Python 1

Introduction to Elementary Programming

Classroom Course Manual

Written, designed, and produced by:
DoIT Software Training for Students
Last Updated 1/16/2017

About Software Training for Students

Software Training for Students is an organization on campus that provides free software training to all students and faculty. Our services include custom workshops, open-enrollment classes, one-on-one project help, and access to Lynda.com. For more information on the Software Training for Students (STS) program, visit our website at **wisc.edu/sts**.

STS is part of the Division of Information Technology (DoIT) - Academic Technology at UW-Madison. For more information regarding DoIT Academic Technology, visit **at.doit.wisc.edu**.

© University of Wisconsin Board of Regents.

This manual and any accompanying files were developed for use by current students at the University of Wisconsin-Madison. The names of software products referred to in these materials are claimed as trademarks of their respective companies or trademark holder.

If you are not a current member of the UW-Madison community and would like to use STS materials for self-study or to teach others, please contact sts@doit.wisc.edu. Thank you.

Topics Outline

1. Introduction
2. Simple Data Types
3. Sequence Data Types
4. Creating Runnable Scripts
5. Functions
6. Standard Library

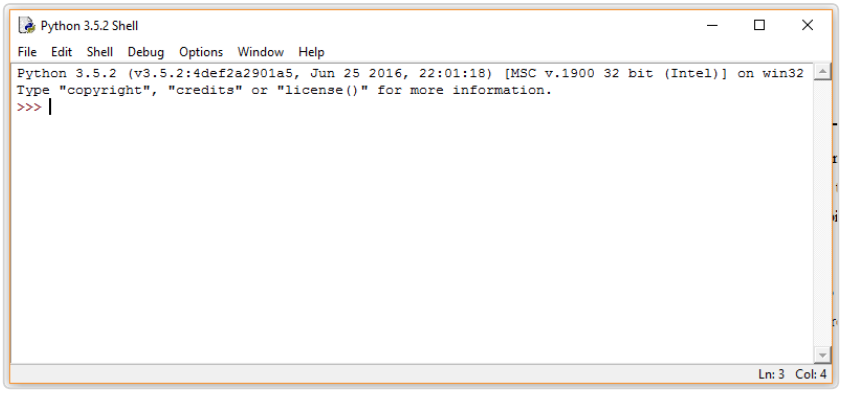
Introduction

Welcome to Python! This course has been designed to give an introduction to the Python programming language. Python is a dynamic programming language that is used in a variety of application domains. Python is based on object-oriented programming, similar to the likes of C++, Java, C#, Perl, Ruby, etc. This manual will include explanatory text along with code examples that you should be typing to follow along. All code is placed within snippets applicable to the language being used.

This manual is based on Python 3.5, and is quite different from Python 2.7. The code snippets and exercises are only applicable to any 3.x version of Python. Python 2.7 is the legacy version and has not been updated since 2010. Python 3 is the present and future of the language and is a better starting point for beginners, with wider applications.

Running Python

If not already installed, Python can be downloaded from the following website: **https://www.python.org/downloads/**. After a straightforward installation process, find and run the **IDLE (Python 3.5)** application. This should open up the Python interpreter shell which allows the user to enter code line by line for testing or trial and error. The interpreter provides many options including being able to open a new script window where complete Python programs can be written in a built in Integrated Development Environment (IDE).

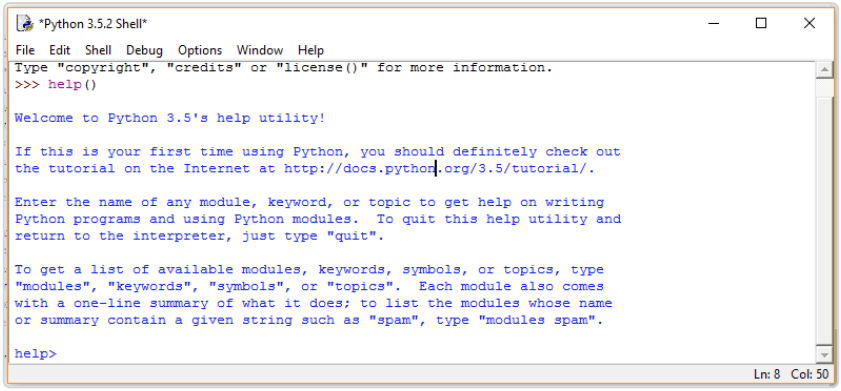


Macs have Python built-in and can be found via the Terminal. Search for and open the Terminal via the applications menu or Spotlight search. In the terminal, type in `python`. Note that this will be version 2.7. It is recommended that you use version 3.4 and above for this course. Please note that, on the DoIT computers, only the Windows partition has Python 3 installed.

Getting help

Python has a very detailed help menu that can be entered through the Python interpreter. Once you have the Python Interpreter opened, type in the following:

```
#helpMenu
>>> help()
```



This will enter you into the Python help utility. You can enter any module, keyword or topic into this utility to get more information. For example, to get more information about strings you would type `str` inside of the help module. This will bring up the help menu for strings specifically. To return to the interpreter, use `quit`.

```
#helpMenuModule
>>> help()
help> str       #help on strings...

help> quit      #quit help menu
```

Another approach to get help on certain modules in Python without the help menu is to insert the module names as an argument when in the interpreter. We do not need to quit from this as it is displayed in the interpreter.

```
>>> help(str)   #help on strings...
```

If the built-in resources do not suffice, additional help can be found over at Lynda.com in the form of video tutorials. The Python website also offers support over at **https://www.python.org/about/help/**

Simple Data Types

As in other programming languages, variables are symbols or names that stand for a specific value. Variables can be many different types in Python: numbers, strings, lists, tuples and dictionaries.

There are some naming conventions that need to be followed when naming a Python variable. Variable names cannot begin with a number or contain punctuation characters such as @, $, and %, with the exception of the underscore character. Python keywords cannot be used as variable names either. A list of such keywords is provided in the code snippet below.

```
#some incorrect variable syntax
>>> 313ph4nt = 349              #variable name begins with number
>>> s.t.s = 420                 #contains special characters except underscore
>>> if = 'Weaponizing Didgeridoos by 2024'      #"if" is Python keyword

#a list of some keywords that cannot be used as variable names
        and       del       from      not       while
        as        elif      global    or        with
        assert    else      pass      yield     is
        break     except    import    print     exec
        in        raise     try       continue  finally
        return    for       lambda    class     def
```

Python is a case sensitive language, so two variables named Buckybadger and buckybadger would be considered different variables. Most programmers follow the **camelCase** naming convention where the first letter of each word in the name is capitalized except for the first word. Variables may contain numbers but must start with either a letter or an underscore. Below is an example that explains the case sensitivity of Python.

```
#caseSensitivity
>>> buckyBadger = 5
>>> buckybadger = 10
>>> buckybadger + buckyBadger
15
```

Variables follow this structure: **variableName = *value***. The text entered after the equals sign depends on the type of variable you are using. To demonstrate, we are going to start by creating some string variables.

Strings

Strings in Python are unique because they are immutable. This means that the string cannot be modified directly. Many mutable data structures provide methods to allow you to dynamically change values of a given variable or data structure. With immutable data types, the developer must create a new instance of the data type (in this case a string) and assign the variable to the new data structure. This may sound complicated, but in practice it is very easy to create new strings by combining other strings. Simply remember that string methods will not directly modify the contents of a variable.

Strings are created by placing text in between either single or double quotes. We will now use the `print` keyword to display text. Try out the snippets below.

```
#displayText
>>> print("Hello World!")
Hello World!
```

Congratulations! You have displayed your first string in Python. Strings in Python are easily combined and these combinations can be modified using the comma (`,`) and plus (`+`) operators. Type the following in the Python interpreter and press return after each:

```
#stringCombine
>>> print("Hello","World!")
Hello World!
>>> print("Hello"+"World!")
HelloWorld!
```

The comma operator will put a space between the strings and the plus operator will not include a space when the strings are combined. Text can also be assigned to variables which makes manipulation of the string easier.

```
#stringVariable
>>> s = "Hello World!"  #assign text to variable named s
>>> print(s)            #display variable s
Hello World!
```

String Methods

Python has a series of built-in methods that allow users to return manipulated string values. There are many more methods built into Python that are used on other data types. A method is considered a subroutine or a procedure that is associated with a given type and performs its function when called. In many other languages these are referred to as member functions or simply functions.

You have already defined the string **s** which we will be applying the string methods to. Perform the snippets below.

```
#stringMethods1
>>> print(s.capitalize())
Hello world!
>>> print(s.upper())
HELLO WORLD!
>>> print(s)
Hello World!
```

Since the string **s** is immutable, that means the string methods are not making changes to **s**. They are simply returning modified values of the string. Let's say that we want to modify the values of **s** so that all the letters in the string are uppercase. The snippet below does this.

```
#stringMethods2
>>> s = s.upper()
>>> print(s)
HELLO WORLD!
```

Being able to search for information or data through string values can be a very useful procedure in Python and other programming languages. The `str.find(value)` method will return a positive value if the string contains the value being searched for; otherwise the method will return -1. The positive number returned if the value is found is the index number at which the first character was found, which is explained in the next section. The value or argument within the `find` method is case sensitive. Test it out below.

```
 #stringFind
>>> s.find("ell")
-1
>>> s.find("ELL")
1
```

Another useful method is replace. This method will take two string values as inputs. It will find the first value, if it exists, and return the string with the second value in place of the first value. The replace method is also case sensitive like the find method. If the first value does not exist in the string, the replace method will just return the original string. Type out the following code in the Python interpreter to demonstrate.

```
#stringReplace
>>> print(s.replace("world","EARTH"))
'HELLO WORLD!'
>>> print(s.replace("WORLD","EARTH"))
'HELLO EARTH!'
```

The first line of code just returns the original value of the string because the replace method is case sensitive. The returned values are not permanent; the string **s** would need to be redefined as done previously to retain any of the changes from the replace method.

String Indices

Each character in a string is identified with an index that denotes its position within the string. The first character is assigned an index of 0. The image below helps explain this better in the case of the string **s**. H, the first character, has index 0. Blank spaces are assigned indices too.

You can return partial values of a string by searching through its indices by using `str[i: j]` where i and j are indices. The j index is non-inclusive meaning that the returned value will be up until the value of j. Try out the snippet below.

```
#stringIndices
>>> print(s[:])
'HELLO WORLD!'
>>> print(s[3:])
'LO WORLD!'
>>> print(s[:8])
'HELLO WO'
>>> print(s[3:8])
'LO WO'
```

In the first line of code, by omitting index values for this syntax, the returned value will start with the 0th index and end with the last index of the string. Similarly, the second line of code begins with the 3rd index ending with the last index of the string. The third line begins with the 0th index and ends with the 7th index. This is because the j index is non-inclusive. The final line of code returns a value that starts with the 3rd index and ends with the 7th index.

The principles of string indices carry over to other data types like lists, dictionaries, and tuples which will be discussed later in this course.

Format String Method

The format method allows for string manipulation. It can insert specific values into placeholders in a string that are indicated by `{value}`. The value inside of the curly brackets can be blank, ordered numbers or names of specific strings. Let's practice this to get a better understanding of how this method works.

```
#stringFormat1
>>> print("{} love the {}.".format("I","python"))
'I love the python'
```

The format method places the strings 'I' and 'python' sequentially into the string. If we were to switch the order of 'I' and 'python' inside of the `format` method, the string would read 'python love the I'. This can be accomplished by using index values inside of the `{ }` brackets. The first value in the format method is indexed 0, the second value is indexed 1, and so on.

```
#stringFormat2
>>> print("{0} love the {1}.".format("I","python"))
'I love the python.'
>>> print("{1} love the {0}.".format("python","I"))
'I love the python.'
```

The returned results from both methods are the same. This is because we switched the index values in both sets of brackets as well as in the format method. You can include variable or string names within the format method for additional flexibility.

```
#stringFormat3
>>> print("{name} loves the python and so do {0}.".format("you", name="Mr. Miers"))
'Mr. Miers loves the python and so do you.'
```

The string 'you' is in the 0 index of the format, therefore it displays inside of the {0} brackets in our string. Any indexed string values inside of the format method must come before any strings that have a temporary variable name given to them. This is the reason that `name="Mr. Miers"` needs to come after `"you"` inside of `format("you",name="Mr. Miers")`.

Numbers and Simple Math

Working with numbers and basic math functions is very similar to other programming languages. Python also has a built-in math library which makes calling math functions very efficient. Before we dive into working with numbers in Python, we should define the different numeric types available:

- **int**: Plain integers are whole numbers with precision limited up to 32 bits.
- **long**: Long integers are whole numbers with unlimited precision.
- **float**: Floating point numbers contain decimal places with precision dependent on the system.
- **complex**: Complex numbers have both a real and imaginary part.

As mentioned earlier in this class, Python variables are typeless so it is easy to change from one number type to another but the output of math functions are dependent on the inputs' data types. Python follows normal mathematical operations. We're going to be using print to display some outputs of using numbers. Type out the following four lines of code to get some outputs:

```
#mathPrint1
>>> print(42)
42
>>> print(40 + 2)
42
>>> print(42 * 2)
84
>>> print(42 / 5)
8.4
```

In Python 2, the last line `print(42/5)` would have given an answer of 8 by rounding off 8.4 to a whole integer. This does not occur in Python 3 and a float is displayed instead. Numbers can be assigned to variables and then be manipulated as follows.

```
#mathPrint2
>>> x = 2
>>> y = 5
>>> sum = x + y
>>> print(sum)
7
>>> print(x+y)
7
```

Python has a few other mathematical operations built-in that can raise a number to a power or find its absolute value.

```
#mathPrint3
>>> print(abs(-4))      #find the absolute value of -4
4
>>> print(pow(2,5))     #raise 2 to the fifth power
32
>>> print(2**5)         #another method to find 2^5
32
```

Math Module Functions

Python has access to a large library (or module) that contains many complex math functions. These functions are defined in the C standard and require real numbers as inputs. There are separate function calls for inputs that use complex numbers.

The math module needs to be imported into Python before the functions can be called. The math module can be imported by typing `import math` into the Python interpreter. This will require `math.` to be appended to the front of the function call. By adding `from math import *` to the interpreter, the entire math library can be called without adding `math.` to the beginning of a function call. This is seen below.

```
#mathModule1
>>> import math           #calling math module
>>> print(math.sqrt(4))   #sqrt function with prefix
2.0
>>> print(math.pi)        #pi function with prefix
3.141592653589793
>>> from math import *    #calling entire library
>>> print(sqrt(4))        #sqrt function without prefix
2.0
>>> print(pi)             #pi function without prefix
3.141592653589793
>>> print(e)              #Euler's number function without prefix
2.718281828459045
```

The fourth line of code tells Python to import all functions from the math module into the interpreter. The ***** is a wildcard and represents all functions. Specific functions could be imported by replacing the ***** with the function name. The advantage of importing math functions is so that you do not need to prepend math. to the front of the function call.

Next, the ceiling function rounds off a number to the next whole number, whereas the floor function rounds down. Let's try them out.

```
#mathModule2
>>> print(ceil(4.2))    #round-up 4.2
5
>>> print(floor(4.2))   #round-down 4.2
4
```

You can also work with trigonometric functions in Python. These functions accept angles in units of radians by default and angle conversion methods can be used to provide angles in degrees.

```
#mathModule3
>>> print(sin(pi/2))
1.0
>>> print(cos(0))
1.0
>>> print(degrees(pi))
180.0
>>> print(radians(180))
3.141592653589793
>>> print(sin(radians(90)))
1.0
```

Sequence Data Types

Python, like many programming languages, has additional data types that are more suitable for sequences of information. In Python, these data types are defined as lists, tuples, and dictionaries.

Lists

The List is one of the most flexible data types in Python. A list in Python is a container that holds a number of other objects in a given order. The unique aspect of lists in Python is that the data types within the list can vary from element to element. In other programming languages, such as Java, list elements all need to be the same data type. The additional flexibility makes Python a very powerful and dynamic language.

```
 #lists
>>> items = [1, 2.0, "three", 'four']
>>> print(items)
[1, 2.0, 'three', 'four']
```

Note that our list includes both an integer and a float, as well as strings that were defined with `" "` and `' '`. We can see that there is a lot of flexibility that comes along with lists. You can even nest lists inside of lists to create 2D and 3D arrays, but that is outside the scope of this course.

List Indices

Just like strings, the elements of lists are identified using indices. An element of a list can be manipulated using the index `i`, and a range of elements using the `i:j` notation. Just like strings, the `j` index is non-inclusive. This is shown below.

```
#listsIndex1
>>> print(items)
[1, 2.0, 'three', 'four']
>>> print(items[0])
1
>>> print(items[2])
three
>>> print(items[1:3])
[2.0, 'three']
>>> print(items[0:3])
[1, 2.0, 'three']
```

The list items contains 4 elements and is therefore indexed to 3. Remember, the j index is not included in the return value of `items[i:j]`. It goes up until index `j`. In our case, if we wanted to include the last element in the list we would leave the `j` index blank to go all the way to the end of the list, or change the range to `i:j+1`. This same syntax applies for the beginning of the list. You can leave the `i` index blank instead of including the 0^{th} index to return values beginning from the start of the list.

```
#listsIndex2
>>> print(items[0:4])
[1, 2.0, 'three', 'four']
>>> print(items[:])
[1, 2.0, 'three', 'four']
```

List Methods

Lists can be modified by using the append and insert methods. The `append` method will add an element to the end of the list, which will have the last index. The `insert` method will insert an element into a given index and then push the remaining list elements over to the right.

```
 #listsMethods1
>>> items.append(6)
>>> print(items)
[1, 2.0, 'three', 'four', 6]
>>> items.insert(4, 5.0)
>>> print(items)
[1, 2.0, 'three', 'four', 5.0, 6]
```

Lists can also be used like a stack by using the `append` method to push items on to the list and the `pop` method to remove items. You can leave the argument for the pop function blank and it will remove the item at the last index of the list. Otherwise, you can define an index for the element to be removed. The function returns the value that is removed from the list, which could then be stored as a variable if desired. Type the following lines of code to see how to use the `pop` method.

```
#listsMethods2
>>> items.pop()
6
>>> print(items)
[1, 2.0, 'three', 'four', 5.0]
>>> items.pop(2)
'three'
>>> print(items)
[1, 2.0, 'four', 5.0]
```

The `pop` method returns the value that is removed from the list. This returned value can be set to a variable, which applies to any method that returns a value. To demonstrate, we are going to create an integer variable that will be assigned the value of a "popped" listed element.

```
#listsMethods3
>>> z = items.pop()
>>> print(z)
5.0
>>> print(items)
[1, 2.0, 'four']
>>> items.append(z)
>>> print(items)
[1, 2.0, 'four', 5.0]
```

Three additional methods that are useful are the `max`, `min`, and `len` (length) methods. The `len` method will return the number of elements in the list. Remember, the last indexed element in the list will be 1 less than the length of the list. The `max` method will return the highest value within the list. Similarly, the `min` method will return the lowest valued element in the list. The max and min functions will not work for the list named items as it contains a mix of strings and numbers. Let's pop the string and add another number to items.

```
 #listsMethods4
>>> items.pop(2)
'four'
>>> items.append(3)
>>> print(items)
[1, 2.0, 5.0, 3]
>>> print(len(items))
4
>>> print(max(items))
5.0
>>> print(min(items))
1
```

Tuples

Another sequence data type in Python are tuples. Tuples are similar to lists in the fact that they can hold multiple data types; however, there is one major difference. Tuples are immutable which means that they cannot be modified. This is drastically different than lists, which have multiple methods to modify elements.

Tuples are defined using open brackets `( )` instead of square brackets `[ ]` which are used for lists. Tuples are essentially frozen lists; they are immutable and cannot be modified. Tuples are usually used when it is known that the values will not change throughout a program or a script. Tuples are faster to process than lists, which makes them preferred in large programs if they do not need to be modified.

```
 #tuples1
>>> itemsFrozen = (1, 2.0, 'three', 4, "five")
>>> print(itemsFrozen)
(1, 2.0, 'three', 4, 'five')
```

While there isn't a way to modify this tuple, you can still return indexes just like lists. Returning indexes for tuples uses the same syntax for returning list indexes.

```
#tuples2
>>> print(itemsFrozen[1:4])
(2.0, 'three', 4)
>>> print(itemsFrozen[1:])
(2.0, 'three', 4, 'five')
>>> print(itemsFrozen[:3])
(1, 2.0, 'three')
>>> print(itemsFrozen[:])
(1, 2.0, 'three', 4, 'five')
```

Dictionaries

A Python dictionary is a container of key-value pairs. The dictionary will store a value with an associated key, and there can be many key-value pairs for a given dictionary. Dictionaries differ from lists or tuples because the keys will be called to return a value rather than an index.

Creating/Modifying Dictionaries

Dictionaries are initialized by using curly brackets `{ }` with a `key:value` syntax. The keys for dictionaries are strings, so either single or double quotes must be included around the key. Quotes must also be included on string values, but not for integers or floats. Let's create a sample dictionary to store information about a student.

```
#dict1
>>> nick = {'name':'nick', 'major':'Computer Science', 'age':20}
>>> print(nick)
{'age': 20, 'name': 'nick', 'major': 'Computer Science'}
>>> print(nick['age'])
20
```

Dictionaries can be updated and key-value pairs can be added as well. The value of a key can be updated by calling the key and resetting the value. This same syntax is used to create a new key-value pair to be added to the dictionary.

```
#dict2
>>> nick['age'] = 21
>>> print(nick['age'])
21
>>> nick['school'] = 'UW-Madison'
>>> print(nick['school'])
UW-Madison
>>> print(nick)
{'age': 21, 'school': 'UW-Madison', 'name': 'nick', 'major': 'Computer Science'}
```

Removing key-value pairs from a dictionary is not the same as removing elements from a list. Dictionary key-value pairs require use of the `del` (delete) method.

```
#dict3
>>> del(nick['school'])
>>> print(nick)
{'age': 21, 'name': 'nick', 'major': 'Computer Science'}
```

Dictionary Methods

There are some useful methods that will return dictionary information. The first method is the `in` method, with syntax `'key_name' in dictionary_name`. This method will return a true or false value depending on if the dictionary has the indicated key. The `keys` method returns a list of the dictionary's keys. Similarly, the `values` method returns a list of the dictionary's values.

```
#dict4
>>> 'name' in nick
True
>>> 'school' in nick
False
>>> print(nick.keys())
dict_keys(['age', 'name', 'major'])
>>> print(nick.values())
dict_values([21, 'nick', 'Computer Science'])
```

Checking and Casting Data Types

Checking and casting data types in Python can be done using some built in methods. In this example, the `type` method will be used to return an input type. The type method will return the variable or parameter input type. To practice, we will use some integers and previous variables that were defined inside of the Python interpreter.

```
#checkingData
>>> print(type(4))
<class 'int'>
>>> print(type(4.0))
<class 'float'>
>>> print(type(nick))
<class 'dict'>
>>> print(type(items))
<class 'list'>
>>> print(type(itemsFrozen))
<class 'tuple'>
```

There are instances that you could come across while writing programs that would require you to change one data type to another. For example, from an integer to a float or a float to a string. This can be done using specific methods. Some example ones include int, float, and str. It should be noted that there are some data type conversion limits. For example, you cannot cast a string into an integer but you can cast an integer into a string.

```
#castingData
>>> int(3.1416) #cast value as an integer
3
>>> float(5)    #cast value as a floating point number
5.0
>>> str(4)      #cast value as string
'4'
```

Creating Runnable Scripts

You may have noticed that the current programming environment we are working in only allows for single Python statements to be written an executed. With a script, multiple statements can be written and then ran at once. In order to write these scripts, you will want to download a *plain text editor*.

Plain Text Editors

There are various plain text editors to choose from. The plain text editors listed here are available on either MacOS, Windows, or both. Find which one you prefer, and use it. We recommend using TextWrangler if you are on a Mac, or Notepad++ if you are on a Windows machine.

Popular plain text editors:

- TextWrangler
- Notepad++
- Brackets
- Atom
- Sublime Text

Creating and Running a Script

Scripts are created by writing and saving Python code in a file with the extension .py. We will create a simple script that prints out a grocery list. Type the following into your plain text editor, and save the file as *script.py* to the Desktop.

```
groceryList = ["Apples", "Bananas", "Milk", "Bread"]
print("Your grocery items are listed below:")
print(groceryList[0])
print(groceryList[1])
print(groceryList[2])
print(groceryList[3])
```

One way to run our script is to use a command line interface like Terminal on MacOS, or cmd on Windows. However, before we run our script we must navigate to our Desktop using the command line interface. Open your operating system's command line interface by searching using Spotlight (MacOS), or the Start Menu (Windows). Whether you are on MacOS using terminal or Windows using cmd, type the following into the command line prompt and hit enter to navigate to your Desktop.

```
cd Desktop
```

Now that we have navigated to our Desktop, we can run our grocery item script. Assuming you have saved the script to the Desktop correctly, execute the Python script using the command line interface by typing the following command.

```
python script.py
```

The script should output the following onto your command line interface window.

```
Your grocery items are listed below:
Apples
Bananas
Milk
Bread
```

For the remainder of the class, we will use this method of creating scripts in a plain text editor and running scripts on a command line interface in order to learn about functions, and the standard library.

You may alternatively just type `python` into the command line interface with no file name after to open up the command line python interpreter.

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reuse. As you know, Python has a number of built-in functions like `print()`, but you can also create your own functions which are called user-defined functions.

Python has a large number of built-in functions. You already have experience calling these functions such as `print(`*`arg`*`)`, `max(`*`arg`*`)`, and `len(`*`arg`*`)`. The text *arg* is a placeholder for the function argument. An argument is a parameter that is passed to the function and is referred to a piece of data while the function executes.

You have also looked at some functions that did not require any arguments, such as `.pop()` when the last item of a list was popped off. Functions can be defined to include arguments or not to include arguments, it just depends on how the functions are being used. For our example, we will be using two functions that include arguments.

User-defined Functions

Before we create a new function, there are rules that need to be followed for defining functions.

- Function blocks begin with the keyword `def` followed the the function name and parentheses. It should look like: `def functionName()`.
- Any input parameters should be placed within these parentheses and separated by commas. You can also define parameters inside of these parentheses.
- The code block for every function starts with a colon and is indented.
- The statement `return` *`expression`* exits a function, optionally passing back a value to the caller. A return statement with no arguments is the same as `return None`.

In your *script.py* file, begin typing this function:

```
# prints both arg1 and arg2 on separate lines
def printBoth(arg1, arg2):
        print(arg1)
        print(arg2)
```

In order to execute your function, you may call it using the following code. You will want to add this code to your script.

```
# call the function previously defined
printBoth("Hello","World")
```

You have now created and ran your first user-defined function in Python. The function `printBoth()` accepts two arguments: `arg1` and `arg2`, which can be any data type, and prints them. The return statement has been omitted from our function because there are no values that we need to return.

Functions with Default Arguments

Function arguments must be included when you call them in the interpreter. If you forgot to include the second argument in the `printBoth` function you would receive the following error:

```
Traceback (most recent call last):
  File "script.py", line 5, in <module>
    printBoth("Hello")
TypeError: printBoth() missing 1 required positional argument: 'arg2'
```

This can be avoided by including default arguments in the function definition. This allows you to include less parameters in the function call and the default parameter will be used. Additionally, you can overwrite the default parameter as you would normally call the function.

Switch back to script.py in your text editor and add the next code snippet below your previous function.

```
# printBoth with a default argument
def printBoth2(arg1, arg2="default"):
        print(arg1)
        print(arg2)
```

You will want to add the code to call the function as well.

```
# call the function, with arg2 taking its default argument
printBoth2("Hello")
```

After you type in the above code, save the file. It should look like the image below.

```
def printBoth(arg1, arg2):
    print(arg1)
    print(arg2)

def printBoth2(arg1, arg2="default"):
    print(arg1)
    print(arg2)

printBoth("Hello","World")
printBoth2("Hello")
```

You can see that default arguments can be overwritten if needed. Feel free to test this out by adding a second argument to the `printBoth2` function call. Default arguments help shorten function calls in some cases. This can be very useful with more complex functions.

Standard Library

Python contains many built-in and importable modules through its standard library. These modules contain functions that can be used while writing Python scripts and other modules. We have already used the Math module earlier in the class and will now take a look at the Random and Tkinter module.

The Random Module

Random numbers are used frequently in computer programming. Python generates random numbers using the random module from its standard library. Nearly all functions in the random module depend on the basic `random()` function. This function generates a random float uniformly in the semi-open range `[0.0, 1.0)`. Python uses an extensively tested random number generator that has a wide range of uses in programming. It is a deterministic function so not all applications are suitable, but those are outside the scope of this course. Let's dive into the `random()` function by typing the following code into the Python interpreter. Remember, opening Terminal or cmd, typing `python`, and hitting enter can get you to this interface.

```
# importing the random module, and generating a random number
>>> import random
>>> random.random()
0.8210122987308486
```

This function produces a random number from 0.0 up to but not including 1.0. This has a number of uses in programming. Another useful function is `uniform(a,b)` which produces a random number from a to b. The end value b may or may not be included based on rounding of the equation `a + (b-a) * random()`.

```
# importing the random module, and generating a random number in a certain range
>>> random.uniform(5,50)
8.290954398059158
>>> random.uniform(0,10)
1.960216421390445
>>> random.uniform(-5,-1)
-1.3961170315343971
```

The random module has a number of other functions that are further detailed in Python's online documentation.

The TkInter Module

Tkinter is a GUI (Graphical User Interface) package which exists inside the Python Standard Library, with a collection of widgets that can be combined in a variety of ways to achieve a specific GUI. Up until now all of our programs interact with the user using a text interface, but most of the applications you use on a daily basis have a GUI (Word, Firefox, even the Python IDE and interpreter).

For our purposes we'll build a super simple GUI in five lines to demonstrate the power and ease of use of building a GUI in Python. The Tkinter package is part of Python Standard Library, so all we have to do is import the package into our workspace. We will use the wildcard notation to import all member functions into our namespace like we did earlier with the Math function. We'll also include the webbrowser package to add some functionality to our GUI. Follow the snippets below to build your script. You will want to create and save a new file to your Desktop for this script, called *GUI.py*.

```
from tkinter import *
from webbrowser import *
root = Tk()
```

The third line of code creates a "parent" or top-level" widget which will act as a container and reference point for all other GUI widgets.

Now we can create button widget. Note that the first argument passed to the widget is a reference to the parent widget we created. The string passed as a text argument will appear on the face of the button. After that, we need to call a special member function of the button to "pack" the button into the application frame. Behind the scenes Tkinter includes a geometry manager which is being called to arrange the specific widget. The `pack()` function allows you to pass layout options to the geometry manager.

```
simpleButton = Button(root, text="Don't click me")
simpleButton.pack()
```

Let's add one more button. We'll create the button in the same way as last time, but let's pass an argument to the pack function to place the second button below the first. This button will also have some added functionality using the command argument, but first we have to define a function to bind the button to.

```
def openBrowser():
        open_new('http://www.wisc.edu/sts')

simpleButton2 = Button(root, text="Click me", command=openBrowser)
simpleButton2.pack(side = BOTTOM)
```

You've probably noticed that none of the GUI widgets we have created are actually visible yet. In order to spawn the window we must initiate the application loop. All GUI's use a constantly running application loop that waits for user input or updates from behind the scenes and immediately updates the GUI accordingly.

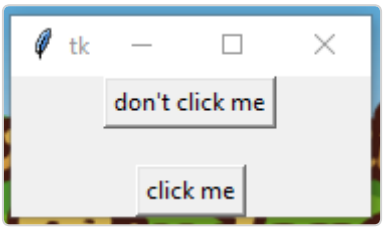
```
root.mainloop()
```

Your script should look like this in the end:

```
from tkinter import *
from webbrowser import *
root = Tk()
simpleButton = Button(root, text="don't click me")
simpleButton.pack()
def openBrowser():
        open_new('http://www.wisc.edu/sts')


simpleButton2 = Button(root, text="click me", command=openBrowser)
simpleButton2.pack(side = BOTTOM)
root.mainloop()
```

Once your script is ready, save it as *gui.py*, if you haven't already done so, and run the script. A small window should pop-up, where the "Don't click me" button should have no effect while the "Click me" button should open up the STS website in your web browser.



With that, we conclude this introductory course to programming in Python. You should now be able to read and understand basic Python code, with the ability to write scripts and functions. Do check out the Python help page at **https://www.python.org/about/help/** and the Lynda.com page at **https://www.lynda.com/Python-training-tutorials/415-0.html** for additional resources. A quick Google search should help with any trouble you encounter. STS has free consultation services that can help a lot as well.

Feel free to also take our Python 2 class to further hone your skills, and learn about advanced Python topics.