**Module 2. Input & Output**

**Learning Objectives**

* Understand the software development process and its importance in creating functional and reliable programs.
* Learn the steps involved in analyzing a problem, defining specifications, designing a solution, implementing the design through coding, and testing and debugging the program.
* Grasp the concepts of input, processing, and output as the fundamental actions driving computer programs.
* Comprehend the role of variables in Python programming, including naming conventions, the significance of case sensitivity, and the avoidance of reserved keywords and overriding built-in functions.
* Understand the concepts of expressions, output statements, assignment statements, the input function, and multiple assignment to manage data flow and manipulation effectively.

# **The Software Development Process**

Running programs that are already written is easy. But creating a program from scratch is much harder. Computers need very precise instructions to do anything, and writing long programs can be very difficult without a step-by-step plan. The software development process is a way to make creating programs easier and more organized. Here are five important steps in this process:

**Understanding the Problem**

Before writing any code, you need to understand the problem you want to solve. This means thinking about what you want the program to do and who will use it. You need to know exactly what the problem is and what the program should achieve. Understanding the problem well is the first and most important step in creating a useful program.

**Defining What the Program Should Do**

Once you understand the problem, the next step is to decide what the program should do, without worrying about how it will do it. This step is like making a list of features and functions the program needs to have. For example, you might decide that your program needs to take user input, process data, and display results. These details are called specifications, and they act as a blueprint for the program.

**Planning the Solution**

With the specifications ready, you can start planning how to build the program. This step involves deciding how the different parts of the program will work together. You choose the methods and tools (like algorithms and data structures) that will help the program run smoothly. Planning is like drawing a map of the program, showing how everything will fit together.

**Writing the Code**

Now it’s time to write the code. Using a programming language like Python, you turn your plan into a real program. This step requires careful attention to detail to make sure the program works correctly and is easy to read. Good coding practices help make the program more understandable and easier to fix if something goes wrong.

**Testing and Debugging the Program**

The final step is to test the program to find and fix any mistakes (bugs). Testing involves running the program in different scenarios to make sure it works as expected. Debugging is the process of finding and fixing errors in the code. After the program is working, it still needs regular maintenance to add new features, improve performance, and keep it up to date.

Following these five steps helps make the complex task of software development more manageable. Each step builds on the one before, ensuring that the program is built on a solid foundation, designed thoughtfully, and improved through testing and feedback.

# **Example Program: Grade Calculator**

Let's walk through the software development process with a real-world example, creating a program that calculates the percentage score of an exam and determines the grade.

**Step 1: Understanding the Problem**

Alex, a student facing end-of-semester exams, wants a quick way to determine his final scores and grades from raw marks. The main issue is converting raw marks into a percentage and then assigning a letter grade.

**Step 2: Defining What the Program Should Do**

The program will accept two inputs: the student's score and the total possible score. The output will be the percentage score and the corresponding letter grade. The relationship between the inputs (student score and total score) and the output (percentage and grade) is defined by calculating the percentage and comparing it against a grading scale.

**Step 3: Planning the Solution**

Alex decides on a straightforward approach:

* + **Input**: Prompt for the student's score and the total possible score.
  + **Process**: Calculate the percentage (student score / total score \* 100) and determine the letter grade based on the percentage.
  + **Output**: Display the percentage and the letter grade.

Alex drafts the algorithm in pseudocode for clarity:

Input student score and total possible score.

Calculate percentage = (student score / total possible score) \* 100

Determine letter grade based on percentage ranges.

Output percentage and letter grade.

**Step 4: Writing the Code**

Translating the pseudocode into Python, Alex writes the following program:

|  |
| --- |
| # exam\_grade\_calculator.py  # Simple program to calculate exam percentage and grade  # by: Alex  def main():  # Input phase  student\_score = float(input("Enter your exam score: "))  total\_score = float(input("Enter total possible score: "))    # Process phase  percentage = (student\_score / total\_score) \* 100    # Determine letter grade  if percentage >= 90:  letter\_grade = "A"  elif percentage >= 80:  letter\_grade = "B"  elif percentage >= 70:  letter\_grade = "C"  elif percentage >= 60:  letter\_grade = "D"  else:  letter\_grade = "F"    # Output phase  print(f"Your percentage is {percentage:.2f}%, which is a(n) {letter\_grade} grade.")  main() |

**Step 5: Testing and Debugging**

To ensure the program works correctly, Alex tests it with known values, such as:

|  |
| --- |
| Enter your exam score: 45  Enter total possible score: 50  Your percentage is 90.00%, which is a(n) A grade. |

These tests confirm the program accurately calculates the percentage and assigns the correct grade. With successful tests, the program is ready for use, streamlining Alex's exam grading process.

This simplified example once again underscores the importance of methodically progressing through the software development stages to create an effective, user-friendly program.

# **Input, Processing, and Output**

In computer programming, everything a program does can be broken down into three main actions: taking input, processing it, and producing output. This sequence is the basic operation of all computer programs, no matter how simple or complex they are.

**Input**

Input is the data that a program receives to work with. This can come from various sources like keyboard input, files, sensors, or other programs. For example, if you're using a calculator program, the numbers you type in are the input.

**Processing**

Processing is what the program does with the input data. This involves following a set of instructions to perform calculations, modify data, or make decisions. The processing stage is where the program applies its logic to transform the input into a useful result. For instance, in a calculator program, processing would be the actual computation, like adding or multiplying the numbers you entered.

**Output**

Output is the result that the program produces after processing the input. This can be displayed on the screen, saved to a file, or sent to another system. Using the calculator example, the output would be the result of the calculation shown on the display.

**Example: Calculating Pay**

Let's look at a simple example to illustrate these steps. Imagine a program that calculates pay based on hours worked and an hourly rate:

1. **Input:** The program asks for the number of hours worked and the hourly rate.
2. **Processing:** It multiplies the hours worked by the hourly rate to calculate the total pay.
3. **Output:** The program then displays the calculated pay on the screen.

This basic pattern of input, processing, and output is the foundation of all computer programs.

# **Variables**

In Python, variables are essential tools for storing and manipulating data in your programs. By using variables, you can assign names to values, making your code easier to understand and manage. These names, known as identifiers, can also be used for modules and functions, enhancing the readability and maintainability of your code.

**Creating Variables**

In Python, you don’t need a special command to create a variable. A variable is created the moment you assign a value to it. Here are some examples:

|  |
| --- |
| a = 10  b = "Hello World!"  print(a) # 10 will be printed  print(b) # “Hello World!” will be printed |

**Naming Variables**

There are specific rules for naming variables in Python:

* A variable name must begin with a letter or an underscore (\_).
* A variable name cannot start with a number.
* A variable name can only contain letters, numbers, and underscores (A-Z, a-z, 0-9, and \_).
* Variable names are case-sensitive, meaning ***height***, ***Height***, and ***HEIGHT*** are all different variables.
* Variable names cannot be Python keywords.

Here are some examples of valid variable names:

|  |
| --- |
| x  temperature  abc  apple  \_temp123  \_abc\_cde\_123 |

In Python, variables are used to store data that can be manipulated and referenced throughout a program. Variables play a pivotal role by allowing us to assign names to values, making code easier to understand and manage. These names, known as identifiers, can be applied not only to variables but also to modules and functions, enhancing readability and maintainability of the code.

Python does not have a specific command to declare a variable. A variable comes into existence as soon as you assign a value to it for the first time. For example,

**Case Sensitivity**

In Python, identifiers are case-sensitive. This means that ***apple***, ***Apple***, ***aPple***, and ***APPLE*** are all different variables. While you have a lot of freedom in naming variables, it's best to choose names that clearly indicate the variable’s purpose or use.

**Reserved Words**

Python has a set of reserved words, or keywords, that are used by the language itself and cannot be used as variable names. These keywords are integral to Python's syntax and functionality. Here are some examples of Python keywords:

* False
* class
* return
* while

Here is a list of Python keywords:

|  |
| --- |
| None break except in raise  False await else import pass  and continue for lambda try  True class finally is return  as def from nonlocal while  async elif if not with  assert del global or yield |

Table 2-1. Python Keywords

**Best Practices for Variable Naming**

Although it’s possible to use the names of Python’s built-in functions (like print()) as variable names, this is strongly discouraged. Doing so can lead to confusion and errors, as it overwrites the built-in functionality, which can cause unexpected results.

Here are some tips for naming variables:

* Use meaningful names that describe what the variable represents.
* Avoid using single letters or generic names unless they are for temporary or loop variables.
* Stick to lowercase letters and underscores for variable names (this is called *snake\_case*).
* Avoid using Python keywords as variable names to prevent conflicts.

By following these guidelines and best practices, you can write clear, readable, and error-free Python programs.

# **Expressions**

In Python, the key to working with data in programs is through expressions. Expressions are snippets of code that generate or calculate new data values. They can be as simple as a literal value or more complex by combining multiple elements. Let's break down the basics:

**Literals**

Literals are the simplest form of expressions. They directly represent specific values in the code. There are two main types of literals:

* **Numeric Literals**: These are numbers written directly in the code.

|  |
| --- |
| 4.5, 1, 9, 32 |

* **String Literals**: These are sequences of characters enclosed in quotation marks.

|  |
| --- |
| "Hello", "Enter a number between 0 and 1: " |

**Evaluation of Expressions**

When you input an expression into the Python interpreter, it evaluates the expression and returns its value. For instance:

|  |
| --- |
| >>> 7  7  >>> "Hello"  'Hello' |

Here, the number ***7*** is evaluated to ***7***, and the string "***Hello***" is evaluated to ***'Hello'***.

Variables can also be used in expressions. If a variable has been assigned a value, Python evaluates it to that value. If not, it raises a ***NameError***.

|  |
| --- |
| >>> x = 10  >>> x  10  >>> print(x)  10  >>> print(y)  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name 'y' is not defined |

**Combining Expressions with Operators**

Expressions can be made more complex by combining them with operators. Python supports various operators for different operations:

* **Addition (+)**: Adds two numbers.
* **Subtraction (-)**: Subtracts one number from another.
* **Multiplication (\*)**: Multiplies two numbers.
* **Division (/)**: Divides one number by another.
* **Exponentiation (**)\*\*: Raises one number to the power of another.

The order in which operations are performed follows the rules of precedence and associativity, which can be controlled using parentheses.

|  |
| --- |
| >>> x = 10  >>> y = 20  >>> z1 = 3 \* x + y  >>> z2 = 3 \* (x + y)  >>> print(z1)  50  >>> print(z2)  90 |

**String Operations**

Python also allows for operations on strings, such as concatenation, which combines strings together.

|  |
| --- |
| >>> "Hello" + " " + "World"  'Hello World' |

This example shows how strings can be joined to form a new string.

Expressions are a fundamental part of Python programming, allowing you to manipulate and create new data values. We'll explore more operations and data manipulations in later chapters.

# **Output Statements - The *print()* Function**

In Python, the ***print()*** function is a fundamental tool used to display information on the screen. This function helps developers output data, which is useful for both debugging and interacting with users. Let's break down how the ***print()*** function works and how to use it effectively.

**Basic Usage**

The general structure of a ***print()*** statement is:

|  |
| --- |
| print(expression, expression, ..., expression)  print() |

* + **Multiple Expressions**: The ***print()*** function can take multiple expressions separated by commas. These expressions are evaluated and printed in sequence.
  + **Empty Print**: Calling ***print()*** with no arguments outputs a blank line because ***print()*** adds a newline character (***\n***) at the end of its output by default.

**Examples**

Consider these output statements:

|  |
| --- |
| print(10 - 2)  print("Value:", 5, "Next value:", 5+1)  print()  print("Final result:", 8) |

This will produce the following output:

|  |
| --- |
| 8  Value: 5 Next value: 6  Final result: 8 |

Here's what happens:

* ***10 - 2*** is evaluated to ***8*** and printed.
* ***"Value:", 5, "Next value:", 5 + 1*** are evaluated and printed in order. By default, spaces are inserted between each expression.
* ***print()*** with no arguments prints a blank line.
* ***"Final result:", 8*** is evaluated and printed.

**Customizing Output with *sep* and *end***

The ***print()*** function has two important keyword arguments: ***sep*** and ***end***.

* ***sep*** specifies the string inserted between each expression.
* ***end*** specifies the string appended at the end of the output.

By default:

* ***sep*** is a space (' ').
* ***end*** is a newline character ('***\n***').

You can change these defaults to customize the output format.

**Example of Customization**

Consider this example:

|  |
| --- |
| print("Step", 1, sep=": ", end="; ")  print("Step", 2, sep=": ", end=".\n") |

This will produce:

|  |
| --- |
| Step: 1; Step: 2. |

Here's what happens:

* The first ***print()*** uses "***:*** " as the separator and "***;*** " as the end character, so ***Step: 1;*** is printed.
* The second ***print()*** uses "***:*** " as the separator and "***.\n***" as the end character, so ***Step: 2.*** is printed and the output ends with a newline.

By modifying ***sep*** and ***end***, you can control how the output is formatted, making your program's output more readable and aligned with your needs.

# **Assignment Statements**

Assignment statements are a core concept in Python, allowing you to assign values to variables. The essence of an assignment is quite simple: it calculates the value of an expression on the right and assigns it to the variable on the left. This operation forms the basis of most programming tasks in Python.

The basic syntax is:

|  |
| --- |
| variable = expression |

This assigns the value of the expression on the right to the variable on the left. Examples:

|  |
| --- |
| area = 3.14 \* radius \*\* 2  temperature = (fahrenheit - 32) \* 5/9  counter = 10 |

You can reassign variables as many times as needed:

|  |
| --- |
| # Initial assignment  currentValue = 5  print(currentValue) # Outputs: 5  # Reassignment  currentValue = 12  print(currentValue) # Outputs: 12  # Update based on current value  currentValue = currentValue + 3  print(currentValue) # Outputs: 15 |

**Variables as References**

In Python, variables act as references to values, not containers. When you reassign a variable, it points to a new value, and the old value is eventually discarded if not referenced elsewhere.

**Gathering User Input with *input()***

The ***input()*** function is used to get input from the user, and it always returns the input as a string.

Syntax:

|  |
| --- |
| variable = input(prompt) |

Example:

|  |
| --- |
| favorite\_color = input("Please enter your favorite color: ")  print(favorite\_color) |

For numerical input, you need to convert the string to a number:

|  |
| --- |
| age = int(input("Enter your age: "))  temperature = float(input("Enter the current temperature: ")) |

Using ***eval(input())*** can be risky, so it's safer to use type-specific conversion functions.

**Multiple Assignment**

Multiple assignment allows you to assign values to multiple variables in a single statement.

Syntax:

|  |
| --- |
| var1, var2, ..., varN = expr1, expr2, ..., exprN |

Example:

|  |
| --- |
| total, difference = a + b, a - b |

This is useful for swapping values:

|  |
| --- |
| a, b = b, a |

And for capturing multiple user inputs:

|  |
| --- |
| print("Enter two exam scores separated by a comma: ")  score1, score2 = map(int, input().split(','))  average = (score1 + score2) / 2  print("The average score is:", average) |

This technique enhances code efficiency and compactness.

**Summary**

1. The software development process includes five essential steps: Problem Analysis, Defining Specifications, Designing the Solution, Implementation and Coding, and Testing, Debugging, and Maintenance.
2. Effective problem analysis is crucial for understanding the objectives, complexities, and constraints of the intended software solution.
3. Defining specifications outlines what the software will do, serving as a blueprint for development.
4. Designing the solution involves selecting algorithms and data structures to meet the specified requirements.
5. Implementation translates design into code, using a programming language to create functional software.
6. Testing and debugging involve identifying and correcting errors, ensuring the software operates as intended.
7. Continuous maintenance is necessary for adapting the software to new requirements and improving functionality.
8. Input operations capture data from users, processing manipulates this data, and output delivers the results.
9. Variables in Python act as references to values and play a critical role in code readability and management.
10. Python's naming conventions for variables enhance code clarity and prevent errors.
11. Python variables are case-sensitive, allowing for flexibility in naming but requiring precision.
12. Reserved keywords in Python have specific functions and cannot be used as variable names.
13. Expressions in Python are used for data manipulation and can range from simple literals to complex operations involving multiple operators.
14. The ***print()*** function in Python is used for outputting data, allowing for customized formatting through parameters like ***sep*** and ***end***.
15. Assignment statements assign values to variables, with the possibility of reassignment to update or change these values.
16. Python's approach to variable assignment emphasizes references over value containment, facilitating efficient memory management.
17. The ***input()*** function in Python gathers user input as a string, which can then be converted to other types as needed.
18. The ***eval()*** function evaluates string expressions but is recommended to be used cautiously due to security concerns.
19. Multiple assignment in Python allows for the simultaneous assignment of values to multiple variables, streamlining code and facilitating operations like variable swapping.
20. Collecting multiple user inputs in a single step can be achieved through multiple assignment, enhancing code efficiency and user interaction.

**Programming Exercises**

**Exercise 1: Variable Naming and Assignment**

Write a Python program that defines variables for storing a person's name, age, and country of residence. Ensure the variable names follow Python's naming conventions.

**Exercise 2: Basic Input/Output**

Create a program that asks the user for their name and age, then prints a message saying how old they will be in 5 years.

**Exercise 3: Arithmetic Expressions**

Write a program that takes two numbers from the user and prints the result of their addition, subtraction, multiplication, and division.

**Exercise 4: Conditional Letter Grades**

Develop a program that prompts the user for a score (0-100). Based on the score, output the corresponding letter grade (A, B, C, D, F), following a standard grading scale.

**Exercise 5: Temperature Converter**

Implement a program that converts temperature from Fahrenheit to Celsius. The user should enter the temperature in Fahrenheit, and the program should output the temperature in Celsius.

**Exercise 6: Simple Interest Calculator**

Write a Python script that calculates simple interest given the principal amount, rate of interest, and time period. Use the formula:

*simple\_interest = (principal \* rate \* time) / 100*

**Exercise 7: Swapping Two Variables**

Create a program that demonstrates swapping the values of two variables using multiple assignment. The initial values should be input by the user.

**Exercise 8: User Input Validation**

Develop a program that continuously prompts the user for a number between 1 and 10. The program should not terminate until the user inputs a valid number within the specified range.

**Exercise 9: Basic Calculator for Two Numbers**

Implement a simple calculator that takes two numbers and an operator (+, -, \*, /) as input from the user, then prints the result of the operation.

**Exercise 10: Debugging Practice**

Given a piece of code that is supposed to calculate the average of three test scores but contains errors, identify and correct the errors to ensure the program runs correctly.

|  |
| --- |
| score1 = input("Enter score 1: ")  score2 = input("Enter score 2: ")  score3 = input("Enter score 3: ")  average = (score1 + score2 + score) / 3  print("The average score is: ", average) |