PROGRAMMING WITH PYTHON

By Randal Root

Module 02

In this module, you learn about **how data is used and processed in a program**.

What is Data?

Your program's data exists in one of two states**: stored on a drive or loaded into memory**.

Programs store their **drive data in a variety of ways**. Some examples include a simple text file, the Window's Registry, or in a database. In the end, though, this is just different types of *files* on a *drive*.

Data stored on a drive **still must load into memory** before you can use it. All **data must be read or modified while in memory**, even simple applications like Notepad do this.

When the program has completed using the **data in memory, it must be saved back as stored data**, or the data is lost when the program ends, or the computer shuts down.

Note that **not all program data must be saved**. For example, often you ask a user to supply **temporary values** as they are using your program. The **Window's Calculator** is a**n example** of this kind of program. After the user provides data, and a calculation is performed, the **input evaporates as the program completes its purpose**.

Whenever you are loading data into memory, either from stored data or from the collected user input, **you need to tell the computer what kind of data is being loaded**. This does two things; **it allows the computer to reserve enough memory space** to hold the data you are going to load, and it allows the computer to **restrict certain types of data from that space**.

For **example**, if you create a space in memory for an **integer**, the program tells the computer to **set aside typically 4 bytes of memory** and **only allow whole numbers** within that memory space, and **no characters**. If you later input characters or non-whole numbers to that memory space, the computer reports an **error or possibly ignore the values after the decimal point**.

Input and Output (I/O)

All computer programming languages have some way of allowing both input and output of data. In Python, the print() and input() commands are your two primary tools for this.

You use the **print function to display data (output) and the input function to capture data**.

```
print("Enter your name:");
strData = input();
print("Saving name as: " + strData);
print("Saving name as: " , strData);
print("Saving name as:" ,strData);
```

Listing 1

Declaring Constants and Variables

Constants and Variables are **both tools for holding data in memory**. In most languages, you need to tell the computer what type of data you wish to store, called **declaring the "data type."** In **Python, "data typing" is done automatically**.

```
int x; //declaring a C# variable
x int; -- declaring a SQL variable
x # declaring a Python variable
```

Listing 2

Constants

Programmers sometimes place restrictions on their program's data so that it cannot change once its value set. These are known as *Constants.*

Constants stop a value from being changed while the program is running, which is different than **a Variable,** who's **value can "vary."**

You **set the value of a constant at the same time you create the constant**. Once your program starts running, the value remains the same as long the program is running.

It is a common programming convention to **name your constants using all uppercase letters**:

```
const int SIMPLEPI = 3.14 //C#
SIMPLEPI = 3.14 #Python
```

Listing 3

Note*: In* ***Python****, Constants can be changed since there is no enforcement provided in the language. This is odd since it means that there is* ***no built-in safeguard****.*

Variables

If the programmer wants to **allow changes to a value while the program is running**, they can create a variable instead.

```
x = 5 #x starts with the value of 5
x = 10 # and x now holds the value of 10
```

Listing 4

Note*: The difference between variables and constants is that, while a program is running, variables accept changes and constants do not accept changes.*

Assigning Values

Python uses the = symbol to **assign values to both variables and constants**. One thing every programmer needs to know is that items **on the left of the = symbol** always receive that which is on the right.

It may seem so simple if you have some programming experience, but **many beginners are confused by this**. I guess that the reason for this is that when we were children, we always saw our teachers write out equations as follows:

4 + 5 = 9

It was not until later that a teacher might write the same equation like this:

9 = 4 + 5

By the time we saw the second example, we had seen the first example for years. However, the second example, "**9 = 4 + 5**", **is the correct way** to write the equation in programming.

Perhaps this is why some people struggle with the fact that **the programming code in Listing 5**, which **always writes out the value of x as 10 and never 5.**

```
x = 5;
y = 10;
x = y; # The variable x is set to the same value as y
print(x);
```

Listing 5

LAB 2-1. Creating Variables

In this lab, you practice creating variables.

1) Open Idle and create a script using the code in Listing 5.

2) Verify that the script prints x as "10."

3) Write down why it works as it does.

Choosing Data Types

Both variables and constants are declared by choosing a name and deciding on the type of data you want to store.

Computers do not deal with subtle distinctions when it comes to data, **but humans do**. **We see a number** and think that it **is very different from a name**. We see **a picture** and think that it **is different from a collection of numbers**. However, computers don’t care. **Even pictures are just all ones and zeros to a computer**.

Still, it is **often essential to force a computer to see these distinctions sometimes**. That way, you can **receive an error message** from the computer when a program tries to set **incompatible values to your variables or constants**. To make a computer understand the distinction between numbers, names, dates, etc., you formally tell the computer the difference.

<u>Although in Python data typing is automatic</u>, **in many other languages, you must decide and declare a variable and constant's data "type**."

A data's "type" is a description of the data. In other words, **a type defines what it can and cannot allow as values**. For example, you can indicate you want to store only numbers in one variable and character data in another.

Variable and constant **data exist at a defined location in a computer's memory**. In almost all languages nowadays, **the location is dynamically assigned**. **In addition to the value of the variable and constant, there is information about its *name, type, and size***.

Each language has a set of pre-defined, or "built-in," data types. Here is a summary of Python 3's built-in types:

Type	**Description**	**Example**
string	String type; a string is a sequence of Unicode characters	**s = "hello"**
int	32-bit signed integral type	**val = 12**
float	Single-precision floating point type	**val = 1.23**

bool	Boolean type; a bool value is either true or false	**val1 = True;**
		val2 = False;

Here are some things to keep in mind when using variables and constants:

- When a new value enters into a variable, the old value is lost.
- Use common sense when choosing a data type!
- The most common types used are integers numbers, floating points numbers, and strings of individual characters.

The type() Function

Use the type function **when you want to know which data type** was chosen for you.

```
intFirstNumber = 5
print(type(intFirstNumber))

intFirstNumber = 5.8
print(type(intFirstNumber))

intFirstNumber = "5"
print(type(intFirstNumber))

intFirstNumber = True
print(type(intFirstNumber))
```

Listing 6

LAB 2-3. Using the Type function

In this lab, you practice using the type() function.

1) Open Idle and create a script using the code in Listing 6.

2) Verify that the script prints:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

3) Write down why it works as it does.

Special Characters

In programming, you use **keywords and symbols to tell a computer to perform actions**. While **many of the keywords or symbols are apparent**, such as "def" for define or "+" for addition, **some of them need further explanation**.

Escape Characters

Programming languages are made up of symbols and characters that have implied meaning. However, if you need **to use at character or symbol without this implied meaning**, you need to "escape" it.

```
print("backslash \\ test ");
print("single quote \' test ");
print("double quote\" test ");
print("tab \t test " );
print("newline \n test ");
print("carriage return \r test ");

'''print:
backslash \ test
single quote ' test
double quote" test
tab     test
newline
 test
carriage return
 test
'''
```

Listing 7

Line Continuation Character

Languages often use symbols and characters for more than one purpose. One example is the Slash (\). The first meaning of this symbol is escaping characters, but the second meaning indicates that a programming statement continues on multiple lines. **Symbols with multiple meanings are known as "*Overloaded Operators*."**

```
print("test") # This works!
print("test"
      ) # Oddly, this works too!
print
("test") # This does NOT work, though there is no error!
print \
("test") #Adding the backslash makes it work!
```

Listing 8

Overloaded Functions

Functions (otherwise known as methods or procedures) **can also have multiple versions of themselves** as well. **Each version performs a slightly different action**. For example, the print() method has a number of versions that you can use. Functions with multiple versions of themselves **are technically called *Overloaded Functions (Methods*.)**

```
print("test")
print("test" * 2)
print("test","msg")
print("test" * 2,"msg")
# Only in 3.x unless you add: from __future__ import print_function
print("test","msg", sep='-')
print("test","msg", sep='-',end=':')
```

Listing 9

The Python help documents indicate when a function has an overload like this:

"**print**(*[object, ...][, sep=' '][, end='\n'][, file=sys.stdout]*)

> Print *object*(s) to the stream ***file***, separated by ***sep*** and followed by ***end***. *sep*, *end* and *file*, if present, must be given as keyword arguments.
>
> All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. …"

The official description can be very technical and off-putting for beginners. So, **remember that there are lots of** other places you can find **help on the internet and** when that fails, **just trying out different combinations of options may be enough for your current needs** in a given program.

LAB 2-3. Using the print function

In this lab, you practice using the print() function.

1) Open Idle and create a script using the code in Listing 9.

2) Verify that the script prints:

```
test
testtest
test msg
testtest msg
test-msg
test-msg:
```

3) Write down why it works as it does.

Operators

Operators are small pre-made functions that use symbols instead of the standard function syntax. A commonly used example is the string concatenation operation "+." **This operator joining two strings together to create a single string, meaning that the (+) symbol is overloaded for addition and concatenation.**

```
print("One " + "long " + "string " + "of " + "characters.")

'''prints:
One long string of characters.
'''
```

Listing 10

While **you could create methods that do the same things as operators**, **but** most people prefer to use operators. In the case of "+" symbol, it looks and acts like the mathematical statements we learned as children, and thus are **often easier for people to read**.

```
x = 5
y = 4
sum = Add(x, y) # If I made a custom function, it would do
sum = x + y # the same thing as this operator
```

Listing 11

If it’s been a while since your last math class, you may not remember what is meant by the phrase, “Operators work with operands.” That being the case, let’s start with a refresher on these two standard terms:

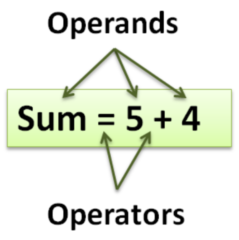


Figure 1. An Operator example

- The operator is the **symbol that indicates the action** or operation to perform
- **You perform actions on and with the operands**
- Some operators require only **one operand** (known as **unary** operators), while some require **two** (known as **binary** operators), and others need **three** operands (known as **ternary** operators)

The "5 + 4" is an **expression** that **evaluates** to nine. The **+** operator has a **higher order of precedence** and therefore runs before the **=** operator. Once the + operator finishes, the value of 9 is assigned to the variable sum using the assignment operator, which is the = symbol.

Remember that in computer programming, the value of the operand on the right side of the assignment operator (=) **always assigns to the operand on the left**.

The Parentheses Operator

The **()** operator is used for defining a list of values and grouping expressions to set the order of precedence and to create a Tuple of values:

```
intSum = (4 * (2 + 2))
print(intSum)

tplWords = ("Test","data")
print(tplWords)
```

Listing 12

The Bracket Operator

The **[]** operator is used for array indexes and to create a List of values:

```
lstWords = ["Test","data"]
print(lstWords[1])
```

Listing 13

The Logical Negation Operator

The ***not*** operator negates the expression. In other words, if the operand was true, it is now false:

```
x = 1
if (not(x == 1)): # It starts as true, but then is changed to false
 print("It is true")
else:
 print("It is false") # This is what prints out
```

Listing 14

The ***!=*** operator negates a comparison as well, meaning that there are two options to choose from when you can to negate an expression.

```
x = 1
if (x != 1): # It starts as true, but then is changed to false
 print("It is true")
else:
 print("It is false") # This is what prints out
```

Listing 15

The Multiplicative Operators

The * operator multiplies operands and returns the results (known as a product).

```
x = 20
y = 10
print(x * y) # Shows 200
```

Listing 16

The / operator divides the operands and returns the results (known as a quotient).

```
x = 20
y = 10
print(x / y) # Shows 2
```

Listing 17

The % operator is known as the "modulo" operator, and it divides the operands and returns any remainder (known as the "modulus").

```
x = 20
y = 10
print(x % y) # Shows 1
```

Listing 18

The Additive Operators

The + and - operators work as you would expect—they add and subtract.

```
x = 20;
y = 10;
print(x + y) # Shows 30
print(x - y) # Shows 10
```

Listing 19

LAB 2-4: Working with operators

In this lab, you practice working with operators.

1. Create a script using IDLE that adds two numbers together and then print the answers to the screen.

2) Write down why it works as it does.

Value vs. Reference Types

Most languages support two categories of types: value types and reference types.

- **Value** types (commonly int or float)
- **Reference** types (commonly string or custom objects)

***Note**: Oddly, in Python, all variables are "reference" types, but they still behave like either value or reference types.*

One of the differences between Value types and reference types is that variables of the **value types store their data in the area of memory where your program is running** (The program's **stack**), whereas variables of **reference types store the data separate from your application, in the** "runtime" area of memory (the "**heap**"). The Python runtime is a general area that multiple Python programs can use at the same time.

To help you understand the difference, consider the kind of things a computer tracks to when you to create a variable or constant. **The computer notes the name, data type, and address in memory where the actual values are stored.** It may be useful to think of what the computer stores as a table of values.

An Example in C#

Since python does not have "true" value types, let's look at recorded information for **two** *variables in Microsoft's C#*, **"x" and "y."** These types are recorded as i**nt** for **x** and **string** for **y**. One noticeable difference is that one holds numbers and one holds characters, but another is that Microsoft chose to make *int a value type* while they made *string a reference type*. As such, their **data is stored in different places (Figure 2)**.

Name	**Type**	**Address or Data**
x	int	stack(value = 42)
y	string	heap(address = #123)

All the programs running the C# code share the managed heap or runtime (Figure 2).

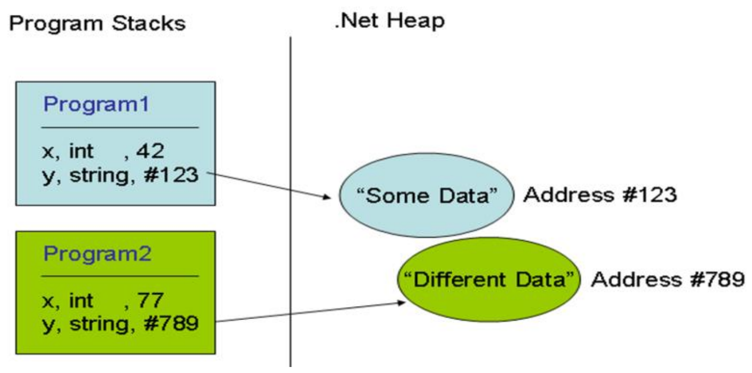


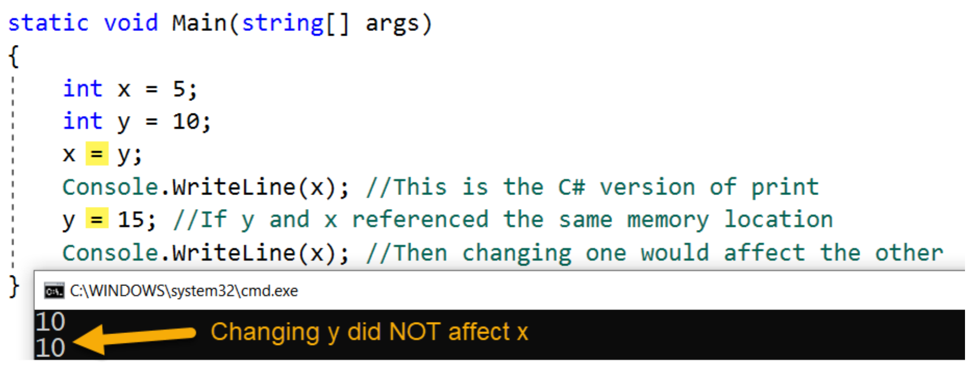
Figure 2. An example of reverence and value types in C#

***Note**: I just made up the address number and simplified it as #123. Technically, computers use an advanced numbering system for addressing, but my simplified example should give you a good idea of how the system works.*

Passing by Value or Reference

The actual location of the data is less important than the way the data behaves when it passes from one variable to another.

Variables defined as **value types pass values**, but variables defined as **reference types pass references**. In figure x, the C# code defines two integer variables, then passes the value of one to the other. The value is passed because **integers are "value" types (Figure 3)**.



```
static void Main(string[] args)
{
    int x = 5;
    int y = 10;
    x = y;
    Console.WriteLine(x); //This is the C# version of print
    y = 15; //If y and x referenced the same memory location
    Console.WriteLine(x); //Then changing one would affect the other
}
```

```
10
10
```

Figure 3. Integers act like value types

Now, remember that in Python, all variable, even simple types like int, are "reference types," so you can picture it like as shown in figure 4.

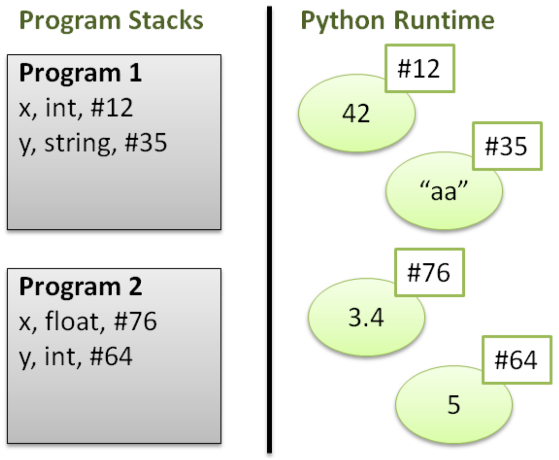


Figure 4. Imagining the Python runtime

However**, Python variables still "act" like ether value types or reference types**, regardless of where the data is stored! Consider the following code and ask yourself what will this code print out?

```
x = 5
y = 10
x = y
y = 15
print(x) # Is x still 10 or has it changed to 15?
```

Listing 20

When the variable "acts" as a **value** type, then the equal sign **assigns a value** to the variable.

If the variable "acts" as a **reference** type, then the equal sign **assigns a reference** link to the same address location in memory, this means that **changing the value of one variable changes both variables since they both point to the same address location**.

In Python, some variables act like a "value type" and other times act like a "reference type" depending on their automatically chosen data type.

In general, if a variable uses **simple data,** like an integer**, then it acts like a value type**. While **complex data**, like a "List" of values, **act like reference types**!

```
# -- An example with integers --
# Simple data
x = 1
y = x
x = 3
print(y) # Print 1, as if only the value was passed!

# Complex data
x = [1, 2] # This is a Python List
y = x
x[0] = 3
print(y) # Prints [3,2], as expected of a reference!

#  -- An example with strings --
# Simple data
x = "Bob"
y = x
x = "Robert"
print(y) # Print "Bob", as if only the value was passed!

#Complex data
x =["Bob", "Sue"]
y = x
x[0] = "Robert"
print(y) # Prints ['Robert', Sue], as expected of a reference!
```
Listing 21

Now, this may seem confusing, but remember that **if you know that this behavior exists**, **you can watch and test for it.**

Always remember that **"if in doubt, test it out!"**

Using an Integrated Development Environment (IDE)

An Integrated Development Environment is a tool you use to develop and test programs.

*"An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger. Most modern IDEs have intelligent code completion." (Wikipedia,2015, **https://en.wikipedia.org/wiki/Integrated_development_environment**)*

So far we have used Idle as our IDE, but **starting in module 3**, you need a better one! The one **we use** in this course is called **PyCharm**.

"The free open-source edition of PyCharm, the premier IDE for pure Python development." (JetBrains, 2015, https://www.jetbrains.com/pycharm/*)*

You can **download and install PyCharm on your PC or Mac.** You will find the instructions here: **https://www.jetbrains.com/pycharm/documentation/**

Summary

In this module, we looked at what the Python language is, how to install it, and how to use it. We also covered the different components that make up a Python script.

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

- What is a variable?
- What is constant?
- How do you assign values to a variable or constant?
- What is a datatype?
- What does the type() function do?
- What is an escape character?
- What is an operator?
- Name some common operators?
- What is an overloaded operator?
- What is difference between a value and reference type?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.