01-python-numeric-data-types

September 22, 2024

1. Numbers in Python

In this section, We'll dive into the world of numbers and learn how to use them effectively in Python.

We'll learn about the following topics:

- 1.1. Types of Numbers in Python
- 1.2. Python Built-in Arithmetic Operators
- 1.3. Python Built-in Arithmetic Functions
- 1.4. Variable Assignment
- 1.5. Variable Reassignment
- 1.6. Determining Variable Type

1.1. Types of Numbers in Python:

Python offers different kinds of numbers. We'll mainly work with integers and floating-point numbers.

Integers are whole numbers, positive or negative. For example: 5 or -5.

Floating-point numbers are numbers with a decimal point or those represented in scientific notation. They can be used to represent a wide range of values, including very large or very small numbers. For example: 4E2 (4 times 10 to the power of 2) or 4.56.

Here is a table of the two main types we will spend most of our time working with some examples:

Integers:

1, 5,- 7, 1400

Floating-point numbers:

1.3, -0.8, 2e2, 4E9

Be aware that Python recognizes 1 an integer, but 1.0 as a float.

To make large numbers easier to read, we often group digits into threes with commas. Python doesn't allow commas within numbers, but you can use underscores _ instead.

[1]:
```
#Integer
1_000_000
```

[1]:
```
1000000
```

[2]:
```
#Float
1_000_000.0
```

[2]:
```
1000000.0
```

1.2. Python Built-in Arithmetic Operators:

[3]:
```
#Addition
6+3
```

[3]:
```
9
```

[4]:
```
#Subtraction
7-2
```

[4]:
```
5
```

[5]:
```
#Multiplication
2*5
```

[5]:
```
10
```

[6]:
```
#Division
9/2
```

[6]:
```
4.5
```

[7]:
```
#Floor Division
9//2
```

[7]:
```
4
```

The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

[8]:
```
#Modulo
12%7
```

[8]:
```
5
```

The % operator returns the remainder after division.

```
[9]: #Power
     5**2
```

```
[9]: 25
```

```
[10]: #Square Root
      16**0.5
```

```
[10]: 4.0
```

While we've used the built-in exponentiation operator for square roots, there are more specialized libraries like NumPy that offer efficient functions for mathematical operations. We'll explore NumPy in detail later in the course.

```
[11]: #Order of Operations followed in Python
      4 + 20 * 20 + 8 / 2
```

```
[11]: 408.0
```

Python has a set of rules for the order in which operations are performed. Multiplication and division have higher precedence than addition and subtraction. So, in the expression 4 + 20 * 20 + 8 / 2, the multiplication is done first, resulting in 1 00. Then, the divisioin is d one. Finally, the additions are performed from left to right.

```
[12]: 4 + (20*20) + (8/2)
```

```
[12]: 408.0
```

You can use parentheses to specify and change orders.

```
[13]: #Using parentheses to specify orders
      (4 + 20) * ((20 + 8) / 2)
```

```
[13]: 336.0
```

1.3. Python Built-in Arithmetic Functions:

- `pow()`: This function in Python is equivalent to the `**` operator.

```
[14]: pow(2, 3)
```

```
[14]: 8
```

The `pow()` function accepts an optional third argument that computes the first number raised to the power of the second number, then takes the modulo with respect to the third number. In other words, `pow(x, y, z)` is equivalent to `(x ** y) % z`.

```
[15]: pow(2, 3, 2)
```

```
[15]: 0
```

- **round()**: It rounds a number to the nearest integer.

```
[16]: round(3.6)
```

```
[16]: 4
```

To round a number to a specific number of decimal places, you can utilize the `round()` function and provide a second argument specifying the desired decimal precision.

```
[17]: round(3.2359, 3)
```

```
[17]: 3.236
```

- **abs()**: Returns a positive number of the same type as its argument.

```
[18]: abs(-5.0)
```

```
[18]: 5.0
```

```
[19]: abs(3)
```

```
[19]: 3
```

1.4. Variable Assignment:

Now that we understand how to perform calculations in Python, let's learn about variables and how to assign values to them.

A variable in Python is a container that stores a value. It's like a box labeled with a name, and you can put different values inside that box. The name of the variable allows you to refer to the value stored within it.

Use the assignment operator (=) to assign a value to a variable.

```
[20]: #Create a variable and assign a value
      a = 25
```

Now if I call a in my Python script, Python will treat it as the number 25.

```
[21]: a
```

```
[21]: 25
```

You can use all basic arithmetic operation using variables containing values.

```
[22]: a + a
```

```
[22]: 50
```

[23]:
```
a * 2
```

[23]:
```
50
```

Variables can also be assigned like this:

[24]:
```
x1, x2, x3 = 4, 5, 6

print(x1)
print(x2)
print(x3)
```

```
4
5
6
```

The names you use when creating these labels need to follow a few rules:

- 1. Names can not start with a number.
- 2. There can be no spaces in the name, use _ instead.
- 3. Can't use any of these symbols :'",<>/?|()!@#$%^&*~-+
- 4. It's considered best practice (PEP8) that names are lowercase.
- 5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names because these characters can be easily confused with other characters.
- 6. Avoid using words that have special meaning in Python like "if" "list" and "str"

Using variable names can be a very useful way to keep track of different variables in Python. For example:

[25]:
```
# Define variables
product_price = 599.99
quantity_sold = 100

#Calculate total revenue
total_revenue = product_price * quantity_sold

#Result
total_revenue
```

[25]:
```
59999.0
```

1.5. Variable Reassignment

In Python, variables can be reassigned to hold different values throughout the execution of a program. This means you can change the value stored in a variable at any time. This feature is also known as **dynamic typing** and it differs from other languages that are statically typed.

Pros of Dynamic Typing in Python:

- Flexibility: You can easily change the data type of a variable as needed, making your code more adaptable to changing requirements.
- Simpler syntax: There's no need for explicit type declarations, which can make your code more concise and easier to read.
- Rapid development: Dynamic typing can speed up development, as you don't have to worry about type checking errors.
- Prototype development: It's well-suited for prototyping and experimentation, as you can quickly iterate and change the data types without worrying about compatibility issues.

Cons of Dynamic Typing in Python:

- Potential for runtime errors: While dynamic typing can be convenient, it can also lead to runtime errors if you accidentally assign an incompatible type to a variable.
- Reduced performance: In some cases, dynamic typing can lead to slightly slower execution compared to statically typed languages, especially when dealing with large datasets or performance-critical applications.
- Debugging challenges: Debugging can be more difficult, as you may need to track the types of variables throughout your code to understand the root cause of errors.
- Reduced code readability: In some cases, the lack of explicit type annotations can make it harder to understand the intent of your code, especially for complex data structures.

```
[26]: a
```

```
[26]: 25
```

```
[27]: a = 15
```

```
[28]: a
```

```
[28]: 15
```

We can also use the variables themselves when doing the reassignment. Here is an example:

```
[29]: a = a - 5
```

```
[30]: a
```

```
[30]: 10
```

```
[31]: a = a + a
```

```
[32]: a
```

```
[32]: 20
```

Some operators combine an arithmetic operation with an assignment, providing a concise way to perform calculations and update variable values in a single step. For example:

`+=:` Adds the right-hand operand to the left-hand operand and assigns the result to the left-hand operand.

```
[33]: a *= 2

      a
```

```
[33]: 40
```

```
[34]: a -= 10  #Meaning: a = a + 10

      a
```

```
[34]: 30
```

1.6. Determining Variable Type:

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include which we will discuss later: * **int** (for integer) * **float** * **str** (for string) * **list** * **tuple** * **dict** (for dictionary) * **set** * **bool** (for Boolean True/False)

```
[35]: a
```

```
[35]: 30
```

```
[36]: type(a)
```

```
[36]: int
```

```
[37]: b = [1,4]
```

```
[38]: type(b)
```

```
[38]: list
```