Topic 1 – Variables, Data Types, and Functions

Author: Jake Palczewski, CCIE #63320
Last Updated: November 29, 2020

Table of Contents

Introduction 2

Variables 2

Data Types 3

Strings 3

"Adding" strings 4

String slices 5

String formatting 5

Numbers 6

Lists 7

Indexing 7

Append method 8

Tuples 9

Dictionaries 9

Return only keys/values 9

Return value by key 10

Functions 10

print() 11

type() 11

range() 12

Methods 12

Conclusion 13

Introduction

Python is an extremely versatile coding language. From software development, to web development, to editing CSV and TXT files, to being a network engineer's best friend, Python can be leveraged for many a purpose. However, before creating the next great thing with Python, it is essential to understand the language's fundamental concepts, including variables, data types, and functions. The purpose of this document is to impart the basics of Python variables, data types, and functions.

Variables

In technical terms, a variable in Python is a reserved location in memory that stores values. In other words, a variable encapsulates some sort of value that can be used and reused in a Python script. Here's an example:

```
2
3   my_var1 = "Hello there"
4
```

Let's break down the example above.

- The **my_var1** to the left of the equal sign is the variable's name. In defining a variable name, there are a few rules to follow:
    - Variables should always start with a letter, usually lowercase, as the example above illustrates
    - The variable name may contain numbers, as the example above illustrates
    - Do not include spaces nor special characters in a variable name, although underscores are okay, as the example above illustrates
    - Python variable names are case sensitive!
    - Lastly, some words cannot be used as a variable name. A list can be found by clicking one of the links at the end of this lesson
- The **my_var1** variable is followed by an equal sign, which acts as an assignment operator.
- To the right of the equal sign is the value encapsulated in the variable.
- In our example, the **my_var1** variable encapsulates the **string** "Hello there" (we will cover **strings** and other data types in the next section).

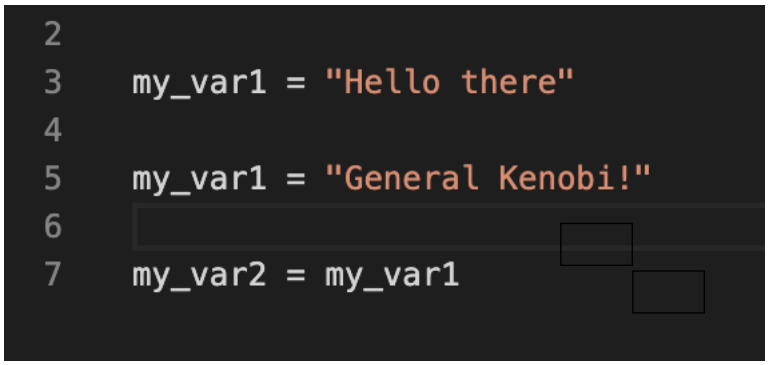
It is possible to give a variable a new value in your script. Here's an example:

```
2
3   my_var1 = "Hello there"
4
5   my_var1 = "General Kenobi!"
```

In this example:

- The string “Hello there” is initially encapsulated in the **my_var1** variable.
- However, we then give **my_var1** a new value: the string “General Kenobi!”.
- The **my_var1** variable is now equal to “General Kenobi!”.

Lastly, we can define a variable and assign a different variable as its value! Here’s an example:



In this example:

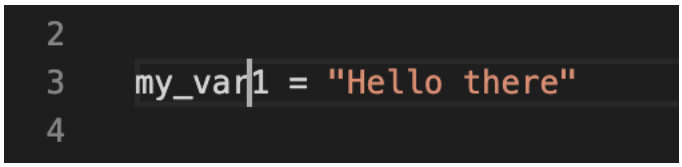
- In the third line of code, we are encapsulating the variable **my_var1** into the variable **my_var2**.
- This is an example of Python’s flexibility!

Data Types

Python leverages different types of data to handle its myriad operations and capabilities. Among the most commonly used data types are **strings**, **integers**, **floats**, **lists**, **tuples**, and **dictionaries**. These data types can be encapsulated within a variable.

Strings

As seen in the previous section, **strings** are a sequence of characters; letters, numbers, special characters, spaces, and more can be part of a **string**. To encapsulate a **string** within a variable, we use quotation marks. The characters within the quotes comprise the **string** value. Here’s an example from the previous section:



Let’s break this down:

- **my_var1** is the variable name, and the equal sign is assigning the **string** “Hello there” to the variable.

In the above example, we used double quotes in the **string**'s definition. However, we can use single quotes instead.

```
2
3    my_var1 = 'Hello there'
4
```

However, the single/double quotes can only encapsulate a one-line **string**:

```
4
5
6   my_var1 = "Darth Vader: If only you knew the power of the Dark Side. Obi-Wan never told you what happened to you father"
7
8
```

What if we wanted to have line breaks within a **string**? In this case, one option is to use a triple quotation:

```
my_var1 = '''
Darth Vader: If only you knew the power of the Dark Side. Obi-Wan never told you what happened to you father.

Luke: He told me enough. He told me you killed him.

Darth Vader: No. I am your father.

Luke: No. That's not true! That's impossible!

Darth Vader: Search your feelings. You know it to be true.

Luke: Noooo! Noooo!'''
```

Be sure to close your **string** with the same quotation pattern with which you start the **string**. For example, if you use a triple quotation to start your **string**, use a triple quote to close your **string**.

This document does not encapsulate everything we can do with **strings**, but here's a few more examples:

"Adding" strings

```
30    my_var1 = "Hello "
31    my_var2 = "there"
32
33    my_var3 = my_var1 + my_var2
34    print(my_var3)
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Hello there
```

- By “adding” **my_var1** to **my_var2** and encapsulating this operation in **my_var3**, we can use the **print()** function (which we will talk about later in this document) to see that **my_var3** is equal to the **string** “Hello there”.

String slices

```
29
30    my_var1 = "Hello there"
31    print(my_var1[0])
32    print(my_var1[0:])
33    print(my_var1[0:6])
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

H
Hello there
Hello
```

- Each character in a **string** has an index value within that **string**, starting with 0.
- By using square brackets, we can “slice” a **string** into parts.
- [0] shows the very first character of a **string**, in the case above “H”
- [0:] will show everything after the first character of a **string** including the first character
- [0:6] will show the first five characters of a **string**

String formatting

```
30   my_var1 = "Hello there"
31   my_var2 = f"I have something to say: {my_var1}"
32   print(my_var2)

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

I have something to say: Hello there
```

- **Strings** are immutable data types. Unlike other data types like **lists** and **dictionaries**, we cannot edit a **string** during a script's operation once the **string** is defined.
- To add flexibility to **strings** and their immutability, we can use **string** formatting to insert variables and their values into **strings**.
- The "f" to the left of the first quote in **var_2** allows us to use curly brackets within the **string** to encapsulate a variable and its value in the **string**.

Numbers

Python can perform all sorts of manipulations with numbers, including addition, subtraction, multiplication, division, greater than, less than, and more! Python has two primary types of numbers: **integers** and **floating-point numbers** (aka **floats**).

Let's take a look at an example:

```
30   my_num1 = 1
31   print(type(my_num1))
32   my_num2 = 2.6
33   print(type(my_num2))
34
35   int(my_num2)
36   print(int(my_num2))

TERMINAL

<class 'int'>
<class 'float'>
2
```

- The "**my_num1**" variable encapsulates the integer "1".

- By using the type() function within the print() function, we can see the data type of my_num1, which shows that my_num1 is an "int" data type: an integer.
- The my_num2 variable encapsulates the float number 2.6. Why is 2.6 a float and not an integer? It has a decimal!
- We can convert a float to an integer. In the example above, we do this using int(my_num2).
- We can see the result of the int() function in the last print() function: 2.6 is rounded down to 2, since integer data types can only equal a whole number.

Lists

With integers, floats, and strings, we've seen how to encapsulate a value into a variable. But what if we want a variable to represent a *series* of numbers or strings? Python has a few options in this scenario, including lists.

Within square brackets, a list is a sequence consisting of elements separated by a comma. You can have various data types within a list, including strings, integers, floating-point numbers, and even other lists to name a few. Let's take a look at an example:

```
39
40    my_list1 = ["Hello there", "General Kenobi", 1, 2.5]
```

- The list above contains four elements: the string "Hello there", the string "General Kenobi", the integer 1, and the floating-point number 2.5.

Using the print() function for this list yields the following:

```
5    my_list1 = ["Hello there", "General Kenobi", 1, 2.5]
6    print(my_list1)

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

['Hello there', 'General Kenobi', 1, 2.5]
```

- The print() function sends the elements of the list to the terminal, enclosed in the square brackets that we used to define the list.

What else can lists do?

Indexing

```
5   my_list1 = ["Hello there", "General Kenobi", 1, 2.5]
6   print(my_list1[2])
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

1
```

- What if we only wanted to **print()** one element of the **list**?
- Each element in the **list** occupies a specific index value, starting with 0.
- In our case, the **string** "Hello there" is at index 0, the **string** "General Kenobi" is at index 1, and so on.
- The above example uses the **print()** function to show what is at index 2 in our **list**.
- Looks like the **integer** "1" is at index 2 of **my_list1**!

Append method

```
5   my_list1 = ["Hello there", "General Kenobi", 1, 2.5]
6   print(my_list1)
7   my_list1.append("R2-D2")
8   print(my_list1)
```

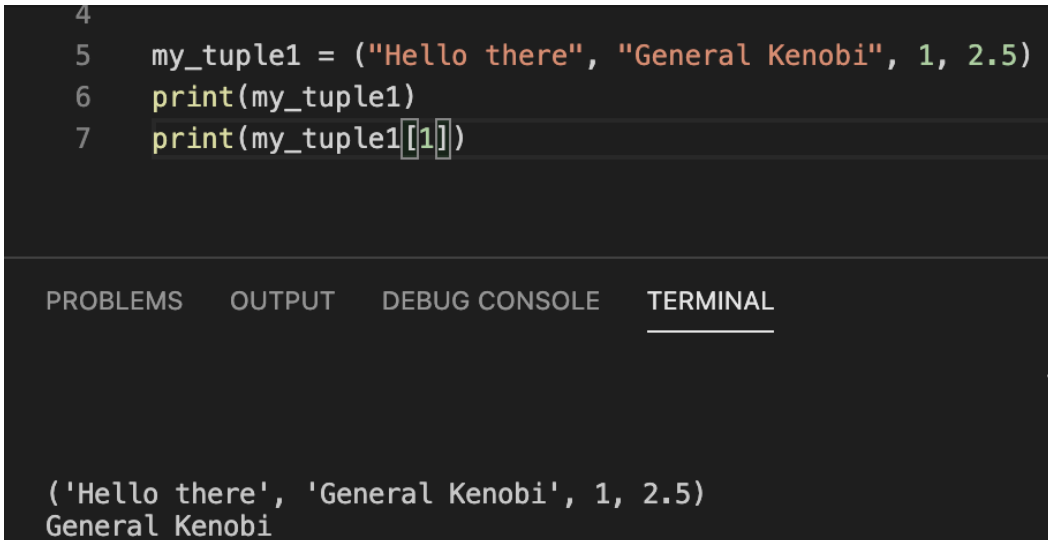
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

['Hello there', 'General Kenobi', 1, 2.5]
['Hello there', 'General Kenobi', 1, 2.5, 'R2-D2']
```

- **Lists** are mutable, meaning we can edit their contents during the execution of a script.
- In other words, we can, for example, configure a **list** and tell our script to append an element to the end of the **list**.
- The above example leverages the **append()** method (methods will be discussed more in-depth later in this lesson).
- We can see the initial **print()** of **my_list1**, the subsequent **append()** method adding the **string** "R2-D2" to the **list**, and then another **print()** of **my_list1** reflecting the **append()**.

Tuples

Tuples, like **lists**, encapsulate a series of elements. Unlike **lists**, **tuples** are immutable, meaning we cannot edit the contents of a **tuples** as your Python script is running. **Tuples** are handy when we want to create an untouchable series of elements. Here is an example:



Dictionaries

As an incredibly useful ally as a network engineer, a **dictionary** is a set of key-value pairs, separated by comma and enclosed by curly braces. Let's take a look at an example:

In the example above:

- "Obi-Wan" (key) and "Hello there" (value) are a key-value pair, as is "General Grievous" (key) and "General Kenobi!" (value).
- The key-value pair allows us to manipulate data by specifying the key.
- However, important to note is that each key must be a unique element!
- In the above **dictionary**, we could not have, say, two keys named "Obi-Wan."

Let's see what else we can do with dictionaries.

Return only keys/values

```
4
5   my_dict1 = {"Obi-Wan":"Hello there", "General Grievous": "General Kenobi!"}
6   print(my_dict1.keys())
7   print(my_dict1.values())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
dict_keys(['Obi-Wan', 'General Grievous'])
dict_values(['Hello there', 'General Kenobi!'])
```

In the example above:

- we use the **print()** function and the keys() method to return the keys in "**my_dict1**".
- We then use the **print()** function and the values() method to return the values in "**my_dict1**".

Return value by key

```
4
5   my_dict1 = {"Obi-Wan":"Hello there", "General Grievous": "General Kenobi!"}
6   print(my_dict1["Obi-Wan"])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Hello there
```

In the example above:

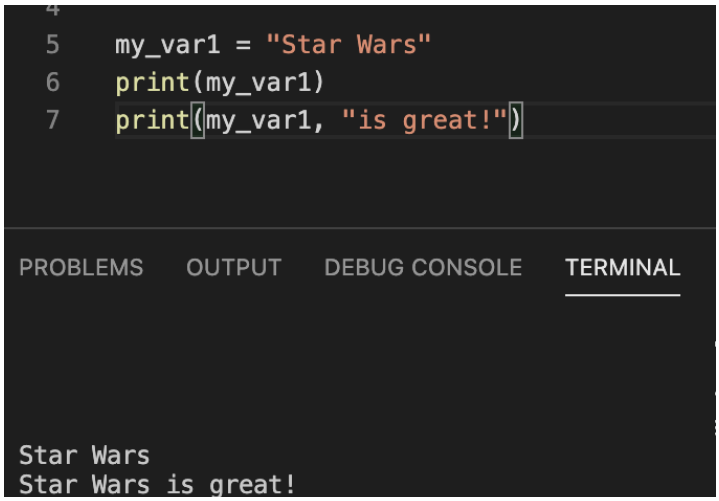
- We use the **print()** function to return the value for the "Obi-Wan" key.

Functions

Functions in Python perform a specific action. In the examples above, we saw the **print()** function in action, which outputs a specified value to the terminal . The **print()** function is an example of a built-in function to Python. It is possible to create custom functions, which will be explored in a future lesson.

With regards to built-in functions, there are many within Python. Here are some the more popular and useful ones:

print()



In the example above:

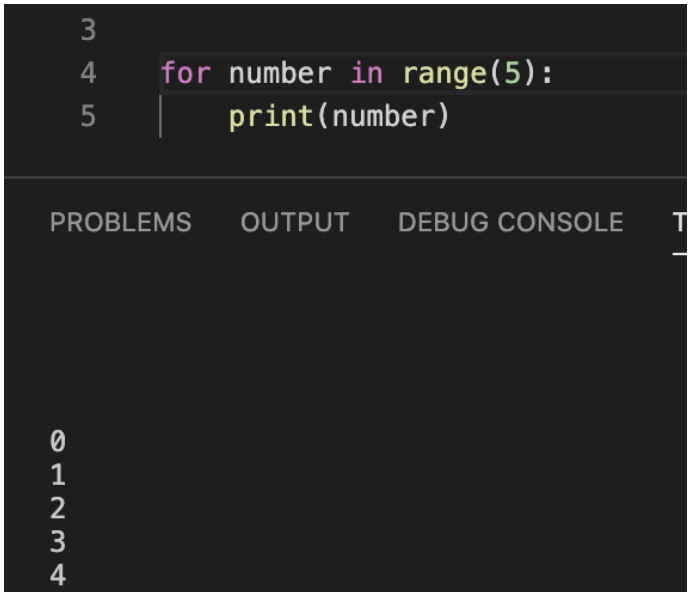
- We use the **print()** function twice.
- The first time, we output the contents of the variable **my_var1**.
- With the second **print()** function, we output the contents of the variable **my_var1** and concatenate the **string** “is great!”!

type()

In the example above:

- we use the **type()** function within the **print()** function to return the data type encapsulated within the **my_var1** variable.
- In this case, the contents of **my_var1** is 'str', a **string**.
- Notice the parenthetical structure of using a function within a function: print(type(my_var1))

range()



In the example above:

- We dip into the contents of our next lesson, which is all about looping in Python.
- For now, know that we use a **for** loop, the **range()** function, and the **print()** function to return a series of **integers** in the range 5.
- This yields the five **integers** 0, 1, 2, 3, and 4.

Methods

At first glance, Python functions look a lot like methods. Python functions carry out specific tasks, and are contained within themselves. Functions may contain zero arguments or more than one argument. The **print()** function is an example.

Methods have a similar syntactical structure to functions, but are associated with specific data types. In other words, the methods we can use are dependent on the type of data upon which we are performing the method.

Let's review an earlier example:

```
4
5   my_dict1 = {"Obi-Wan":"Hello there", "General Grievous": "General Kenobi!"}
6   print(my_dict1.keys())
7   print(my_dict1.values())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
dict_keys(['Obi-Wan', 'General Grievous'])
dict_values(['Hello there', 'General Kenobi!'])
```

In this case:

- **keys()** and **value()** are methods.
- These methods are dependent on the dictionary data type; **keys()** and **values()** call the keys and values in a **dictionary**, and would not work for a different data type, like a **list**.
- The function **print()** is independent of the **dictionary** data type; the print() functions will work with any data type.

Conclusion

A lot of information was imparted through this lesson. However, it is of paramount importance to understand variables, data types, and functions: the building blocks of Python. The accompanying **Lesson1-variables_datatypes_functions.py** file shows examples of variables, data types, and functions to help solidify these concepts. This lesson does not touch every data type, nor does it impart everything about functions nor variables, but below is a list for continued reading and research:

- Variables:
  - Python variables
  - More about variables, including words you cannot use as variable names
- Data Types:
  - Data types
  - More about data types
- Functions:
  - Built-in Python functions with examples
  - More examples