Variables and Data Types

This segment is a natural extension dedicated to the dynamic realm of 'Variables and Data Types.' Let's further understand how to ascertain the type of a variable.

The type() function is our tool of choice for this task, allowing us to dynamically determine the data type of a variable at runtime.

The type() function is a handy tool that helps us figure out the data type of a variable. Imagine you have a variable, but you're not sure if it's a number, a word, or something else entirely. That's where type() comes in.

Here's how it works:

- number1 = 42

  print(type(number1))

  Output: <class 'float'>(**Show the output**)

  In this example, we declare a variable number1 and assign it the value 42, which is an integer. By using type(number1), we ask Python to tell us the type of the variable, and it prints <class 'int'>, indicating that it's an integer.

Let’s see another example

- number2 = 3.14

  print(type(number2))

  Output: <class 'float'>(**Show the output**)

In this example, the variable number2 is a floating-point variable (a number with a decimal point). The type(number2) shows <class 'float'>.

Let’s see one more example

- word = "Hello, Python!"

print(type(word))

Output: <class 'float'>(Show the output)

Lastly, in this example, we have a string variable word. The type(word) reveals <class 'str'>, indicating that it's a string.

So, the type() function helps us dynamically figure out what kind of data we're working with, making it a valuable tool in our Python journey. Feel free to use it whenever you're curious about the type of variable in your code!

Let's continue and move on to accessing values in variables of different data types, focusing on indexing.

So, indexing is how we point to or choose a particular item from a group by referring to its position or order. An index is a numerical value representing an item's position in a group, helping us locate and access that specific item easily.

Let’s understand this with an example:

Think of a list of your favorite snacks: Pizza, Popcorn, Ice Cream, Chips, Cookies



Now, imagine you want to talk about your second favorite snack:

You'd say, "My second favorite snack is Popcorn." In this case, you're using the order or position of Popcorn in the list to talk about it. This idea of talking about things in order without using specific numbers is like indexing. The "second favorite" part is the index—it helps you pick out the right snack from the list.

Now, let's apply the concept of indexing to Python. In Python, you often work with variables of different data types, and you can use indexing to access specific elements within those variables.

Let's consider a simple example using a Python list:

(In Python)

- favourite_snacks = ["Pizza", "Popcorn", "Ice Cream", "Chips", "Cookies"]

Just like before, this is a list of your favorite snacks.

If you want to talk about your second favorite snack which is Popcorn, you can do it like this:

- second_favourite_snack = favorite_snacks[1]

The square brackets ([]) are used for indexing, and the index starts from 0. So, to access the second element ("Popcorn"), we use favorite_snacks [1], so the second position has an index of 1.

The value of second_favourite_snack is then printed, which will output: "My second favorite snack is Popcorn."

- print("My second favorite snack is", second_favorite_snack)

If you want to talk about your fourth favorite snack which is Chips, you can use:

- fourth_favorite_snack = favorite_snacks[3]

So, the generalized syntax for indexing in Python is as follows:

element = iterable[index]

Here,

'iterable' refers to the name of the data structure you're working with, such as a list, tuple, or string. The index represents the position of the element you want to access, Remember that in Python, indexing starts from 0, so the first element in the list has an index of 0, the second element has an index of 1, and so on.

Now, let's suppose we want to access an element that doesn't exist in the list:

Let’s try:

- fourth_favorite_snack = favorite_snacks[5]

print(fourth_favorite_snack)

On printing the fourth_favorite_snack we will get an error.

Because, in this case, we're trying to access the element at index 5, which goes beyond the limits of the list. Remember that the list is like a row of boxes, and we start counting from 0. So, for our list of favorite snacks, the valid box numbers (indices) are 0, 1, 2, 3, and 4. Trying to grab something from box 5 doesn't work because there's no box number 5 in this case.

So, the error is a way for Python to tell us that we're trying to reach for something beyond the end of our list.

Now, let's explore a handy feature called negative indexing. In simple terms, negative indexing allows us to count elements from the end of the list, starting with -1 for the last element, -2 for the second-to-last element, and so forth. This is a convenient way to access elements from the end of the sequence without explicitly calculating the position.

Building upon our previous example of favorite snacks:

- favorite_snacks = ["Pizza", "Popcorn", "Ice Cream", "Chips", "Cookies"]

If we want to refer to our last favorite snack, "Cookies," we can use negative indexing:

- last_favorite_snack = favorite_snacks[-1]

Now, 'last_favorite_snack' holds the value "Cookies," representing the snack at the end of our list. Let’s print it and see for ourselves.

- print(last_favorite_snack)

And see the output is

- Cookies.

So positive indexing is useful when you know the position from the beginning, while negative indexing is helpful when referring to positions relative to the end of the sequence.

Now, let's look at some examples of how we access elements from variables of different data types:

Let’s take an example of string datatype:

We start by creating a variable named favorite_word and assigning it the string value "Python".

(In Python)

- favorite_word = "Python"
- first_letter = favorite_word[0] # Positive indexing

first_letter = favorite_word[0]: Positive indexing is used to access the element at position 0 in the string. In Python, indexing starts from 0, so the first letter "P" is at index 0. The value "P" is assigned to the variable first_letter.

- last_letter = favorite_word[-1] # Negative indexing

last_letter = favorite_word[-1]: Negative indexing is used to access the last element in the string. The index -1 refers to the last position, -2 to the second-to-last, and so on. In this case, it retrieves the last letter "n" and assigns it to the variable last_letter.

- print("First letter:", first_letter)

The output of the first print comes out to be "First letter: P"

- print("Last letter:", last_letter)

The output of the second print comes out to be "Last letter: n"

Now let’s consider an example for retrieving elements in a tuple.

Suppose we have a tuple of favorite numbers as:

(In Python)

- favorite_numbers = (3, 7, 1, 8, 4)

Now let’s suppose we want to access the third number and last number from favorite_numbers

- third_number = favorite_numbers[2]
- second_last_number = favorite_numbers[-2]

Now let’s print this third_number and second_last_number

- print("Third number:", third_number)
- print("Second-to-last number:", second_last_number)

So the output that we get is :

Third number: 1 and

Second-to-last number: 8

Now let’s see how we can access elements in a dictionary.

In Python, dictionaries are collections of key-value pairs, and you can access elements in a dictionary using the keys. Each key is associated with a specific value. Here's how you can access elements in a dictionary:

Suppose we have a dictionary of favorite movies:

- favorite_movies = {"Action": "Inception", "Comedy": "Dumb and Dumber", "Drama": "The Shawshank Redemption"}

To access elements, you can directly use the key to retrieve the associated value:

Suppose we want to access values associated with Comedy and Action:

- comedy_movie = favorite_movies["Comedy"]

action_movie = favorite_movies["Action"]

And when we print it:

- print("Favorite Comedy:", comedy_movie)
- print("Favorite Action:", action_movie)

This is the output that we get:

Favorite Comedy: Dumb and Dumber

Favorite Action: Inception

However, attempting to access a key using its value directly is not directly supported by the standard dictionary methods.

Now let’s see an example to retrieve elements in Sets.

We have a set of favorite_colors

- favorite_colors = {"Red", "Green", "Blue"}

print(favorite_colors[0])

Attempting to use indexing on a set, will result in an Error because sets do not support indexing.

In Python, sets are unordered collections of unique elements. Unlike sequences (such as lists or tuples), sets do not have a specific order, and there is no concept of indexing to access elements by position. The primary purpose of a set is to provide a collection of distinct and unordered elements.

Since sets are unordered, there is no guarantee of the order in which elements are stored internally, and therefore, there is no direct way to access elements by an index.

Now, let's delve into another powerful concept in Python called slicing. Slicing allows us to extract a portion, or a "slice," of elements from an iterable, such as a list, tuple, or string. It provides a convenient way to work with multiple elements at once.

In Python, the basic syntax for slicing is as follows:

subset = iterable[start:stop]

Here, start represents the index of the first element you want in the slice, and stop represents the index of the first element you don't want in the slice. The resulting subset includes elements from the start index up to, but not including, the stop index.

Let's break down the explanation with a simple example:

Imagine you have a list of fruits:

- fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi']

Now, let's use the subset notation iterable[start:stop] to create a subset of this list:

- subset = fruits[1:4]

In this example:

start is the index of the first element you want in the slice, which is 1. So, we start with the element at index 1, which is 'banana'.

stop is the index of the first element you don't want in the slice, which is 4. So, we include elements up to, but not including, the element at index 4. The element at index 4 is 'kiwi', but we stop before including it.

The resulting subset includes elements from index 1 up to, but not including, index 4. Therefore, the subset will be:

- ['banana', 'orange', 'grape']

So the subset contains the elements from the original list starting from 'banana' up to, but not including, 'kiwi'.

Let's continue with the slicing explanation, introducing the concept of step size. In slicing, the step size determines the interval between elements that are included in the subset. The syntax for slicing with a step size is as follows:

subset = iterable[start:stop:step]

Here, the step parameter indicates the number of indices between elements. If not specified, it defaults to 1. Let's use a numerical example to illustrate this:

Let’s consider a list of numbers

- numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Now, let's create a subset without a step size:

- sub = numbers[1:8]

In this case:

start is 1, so we start with the element at index 1, which is 1.

stop is 8, so we include elements up to, but not including, the element at index 8.

The resulting subset includes elements from index 1 up to, but not including, index 8.

Therefore, the subset which is sub will be:

- [1, 2, 3, 4, 5, 6, 7]

Now, let's proceed with the next step, introducing the concept of step size:

- subset = numbers[1:8:2]

As mentioned earlier, this subset includes elements from index 1 up to, but not including, index 8 because the start is 1 and the stop is 8.

Since step is 2, we include every second element in the specified range.

The resulting subset includes elements from index 1 up to, but not including, index 8 with a step size of 2.

Therefore, a subset will be: [1, 3, 5, 7]

Diving into a crucial concept in programming, let's explore mutable and immutable data types.

Mutable means changeable. Once you create a mutable variable, you can modify its value.

Let's consider a simple example of a mutable data type, specifically a list in Python. Here's an example:

- mutable_list = [1, 2, 3, 4, 5]

  # Modifying the value at index 2

  mutable_list[2] = 10

  # Printing the modified list

print("Modified List:", mutable_list)

In this example, we start with a list [1, 2, 3, 4, 5]. Lists in Python are mutable, which means you can change their elements after creation. The line mutable_list[2] = 10 modifies the value at index 2 (the third element) of the list, changing it from 3 to 10. After this modification, the list becomes [1, 2, 10, 4, 5].

Let’s consider an example of a dictionary **Dictionary**representing the marks of 5 students, and we'll modify the marks of the 2nd student:

- # Creating a dictionary with marks of 5 students

  student_marks = {'student1': 85, 'student2': 92, 'student3': 78, 'student4': 95, 'student5': 88}

  # Displaying the original dictionary

  print("Original Student Marks:", student_marks)

  # Modifying the marks of the 2nd student

  student_marks['student2'] = 88

  # Displaying the modified dictionary

  print("Modified Student Marks:", student_marks)

In this example:

We start with a dictionary student_marks where keys are the names of students and values are their respective marks. The original dictionary is displayed using print("Original Student Marks:", student_marks). We then modify the marks of the 2nd student using the line student_marks['student2'] = 88. This changes the marks from 92 to 88.

Finally, we display the modified dictionary using print("Modified Student Marks:", student_marks).

After the modification, the dictionary student_marks is updated to {'student1': 85, 'student2': 88, 'student3': 78, 'student4': 95, 'student5': 88}, reflecting the change in marks for the 2nd student.

Sets in Python are mutable as well. You can add or remove elements from a set.

Now, let's try to modify a value in a tuple.

- # Original tuple of favorite numbers

  favorite_numbers = (3, 7, 1, 8, 4)

  # Attempt to modify the third number (this will result in an error)

  favorite_numbers[2] = 5

In this example, we start with an original tuple named favorite_numbers, which contains the numbers (3, 7, 1, 8, 4).

We then attempt to modify the third number in the tuple at index 2, trying to change the value from 1 to 5. However, this operation will result in an Error.

This error occurs because tuples in Python are **immutable** , meaning their elements cannot be changed or modified after the tuple is created.

Just like tuples, integers, and strings are also immutable, i.e. Once assigned, their values remain fixed.

my_integer = 42

print(id(my_integer))

Creating an integer

my_integer = 90

print(id(my_integer))

At this point, my_integer refers to the integer object with the value 42. Now, a new integer object with the value 90 is created, and the variable check both have different IDs which proves a new object has been created,

showing that integers are not mutable.

That wraps up our exploration of mutable and immutable data types. Where mutable data types, such as lists, dictionaries, and sets, allow for dynamic changes to their elements after creation. On the other hand, immutable data types, including tuples, integers, and strings, resist change once assigned i.e. their values remain fixed.

As you navigate the programming landscape, keep in mind the characteristics of mutability and immutability. Choose wisely based on your program's needs, balancing adaptability with stability.