Intro to Python

Introduction to Variables

In Python, variables are used to store and manage data. They act as placeholders or labels for values, making it easier to work with data in your programs. To use a variable, you need to give it a name and assign a value to it. Here's an introduction to variables in Python:

Declaring a Variable:

In Python, you declare a variable by giving it a name and using the assignment operator (**=**) to assign a value to it. Variable names should follow some rules:

- They must start with a letter (a-z, A-Z) or an underscore (_).
- The rest of the name can contain letters, numbers, and underscores.
- Variable names are case-sensitive, meaning **myVar** and **myvar** are considered different variables.

Examples of variable declaration:

```
name = "Alice" # A variable named 'name' with the value "Alice"

age = 30       # A variable named 'age' with the value 30

pi = 3.14159   # A variable named 'pi' with the value 3.14159

is_student = True  # A variable named 'is_student' with the value True
```

Data Types:

Variables in Python are dynamically typed, which means you don't need to explicitly declare the data type of a variable. Python infers the data type based on the value assigned to it. Common data types in Python include:

- **Integers (int):** Whole numbers without a decimal point.
- **Floating-Point Numbers (float):** Numbers with a decimal point.
- **Strings (str):** Sequences of characters, typically enclosed in single or double quotes.
- **Booleans (bool):** Represents either **True** or **False**.
- **Lists, Tuples, and Dictionaries:** More complex data structures for storing collections of values.

Using Variables:

You can use variables in various ways, such as in mathematical expressions, string manipulation, and logical operations.

x = 5

y = 3

sum = x + y # sum is now 8

greeting = "Hello, " + name # greeting is now "Hello, Alice"

is_adult = age >= 18 # is_adult is True if age is 18 or greater

Reassigning Variables:

You can change the value of a variable by reassigning it with a new value. The variable will then hold the new value.

x = 10 # x is 10

x = 20 # x is now 20

Variable Naming Conventions:

To write clean and readable code, it's essential to follow naming conventions for variables:

- Use descriptive names that indicate the variable's purpose (e.g., **total_price** instead of **tp**).
- Use lowercase letters and separate words with underscores for readability (e.g., **student_age**).
- Avoid using reserved words or built-in function names as variable names (e.g., **print**, **if**, **while**).
- Follow the PEP 8 style guide for Python code to maintain consistency.

Casting

Casting in Python refers to the process of converting one data type into another. It's also known as type conversion or typecasting. Python allows you to cast data from one type to another, provided the conversion is valid. Here are some common types of casting in Python:

- **Implicit Casting (Automatic Conversion):**
- Implicit casting occurs automatically when Python converts data types to complete an operation or expression. For example, when you perform arithmetic operations between different data types, Python will automatically convert them to a common data type, following a hierarchy. For example, in numeric calculations, integers may be automatically converted to floats to preserve precision:

x = 5   # int

y = 2.5 # float

z = x + y # z will be a float (7.5)

Explicit Casting (Manual Conversion):
Explicit casting is when you intentionally convert a data type to another using casting functions. Python provides several built-in functions for explicit casting:

- **int()**: Converts to an integer.
- **float()**: Converts to a float.
- **str()**: Converts to a string.
- **bool()**: Converts to a boolean.

Example of explicit casting:

x = 5.6

y = int(x)   # y will be an integer (5)

z = str(x)   # z will be a string ("5.6")

Data Types

Introduction to Data Types

Data types are fundamental concepts in programming. They define the kind of data that a variable can hold, how the data is stored in memory, and what operations can be performed on it. Data types are essential because they help ensure the integrity and accuracy of data in a program. In Python, a dynamically-typed language, you don't need to declare the data type explicitly; it's determined automatically based on the value assigned to the variable.

Here's an introduction to common data types in Python:

- **Numeric Data Types:**
- **int (Integer)**: Represents whole numbers, both positive and negative. For example, **5**, **-10**, **0**.
- **float (Floating-Point)**: Represents numbers with a decimal point or in scientific notation. For example, **3.14**, **-0.5**, **2e3** (2000.0).
- **Text Data Type:**
- **str (String)**: Represents sequences of characters, enclosed in single (') or double (") quotes. For example, **"Hello, World!"**.
- **Boolean Data Type:**
- **bool (Boolean)**: Represents binary values, **True** or **False**. Booleans are often used for logical comparisons and conditional statements.
- **Sequence Data Types:**

- **list**: Represents ordered, mutable (changeable) collections of items. Lists are defined by square brackets, and items can be of different data types.
- **tuple**: Represents ordered, immutable (unchangeable) collections of items. Tuples are defined by parentheses.
- **range**: Represents an immutable sequence of numbers, often used in loops and for creating ranges.
- **Set Data Types:**
- **set**: Represents unordered, mutable (changeable) collections of unique items. Sets are defined using curly braces.
- **frozenset**: Represents unordered, immutable (unchangeable) collections of unique items.
- **Mapping Data Type:**
- **dict (Dictionary)**: Represents a collection of key-value pairs. Dictionaries are defined using curly braces and colons (**:**) to separate keys and values.
- **None Data Type:**
- **NoneType (None)**: Represents the absence of a value or a null value. It's often used to indicate that a variable has no value assigned to it.

Integers

Integers, often abbreviated as **int**, are a fundamental data type in Python and most programming languages. In Python, integers are used to represent whole numbers, both positive and negative, without any decimal points or fractions. Here's some essential information about integers in Python:

- **Declaring Integers:**
- You can declare integer variables by assigning whole numbers to them. For example:

```
age = 25

quantity = -10

population = 7_800_000_000  # You can use underscores for readability
```

Mathematical Operations:

You can perform various mathematical operations using integer variables, including addition, subtraction, multiplication, division, and more.

```
x = 10

y = 5

sum_result = x + y    # Sum of x and y

diff_result = x - y   # Difference between x and y

prod_result = x * y   # Product of x and y
```

```
div_result = x / y    # Division of x by y (result will be a float)
```

- **Data Type:**
- The **int** data type represents integers, and Python automatically assigns this data type when you declare a variable with a whole number.
- **Arithmetic Operators:**
- Python supports various arithmetic operators for working with integers, such as **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), **//** (integer division), **%** (modulo or remainder), and ****** (exponentiation).

```
a = 10

b = 3

div_result = a / b        # Floating-point division (3.333...)

floor_div_result = a // b # Integer division (3)

remainder = a % b         # Remainder after division (1)

power_result = a ** 2     # Exponentiation (a squared, 100)
```

- **Limits:**
- In Python, integers can be as large or small as your system's memory allows. Python 2 had a separate **long** data type for arbitrarily large integers, but in Python 3, **int** can represent integers of arbitrary size.
- **Type Conversion:**
- You can convert other data types into integers using the **int()** constructor. For example:

```
number_as_string = "42"

integer_number = int(number_as_string)  # Convert a string to an integer
```

- **Bitwise Operations:**
- Python supports bitwise operations on integers, such as bitwise AND (**&**), OR (**|**), XOR (**^**), shift left (**<<**), and shift right (**>>**). These are useful for low-level operations and working with binary data.
- **Use Cases:**
- Integers are used in a wide range of applications, including counting, indexing, loop control, and representing numerical data in mathematical and scientific computations.
- **Overflow:**
- While Python's **int** type has no fixed size, it's still limited by the available memory. In extreme cases, very large integer calculations can lead to memory issues.

Booleans

Booleans, often referred to as **bool**, are a fundamental data type in Python and many other programming languages. Booleans are used to represent binary values, typically indicating one of two possible states: **True** or **False**. Booleans are essential for decision-making and logical operations in programming. Here's what you need to know about Booleans in Python:

- **Declaring Booleans:**
- In Python, you can directly declare Boolean variables with the values **True** or **False**. For example:

```
is_student = True

is_adult = False
```

Logical Operations:

Booleans are primarily used for logical operations, such as comparisons and conditional statements. These operations return Boolean values. For example:

```
x = 10

y = 5


is_greater = x > y  # True, as 10 is greater than 5

is_equal = x == y  # False, as 10 is not equal to 5
```

- **Comparison Operators:**
- Python provides various comparison operators to compare values and return Boolean results. These operators include **>**, **<**, **>=**, **<=**, **==** (equality), and **!=** (inequality).
- **Logical Operators:**
- You can perform logical operations on Booleans using operators like **and**, **or**, and **not**. These operators are used to combine and manipulate Boolean values.

```
a = True

b = False


result_and = a and b  # False (both 'a' and 'b' must be True)

result_or = a or b    # True (either 'a' or 'b' is True)

result_not_a = not a  # False (negation of 'a')
```

- **Data Type:**
- The **bool** data type represents Boolean values in Python. When you perform a logical operation or comparison, the result is automatically of type **bool**.
- **Truthiness and Falsiness:**

- In Python, values have an inherent truthiness or falsiness. For example, numeric values other than **0** are considered **True**, and an empty sequence (e.g., an empty list or string) is considered **False**.
- **Use Cases:**
- Booleans are commonly used in conditional statements, loops, and decision-making processes. They determine which branch of code to execute based on a condition.

Type Conversion:

You can convert other data types into Booleans using the **bool()** constructor. In general, an object is considered **True** unless it has one of the following:

- The value **False**.
- The integer **0**.
- An empty sequence (e.g., an empty string, list, or dictionary).
- **None**.

Floating point numbers

Floating-point numbers, often referred to as **float**, are a fundamental data type in Python and most programming languages. Floating-point numbers are used to represent real numbers with decimal points or in scientific notation. They are essential for performing precise mathematical calculations and working with non-integer values. Here are some important points to understand about floating-point numbers in Python:

- **Declaring Floating-Point Numbers:**
- In Python, you can declare floating-point variables by assigning values that include decimal points. For example:

```
pi = 3.14159

temperature = -7.5

my_float = 2.0  # Even if there's no decimal part, it's still a float
```

- **Data Type:**
- The **float** data type represents floating-point numbers in Python. Python automatically assigns this data type when you declare a variable with a value that includes a decimal point.
- **Arithmetic Operators:**
- Python supports various arithmetic operators for working with floating-point numbers, including **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), and ****** (exponentiation).

```
a = 3.0

b = 1.5
```

power_result = a ** b # Exponentiation (a raised to the power of b)

- **Precision and Rounding:**
- Floating-point numbers in Python are implemented using the IEEE 754 standard, which can lead to precision issues when performing certain calculations. It's important to be aware of this when working with floating-point numbers. You can use functions like **round()**, **math.ceil()**, and **math.floor()** to control the precision of floating-point values.
- **Type Conversion:**
- You can convert other data types into floating-point numbers using the **float()** constructor. For example:

num_as_string = "3.14"

float_number = float(num_as_string) # Convert a string to a float

Complex Numbers

Complex numbers are a mathematical concept that extends the real number system to include numbers with both a real part and an imaginary part. In Python, complex numbers are represented using the **complex** data type. A complex number is written in the form **a + bj**, where **a** is the real part, **b** is the imaginary part, and **j** is the square root of -1, denoted as **1j**.

Here's how you can work with complex numbers in Python:

Declaring Complex Numbers:

You can declare complex numbers directly by using the **complex()** constructor, or by writing them in the form **a + bj**. For example:

z1 = complex(3, 4) # Represents 3 + 4j

z2 = 2 - 1j   # Represents 2 - 1j

Real and Imaginary Parts:

You can access the real and imaginary parts of a complex number using the **.real** and **.imag** attributes, respectively:

z = 3 + 2j

real_part = z.real # real_part is 3.0

imag_part = z.imag # imag_part is 2.0

Conjugate:

The conjugate of a complex number **a + bj** is **a - bj**. You can find the conjugate of a complex number using the **.conjugate()** metho

z = 4 + 3j

conjugate_z = z.conjugate() # Represents 4 - 3j

Complex Arithmetic:

You can perform arithmetic operations on complex numbers, including addition, subtraction, multiplication, and division, just like with real numbers. Python handles complex arithmetic seamlessly:

z1 = 3 + 4j

z2 = 1 - 2j

sum_result = z1 + z2   # Represents 4 + 2j

difference_result = z1 - z2 # Represents 2 + 6j

product_result = z1 * z2   # Represents 11 - 2j

quotient_result = z1 / z2   # Represents (-2 - 1j)

Absolute Value:

You can find the absolute value (also called the modulus or magnitude) of a complex number using the **abs()** function:

Strings

Strings, often referred to as **str**, are a fundamental data type in Python and many other programming languages. Strings are used to represent sequences of characters, such as text or symbols. They are versatile and extensively used for various purposes in programming. Here's what you need to know about strings in Python:

- **Declaring Strings:**
- You can declare string variables by enclosing characters in either single (**'**) or double (**"**) quotes. For example:

name = "Alice"

message = 'Hello, World!'

String Concatenation:

You can combine strings using the **+** operator, known as string concatenation:

first_name = "John"

last_name = "Doe"

full_name = first_name + " " + last_name # "John Doe"

String Indexing:

Strings are indexed, meaning each character in a string has a position or index starting from 0. You can access individual characters by specifying their index:

```
text = "Hello"

first_character = text[0]  # "H"

second_character = text[1]  # "e"
```

String Slicing:

You can extract a portion of a string using slicing. Slicing allows you to create a new string that includes a range of characters:

```
text = "Python Programming"

sub_string = text[7:18]  # "Programming"
```

String Length:

You can find the length of a string (the number of characters it contains) using the **len()** function:

```
text = "Hello, World!"

length = len(text)  # 13
```

String Methods:

Python provides a wide range of string methods for manipulating and formatting strings. These methods include **lower()**, **upper()**, **strip()**, **split()**, **replace()**, and many more.

```
text = "   Python Programming   "

trimmed_text = text.strip()  # Removes leading and trailing spaces

words = text.split()  # Splits the string into a list of words
```

String Escaping:

If you need to include special characters within a string, you can escape them using the backslash (\) character. For example, to include a quotation mark within a string enclosed by double quotes:

```
message = "She said, \"Hello!\""
```

String Formatting:

You can format strings using various techniques, such as f-strings, the **.format()** method, or the **%** operator:

```
name = "Alice"

age = 30

formatted_text = f"My name is {name} and I am {age} years old."
```

Lambda Expressions

In Python, a lambda expression is a small, anonymous function defined using the **lambda** keyword. Lambda expressions can take any number of arguments but can only have one expression. They are typically used for short, simple operations and are often employed in situations where you need a small, throwaway function without the need to give it a formal name.

The general syntax of a lambda expression is as follows:

lambda arguments: expression

Here's an example of a simple lambda expression: add = lambda x, y: x + y

result = add(3, 4) # result will be 7

In this example, the lambda expression takes two arguments **x** and **y**, and it returns their sum. The lambda expression is assigned to the variable **add**, and you can then call it like a regular function.

Lambda expressions are often used in conjunction with higher-order functions, such as **map()**, **filter()**, and **sorted()**, which accept functions as arguments. Here are some common use cases for lambda expressions:

- **Sorting Lists of Complex Objects:**
- You can use lambda expressions as a key function when sorting a list of complex objects based on a specific attribute. For example, to sort a list of dictionaries by the "age" key:

people = [{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]

sorted_people = sorted(people, key=lambda person: person["age"])

Filtering Lists:

You can use lambda expressions to filter items in a list based on a certain condition. For example, to filter even numbers from a list of integers:

numbers = [1, 2, 3, 4, 5, 6]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

Mapping Values:

Lambda expressions can be used to create new lists by applying a function to each item in an existing list. For example, to square each number in a list:

numbers = [1, 2, 3, 4, 5]

squared_numbers = list(map(lambda x: x ** 2, numbers))

Introduction to Operators

In Python, operators are symbols that allow you to perform operations on data, variables, and objects. Python supports a variety of operators, which can be broadly categorized into the following types:

- **Arithmetic Operators:**
- Arithmetic operators are used to perform mathematical operations. These operators include:
- Addition (**+**): Adds two values.
- Subtraction (**-**): Subtracts the right operand from the left operand.
- Multiplication (*****): Multiplies two values.
- Division (**/**): Divides the left operand by the right operand.
- Modulus (**%**): Returns the remainder of the division.
- Exponentiation (******): Raises the left operand to the power of the right operand.
- Floor Division (**//**): Performs division and returns the largest whole number less than or equal to the result.

```
a = 5

b = 2

addition = a + b  # 7

subtraction = a - b  # 3

multiplication = a * b  # 10

division = a / b  # 2.5

modulus = a % b  # 1

exponentiation = a ** b  # 25

floor_division = a // b  # 2
```

Comparison Operators:

- Comparison operators are used to compare values and return a Boolean result (True or False). These operators include:
- Equal to (**==**): Checks if two values are equal.
- Not equal to (**!=**): Checks if two values are not equal.
- Greater than (**>**): Checks if the left operand is greater than the right operand.
- Less than (**<**): Checks if the left operand is less than the right operand.
- Greater than or equal to (**>=**): Checks if the left operand is greater than or equal to the right operand.
- Less than or equal to (**<=**): Checks if the left operand is less than or equal to the right operand.

Example:

```
x = 5

y = 3

equal = x == y  # False

not_equal = x != y  # True

greater_than = x > y  # True

less_than = x < y  # False

greater_than_or_equal = x >= y  # True

less_than_or_equal = x <= y  # False
```

Logical Operators:

- Logical operators are used to perform logical operations on Boolean values. These operators include:
- Logical AND (**and**): Returns True if both operands are True.
- Logical OR (**or**): Returns True if at least one operand is True.
- Logical NOT (**not**): Returns the opposite of the operand's value.

Example:

```
p = True

q = False

logical_and = p and q  # False

logical_or = p or q  # True

logical_not_q = not q  # True
```

Assignment Operators:

- Assignment operators are used to assign values to variables. The basic assignment operator is **=**.

Example:

```
a = 10
```

Bitwise Operators:

- Bitwise operators are used to manipulate individual bits in integer values. These operators include:
- Bitwise AND (**&**): Performs a bitwise AND operation.
- Bitwise OR (**|**): Performs a bitwise OR operation.
- Bitwise XOR (**^**): Performs a bitwise XOR (exclusive OR) operation.
- Bitwise NOT (**~**): Performs a bitwise NOT operation (bitwise inversion).

- Left Shift (**<<**): Shifts bits to the left.
- Right Shift (**>>**): Shifts bits to the right.

Example:

```
a = 5  # Binary: 0101

b = 3  # Binary: 0011

bitwise_and = a & b  # Result: 1 (Binary: 0001)

bitwise_or = a | b  # Result: 7 (Binary: 0111)

bitwise_xor = a ^ b  # Result: 6 (Binary: 0110)

bitwise_not_a = ~a  # Result: -6

left_shift = a << 1  # Result: 10 (Binary: 1010)

right_shift = a >> 1  # Result: 2 (Binary: 0010)
```