**Module 2 -** Variables and Expressions

**Learning Objectives**

* Explain and apply core programming concepts such as input, processing, output, variables, expressions, assignment statements, and data flow management using Python syntax and conventions.
* Identify and use Python data types effectively, including numeric (int, float, complex), text (str), and Boolean (bool) types, and apply appropriate operations and functions like type() for data inspection.
* Demonstrate proficiency in manipulating strings and numeric values, including operations such as concatenation, slicing, formatting, arithmetic, and type conversion using built-in functions like int(), float(), and str().
* Apply logical and comparison operators in decision-making, using Boolean expressions and control flow statements to guide program behavior based on user input and variable conditions.

Contents

[**1.** **Inputs and Outputs** 1](#_Toc202189671)

[**2.** **Example Program: Grade Calculator** 2](#_Toc202189672)

[**3.** **Input, Processing, and Output** 4](#_Toc202189673)

[**4.** **Variables** 5](#_Toc202189674)

[**5.** **Expressions** 7](#_Toc202189675)

[**6.** **Output Statements - The *print()* Function** 9](#_Toc202189676)

[**7.** **Assignment Statements** 11](#_Toc202189677)

[**8.** **Numeric Data Types** 14](#_Toc202189678)

[**9.** **Strings** 17](#_Toc202189679)

[**10.** **Boolean** 21](#_Toc202189680)

[**11.** **Type Conversion** 23](#_Toc202189681)

[**12.** **Operators** 25](#_Toc202189682)

[**Reference and Additional Materials:** 31](#_Toc202189683)

# Inputs and Outputs

Running programs that are already written is easy. But creating a program from scratch is much harder. Computers require precise instructions to perform any task, and writing lengthy programs can be challenging without a step-by-step plan. The software development process is a method for making program creation 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](https://www.geeksforgeeks.org/python/input-and-output-in-python/), 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](https://code.visualstudio.com/docs/python/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](https://realpython.com/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](https://programwithus.com/learn/python/python-operators-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.

# Numeric Data Types

Data types are fundamental in programming, as they define the kind of data variables can store and the operations they support. Python includes default built-in data types grouped into several categories. These types are used to store different kinds of data. This module will explore commonly used simple data types, including text types, numeric types, and Boolean types.

|  |  |
| --- | --- |
| **Data Types** | **Keywords** |
| Text Type | **str** |
| Numeric Types | **int, float, complex** |
| Sequence Types | **list, tuple, range** |
| Mapping Type | **dict** |
| Set Types | **set, frozenset** |
| Boolean Type | **bool** |
| Binary Types | **bytes, bytearray, memoryview** |
| None Type | **NoneType** |

When computers were initially created, their primary role was to perform complex calculations, and this remains a significant use today. Problems that involve mathematical formulas are straightforward to convert into Python programs. In this section, we will delve deeper into programs specifically designed for numerical computations.

Numbers in Python are divided into three main types: integers, floating-point numbers, and complex numbers. Integers are whole numbers without a fractional component, while floating-point numbers contain a decimal point, making them suitable for representing more precise values. Complex numbers consist of a real part and an imaginary part, denoted by the suffix 'j'. Python provides a wide range of operations for working with these number types, including arithmetic operations, type conversion, and built-in functions for mathematical calculations.

Numeric variables are created when you assign a value to them. To check the type of any object in Python, you can use the ***type()*** function.

|  |
| --- |
| x = 10 # int  y = 3.14 # float  z = 1j # complex  print(type(x))  print(type(y))  print(type(z)) |

***Integer***

Integers (***int***) in Python refer to whole numbers that can be positive or negative, without decimal points, and can be of unlimited size.

|  |
| --- |
| x = 0 # zero  y = 42233435353211211 # positive integer  z = -17 # negative integer  print(type(x))  print(type(y))  print(type(z)) |

***Float***

A float, or "floating point number," in Python is a number that can be positive or negative and contains one or more decimals.

|  |
| --- |
| x = 0.5  y = -10.5  z = 3.14  print(type(x))  print(type(y))  print(type(z)) |

The following program demonstrates basic input handling, arithmetic operations, and output formatting for calculating and displaying the sum and average of three test scores.

|  |
| --- |
| # test\_scores.py  # A program to calculate the sum and average of three test scores  def main():  print("Test Scores Calculator")  print()    # Prompting the user to enter three test scores  score1 = float(input("Enter score 1: "))  score2 = float(input("Enter score 2: "))  score3 = float(input("Enter score 3: "))    # Calculating the sum of the scores  total = score1 + score2 + score3    # Calculating the average of the scores  average = total / 3    print()  print("Sum of the scores:", total)  print("Average of the scores:", average)  # Calling the main function to execute the program  main() |

Example output:

|  |
| --- |
| Test Scores Calculator  Enter score 1: 85  Enter score 2: 92  Enter score 3: 78  Sum of the scores: 255.0  Average of the scores: 85.0 |

In the program, the *main()* function prompts the user to enter three test scores (*score1*, *score2*, and *score3*) using *float(input())* to handle decimal input. Then, it calculates the sum of the three scores using *total = score1 + score2 + score3*. In addition, it calculates the average of the three scores by dividing the total sum by 3 (*average = total / 3*). Finally, it prints out the calculated sum and average of the scores.

***Type Conversion***

In Python, you can convert between different data types using *int()* and *float()* methods. The *int()* function is used to convert a number or a string containing a numeric value to an integer.

|  |
| --- |
| num\_float = 3.75  num\_int = int(num\_float)  print(num\_int) # Output: 3 |

The *float()* function converts a number or a string containing a numeric value to a float.

|  |
| --- |
| num\_int = 5  num\_float = float(num\_int)  print(num\_float) # Output: 5.0 |

These methods are handy when you need to change the type of a variable or handle different types of numeric inputs in your Python programs.

***Random Numbers***

Python does not have a built-in function to make a random number, but Python has a built-in module called ***random*** that can be used to make random numbers. You need to import the random module to generate random numbers. The following code generates and displays a random number between 1 and 6.

|  |
| --- |
| import random  print(random.randrange(1, 7)) |

# Strings

In Python, strings can indeed be enclosed in either single (' ') or double (" ") quotation marks, and both are functionally equivalent. This flexibility allows you to choose the style that best fits your needs, especially when dealing with strings that contain quotation marks.

Here are a few examples to demonstrate different usages:

|  |
| --- |
| print("Hello") #Output: Hello  print('Hello') #Output: Hello |

If your string contains a single quote, you can use double quotes to avoid needing to escape the single quote, and vice versa.

|  |
| --- |
| print("It's a beautiful day!")  print('He said, "Python is great!"') |

If you need to use the same type of quotation mark inside the string, you can escape it with a backslash (\):

|  |
| --- |
| print('It\'s a beautiful day!')  print("He said, \"Python is great!\"") |

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

|  |
| --- |
| a = "Hello World"  print(a) |

***Multiline Strings***

Multi-line strings in Python can be defined using triple quotes, either """ or '''. These are particularly useful for strings that span multiple lines, such as long text blocks, code snippets, or formatted text.

|  |
| --- |
| a = """Lorem ipsum dolor sit amet,  consectetur adipiscing elit,  sed do eiusmod tempor incididunt  ut labore et dolore magna aliqua."""  print(a) |

***Slicing Strings***

Slicing is a powerful feature in Python that allows you to extract a subset of characters from a string using a specified range of indices. The general syntax for slicing is ***string[start:end]***, where ***start*** is the index of the first character you want to include, and ***end*** is the index of the character you want to stop before (not included in the slice).

Here's an example that demonstrates how to slice a string to get characters from position 2 to 5 (not included):

|  |
| --- |
| b = "Hello, World!"  print(b[2:5]) #Output: llo |

The syntax for slicing can be broken down into three parts:

* ***start*** (optional): The starting index of the slice. The character at this index is included in the slice. If omitted, the slice starts from the beginning of the string.
* ***end*** (optional): The ending index of the slice. The character at this index is not included in the slice. If omitted, the slice goes until the end of the string.
* ***step*** (optional): The step size, or the interval between indices. If omitted, the default step is 1.

|  |
| --- |
| b = "Hello, World!"  # From index 2 to 5 (5 not included)  print(b[2:5]) # Output: llo  # From start to index 5  print(b[:5]) # Output: Hello  # From index 2 to end  print(b[2:]) # Output: llo, World! |

You can also specify a step size to skip characters:

|  |
| --- |
| b = "Hello, World!"  # Every second character from index 0 to 5  print(b[0:5:2]) # Output: Hlo  # Reverse the string  print(b[::-1]) # Output: !dlroW ,olleH |

Negative indices can be used to count from the end of the string:

|  |
| --- |
| b = "Hello, World!"  # From index -5 to -2 (not included)  print(b[-5:-2]) # Output: orl |

***String Concatenation***

String concatenation in Python allows you to combine two or more strings into one. This can be done using the ***+*** operator. Here are some examples to illustrate this concept:

To concatenate two strings, you simply use the ***+*** operator:

|  |
| --- |
| a = "Hello"  b = "World"  c = a + b  print(c) #Output: HelloWorld |

If you want to add a space between the concatenated strings, you include the space within the quotation marks:

|  |
| --- |
| a = "Hello"  b = "World"  c = a + " " + b  print(c) #Output: Hello World |

***String Format***

In Python, we cannot combine strings and numbers like this:

|  |
| --- |
| age = 25  txt = "Hello, my name is John, and I am " + age #ERROR  print(txt) |

However, strings and numbers can be combined by utilizing ***f***-strings or the ***format()*** method.

Using f-strings is an efficient and powerful way to format strings in Python. They allow you to embed expressions inside string literals using curly braces ***{}***. This makes it easy to combine strings and variables, perform calculations, and format values.

To create an f-string, prefix the string with the letter f and use curly braces {} to include variables or expressions:

|  |
| --- |
| age = 25  txt = f"Hello, my name is John, and I am {age}."  print(txt) #Output: Hello, my name is John, I am 25. |

You also can include variables directly within the curly braces:

|  |
| --- |
| price = 35  txt = f"The price is {price} dollars."  print(txt) #Output: The price is 35 dollars. |

The ***format()*** method provides another way to format strings, which is particularly useful for creating complex formatted strings. This method allows you to insert values into a string using placeholders defined by curly braces ***{}***.

The simplest use of the ***format()*** method involves placing curly braces ***{}*** within a string and passing the values you want to insert as arguments to the ***format()*** method.

|  |
| --- |
| name = "John"  age = 25  txt = "My name is {}, and I am {} years old.".format(name, age)  print(txt) #Output: My name is John, and I am 25 years old. |

You can use positional arguments by placing numbers inside the curly braces, indicating the position of the arguments.

|  |
| --- |
| name = "John"  age = 25  txt = "My name is {0}, and I am {1} years old. {0} is learning Python.".format(name, age)  print(txt) #Output: My name is John, and I am 36 years old. John is learning Python. |

# Boolean

In Python, Booleans are used to evaluate expressions and make decisions as same as in other programming languages. The Boolean data type has two possible values: ***True*** and ***False***.

When you compare two values, the expression is evaluated, and Python returns a Boolean value (True or False).

Python supports several comparison operators that return Boolean values:

|  |  |
| --- | --- |
| > | greater than |
| < | less than |
| == | equal to |
| != | not equal to |
| >= | greater than or equal to |
| <= | less than or equal to |

|  |
| --- |
| print(20 > 10) # True  print(20 == 10) # False  print(20 < 10) # False |

Booleans are often used in control flow statements like if statements to determine which block of code should be executed.

|  |
| --- |
| x = 55  y = 26  if y > x:  print("y is greater than x")  else:  print("y is not greater than x")  #Output: y is not greater than x |

The ***bool()*** function allows you to evaluate any value and returns ***True*** or ***False***. This can be particularly useful for checking the truthiness of values.

|  |
| --- |
| print(bool("Hello World")) # True  print(bool(25)) # True  print(bool("")) # False  print(bool(0)) # False |

Python also provides logical operators to combine Boolean expressions:

* ***and***: Returns ***True*** if both operands are true
* ***or***: Returns ***True*** if at least one operand is true
* ***not***: Returns ***True*** if the operand is false

|  |
| --- |
| x = True  y = False  print(x and y) # False  print(x or y) # True  print(not x) # False |

# Type Conversion

In Python, you can specify a type for a variable using casting, which involves converting a value from one type to another. Python, being an object-oriented language, uses classes to define data types, including its primitive types. Casting is performed using constructor functions like ***int()***, ***float()***, and ***str()***.

***Casting to Integer***

The ***int()*** function converts a value to an integer. This can be done from an integer literal, a float literal (by removing all decimals), or a string literal (provided the string represents a whole number).

|  |
| --- |
| # From an integer literal  x = int(5)  print(x) # Output: 5  # From a float literal (decimals are truncated)  y = int(5.99)  print(y) # Output: 5  # From a string literal  z = int("10")  print(z) # Output: 10  # From a float in string format  # This will raise a ValueError  try:  w = int("10.5")  except ValueError as e:  print(e) # Output: invalid literal for int() with base 10: '10.5' |

***Casting to Float***

The ***float()*** function converts a value to a float. This can be done from an integer literal, a float literal, or a string literal (provided the string represents a float or an integer).

|  |
| --- |
| # From an integer literal  x = float(5)  print(x) # Output: 5.0  # From a float literal  y = float(5.99)  print(y) # Output: 5.99  # From a string literal  z = float("10.5")  print(z) # Output: 10.5  # From an integer in string format  w = float("10")  print(w) # Output: 10.0 |

***Casting to String***

The ***str()*** function converts a value to a string. This can be done from strings, integer literals, float literals, and many other data types.

|  |
| --- |
| # From a string literal  x = str("Hello")  print(x) # Output: Hello  # From an integer literal  y = str(10)  print(y) # Output: 10  # From a float literal  z = str(10.5)  print(z) # Output: 10.5  # From a boolean literal  w = str(True)  print(w) # Output: True |

***Practical Use Cases***

When reading input from a user, it is often read as a string. You might need to convert it to another type for further processing.

|  |
| --- |
| user\_input = input("Enter a number: ")  number = int(user\_input)  print(f"The number you entered is {number}") |

When dealing with numbers in string format, you need to convert them to the appropriate numeric type before performing calculations.

|  |
| --- |
| num1 = "10"  num2 = "20.5"  # Convert to appropriate numeric types  int\_num1 = int(num1)  float\_num2 = float(num2)  # Perform calculation  result = int\_num1 + float\_num2  print(result) # Output: 30.5 |

When you need to format numbers as strings for output purposes, you can use ***str()***.

|  |
| --- |
| age = 25  txt = "My name is John, and I am " + str(age) + " years old."  print(txt) # Output: My name is John, and I am 25 years old. |

# **Operators**

Operators are fundamental in programming as they enable you to perform various operations on variables and values. Python supports several types of operators, including arithmetic, comparison, logical, bitwise, assignment, identity, and membership operators.

***Arithmetic Operators***

Arithmetic operators are used to perform common mathematical operations.

|  |
| --- |
| x = 10  y = 5  # Addition  print(x + y) # Output: 15  # Subtraction  print(x - y) # Output: 5  # Multiplication  print(x \* y) # Output: 50  # Division  print(x / y) # Output: 2.0  # Modulus (remainder)  print(x % y) # Output: 0  # Exponentiation  print(x \*\* y) # Output: 100000  # Floor division  print(x // y) # Output: 2 |

***Comparison Operators***

Comparison operators compare two values and return a Boolean result (***True*** or ***False***).

|  |
| --- |
| x = 10  y = 5  # Equal to  print(x == y) # Output: False  # Not equal to  print(x != y) # Output: True  # Greater than  print(x > y) # Output: True  # Less than  print(x < y) # Output: False  # Greater than or equal to  print(x >= y) # Output: True  # Less than or equal to  print(x <= y) # Output: False |

***Logical Operators***

Logical operators are used to combine conditional statements.

|  |
| --- |
| x = True  y = False  # AND  print(x and y) # Output: False  # OR  print(x or y) # Output: True  # NOT  print(not x) # Output: False |

***Bitwise Operators***

Bitwise operators are used to perform bit-level operations on integers.

|  |
| --- |
| x = 10 # Binary: 1010  y = 4 # Binary: 0100  # AND  print(x & y) # Output: 0 (Binary: 0000)  # OR  print(x | y) # Output: 14 (Binary: 1110)  # XOR  print(x ^ y) # Output: 14 (Binary: 1110)  # NOT  print(~x) # Output: -11 (Binary: -1011)  # Left shift  print(x << 2) # Output: 40 (Binary: 101000)  # Right shift  print(x >> 2) # Output: 2 (Binary: 0010) |

***Assignment Operators***

Assignment operators are used to assign values to variables.

|  |
| --- |
| x = 10  # Assign  x = 5  print(x) # Output: 5  # Add and assign  x += 3  print(x) # Output: 8  # Subtract and assign  x -= 2  print(x) # Output: 6  # Multiply and assign  x \*= 2  print(x) # Output: 12  # Divide and assign  x /= 3  print(x) # Output: 4.0  # Modulus and assign  x %= 3  print(x) # Output: 1.0  # Exponentiation and assign  x \*\*= 2  print(x) # Output: 1.0  # Floor division and assign  x //= 1  print(x) # Output: 1.0 |

***Practical Examples***

You can combine different types of operators to create complex expressions.

|  |
| --- |
| x = 10  y = 5  z = 2  # Combine arithmetic and comparison operators  result = (x + y) > (z \* y)  print(result) # Output: True  # Combine logical operators  is\_valid = (x > y) and (y > z)  print(is\_valid) # Output: True |

Operators are often used in conditional statements to control the flow of the program.

|  |
| --- |
| age = 20  if age >= 18:  print("You are an adult.")  else:  print("You are a minor.")  #Output: You are an adult. |

**Programming Exercises**

**Exercise 1: Basic Numeric Operations**

Write a program that prompts the user to enter two numbers. Calculate and display the sum, difference, product, and quotient of these two numbers. Check and display the data types of the entered numbers and the results.

**Exercise 2: Random Number Generator**

Write a program that imports the random module. Generate and print five random integers between 1 and 100. Then, generate and print a random floating-point number between 0 and 1.

**Exercise 3: Type Conversion**

Write a program that prompts the user to enter a floating-point number. Convert the floating-point number to an integer and print the result. Then, convert the integer back to a floating-point number and print the result.

**Exercise 4: Test Scores Calculator**

Write a program that prompts the user to enter three test scores. Calculate the sum and average of the test scores. Print the sum and average of the scores with appropriate labels.

**Exercise 5: String Handling and Quotation Marks**

Write a Python program that assigns the following text to a variable and prints it: He said, "Python's flexibility with strings is amazing!". Use both single and double quotation marks appropriately to avoid escape characters.

**Exercise 6: String Concatenation**

Write a Python program that asks the user to input their first name and last name separately. Concatenate these inputs with a space in between to form their full name, and print the full name with the message: "Your full name is: <full\_name>".

**Exercise 7: String Formatting with f-strings**

Develop a Python program that prompts the user to enter their name, age, and favorite color. Use an f-string to format and print a message that says: "Hello, my name is <name>, I am <age> years old, and my favorite color is <color>."

**Exercise 8: Boolean Expressions and Comparison Operators**

Write a Python program that asks the user for the price of two different items. Compare the prices using comparison operators and print out which item is more expensive, or if they are priced the same. Use the format method to create the output message.

*This OER module incorporates and adapts content from CSCI1301 (****Introduction to Computer Programming, Application Development, Computer Science I & II****) Open Course**materials by Myungjae Kwak, Jonathan Jenkins, Joobum Kim, and Kevin Floyd from Middle Georgia State University, available on the ALG Manifold platform. Updates have been made to align the material with our course objectives, instructional approach, and current technological practices.*

## Reference and Additional Materials:

<https://alg.manifoldapp.org/projects/itec2260_csci1301-mga>

<https://www.geeksforgeeks.org/python/input-and-output-in-python/>

<https://code.visualstudio.com/docs/python/debugging>

<https://realpython.com/python-variables/>

<https://programwithus.com/learn/python/python-operators-expressions>

Video Tutorials: YouTube link

Leet Code Practice Link:

**Programming Exercise Solutions**

1.

|  |
| --- |
| num1 = float(input("Enter the first number: "))  num2 = float(input("Enter the second number: "))  sum\_result = num1 + num2  difference = num1 - num2  product = num1 \* num2  quotient = num1 / num2 if num2 != 0 else "undefined (division by zero)"  print("Sum:", sum\_result)  print("Difference:", difference)  print("Product:", product)  print("Quotient:", quotient)  print("Type of num1:", type(num1))  print("Type of num2:", type(num2))  print("Type of sum:", type(sum\_result))  print("Type of product:", type(product)) |

2.

|  |
| --- |
| import random  print("Five random integers between 1 and 100:")  for \_ in range(5):  print(random.randint(1, 100))  print("\nRandom floating-point number between 0 and 1:")  print(random.random()) |

3.

|  |
| --- |
| num = float(input("Enter a floating-point number: "))  int\_num = int(num)  print("Converted to integer:", int\_num)  float\_num = float(int\_num)  print("Converted back to float:", float\_num) |
|  |

4.

|  |
| --- |
| score1 = float(input("Enter the first test score: "))  score2 = float(input("Enter the second test score: "))  score3 = float(input("Enter the third test score: "))  total = score1 + score2 + score3  average = total / 3  print("Sum of test scores:", total)  print("Average of test scores:", average) |

5.

|  |
| --- |
| message = 'He said, "Python\'s flexibility with strings is amazing!"'  print(message) |

6.

|  |
| --- |
| first\_name = input("Enter your first name: ")  last\_name = input("Enter your last name: ")  full\_name = first\_name + " " + last\_name  print("Your full name is:", full\_name) |

7.

|  |
| --- |
| name = input("Enter your name: ")  age = input("Enter your age: ")  color = input("Enter your favorite color: ")  print(f"Hello, my name is {name}, I am {age} years old, and my favorite color is {color}.") |

8.

|  |
| --- |
| price1 = float(input("Enter the price of the first item: "))  price2 = float(input("Enter the price of the second item: "))  if price1 > price2:  print("Item 1 is more expensive: {}".format(price1))  elif price2 > price1:  print("Item 2 is more expensive: {}".format(price2))  else:  print("Both items cost the same: {}".format(price1)) |