# **Programming with Python**

# Module 2

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

Topic Covered:

* What is Data?
* Declaring Constants and Variables
* Assigning Values
* Choosing Data Types
* Escape Characters
* Line Continuation Character
* Overloaded Methods
* Concatenation
* Operational Constructs
* Value vs. Reference Types

## What is Data?

Your program’s data will exist 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 some type of database. In the end though, this is really just different types of *files* on some kind of *drive*.

When data is stored one a drive, the computer still has to load it into memory before it can actually be used. This is because 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 it is lost when the program ends or the computer shuts down.

It should be noted that not all data you use in a program must be saved. Often you will just ask the user to supply you with temporary values as they are using your program. After the user provides this data, it is your choice to save it somewhere or to just let it evaporate as the program closes. The Window's Calculator is an example of this kind of program.

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 will tell the computer to set aside typically 4 bytes of memory and only allow whole numbers within that memory space. If you later try to add characters and not whole numbers to that memory space, the computer will report an error or just possibly ignore the values after the decimal point.

## 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, but in Python this is done automatically.

Programmers will sometimes place a restriction on their program’s data so that it will not change once it is 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.”

The initial value of a constant must be set on the same line of code where you create the constant. Once your program starts running, the value will remain the same as long as it is loaded into memory. It is a common programming convention to name your constants using all uppercase letters:

SIMPLEPI = 3.14

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

If the programmer wants to allow changes to a value while the program is running, they can create a variable instead. A *variable* allows for both reading and writing of data.

x = 5 #x starts off with the value of 5

x = 10 # and x now holds the value of 10

*Note: the difference between variables and constants is that "constant" data is set with an initial value that cannot* (should not in python) *be changed while the program is running and variable are made to accept changes.*

## Assigning Values

Python uses the **=** symbol to assign values and references to both variables or 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.

**Note:** It may seem so simple if you have some programming experience; but many beginners are confused by this. It is my 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. Perhaps this is why some people struggle with the fact that programming code like the following will always write out the value of x as 10 and that y will never be 5:

x = 5;

y = 10;

**x = y**; # The variable x will now be set to the same value as y

print(x);

## 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 really deal in 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 collections of numbers. However, computers don’t care. Even pictures are just all ones and zeros to a computer.

Still, it is important 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 must formally tell the computer the difference. In Python, this is done automatically*,* but in many other languages you have to decide manually by declaring a data’s *type*.

A *type* is a description of data. In other words, a type defines that one memory location will allow only numbers while another can only hold characters.

Each variable exists in a memory location that contains information about the variables such as its *name, type, size, and value*.

Please note:

* When a new value is entered, 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.

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;** |

### 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) )

### 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 ");

## Line Continuation Character

Languages often use symbols and characters for more the one purpose. One example is the Slash (\). This symbol is used for escaping characters as well as for indicating that code which is continued on multiple lines should be considered as part of a single line of code. This means that the (\) symbol has multiple meanings. Symbols like this are known as an *Overloaded Operators*.

print("test"); #this works!

print("test

); " #Oddly, this does work!?

print

("test"); #This one does not work!

print \ #So add this!

("test");

## Overloaded Methods (Multiple Versions)

Many methods (otherwise known as functions) can have multiple versions of themselves as well. For example, the print() method has a number of version that you can use. Methods with multiple versions of themselves are technically called O*verload 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=':');

The Python help documents indicate when a method 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()](mk:@MSITStore:C:\Python27\Doc\python273.chm::/library/functions.html#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. If no *object* is given, [print()](mk:@MSITStore:C:\Python27\Doc\python273.chm::/library/functions.html#print) will just write *end*.

The *file* argument must be an object with a write(string) method; if it is not present or None, [sys.stdout](mk:@MSITStore:C:\Python27\Doc\python273.chm::/library/sys.html#sys.stdout) will be used. Output buffering is determined by *file*. Use file.flush() to ensure, for instance, immediate appearance on a screen.

**Note**: This is something unique to Python 3.x, so does not apply to 2.x.

## Concatenation

String concatenation is the operation of joining two strings together. (Note that this means that the (+) symbol is overloaded for addition and concatenation.)

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

## Input and Output (I/O)

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

print("Enter your name:");

strData = input();

print("Saving name as: " + strData);

print("Saving name as: " , strData);

print("Saving name as:" ,strData);

## Operational Constructs



Operations come in three basic forms: *methods, properties, and operators*.

*Methods* are a named set of programming statements, *Properties* are specialized methods used to set or get values for a variable, and *Operators* are methods that use a symbol instead of a name.

### Operators

Operators are small pre-made methods that use symbols instead of the standard method syntax.

While most operators are made up of one symbol, some operators are made up of two or even three symbols.

Although you could create methods that do the same things as operators, most people prefer to use operators. This is because the symbols look and act like the mathematical statements we learned as children, and thus are often easier for people to read. For example, while you could create a method that would take in two numbers, add them together, and return the results like so:

#Here is an imaginary example of a Add method

x = 5

y = 4

sum = MyCustomClass.***Add***(x, y)

Most people would find the following code more intuitive:

sum

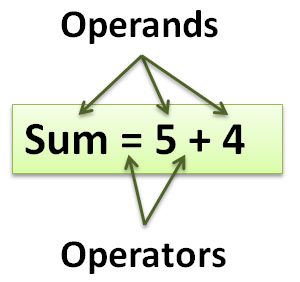
x = 5

y = 4

sum = x ***+*** y

Still, either one would fundamentally perform the same operation, so you could choose to implement your code by using either one.

If it’s been awhile 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 common terms:



* The operator is the symbol that indicates the action or operation to perform.
* The operands are the items that you wish to perform the action against.
* 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 is finished, 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 (=) will always be assigned 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)

### The Bracket Operator

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

lstWords = ["Test","data"]

print(lstWords[1])

### The Logical Negation Operator

The *not* operator negates the expression. In other words, if the operand was true, it will now be 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 will print out

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 will print out

### The Multiplicative Operators

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

x = 20

y = 10

print(x \* y) #Shows 200

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

x = 20

y = 10

print(x / y) #Shows 2

The % operator is known as the “modulo” operator and it divides the operands and return any remainder (known as the “modulus”).

x = 20

y = 10

print(x % y) #Shows 1

### 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

### LAB 2-1: Working with variables

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

## Value vs. Reference Types

Most languages support two categories of types: value types and reference types. (In Python, all variables are officially reference types, but sometimes behave like value types).

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

One of the differences between Value types and reference types in 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 separately where the rest of the runtime is doing other task( examples include the .Net Heap and the Python runtime).

To help you understand the difference, consider the kind of things a computer must track to allow you to use a variable or constant. The computer would need 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.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Address or Data** |
| x | Int32 | stack(value = 42) |
| y | string | heap(address = #123) |

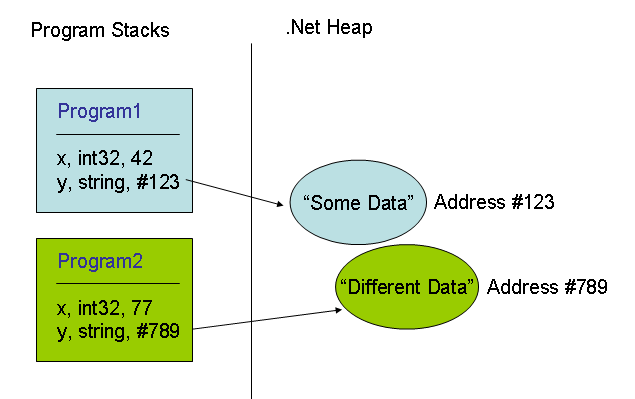
Here I have recorded information about two *C# variables (since python does not have true value types)*, x and y. The types are recorded as Int32 for x and string for y. One obvious difference is that one will hold numbers and one will hold characters, but another is that Microsoft chose to make *Int32 a value type* while they made *string a reference type*.

Remember that for value types, the program itself is responsible for managing the memory where the data is stored. This area of memory is known as the *stack* of memory that the program is running in.

Value types store the value of their data directly with the program’s stack. In the table above, we indicated that x is holding the value of 42. Yet, for reference types, runtime engine is responsible for managing the memory where the data is stored. Microsoft’s .NET Framework uses a special space of reserved memory known as the *.NET heap*. Python uses something similar called the *Python Runtime*.

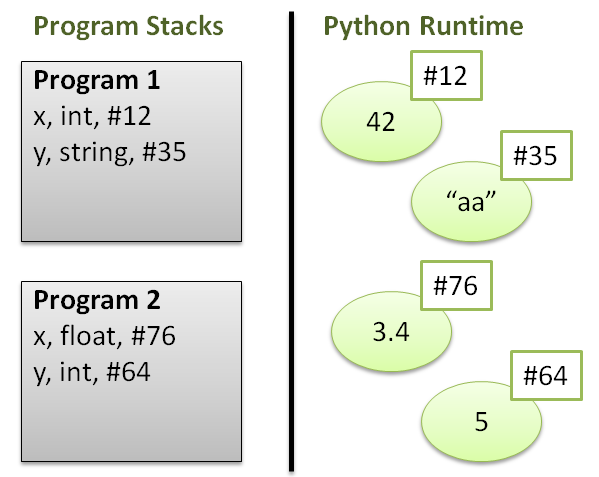
Reference types are stored and managed by this runtime not by the program itself. When your program refers to a variable or constant that uses a reference type, the runtime engine will retrieve the data for it. The actual data is stored in the .NET managed heap or runtime, so all your program needs to store is some reference to the address within the heap.

All of the programs running the same type of code will share the managed heap or runtime. This means that if you have two programs running C# code on the same computer, both will store the reference type data on the .NET heap.



**NOTE**: Technically, .NET uses an advanced numbering system for addressing, but we simplified our example by showing y as holding an address of #123 in the table and not a true memory address. Although we just made this number up, this should give you a good idea of how the system works.

Now, remember that in Python, all variable, even simple types like int, are "reference types", so you can picture it like this:



However, when you use the variables they will act like ether act like value types are reference types, regardless of where the data is stored! Consider the following code and ask yourself what will this code print out?

x = 1

y = x

x = 6

print(y) # what will the answer be, 1 or 6?

If the variable acts like a value type, then the equal sign will assign a value to the variable. If the variable acts like a reference type, then the equal sign will assign a reference to a location to memory. This means that if you change the value of a variable and another variable is point to its same memory location, then both variables will have to new value.

In Python variable sometimes act like a "value type" and other times act like a "reference type." In general, if a variable is pointing to a simple data type then it acts like a value type. For example, integers store a single value and act like a value type, while Lists store multiple values and 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]

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!

Now this will seem confusing, but remember that as long as you know that this behavior exists, and remember to test your code, it is just something you will get used to. (If in doubt, test it out!)

# Let’s review Chapter 2

* Overview
* Introducing the Useless Trivia Program
* Using Quotes with Strings
* Using Escape Sequences with Strings
* Concatenating and Repeating Strings
* Working with Numbers
* Understanding Variables
* Getting User Input
* Using String Methods
* Using the Right Types
* Converting Values
* Back to the Useless Trivia Program
* Summary

# What's Next?

## Using an Integrated Development Environment (IDE)

“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>)

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

## For module 3, you will need to download and install PyCharm on your PC or Mac. You will find the instructions here: <https://www.jetbrains.com/pycharm/documentation/>