

Intro to Scripting - Python

Module 2

©2014 - 2019 by Irv Kalb. All rights reserved.

White space:

++++ Concept of “white space”. All invisible characters are ignored by Python. You can add spaces, returns, in between things anywhere you want. They will be ignored by Python. When your Python programs start to get long, adding blank lines can aid in readability. (You will hear me talk about readability a lot!)

++++ Just as we talked about a naming convention for variables, as a convention most programmers will put spaces around math operators:

```
var1 = 4
var2 = 5
var3 = 6
myVariable = var1 + var2 + var 3 #space on either side
```

could write it like this

```
myVariable=var1+var2+var3      #no spaces
```

could write it like this

```
myVariable    =    \
```

```
var1 + var2      +    var3 #lots of spaces
```

They will all do the same thing. The extra spaces are “white space” and are ignored by the compiler. Adding these characters makes your code more readable by humans.

Spaces will become important later

Errors (3 types):

++++ 1) Syntax or Compile Errors

```
a = 5 5
```

This breaks the rules of Python. You get an error dialog and a red highlight on your source code indicating where Python thinks your error is. You need to figure out what went wrong and fix it. You may have seen this type of error in your first homework if you typed something wrong.

Another example:

```
y = (5 +
```

```
x = 1
```

Did not close the parentheses correctly. Python is searching for the close parenthesis, and gives you the red box where it thinks the error is – actually on the next line.

++++ 2) Run Time or Exception error

```
x = 5 + 'abc'
```

You can't really add a number and a string. So, Python gives you an error when you try to run this line.

Another example:

Assume that you have not used a variable named: xyz

```
print(xyz)
```

or

```
y = xyz + 1
```

These are valid Python statements, but since xyz was not defined before running these lines, Python gives you an error message – a traceback.

Most commonly, when you see this type of error, you have spelled or capitalized a variable wrong. Remember – all uppercase/lowercase letters in a variable name must be the same every time it is used. Variables ABC and abc are completely different variables.

```
+++ 3) Wrong answer
```

```
# Attempt to ADD 2 and 5
```

```
total = 2 * 5
```

```
print(total)
```

The above lines of code are valid Python statements. And the program will run without any error messages. But it produces

an incorrect answer. This type of error is difficult to track down.

It is obvious what is wrong in this example. But when you start to write larger programs, it gets more difficult to find such errors. Generally you add print statements to write out the value of your variables as the program runs. Then you compare the output of your print statements against what you expect. This helps narrow down the location of the error.

Overview of built-in functions:

Let me explain what a built-in function is by using an analogy. My car has a radio built-in. It's not really the car itself, it was developed separately. But when I bought the car, it had a radio in it. When I want the radio to do something, I press its buttons, or turn its dials, and the radio responds appropriately. I don't need to know HOW the radio works, I just need to know how to use the radio's controls.

Python has a number of things like this that are called built-in functions. They are pieces of code that are available for you to use in your programs. (The real power of programming comes when you build your own functions, and we will get to that very soon.) Let me show you how a built in function works.

Function call:

Using a function is known as “calling” a function, or making a “function call”, or making a “call” to a function. To call a

function, you supply the name of the function followed by a set of open/close parentheses. Here is the generic form:

```
+++++  
<functionName>( )
```

However, most built-in functions will expect you to supply one or more pieces of information in addition to the function name. This is referred to as “passing” data to a function. Each piece of information that is passed is known as an argument.

Arguments:

Definition: An Argument is a value that is passed when you call a function.

Inside the parentheses after the function call, you put any data you want to send to that function. Here’s what a generic call to a function with arguments looks like:

```
+++++  
<functionName>(<argument1>, <argument2>, ...)
```

Results:

When you call a built-in function, the function does its work, and it hands back a result. When the function is finished, the result replaces the call and its arguments. That is, the line continues to execute using the returned value in place of the call. When you make a call to a function, you will often assign the returned value to a variable using an assignment statement:

<variable> = <functionName>(<argument1>, <argument2>, ...)

Example: Built-in type function:

There is a built in function that can tell us the data type of any variable. Not surprisingly, it is called 'type'.

Let's start by creating a variable and giving it a value of the type of the number 10 (using an assignment statement):

```
+++++
typeOf10 = type(10)
print(typeOf10) # shows <type 'int'>
```

What has happened here is that we are calling the type function, and we are 'passing' in a value (10). The type function does whatever it needs to do, and gives us back an answer. We put that returned answer into the variable typeOf10.

Now we'll do a similar thing using a variable:

```
age = 18
print(type(age)) # shows <type 'int'>
```

Here we are calling the same built-in function, but we used a variable. In this case, Python passes in the value of the variable: age (which shows this is an integer also).

We don't know how the type function does what it does, and we don't care. It's kind of like how most of think about a microwave oven, or TV, or phone, etc. Most of us don't really know how these things work internally, but we are happy using them as long as they continue to do what we need them to do.

Now, let's try this:

```
print(type(123.45))
```

```
name = 'lrv'  
print(type(name))
```

Whenever you use a variable, the program uses the value of that variable. So when you say:

```
print(type(name))
```

It supplies the value of the variable name, and does this:

```
print(type('lrv'))
```

Now, let's try to confuse you:

```
myVar = '1234'
```

What do you think the type of myVar is??

Let's ask Python to tell us:

```
print(type(myVar))
```

(shows string)

Now, let's execute this:

```
myVar = 1234
```

I've changed the contents of the variable myVar, and the variable is now an integer number variable:

```
print(type(myVar))
```

(shows integer)

The thing to notice is that 1234 and '1234' are very different things. 1234 is an integer number, and '1234' is a string of characters. These two are totally different values

I'll show how to switch between them in a minute.

Getting input from the user:

The typical t, Work with the data (do computation(s)), and Output some answer.

In Python we can get input from the user using another built in function called "input". Here's how we use it:

```
+++++
```



```
<variable> = input(<prompt string>)
```

Example:

```
favoriteColor = input('What is your favorite color? ')
print(favoriteColor)
```

The variable `favoriteColor` is a string variable, because anything the user types will be a string of characters.

When this statement runs:

- The prompt is printed to the Shell
- The program stops and waits for the user to type a response.
- The user types some sequence of characters as an answer.
- When the user presses Enter (PC) or Return (Mac), Whatever the user typed is returned as the result.
- Typically, we use an assignment statement to put the user's response into a variable.

Question: What data type is the variable???

+++++

That works great. But what if you want the user to enter a number? Try:

```
nDollars = input('How many dollars do you have? ')
print(type(nDollars))
```

nDollars is a string, because anything the user types in response to input is those characters the user typed, even if those characters are digits.

Assuming that we want to do some math with the variable nDollars, we probably want to have that variable as an integer

Conversion functions:

Python has three “conversion” functions that can change a value from one data type to another:

++++1) To convert from a string to an integer:

```
nDollars = input('How many dollars do you have in your wallet?')  
nDollars = int(nDollars)
```

This converts the variable nDollars from a string into an integer, and puts the resulting integer value back into the same variable nDollars. Now we can do some math with this variable knowing that it is a integer.

++++2) To convert from a string to a float:

```
thePrice = input('Enter the price: ')  
thePrice = float(thePrice)
```

This converts the variable thePrice from a string into a float, and puts the resulting float value back into the variable thePrice. Similar to using the int function, now we can do some math with the price variable, knowing that it is a floating point value.

++++3) To convert from an integer or float to a string:

```
myAge = 37
myAge = str(myAge)
aPrice = 150.75
aPrice = str(aPrice)
```

These lines convert from an integer or float back into a string. We will use this later when we are building long strings for output.

++++

In summary, these five built in-functions in Python are very useful:

type(<valueOrVariable>)
returns data type of value passed in

input(<promptString>)
returns user's response

int(valueOrVariable)
returns an integer version of value passed in

float(valueOrVariable)
returns a float version of value passed in

`str(valueOrVariable)`
returns string of value passed in

Also:

`print(<thing>, <thing>, ...)` # is a function

Python programmers will typically do these conversions and assign the resulting value back into the same variable name. This is perfectly legal because an assignment statement works by taking the right hand side which calls the conversion function, and then assigns the result back into the variable on the left of the equals sign.

First real program:

As an in class exercise, I want you to try to write a program that will:

prompt the user to enter a number

prompts the user to enter a second integer

add them (using a third variable)

print out the result.

```
# Add two numbers
value1 = input('Enter a number: ')
value2 = input('Enter another number: ')
value1 = int(value1)
value2 = int(value2)
total = value1 + value2
print('The sum of', value1, 'and', value2, 'is', total)
```

Second in-class assignment: build simple cash register program.

- prompts the user for the price of an item
- prompts the user to enter an amount paid
- calculates the change to give back
- reports the change to give back

```
# Simulate a cash register
itemAmount = input('Price of item')
itemAmount = float(itemAmount)
amountGiven = input('Cash given')
amountGiven = float(amountGiven)
change = amountGiven - itemAmount
print('Change is: ', change)
```

Concatenation:

We know that there are a number of operations you can do with numbers (+, -, *, /, **, %). But if you try to add strings, that wouldn't really make sense.

$$\begin{array}{r} \text{Joe} \\ + \text{Schmoe} \\ \hline \text{??????} \end{array}$$

Or would it? Well, you can't really add strings in the numerical sense, it doesn't really make sense to add a string like 'Joe' and a string like 'Schmoe'. But it really isn't much of a stretch to think of adding a string to a string to have a slightly different meaning. That is, take the first string, and add the second string to the end of the first one.

Imagine if you had two strings and you wanted to put them together. For example, if you had:

```
firstString = 'Hot'
secondString = 'Coffee'
```

And you wanted to put them together.

++++DEF: Concatenate means take a string, and add on another string:

In Python, along with most other languages, the “+” is called the concatenation operator.

+++++Example:

```
firstString = 'Hot'
secondString = 'Coffee'
```

```
concatenatedStrings = firstString + secondString
print(concatenatedStrings)
```

In class assignment:

Open a new Python file (remember to save with an extension of .py), and enter this program:

```
# Ask the user for their first name
# Ask the user for their last name
# Concatenate to create a full name (use a third variable)
# Output a greeting
# Example:
#   Hello Irv Kalb I hope you are doing well.
```

```
firstName = input('Enter your first name: ')
lastName = input('Enter your last name: ')
fullName = firstName + ' ' + lastName

print('Hello', fullName, 'I hope you are doing well.')
```

Concatenate the first name, a space character, then concatenate the last name.

Let's talk about code – and get into building programmer-defined functions. Software is a series of detailed instructions. Similar to driving directions – the more detailed the better. Many individual steps can be broken down into smaller and smaller steps.

A recipe as an analogy for building software:

Let's take a look at a recipe, and see how we can also look at a recipe as a straightforward set of instructions:

++++Baking a cake

Ingredients:

- 1 Box of cake mix (Chocolate)
- 1 Box of Jello Instant Pudding (Chocolate)
- ¼ pound chocolate chips
- 4 eggs

¾ cup of water

1/3 cup of oil

Instructions:

Preheat oven to 350 degrees

Crack eggs into bowl (4)

Blend (high, 4 minutes)

Add in water

Add in oil

Blend (medium, 1 minute)

Add in cake mix

Add in Jello mix

Blend (medium, 10 minutes)

Add in chocolate chips

Blend (low, 1 minute)

Grease bundt pan

Pour mixture into pan

Bake at 350 for 45 minutes

Remove from oven.

+++++ Notice that the recipe contains ingredients – nouns, such as eggs, water, oil. Think of these as data. Then it has instructions that are actions or verbs – such as Blend, Add In, Grease, Pour, etc. Think of these as code.

Many of these steps can be broken down further and further. This is a process call “stepwise refinement”. For example, “Crack Eggs into bowl” can be broken down as:

+++++ For each egg:

Pick egg out of carton.

Hit gently on surface

- Move egg over bowl
- Crack open
- Dump until all egg goop goes into bowl
- Discard shell

Once you have developed the detailed steps you need to take to describe “Crack Eggs into Bowl”, and you have tested the steps to know that they work correctly, you can “Crack Eggs into Bowl”. If this recipe had the need for two different steps where you had to crack eggs, now we can do this one set of steps every time needed to crack eggs. Further, if we had a need in other recipes to crack eggs in a bowl, now we have a procedure for doing it. Additionally, if we find a more efficient way to crack eggs, we only have to modify some steps in “Crack eggs into bowl in once place, and all our recipes become more efficient.

++++ Notice in our recipe how many times we see Blend. In this recipe, there are four cases where we want to do a blend. Notice further that each time we do a blend, we are specifying slightly different information.

++++ So, let’s take a closer look at “Blend”. We could break down – or define – “blend” into something like

Blend setting, nMinutes

- Turn mixer onto given setting
- Start timer for nMinutes
- Until time is up
 - Stir with spatula
 - Break up any lumps
 - Scrape sides of bowl

This is the basic idea behind how we can build code. You often create small groups of statements that are called “routines” (also known as subroutines). And once you know that they work the right way, you can call them multiple times to do what they are supposed to do. Each of these sub routines, might call yet another subroutine to do a smaller task. For example, “Stir” could be broken down into a more detailed list of operations like, grab spatula, place under mixture, rotate mixture upwards towards beaters, etc.

Definition of a function

A group of “statements” like this in Python is called a “function”.

++++ DEF: A Function is a series of related steps that make up a larger task, which is often called from multiple places in a program.

```
+++++def <functionName>(<optionalParameters>):  
    #notice the colon  
    <indented function statements>
```

“def” is short for definition. This the definition of a function. You give the name of the function (a name of your choosing), a set of parentheses, let’s ignore the <optionalParameters> for now, and finally a colon.

All the statements that are contained inside the function are indented. Python uses indenting to show a grouping of lines. The convention is to indent 4 spaces (you can change this in the IDLE Preferences, but you will see that it defaults to 4). If you have ever seen C or Java or Javascript code, they use open and close braces {} AND often also use indenting for defining blocks of code. Using indenting only instead of braces makes Python code much more readable than those languages.

Building our first function:

++++ Please open a new file, save it as Groceries.py and type:

```
def getGroceries():  
    print("At the grocery store I need to buy")  
    print("cake mix")  
    print("jello")  
    print("oil")  
    print() #blank line
```

This is the definition of a function. It is the detailed implementation of something that the computer can do. It is like “crack an egg” in our recipe. It contains a series of steps to perform. So, let’s save and run our program.

Woohoo, nothing happens. Why – because we didn’t ask our program to do anything. It’s like saying that you know how to throw a ball, or take a drink, or open a door, but if I don’t ask you to do anything, you will just sit there.

Calling a programmer-defined function:

Therefore, if we want a function to run, we have to write call to tell the computer to run it. As we talked earlier with built-in

functions, this is known as “calling” the function. Python has a number of built-in functions (like type, input, int, float, str, etc.), but they don’t do anything until you call one in your code.

Just as we did with built-in functions, you call a function that you write by typing its name with a set of parentheses, and include any data you want the function to act on:

```
<functionName>(<data to pass>)
```

++++ To call the getGroceries function, we do this:

```
getGroceries() # calling the function, must have  
parentheses, even if there is no data to pass.
```

Type that in after your function, save and run.

Can be called multiple times:

```
getGroceries()  
getGroceries()  
getGroceries()
```

Until now, we have only seen code that runs top down. Now with the ability to create and use functions, we can affect the ‘order of execution’. That is, we can jump around a little.

Receiving data in a programmer-defined function - parameters:

But this function is not very useful because it is “hard-coded”. That is, every time you call getGroceries, it does the exact

same thing. It would be better to have a function will do something different depending on the data that is passed in.

Remember in my recipe for chocolate cake when we did a blend? The blend wanted two pieces of information, a setting and a number of minutes. When you pass values in a call, the function can do different work and/or generate different results depending on the information that is passed in. When you make a call, remember that each value passed into a function is an argument.

Now, let's look at the other side of a call, and look at how to receive the data that is passed in to a function:

DEF: A Parameter - a variable (defined in the 'def' statement of a function) that is given a value when a function starts.

++++ Example (from chocolate cake recipe):

```
# example of function definition with parameters:  
def blend(setting, nMinutes):
```

```
    <indented block of code>
```

```
# examples of function calls with arguments:
```

```
blend('high', 10)
```

```
blend('medium', 1)
```

```
blend('low', 1)
```

```
desiredSetting = 'high'
```

```
numberOfMinutes = 8
```

```
blend(desiredSetting, numberOfMinutes)
```

We are calling the same function (blend), but we are using different arguments in each of these calls

++++++ When we call the function, the values of two arguments. the two values that are passed in, are assigned into the parameter variables: setting and nMinutes.

function definition

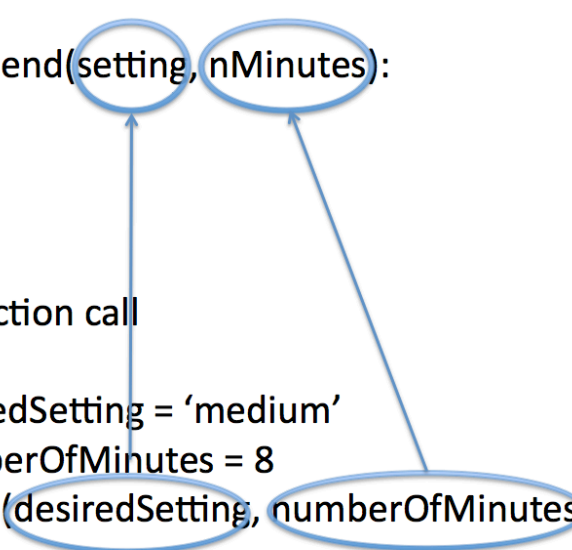
```
def blend(setting, nMinutes):
```

function call

```
desiredSetting = 'medium'
```

```
numberOfMinutes = 8
```

```
blend(desiredSetting, numberOfMinutes)
```



It is as though there is an assignment statement from each argument's value to the associated parameter variable.

Let's do this for real in Python. In our getGroceries function, let's replace the first item, milk, with a parameter – something that you pass in when you make the call. Again, each parameter is a variable whose value gets set from the associated value at the calling line of code.

++++

```
def getGroceries(storeName):
```

```
#notice the use of a parameter variable
print("At", storeName, "I need to buy")
print("cake mix")
print("jello")
print("oil")
print() #blank line
```

Now we can call getGroceries and using different argument values:

```
getGroceries("Safeway")
getGroceries("Luckys")
getGroceries("Amazon.com")
```

[Add in extra 'separateRuns function']

[Step through the order of execution. With PythonTutor:]

<http://pythontutor.com/visualize.html>

Replace all items with parameters. Pass in 4 items when you make the call:

```
def getGroceries(storeName, item1, item2, item3):
```

```
    print("At", storeName, "I need to buy")
    print(item1)
    print(item2)
    print(item3)
```

```
# Now call the function with different arguments
getGroceries("Safeway", "soap", "apples", "cat food")
getGroceries("Luckys", "milk", "soda", "peas")
```


can also call with variables

```
mustGet = 'book'  
mustAlsoGet = 'dvd'  
getGroceries('Amazon.com', mustGet, mustAlsoGet, 'cat food')
```

++++

Building a simple function that does addition:

Example of numeric parameter – pass in a number value

```
def addTwo(startingValue):  
    endingValue = startingValue + 2  
    print('The sum of ', startingValue and 2 is:', endingValue)
```

```
addTwo(5)  
addTwo(10)
```

The sum of 5 and 2 is: 7

The sum of 10 and 2 is: 12

Add the ability to get a number from the user using the input function. Then use that number in a call to addTwo.

Parameter variables take on the values that were passed in. e.g., startingValue would get 5 from the first call, then startingValue would get 10 in the second call. If we make a call with a variable, then the value of the variable is passed and is given to the parameter.

Notice also that I am showing a pattern of putting an functions at the top, then having calls to the function below. This like a story where you must introduce a character – define the characteristics before you have that character have some dialog. You must define a function before you try to call it. If you do it the other way around, then Python will give you an error.

Here is a function that expects four number parameters:

```
def calculateAverage(value1, value2, value3, value4):  
    total = value1 + value2 + value3 + value4  
    average = total / 4.0  
    print("Average value is: ", average)
```

```
calculateAverage(2, 3, 4, 5)  
calculateAverage(-3, 5, 15, 1000)  
calculateAverage(1.4, -2.5, 14.3, 200.5)
```

Add to our definition of a function: a function can return no value, 1 value, or any number of values.

Returning a value from a function – the return statement

Python recognizes the end of a function by the indenting – or more accurately, by the lack of indenting. As soon as a line is found that has the same indenting as the def statement that started the definition of the function, Python knows that it has

reached the end of the definition of a function. After the last indented statement is run, control passes back to a point just past where the function was called.

In the small functions shown so far, each function ends by executing a print statement to print out some result. However, in most cases, programmer-defined functions are typically built to do some calculation(s) to generate an answer, and to give that answer back to the caller. Remember that this is the way that all of the built-in functions that we have seen operate.

Let me give an analogy. Assume that I am a manager, and I call in one of my employees, and I say, go off and figure out the cost of a widget, and I specify some values, like length, width and height of a widget. As a good manager, I don't really care HOW you do your work I just want the final answer given back to me so I can do some further work with it.

A function works in a similar way. If you want the function to give back an answer, you use the 'return' statement with a value. If you want the function to give back a result, then you use a "return" statement to say what your function will return.

```
def someFunction():  
    <body of the function>  
    # return a value to the place  
    # where the call was made  
    return <valueToReturn>
```

For example, here is a modified version of the addTwo function that will return a single number value:

```
def addTwo(startingValue):  
    endingValue = startingValue + 2
```

return endingValue #returns rather than prints

```
sum1 = addTwo(5)
print(sum1)
sum2 = addTwo(10)
print(sum2)
sum3 = sum1 + sum2
print(sum3)
```

By default, after a function executes the last line of code, control is given back to the line after the call. Further, when a function ends, the function does not do anything else.

In our calculateAverage example, let's modify the function to return its result, rather than printing it. Here's how we can do that:

```
def calculateAverage(nValue1, nValue2, nValue3,
nValue4):
    total = nValue1 + nValue2 + nValue3 + nValue4
    average = total / 4.0
    return average
```

The place where the function is called is replaced by the returned value. Very often, you use an assignment statement to capture the returned value:

```
<variable> = <functionName>(<args>)
```

Then we would call the function, passing in four values as arguments, and the function returns a value. The value is then stored into a variable:

```
nAvg1 = calculateAverage(2, 3, 4, 5)
nAvg2 = calculateAverage(-3, 5, 15, 1000)
nAvg3 = calculateAverage(1.4, -2.5, 14.3, 200)
print(nAvg1, nAvg2, nAvg3)
```

Returning no value:

You can use the return statement without specifying any values:

```
return # no values specified
```

When you write it this way, Python actually returns a special value called None.

The expectation is that when you call a function that does not return a value, you do use an assignment statement with for any returned value.

Returning more than one value:

Python is different from any other language that I know, in that it allows you to return any number of values:

```
return <value1>, <value2>, ...
```

And it would be called like this:

`<varA>, <varB>, <varC>, ... = functionName(arg1, arg2, arg3, ...)`

Temperature conversion functions:

Let's build a simple temperature converter: C to F, and F to C

++++

Fahrenheit to Centigrade:

$$C = (F - 32) \text{ times } 5/9$$

Centigrade to Fahrenheit formula:

$$F = (1.8 \text{ times } C) \text{ plus } 32$$

++++

```
def F2C(nDegreesF):  
    nDegreesC = (nDegreesF - 32) * (5 / 9)  
    return nDegreesC
```

```
print(F2C(22))
```

```
def C2F(nDegreesC):
```

```
nDegreesF = (1.8 * nDegreesC) + 32
return nDegreesF
```

```
print(C2F(100))
```

Modify to ask the user for input and print the result

```
# ask the user for input
nDegreesF = input('Enter the number of degrees Fahrenheit: ')

# convert to an integer
nDegreesF = int(nDegreesF)

# call the F2C function to convert from Fahrenheit to
Centigrade
nDegreesC = F2C(nDegreesF)

# output the results
print(nDegreesF, 'degrees F is', nDegreesC, 'degrees C.')
```
