1. Getting Around

1. In a terminal window, list the files in the root directory. The root directory is referred to with a single slash (`/`).

2. Create a new folder below your home directory using the `mkdir` command.

3. Remove the folder using the `rmdir` command. Note that the `rmdir` will fail for directories that aren't empty. To remove a directory that isn't empty, use the `rm -rf` command instead.

4. If you are using OS X, open a Finder window showing your home directory using the `open` command.

   If you are using Windows, open an Explorer window showing your home directory using the `explorer` command.

Getting Help on OS X

OS X comes with an extensive set of documentation "manuals" known as *man pages*. You use the `man` command to access the man pages. You can learn about the multitude of options available for various commands in the terminal by typing `man` followed by the name of the command you want to learn about. For example, you'd type `man ls` (and press return) to access the documentation for the `ls` command. The `man` command presents documentation one screen at a time, in the same way that output is presented using the `more` command. You can press the spacebar to view the next page and `q` to exit the `man` command.

If you'd like to view the documentation in a more visually pleasing style in the Preview application, you can use one of the options for the `man` command in conjunction with what is known as a pipe. A pipe lets you take the output of one command and use it as the input for another command. To view the help for the `ls` command in this manner, you would type the following (the vertical `|` is referred to as the pipe character.)

```
man -t ls | open -f -a /Applications/Preview.app
```

It is more common (and faster) to just view the man pages in the terminal.

Getting Help in Git BASH on Windows

Many, but not all, commands that come with the Git BASH environment have built-in help that can be accessed via the `--help` option. For example, to get help on the `ls` command, you could type `ls --help` and press return. If you want to view the help documentation one screenful at a time, you can use what is known as a pipe. A pipe lets you take the output of one command and use it as the input for another command. So, to view the help for `ls` one screen at a time you could use the following (the vertical `|` is referred to as the pipe character.)

```
ls --help | less
```

When viewing the help content in this manner, you can press the spacebar to view the next page and `q` to exit the command.

2. Hello World

1. Install Python 3 on your system using the procedure demonstrated in the **Hello World** video.

2. In a terminal window, launch a Python 3 session and use the `print` statement to display the words `"hello world"`

3. Use a single `print` statement to print the values `1`, `2`, and `3` on separate lines. The output should appear like this

   ```
   1
   2
   3
   ```

   Hints:

   Consult the help documentation for `print` in a Python session or using the separate help content installed with Python. Specifically look for help related to the `sep` argument.

   The newline character is specified as `'\n'`

3. Interactive Use and Saving Your Work

1. Launch Python in a terminal session and use it to perform some simple calculations (addition, multiplication)

2. Define a function that takes no arguments and prints the word `"demo"` when the function is called.

3. The `print` function can take multiple arguments separated by commas. When given multiple arguments, the `print` function will print all arguments separated by spaces. For example, if you were to execute `print("hello", "world")`, the result would be the following

   ```
   hello world
   ```

   Define a function called `hello` that has a single parameter called `name`. The function should take advantage of the `print` command's support for multiple arguments to print the word `hello` followed by a space followed by the `name` passed into the function. For example, if you were to invoke `hello("george")` the result would be

   ```
   hello george
   ```

   Note: This is the same behavior demonstrated in the video but in the sample function shown there, the print function was called as `print("hello " + name)`. It is often the case that more than one approach is available to achieve a particular result.

4. Install the text editor for your platform (TextWrangler on OS X, Programmer's Notepad on Windows) per the instructions shown in the video

5. Using the text editor installed in step 4, create a new file called `demo.py` in your Desktop folder. Enter the following code into `demo.py` and save the file.

   ```
   from datetime import date
   print(date.today())
   ```

   In a terminal window, navigate to your Desktop folder and execute the following command:

   On Windows:

   ```
   python demo.py
   ```

   On OS X (see note below):

   ```
   python3 demo.py
   ```

This is a common way of running Python programs in a Bash shell (or similar) environment.

Another way that is a perhaps a bit more convenient for programs that are run frequently, is to use a special comment on the first line of a Python file that looks like this:

On Windows:

```
#!/usr/bin/env python
```

On OS X (see note below):

```
#!/usr/bin/env python3
```

Once this comment has been added, you then give the script "execute" permission by typing the following command

```
chmod +x demo.py
```

You can then run the script in a Bash shell as follows:

```
./demo.py
```

The above command assumes that you are running the script in the same directory as the script itself. You would use the full path to a script to run it from another directory. For instance, if the demo.py script is stored in the Desktop folder of the user programmer, you could invoke the script as follows

```
~programmer/Desktop/demo.py
```

NOTE: The `demo.py` program will run using Python 2 or Python 3, but you should get in the habit of using python3 on OS X

4. Data Types, Operators, and Variables

1. Launch Python in a terminal session and use it to explore integer, float, and string data types. Exercise some of the operations available: addition, subtraction, multiplication, division, floor division, modulus, exponentiation and absolute value.

2. Define a function called `square` that takes a single numeric argument and returns the value of the argument squared. So, for example, if you were to call `square(10)`, the function would return the value `100`. Use the `**` operator to compute the result that is returned.

3. Write a function called `uppify` (see note below) that has two parameters named `message`, and `letter`. The purpose of `uppify` is to return a string that consists of the `message` passed in where every occurrence of `letter` in the `message` is replaced by the upper case version of `letter`. Here's an example of what using `uppify` could look like

   ```
   >>> uppify('hello world', 'o')
   'hellO wOrld'
   ```

   Hint: You'll probably want to use the `upper()` string method as part of your solution.

   Another hint: Consult the documentation on String Methods under Text Sequence Types in the Library Reference to see what method can be used to replace values in a string.

NOTE: Uppify isn't a real word. I just made it up. Computer programmers can be like that.

5. Data Collections

1. In a Python session, define a string variable called `letters`, a list variable called `letters_list`, and a tuple variable called `letters_tuple` as follows

   ```
   letters = 'abcdefghijklmnopqrstuvwxyz'
   letters_list = list(letters)
   letters_tuple = tuple(letters)
   ```

   Be sure to perform exercises 2 though 6 in the same Python session.

2. Examine the contents of `letters_list` and `letters_tuple` by typing those variable names at the Python prompt and pressing return after each name. Note that the list and the tuple contain all of the characters in the `letters` string.

3. Examine the element at index 8 of `letters`, `letters_list`, and `letters_tuple`.

4. Examine the result of invoking `.index('i')` on `letters`, `letters_list`, and `letters_tuple`

5. Examine the `[13:15]` slice of each of the variables defined. Note that the slices of the list and the tuple contain the same letters as the slice of the string.

6. An optional third argument can be specified when creating a slice from a sequence type. The third argument specifies the step value for the slice. The step value can be positive or negative. This is sometimes referred to as extended slicing. Examine the result of the following extended slices

   ```
   letters[1::2]
   letters_list[1::2]
   letters_tuple[1::2]
   ```

   Examine slices [:-2:2] and [::3] for `letters`, `letters_list`, and `letters_tuple`.

7. Examine the length of `letters`, `letters_list`, and `letters_tuple` using the `len()` function.

8. Define a function called `reverse` that takes a single parameter called `a_string`. The function should return a string containing all of the letters in `a_string`, but in reverse order. Example usage

   ```
   >>> reverse('abc')
   'cba'
   ```

   One approach: In the body of your function, you may find it helpful to create a list of the letters in `a_string` as your first step

   Hint: You can use the `join` method on an empty string to concatenate strings in a list. For example

```
>>> ''.join(['a', 'b', 'c'])
'abc'
```

Another approach: The reverse of a string can be obtained very efficiently using extended slicing.

9. Define a dictionary, `topping_costs`, which maps pizza toppings to cost in dollars as follows (Note that in a real application where financial calculations are performed, floats should not be used to represent monetary values.)

```
topping_costs = {'extra cheese': 1.00, 'sausage': 1.50, 'pepperoni':
0.75}
```

Enter the following lines of code

```
cost = 0
cost = cost + topping_costs['pepperoni']
cost = cost + topping_costs['extra cheese']
cost = cost + topping_costs['sausage']
print(cost)
```

6. Functions

1. Configure the settings for the text editor used on your platform (TextWrangler on OS X, Programmer's Notepad on Widows.) Consult video number 6 for instructions.

   On Windows:

   Create the `edit` command per the instructions in Functions video. Be sure to create the `edit` file in the `bin` directory below your home directory.

2. Create a folder called `Exercises` below your home folder or some other location that you choose. In the `Exercises` folder, create a new file called `circle.py` using the `edit` command in a terminal window.

3. In `circle.py`, define a function called `info` that takes a single parameter, `radius`. The function should print out the diameter, circumference, and area of a circle with the given radius. For example, if `info(10)` is called, the following output should be produced

   ```
   diameter: 20
   circumference: 62.83185307179586
   area: 314.1592653589793
   ```

   Launch a Python interactive session in the same directory as the `circle.py` file. Import `circle` into the Python session and call the `info` function with various radius arguments. Remember that you can use the `reload` function in the `importlib` module to reload your functions module.

   Hint: You can use the value of `pi` located in the `math` module.

4. Introduce three more functions in `circle.py` called `diameter`, `circumference`, and `area`. Each function should take a single parameter called `radius` and return its result to the caller.

5. Rewrite the `info` function so that if makes use of the `diameter`, `circumference`, and `area` functions. Exercise the revised `info` function in an interactive session.

6. The `sys` module provides a mechanism for accessing the command line arguments a user has supplied when running a Python program. The command line arguments are available as a list in the `sys` module's `argv` property. For example, if you were to execute a Python program as

   ```
   python circle.py one two three
   ```

   then `sys.argv` would be populated as

   ```
   ['circle.py', 'one', 'two', 'three']
   ```

Note that the first element of `sys.argv` is the name of the script being executed. The remaining elements are the arguments supplied after the name of the script. The contents of `sys.argv` would be populated in the same manner if the script is executed via the `#!` mechanism described in exercise 3.5.

Launch an interactive Python session using the command line arguments "`- one two three`". Import the `sys` module and then print the contents of `sys.argv`.

7. Add the `#!` python line to the top of the `circle.py` file and make it executable per the instructions in exercise 3.5.

8. Add in import statement for the `sys` module near the top of `circle.py`. At the bottom of `circle.py`, add code to invoke the info() function using the argument supplied on the command line as the radius.

   Hint: You'll need to convert the supplied argument to a numeric value before passing it to the info function. One way to do this, is by using `float()`.

   Now, you should be able to run the `circle.py` script as follows (assuming that you are working in the directory that contains `circle.py`)

   ```
   ./circle.py 10
   diameter: 20.0
   circumference: 62.83185307179586
   area: 314.1592653589793
   ```

   Note: If you were to execute `./circle.py abcdef`, the program will fail due to a `ValueError`. Dealing with these sorts of errors is not part of this exercise.

7. Conditional Logic

1. Modify the code in the `circle.py` file from the previous exercises to check for the presence of a radius argument on the command line. If no radius argument is supplied, the program should print out a help message such as

   ```
   Please specify a radius value.
   ```

   If a `radius` value is supplied, the `info` function should be called.

   Exercise the `circle.py` script from the command line with and without supplying an argument for `radius`.

2. The `input` function can be used to gather input from the user. It is used as follows

   ```
   number = input("Enter a number: ")
   ```

   When the above code is executed, the user will be prompted to enter a number. When the user presses the Enter key, the supplied value will be assigned to the variable on the left side of the equals sign. Note that the `input` function always returns a string value, regardless of the name of the variable used in the assignment statement.

   Modify the code in `circle.py` such that the user is prompted to enter a `radius` if they don't supply one on the command line.

   Exercise the `circle.py` script from the command line with and without supplying an argument for `radius`.

3. Write a function called `weather` that has a single parameter called `temperature`. When the `weather` function is called, it should have the following behavior:

   If temperature is less than or equal to 32, print "It's freezing."

   If temperature is less than or equal to 65, print "You might want to wear a jacket."

   If temperature is less than or equal to 85, print "It's not too hot."

   For all other cases, print "Looks like a day for short sleeves."

   Only one `print` statement should be executed for any given value of `temperature`.

   You'll probably want to define this function in a new Python module. Exercise the `weather` function in an interactive Python session.

4. Import the circle module into a Python interactive session. Note that you are prompted to enter a radius when the import is performed. This is not desirable behavior. So, let's fix this.

When a module is loaded by the Python interpreter, a special property called `__name__` is set on the module. For modules that are executed as the top-level scope, such as when they are executed at the command line, the `__name__` property will be set to `'__main__'`. Modify the `circle.py` file such that the logic related to processing of command line arguments and the invocation of `info` is only run if it is running as `'__main__'`.

Import the `circle` module into a fresh interactive session. You should not be prompted to enter a radius when the module is imported.

5. Add a triple-quoted document string to top of the `circle.py` file. Add a triple-quoted document string to each of the functions in `circle.py`.

   Import `circle` into an interactive session and execute the following

   ```
   help(circle)
   ```

   Note how the document strings that you entered are presented in the help content.

8. Looping

1. In the terminal, `cd` to the directory containing the `circle.py` module created in previous exercises. Launch Python and import the `circle` module. Using a `for` loop, invoke `circle.area` for a range of radius values from 1 to 10. Do the same thing using a while loop.

2. Create a new file called `utilities.py`. In `utilities.py`, define the functions described in the table below

Function	Purpose
`sum(sequence)`	Returns the sum of values in a list or tuple. For example, `sum([3, 2, 1])` should return `6`.
`max(sequence)`	Returns the maximum value in a list or tuple. For example, `max([3, 2, 1])` should return `3`.
`min(sequence)`	Returns the minimum value in a list or tuple. For example, `min([3, 2, 1])` should return `1`.
`squares(sequence)`	Returns a list containing each element in the supplied sequence raised to the power of 2. For example, `squares([3, 2, 1])` should return `[9, 4, 1]`.

Import the `utilities` module into a Python session and exercise each of the functions.

3. In an interactive session define a dictionary, `topping_costs`, which maps pizza toppings to cost in dollars as follows (Note that in a real application where financial calculations are performed, floats should not be used to represent monetary values.)

```
topping_costs = {'extra cheese': 1.00, 'sausage': 1.50, 'pepperoni':
0.75}
```

In `utilities.py`, write a function called `total` that accepts a dictionary such as `topping_costs` as an argument and returns a tuple with two elements. The first element in the tuple will be the total cost of the toppings. The second element will be the name of the most expensive topping. Import or reload the `utilities` module and exercise the `total` function in an interactive Python session.

9. Exceptions

1. In a terminal, navigate to the directory that contains the `circle.py` file that you created in a previous exercise and execute the following

   ```
   ./circle.py x
   ```

   This should result in an exception since `x` is not a valid number.

   Modify the code in `circle.py` to guard against these sorts of exceptions and display the following message if an invalid number is supplied:

   ```
   You must supply a valid number for radius.
   ```

10. Classes

1. Create a new Python module called `shapes.py`. In `shapes.py`, define a class called `Shape` that has an `area` method and `perimeter` method, both with empty implementations.

2. Read the description of `object.__str__` in the Python documentation. The Python `print` function uses a class's `__str__` implementation when it prints an object. Add the following `__str__` implementation to the `Shape` class

   ```
   def __str__(self):
       return '{name}: area={area}, perimeter={perimeter}'.format(
                                         name=self.__class__.__name__,
                                         area=self.area(),
                                         perimeter=self.perimeter())
   ```

   The above implementation makes use of the `format` method on a string. When using this method, each keyword surrounded by curly braces in the string being formatted, is replaced by the value assigned to the keyword argument passed to format. So, in the code above, the `{name}` in the string is replaced by the value passed in for the `name` parameter. Likewise, substitutions are performed for `{area}` and `{perimeter}`. Consult the documentation for `str.format` for further details.

3. Define a subclass of `Shape` called `Rectangle`. The `Rectangle` class should have the following initializer method

   ```
   def __init__(self, width, height):
       self.width = width
       self.height = height
   ```

   Implement `area` and `perimeter` for the `Rectangle` class.

4. Define a subclass of `Rectangle` called `Square`. Implement `Square` in such a way that only the `__init__` method, which should take a single `width` argument, is implemented. The `__init__` method for `Square` should invoke the `__init__` method for `Rectangle`.

5. Define a subclass of `Shape` called `Circle`. Implement `__init__` for the `Circle` class that makes use of a single `radius` parameter. Implement `area` and `perimeter` for the `Circle` class.

6. Define a class called `ShapeCollection` that is used to hold any number of `Shape` objects. Implement methods `add` and `remove` on `ShapeCollection`. These methods should take a single argument and allow for adding and removing shapes from the collection.

7. Define an `area` method on `ShapeCollection` that returns the total area of all shapes contained in the collection.

8. Define a `perimeter` method on `ShapeCollection` that returns the total perimeter of all shapes contained in the collection.