PROGRAMMING WITH PYTHON

By Randal Root

Module 03

Programming languages all feature various programming **statements and constructs**. In this module, you learn some of the **most common ones used in programming**.

Using an IDE

While you can code with any text editor, you will find coding easier, and less prone to errors, if you use 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 this module, you need to download and install PyCharm on your PC or Mac. You can find the instructions here:

- **https://www.jetbrains.com/help/pycharm/requirements-installation-and-launching.html** (Links to an external site)
- **https://www.jetbrains.com/help/pycharm/quick-start-guide.html**(Links to an external site)
- **https://www.jetbrains.com/pycharm/documentation/**(Links to an external site)

Pseudo-Code

Pseudo-code is a **fancy term for outlining your program's logic**. You often see pseudo-code in textbooks or technical documents as an example code, which **represents an example of code** a programmer might write, but code **that does not necessarily run**.

*" Textbooks and scientific publications related to computer science and numerical computation **often use pseudo-code in the description of algorithms**, so that all programmers can understand them, even if they do not all know the same programming languages. In textbooks, there is usually an accompanying introduction explaining the particular conventions in use. The level of detail of the pseudo-code may in some cases approach that of formalized general-purpose languages." (Wikipedia, 2015, **https://en.wikipedia.org/wiki/Pseudo-code**)*

While you create a programming script, **you should start by writing pseudo-code in the form of comments that indicate what you are trying to do**.

For example, you might first write out the following comment before you add code.

```
# Get user data
# Print user data
```

Listing 1

Once the comments are typed, you **add the actual programming code to perform each noted action**.

```
# Get user data
strData = input("Enter your data: ")
# Print user data
print(strData)
```

Listing 2

You **may also see code that could do something useful but is commented out**. This is code that it **can easily be un-commented** into a working statement **if you want to demo it**, but currently, it's just a note that is not really part of your script's purpose.

```
# Get user data
strData = input("Enter your data: ")
# Print user data
print(strData)
# input("Tip: Use input() to keep a console window from closing!")
```

Listing 3

Conditional Statements

There are **two conditional statements included in most programming languages**. The first is the "**if-else**" construct, and the second is "**switch-case**." While Python does not have a switch-case statement, but it does have **something similar called, "elif"** clause, which we look at in just a bit.

Using if-else

Using if-else is the most common way to create a "conditional" set of statements. The following **pseudo-code** shows the ***general pattern*** of an if-else statement:

```
if (condition1 == true)
{
   // statements executed only if condition1 is true.
}
else
{
   //statements executed only if condition1 is NOT true.
}
```

Here is a simple Python example:

```
intVar = 1
if (intVar == 1):
  print("1") # Statements executed only if condition1 is true.
else:
  print("other")
```

Listing 4

***Note**: In Python, like many other languages, the **single equals operator (=) is used to assign a value** of 1 from the right side of the operator to the variable x on the left side of the operator. However**, to compare** the validity of the question "Does x equal 1," you use the equity operator of **two equal signs** (**==**).*

Each **if** or **else** block **can hold one or more statements**.

```
intVar = 1
if (intVar == 1):
  print("Statement 1")
  print("Statement 2")
else:
  print("another statement")
```

Listing 5

If there is only one statement, you can place it on the same line as the if or else keywords.

```
intVar = 1
if (intVar == 1): print("Only 1 Statement")
else: print("another statement")
```

Listing 6

if - elif statements (Python's Switch-Case option)

*"In **most languages, a switch statement is defined** across many individual lines using one or two keywords." (Wikipedia 2015, **https://en.wikipedia.org/wiki/Switch_statement**)*

The following **pseudo-code** show the ***general pattern*** of a Switch-Case:

```
switch(expression) {
  case n:
    code block
    break;  //Do not 'fall through' to check the next option
  case n:
    code block
    break;  //Do not 'fall through' to check the next option
  default:
    default code block
}
```

*"An **if ... elif ... elif ... sequence is a substitute for the switch or case statements** found in other languages." (Python Docs, 2015, **https://docs.Python.org/2/tutorial/controlflow.html**)*

Here is an example of a Python if-elif statement

```
intVar = 2
if (intVar == 1):
  print("1")
elif (intVar == 2):
  print("2")
else:
  print("Please choose 1 or 2")
```

Listing 7

Multiple Conditions

You can include more than one condition by using the operators "**or"** and **"and."**

```
intVar = 1
strName = "Bob"
if (intVar == 1 and strName == "Bob"):
  print("1 and Bob")
else:
  print("other")

intVar = 2
strName = "Bob"
if (intVar == 1 or strName == "Bob"):
  print("1 and Bob")
else:
  print("other")
```

Listing 8

Nested if statements

Nested If statements allow you to **test multiple "nested" conditions**.

```
intVar = 1
strName = "Bob"
if (intVar == 1):
  print("1") # Statements executed only if condition1 is true.
  if(strName == "Bob"):
      print("Bob")
else:
  print("other")
```

Listing 9

LAB 3-1: If statements

In this lab, you create an test a Python script.

1. Type and test the following code.

```
intVar = 2
if (intVar == 1):
 print("1")
 if (intVar == 2): # This is a nested if statement.
   print("2")
else:
  print("other")
```

Listing 10

2. Write down what it does and does not do.

Comparison operators

"There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, x < y <= z is equivalent to x < y and y <= z, except that y is evaluated only once (but in both cases z is not evaluated at all when x < y is found to be false)." (Python doc, 2015, **https://docs.Python.org/3.5/library/stdtypes.html***)*

Here is a table of the eight comparison operators:

Operator	Meaning	Example
<	something is **less** than another	5 < 10
<=	something is **less** than **or equal** to another	5 <= 10
>	something is **greater** than another	10 > 5
>=	something is **greater** than **or equal** to another	10 >= 5
==	something is **equal** to another	10 == 10
!=	something is **not equal** to another	10 != 5
is	something is the **same** object	objectX is objectY
is not	something is **not the same** object	objectX is not objectY

Table 1

Note: An "object," in this case, refers to data in memory. The "is" operator verifies if two variables are using the same memory address.

Comparing strings

Strings are case-sensitive, so you need to be somewhat careful when comparing them.

```
strName = "bob"

if (strName == "Bob"): print("true")
else: print("false")

# It is common to covert data to a upper or lower case for comparisons
if (strName.lower() == "Bob".lower()): print("true")
else: print("false")
```

Listing 11

The *not* operator

This operator reverses the results of a Boolean expression. The ***"!"*** is a common notation for ***not*** in most languages.

```
strVar = "Bob"

if (strVar == "Bob"): print("true")
else: print("false")

# not reverses true to false
strVar = "Bob"
if not(strVar == "Bob"): print("true")
else: print("false")

# The ! is a common symbol for not
if (strVar != "Bob"): print("true")
else: print("false")
```

Listing 12

Comparing Objects

When you make **objects using a custom class** in Python, the comparison is made on the ***address in memory and not their content***. *Although creating a custom class is an advanced topic we cover later in the course, let's look at a simple example anyway.*

```
class Demo: # This line of code defines a class named "Demo"
  pass  # This line tells Python that there is no code to process yet!

objD1 = Demo() # Make a new Demo object
objD2 = Demo() # Make another new Demo object
if (objD1 == objD2):
  print("same")
else:
  print("different")

objD1 = objD2 # Have the objD1 variable point to the objD2 address
if (objD1 == objD2):
  print("same")
else:
  print("different")
```

Listing 13

The "is" operator

The operators "is" and "is not" also test for object identity. So, the expression "*x **is** y*" is true if x and y are the same object in memory.

```
class Demo: # This line of code defines a class named "Demo"
  pass  # This line tells Python that there is no code to process yet!

objD1 = Demo() # Make a new demo object
objD2 = Demo() # Make another demo object
if (objD1 is objD2):
  print("same")
else:
  print("different")

objD1 = objD2 # Have the objD1 variable point to the objD2 address
if (objD1 is objD2):
  print("same")
else:
  print("different")
```

Listing 14

The exception to these is when you are **comparing two objects that act like "Value" types,** like strings.

```
strName = "Bob"
strNickName = "Bob"

if (strName == strNickName): # Compares the VALUE not the address
  print("same")
else:
  print("different")

if (strName is strNickName): # Compares the VALUE not the address
 print("same")
else:
  print("different")

# Now, lets turn the string into a list
strName = ["Bob"]
strNickName = ["Bob"]

if (strName == strNickName): # STILL, compares the VALUE not the
address
  print("same")
else:
  print("different")

if (strName is strNickName): # Compares the ADDRESS not the value!
 print("Same")
else:
  print("different") # Will be different this time!
```

Listing 15

NOTE: Just like the last time we talked about the differences between Value and Reference types, this may be confusing. Remember that if you know that this behavior exists, and **remember to test your code**, it is just something you get used to.

(If in doubt, test it out!)

Boolean Values

In most languages, **zero evaluates as a "Boolean" False,** while any other number evaluates as True. The **same is true for empty strings** (indicated by two quotation marks with no characters in between them).

However, **what is correct in Python? Let's test some code to find out in the next lab!**

LAB 3-2: If statements

In this lab, you create an test a Python script.

1. Type and test the following code.

```
print("using Boolean values")
if(True): print("T")
else: print("F")

if(False): print("T")
else: print("F")

print("Using Numbers")

if(0): print("T")
else: print("F")

if(1): print("T")
else: print("F")

if(2): print("T")
else: print("F")


print("using Strings")
if("abc"): print("T")
else: print("F")

if(""): print("T") # Note we are using an empty string!
else: print("F")
```

Listing 16

2. Write down what it does and does not do.

Using numbers for Booleans

Be careful, in Python 0 is always equal to False, but only 1 is equal to True.

```
if(True == 2):  # Is True the same as 2?
  print("Yes")
else:
  print("No")
```

Listing 17

The reason you need to know this because sometimes programmers use numbers to stand in for boolean values. Here is an example:

```
intX = 100
intY = 100

print("Zero test")
if( 0 == (intX == intY) ): # Does False == (True)
  print("true")
else:
  print("false") # Will be false

print("One test")
if( 1 == (intX == intY) ): # Does True == (True)
  print("true") # Will be True
else:
  print("false")

print("Neither Zero or One test")
if( 2 == (intX == intY) ): # However what about this one?
  print("true")
else:
  print("false") # Will be False

print("Boolean True test")
if( True == (intX == intY) ): # …and how about this one?
  print("true") #Will be True
else:
  print("false")
```

Listing 18

The keyword None

In Python, the keyword *None* is used to **indicate that a variable is not assigned a value** yet. (Other languages use the word Null)

```
strData = None
if(strData):
  print("T")
else:
  print("F") # Will be false
```

Listing 19

Loops

Loops allow you to **perform a command multiple times**, a task that is so handy that all programming languages have some way of doing this. However, **each language may use different commands**, and **some have more looping options** than others. **In Python, there are only two options, the "while loop" and the "for loop**."

while loop

The most common loop in programming is the "while loop." It **looks and works very much the same in any language** you decide to learn.

The way it works is to **execute one or more statements repeatedly** when an expression is true. **You need to be careful to tell it to stop, or it loops indefinitely!**

One way to control the loop is to use **a counter variable**. You **set an initial value** for the variable, **compare the current value** to see if the loop should continue, and then **increment the value** and compare it again. **When the comparison no longer evaluates to True, the loop stops!**

```
intCounter = 0
while(intCounter < 3):
  print(intCounter)
  intCounter = intCounter + 1
```

Listing 20

Another common pattern is using a "**Flag" value** to stop the loop, which **allows a user to change the "Flag" value as the loop executes its code**.

```
strUserInput = input("Type in a string to echo (Enter 0 to quit!)")
while(strUserInput != "0"): #Make sure to use Quotes!!
  print(strUserInput)
  strUserInput = input("Type in a string to echo (Enter 0 to quit!)")
```

Listing 21

break

The break statement **immediately breaks out of the loop**. This design also **allows a user to exit the loop as its code**.

```
strUserInput = ""
while(True):
  strUserInput = input("Type in a string to echo (Enter 0 to quit!)")
  if(strUserInput == "0"): break
  else: print(strUserInput)
```

Listing 22

continue

The **continue statement forces the loop to jump back to the beginning** and reevaluate the condition.

```
strUserInput = ""
while(True):
  strUserInput = input("Type in a string to echo (Enter 0 to quit!)")
  if(strUserInput == "0"): break
  if(strUserInput == "C"): continue
  print(strUserInput)
```

Listing 23

These "Jump" statements are often used to give users a choice in the way a program runs. For example, we could **create a menu**, allowing users to choose the time of output they want to see when a program run.

LAB 3-3: Creating a menu

In this lab, you create a simple text menu.

1. Type and test the following code.

```
strUserInput = ""
while(True):
    print("Enter an option to process data:")
    print("1 = Add Data")
    print("2 = Delete Data")
    strUserInput = input("(Enter 0 to quit!)")
    if(strUserInput == "1"): print("Adding")
    elif(strUserInput == "2"): print("Deleting")
    elif(strUserInput == "0"): break

print(strUserInput)
```

Listing 24

2. Write down what it does and does not do.

Writing Data to a File

Writing data to files is a practical use of while loops and if statements. Here is an example of how to write to a file:

```
objFile = open("C:\\_PythonClass\\TestData.txt", "a")
objFile.write(input("Enter your data: ") + "\n")
objFile.close()
```

Listing 25

Now we add a loop and conditional to the program.

```
objFile = open("C:\\_PythonClass\\TestData.txt", "a")
print("Type in a string to write (Enter 'Exit' to quit!)")
while(True):
  strUserInput = input("Enter your data: ")
  if(strUserInput.lower() == "exit"): break
  else: objFile.write(strUserInput + "\n")
objFile.close()
```

Listing 26

LAB 3-4: Working with Conditionals, Loops, and Files

In this lab, you will create a script that using many of the techniques you learned it this module.

1. Create a script that lets a user add two numbers together and saves the answer to a file. Let the user continue adding numbers together until they type in the word "exit."

Hint: Use Pseudo-code and the input() function.

Lists of values

You often **work with more than a single value** when programming. When that happens, you can group the values into a list and work with the list **as a single object**. You can use the statement in Listing x to put values in a list.

```
lstData = ['red','green','blue']
```

Listing 27

Once data is in the list you can **access the individual values using their location** in the list, **starting with zero**.

```
lstData = ['red','green','blue']
print(lstData[0]) # prints red
```

Listing 28

Note: List objects are essential programming tools. We look at them **closely in the 4th** and **5th module.**

Program Arguments

You may **remember this command from module01, "python -?."** In that module, we saw how this command prints out helpful information about the Python program's options (Figure 1).

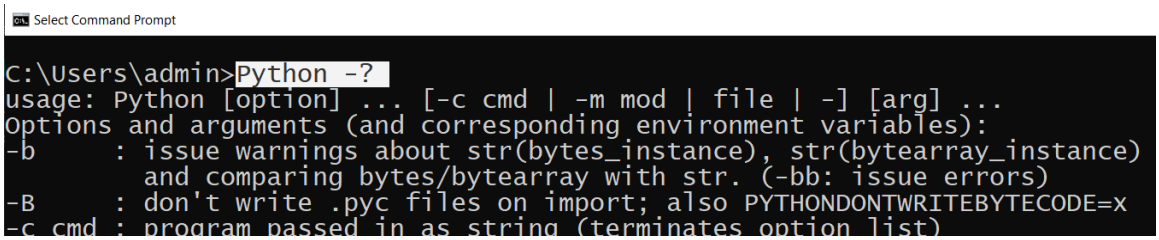

Figure 1. Using a program argument

Program arguments let configure your program at the moment it executes. Pin the case of the Python executable, using the "?" argument changed the way the executable ran. Without the argument, Python starts in interactive mode instead of printing out help text.

When using programming arguments, a programmer checks to see if any values were passed to an application and respond to what they find. **Let's consider a couple of examples!**

Here is the **first** one, which **does not use programming augment** to get configuration data. Instead, it **uses the input function**, you learned earlier, to capture a user's data:

```
#Example 1: Pause and Ask for input
strData = input("Enter your data") # Get user input by pausing the
program
print(strData) # Print user data
```

Listing 29

When this script runs, it **pauses**, ***waits for user input***, and **then prints** the data to the screen.

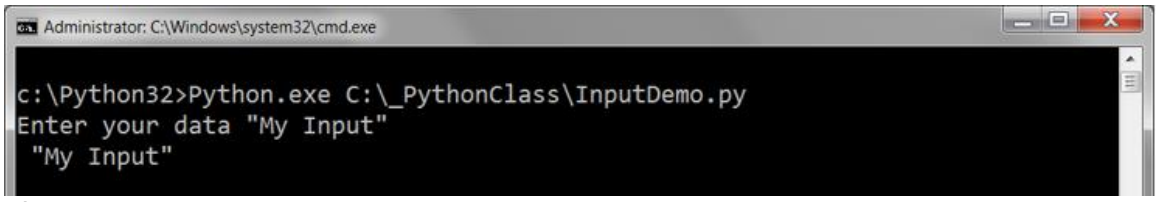

Figure 2

Now, let's looks at a **second example**. This example **is similar, except that <u>it never pauses to ask the user for data</u>**:

```
import sys #This forces your script to reference Python's system module
#Example 2: Input is added when the script starts
strData = sys.argv[1] # Get user input as a script argument
print(strData) # Print user data
```

Listing 30

In the **second example**, the user **includes the script's name after the Python executable** values, **then program arguments after the script's name**. This captures the user's input as the script starts.

Figure 3

Important. Make sure you include the Python executable, the script name, and the arguments to run the script.

Accessing Program Arguments

When you add arguments to the command, they are passed in as a list of values. In Python, these values are stored **in a built-in system variable called** "***<u>argv</u>*" (**which stands for "Argument Values"), but **something similar is used in other languages**.

You access the individual values in argv, by indicating an id number, or "index."

The pattern is:

- The value of **argv[0]** will be the **script name** (and path on a PC).
- The value of **argv[1]** will be the **first argument passed into the script**.
- The value of argv[2] will be the second argument passed into the script after a "space" character.
- The value of argv[n] will be the n^{th} argument passed into the script after a space.

```
import sys
strData = sys.argv[0] # Get Script name
print(strData) # Print Script name

strData = sys.argv[1] # Get Argument 1
print(strData) # Print Argument 1 data

strData = sys.argv[2] # Get Argument 2
print(strData) # Print Argument 2 data
```

Listing 31

When the script in Listing x runs, it displays the file name and passed arguments (Figure 4).

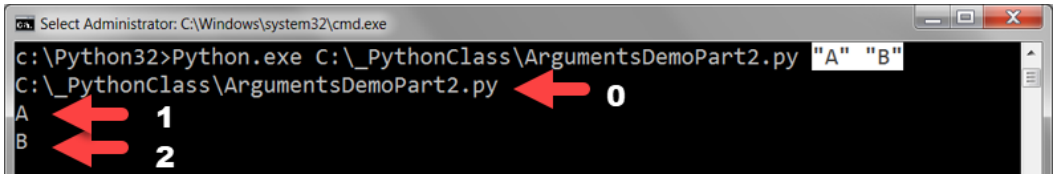

Figure 4

When using this feature, **you should always check that arguments were passed into the program**. If there are **no arguments, then the expression "len(sys.argv) == 0" will evaluate to true**, since the length function, len(), returns the number of items in the collection.

```
import sys
if(len(sys.argv) > 2):
    intArg1 = int( sys.argv[1] ) # Get Argument 1
    intArg2 = int( sys.argv[2] ) # Get Argument 2
    strData = str(intArg1 + intArg2) # Perform some Processing
    print("The Sum of the first and second arguments is: " + strData)
else:
    print("This script requires two argument to run")
    print("Ex: MyScript Arg1 Arg2")
```
Listing 32

Why use arguments?

Arguments are **convenient in Console applications, for they allow you to run scripts without pausing for user input.**

As an example, consider a script scheduled to run automatically, without human interaction. You could **use arguments to configure how the program behaves on a given execution**. This way the same script can be used in different ways. Figure 5 shows a script called "CheckForNewLogFile.py" being sent the configuration option to force the log file to be overridden.

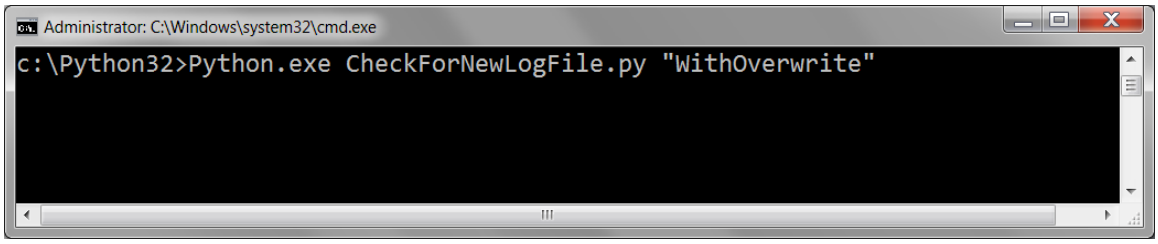

Figure 5

Summary

In this module, we looked at common statements, expressions, and features in the Python language.

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 pseudo-code?
- What are "conditional" statements?
- How do you use multiple conditional expressions?
- What are comparison operators?
- What are the boolean values of the integers 0, 1, and 2?
- What is a Loop?
- Name the most common programming loop?
- What do "break" and "continue" do?
- What statement allows you to write data into a file?
- How do you pass arguments into a script?

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.