances from any point on the ellipse to the foci. It contains two methods (functions of a class): the constructor `__init__` (default name) and the method `contains_point`. The constructor initializes the ellipse by storing the foci and radius sum, and it checks that the radius sum is greater than the distance between the foci to ensure a valid ellipse. The method `contains_point` returns `True` if the point (x, y) lies inside or on the ellipse, and `False` otherwise. The use case demonstrates creating an ellipse with foci at $(0, 0)$ and $(4, 0)$ and a radius sum of 6. It then tests whether the point $(3, 0)$ is inside the ellipse.

4.3 Vibe Coding with Python

Vibe coding is a recent concept introduced by Andrej Karpathy. It is an approach in which developers use LLMs to generate functional code from natural language prompts, rather than writing it manually. Instead of reviewing or editing the code directly, developers evaluate it through execution results and iteratively prompt the LLM for improvements.

Since LLMs are already powerful tools for programming, this course does not expect students to write Python programs from scratch. Instead, students are expected to formulate clear prompts, test the generated code, and guide the LLM toward correct and improved solutions.

Exercise

1) Use an LLM to generate a Python program that acts as a calculator. Start with basic operations: addition, subtraction, multiplication, and division. Then:

    - Test the calculator with different inputs to verify correctness.
    - Extend the prompt to handle invalid inputs (*e.g.,* division by zero, non-numeric input).
    - Optionally, prompt the LLM to add more features, such as exponentiation, square root, or parentheses handling.