

# Python Programming For Beginners

## Module 6 Strings, File I/O, OOP

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

Go over homework.

Review list concepts, range function to build a list on the fly. – Ask for questions

Alternative, using built in sort approach:

```
myList.sort()
print('Minimum value is: ', myList[0])
print('Maximum value is: ', myList[-1]) #last element
```

## Strings

We started using strings on day one with this statement:

```
print('Hello World')
```

Then I showed how to concatenate strings like this:

```
str1 = 'Hello'
str2 = 'There'
str3 = str1 + ' ' + str2
print(str3)
```

When you get input from the user that you want as a number:

```
age = input('Enter your age: ')
```

Needs to be converted to an int or a float before being used:

```
age = int(age)
```

## Strings are like lists

It turns out that while it may not seem obvious, strings are very similar to lists. Think of a string as a list of characters. That's worth saying again, think of a string as a list of characters. If you think of them that way, then many of the operations that you can do with lists, you can also do with strings.

## len function applied to strings

For example, like a list, a string can be any length. To find out how many elements are in a list, you use the len built-in function. But len can also be used on a string.

```
state = 'Mississippi'
theLength = len(state)
print(theLength)    # prints 11
```

There is also the empty string, which has a length of zero

```
myString = ""
print(len(myString)) # prints 0
```

## Indexing characters in a string

Again, if you think of a string as a *list* of characters, then you can think of each character as an element. Further, we can use an index to refer to a character or characters in a string the same way we indexed the elements in an array. Remember from the definition, the index is position of an element.

```
+++++
```

Index of each character in a string:

0	1	2	3	4	5	6	7	8	9	10
M	i	s	s	i	s	s	i	p	p	i

This string has 11 characters. Notice that the characters in a string are numbered (or indexed) identically to the elements in a list. What we humans would think of as the first character, in Python, it is considered the ‘zeroeth’ character – or the character at index zero. And the last character in the string is at index 10 (which is the length -1).

Also notice, that Python allows you to also use negative indexing if you want. Think of the negative index as the length of the string plus the negative index. This can be confusing. But it is useful if you want to get the last character in a string, it’s index is always -1.

0	1	2	3	4	5	6	7	8	9	10
M	i	s	s	i	s	s	i	p	p	i
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

## Accessing characters in a string

Then like a list, we can also use the bracket syntax to identify a character at a specific the index:

```
print(state[0]) # prints ‘M’
```

```
print(state[2]) # prints ‘s’
```

```
print(state[-1]) # prints the last 'i'
```

If you try to access a character that is beyond the end of a string, you get an error:

```
print(state[1000])) # error
```

```
++++++
```

## Iterating through a string

Here is some code to print each letter of a string on a separate line. Could loop through a string like this  
(THIS IS NOT PYTHONIC, JUST FOR SHOWING A CONCEPT)

```
myString = 'Mississippi'
myIndex = 0
while myIndex < len(myString):
    print(myString[myIndex])
    myIndex = myIndex + 1
```

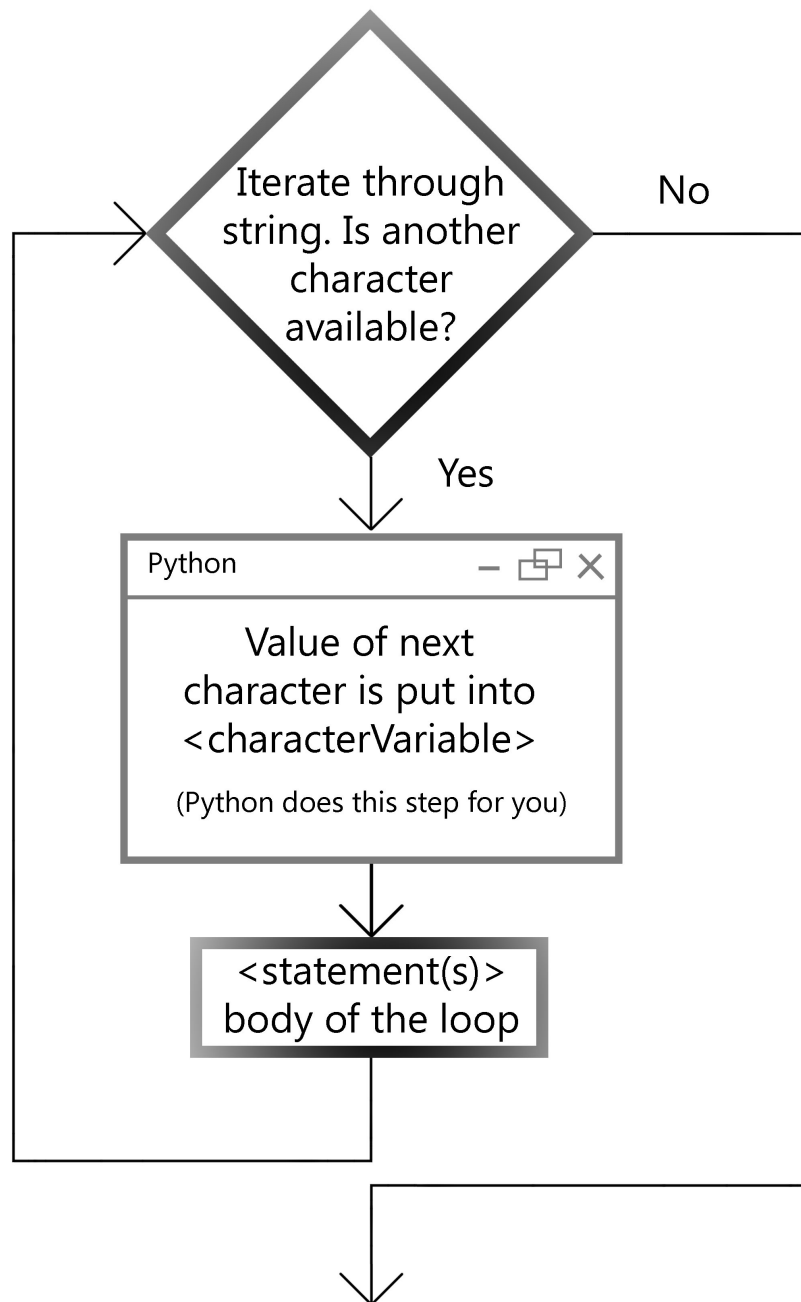
However, just like a list, the “for” statement allows you to easily loop through (or iterate through) all characters in a string.

```
++++++
```

It uses the same syntax as the for loop with a list:

## For loop with a string

```
++++++
for <charVariable> in <string>:
    <do whatever with <charVariable> >
```



Example:

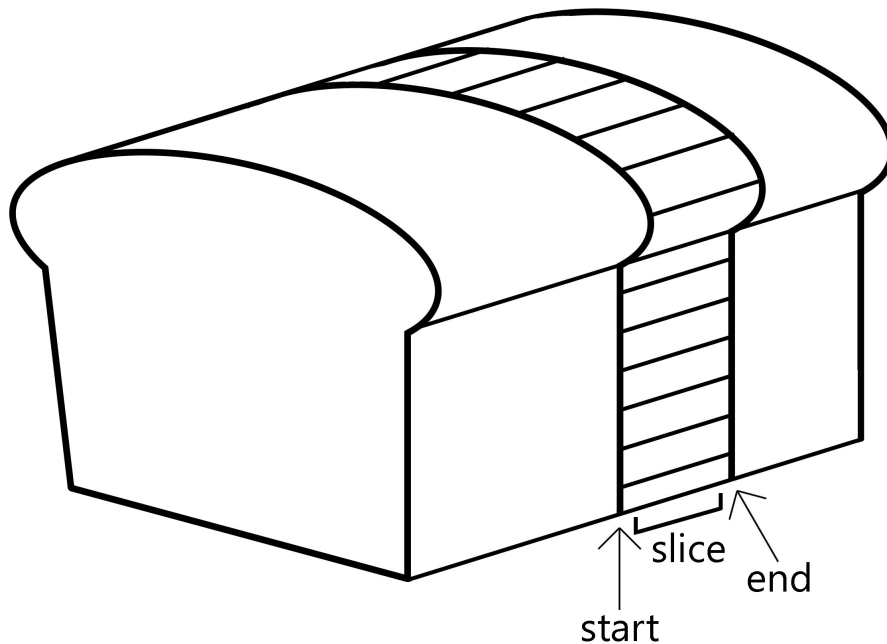
```
myString = 'abcdefg'
for letter in myString:
    print(letter)
```

# Creating a substring – a slice

Very often, when you are dealing with strings, you want to extract a string from a longer string. For example, when you want to find a particular piece of information within a string. In programming, this is generally called creating a substring.

In Python, we do this by taking a 'slice' of a string. In the Python world, this is commonly referred to as 'slicing'.

+++++++



+++++

To make a slice of a string, we have to specify the starting index and the ending index of the slice that we want to create. Think of the string as a loaf of bread, and the analogy is that you are making a slice within the loaf. However, think of this as a magical loaf of bread. When you make a slice, Python makes a copy of the characters in the slice, it does NOT remove those characters from the string. To specify a slice, Python provides the following syntax:

```
++++++  
<string>[<startIndex>: <upToButNotIncludingEndingIndex>]
```

The character at <startIndex> IS included at the first character of the substring. But notice the <upToButNotIncludingEndingIndex>. Here, we specify the first index that is NOT included in our slice. For example, if we have a string like this:

```
++++  
myName = 'Joe Schmoe'
```

and we want to take just the first name, then we want to take a slice starting at index 0 (the J), through index 2 (the e).  
If we want to get just the first name, we would ask for this:

```
print(myName [0:3]) #prints Joe
```

To get the last name, we would use this:

```
nChars = len(myName)  
print(myName [4:nChars]) #prints Schmoe
```

Notice that you can use constants or variables in slices.

## In class assignment:

Write a program that takes in a month number, and outputs a 3 letter abbreviation for a month. Start with:

```
months = 'JanFebMarAprMayJunJulAugSepOctNovDec'
```

```
#scaffolding:
```

```
nMonth = input('Enter a month number (1-12): ')  
nMonth = int(nMonth)
```

```
# Some code that creates a slice based on nMonth
```

```
print(monthAbbrev)
```

Can be done in 2 or 3 lines of code. (Do not use 12 'if' statements)

Hint: Think about what slice would have to be created for Jan, what slice would have to be created for Feb, for Mar, for Apr. Is there a pattern? Can you write some simple code that generates a solution to this pattern?



solution:

```
nMonth = input('Enter a month number (1-12): ')
nMonth = int(nMonth)
startIndex = (nMonth - 1) * 3
endIndex = startIndex + 3
monthAbbrev = months[startIndex : endIndex]
print(monthAbbrev)
```

or even shorter:

```
nMonth = input('Enter a month number (1-12): ')
nMonth = int(nMonth)
startIndex = (nMonth - 1) * 3
monthAbbrev = months[startIndex : startIndex + 3]
print(monthAbbrev)
```

and yet another way:

```
nMonth = input('Enter a month number (1-12): ')
nMonth = int(nMonth)
endIndex = nMonth * 3
startIndex = endIndex - 3
monthAbbrev = months[startIndex : endIndex]
print(monthAbbrev)
```

++++

## Additional syntax for slicing:

[:<upToButNotIncluding>]

means start at the beginning, go to <upToButNotIncluding> character

[<start>:]

means start at the start index, go to the end

`[:]`

means a copy of the whole string

## Slicing applied to a list

++++++

Slicing works the exact same way with lists. To create a slice of a list, you use the exact same syntax:

```
myList[<startingIndex>:<upToButNotIncludingIndex>]
```

Example:

```
startingList = [22, 104, 55, 37, -100, 12, 25]
```

```
newList = existingList[3:6]
```

```
print(newList)
```

```
#prints [55, 37, -100]
```

## Strings are not changeable

One big difference between lists and strings is that strings are NOT changeable. Remember that we said that lists are changeable, or 'mutable'. In Python terms, strings are 'immutable'. That is, you cannot set or change an individual character in a string. For example, let's say that you wanted to change the second character of a string. This will give you an error:

++++++

```
someString = 'abcdefghijkl'
```

```
someString[2] = 'x'    #error
```

Instead, you have to reassign a string, or create a new string:

```
someString = someString[:2] + 'x' + someString[3:]
```

(Remember the concatenation operator)

++++

## In class assignment:

Build a function called `countCharInString`. It will be passed two parameters:

`findChar` – a character to find

`targetString` – a string to be searched

It should return the number of times `findChar` is found in `targetString`

```
print(countCharInString ('s', 'Mississippi')) #expect 4
```

```
print(countCharInString ('i', 'Mississippi')) #expect 4
```

```
print(countCharInString ('p', 'Mississippi')) #expect 2
```

```
print(countCharInString ('q', 'Mississippi')) #expect 0
```

solution:

```
# Count a single char in another string
def countCharInString(findChar, targetString):
    count = 0
    for letter in targetString:
        if findChar == letter:
            count = count + 1

    return count

print(countCharInString ('s', 'Mississippi')) #expect 4

print(countCharInString ('p', 'Mississippi')) #expect 2

print(countCharInString ('q', 'Mississippi')) #expect 0
```

## Built-in string operations

While it is fun to build these types of functions, it turns out that we don't have to. The people who built Python have done all this work for us. In fact, there is a whole 'library' of these types of routines built in.

In fact, strings are smart. Because these functions are built in, strings already know how to do many operations like this. Go to the shell and type this:

```
myString = 'abc'
dir(myString)
```

You will get a long list back. At the end, there are a number of names that are human readable. These are the built in functions that strings know how to do.

To use them, you use this syntax:

```
<string>.<functionName>(<optionalArguments>)
```

There are many operations that can be done on strings that are part of the built in library of strings. That is, every string can do these operations just because they are strings. You do not need to import any special package to use these.

Here are some of the most important string functions:

- <string>.count(substring) Finds number of occurrences of substring
- <string>.index(substring) Finds first occurrence of substring in string
- <string>.lower() Returns lower case version of string
- <string>.upper() Returns upper case version of string
- <string>.title() Returns string with upper case of first word,  
lower case of everything else
- <string>.strip() Returns string without white space
- <string>.lstrip() Returns string with left white space removed
- <string>.rstrip() Returns string with right white space remove
- <string>.replace(<old>, <new>) Returns edited version of string.
- <string>.startswith(<string>) Does a string start with another string.  
<string1> in <string2> Does string1 appear in string2?  
Returns a Boolean

Example: You can ask the user a question asking the user to enter yes or no, take the answer and apply the string.lower() function to it before comparing their answer to 'y' or 'n'.

## In class assignment

In class assignment

Breaking up a user name:

Ask the user to enter their full name:

firstName<space>lastName

Find the <space> character in the name

Then use slices to find the first name and the last name and store each into a variable.

Build a "directory style" version of their name and print it:

lastName, firstName

# First name last name, directory style:

```
fullName = input('Please enter your full name: ')
indexOfSpace = fullName.index(' ')
firstName = fullName[:indexOfSpace]
lastName = fullName[indexOfSpace + 1:]
print(lastName + ', ' + firstName)
```

# First name last name, directory style with error detection:

```
while True:
    fullName = input('Please enter your full name (first name space last name): ')
    fullName = fullName.strip() # remove any leading or trailing while space
    nSpaces = fullName.count(' ')
    if nSpaces == 1: # OK found a single space
        break # exit the loop
    print('Please try again, make sure that you have one space in your name')
    print()

indexOfSpace = fullName.index(' ')
firstName = fullName[:indexOfSpace]
lastName = fullName[indexOfSpace + 1:]
directoryStyleName = lastName + ', ' + firstName

directoryStyleName = directoryStyleName.title()

print(directoryStyleName)
```

# File I/O

## File I/O (Input/Output)

In every program we have talked about so far, when the program ends, the computer forgets everything that happened in the program. Every variable you created, every string you used, every Boolean, every list – it's all gone. But what if you wanted to keep some data that you generate in a program, and save it for when you run the program later. Or, to have some other program use the data.

If you want to keep some information around between runs of a program, you need a way of having what is called “persistent” data – you want the data to remain available on the computer. To do this, we need to have the ability to write to, and read from a file on the computer.

## Saving files on a computer

There are many examples of this concept that you are already familiar with. Think about a word processor or spreadsheet file. You create a file in your word processor or spreadsheet program, and you save it, and quit your program. Later you reopen the word processor or spreadsheet program, reopen the saved file, and all the information is brought back.

In fact, Python source files that you write are like this. You open the Python editor (IDLE), and you create a Python source file (a document). As you edit the file in IDLE, the content is kept in the memory of the computer. When you hit the save button, the content – which, if you think about it, is really just a string of text – is written to a file on the computer. You can then quit IDLE. Later, you can come back into IDLE, open the file, and the text of your program is read in, perhaps formatted in a special way, then displayed to you, and you can edit again. Whenever you do a save, the program is written out to a file.

Up until now, when we run a program, and stopped it, or quit IDLE, the source file would still be around, but any data that we might have

manipulated in the program went away. However, Python allows programmers to write and read files of text.

## Defining a path to a file

You identify a file by specifying the file's name and say that you want to open the file for reading or for writing. Then you read the text into a string variable, or write the contents from a string variable. Finally you close the file.

++++

DEF: Path is a string that uniquely identifies a file on a computer. (Sometimes called a 'filespec')

+++++

There are two types of paths: absolute and relative. An absolute path is one that starts at the top level of the file system of your computer, and ends in name of the file. The other type of path is called a relative path. For this class, we will assume that all paths are relative to where the Python source file lives. That means that any file we want to use or create lives either in the same folder as your Python source file, or in a folder inside that folder. This makes our programs portable across different systems

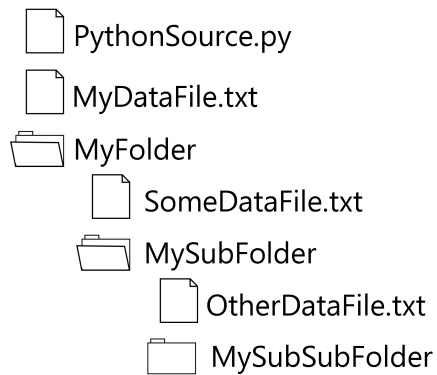
+++++

Let's assume a folder structure like this.



## Folder structure

---










We have a `PythonSource.py` file with a file called `MyDataFile.txt` in the same folder. There is also a folder called `MyFolder`.

Within `MyFolder` there is another data file called `SomeDataFile.txt` and a folder called `MySubFolder.txt`.

Within `MySubFolder`, there is a data file called `OtherDataFile.txt` and perhaps another sub sub folder.

From the point of view of `PythonSource.py` file, here are the paths to the 3 different data files:

Folder structure	Path to the text file
 PythonSource.py	
 MyDataFile.txt	' MyDataFile.txt '
 MyFolder	
 SomeDataFile.txt	' MyFolder/SomeDataFile.txt '
 MySubFolder	
 OtherDataFile.txt	' MyFolder/MySubFolder/OtherDataFile.txt '
 MySubSubFolder	

If you are running a Python program, and you want to use a file in the same folder, then the path for the data file would just be the name of the file as a string:

`'MyDataFile.txt'`

If you want to get to a file that is inside a folder where the Python program lives, then you specify the folder name, a slash ("/"), then the file name, all as a string.

`'MyFolder/SomeDataFile.txt'`

Any time you need to go down another folder level, you specify the folder and then a trailing slash, like this:

`'MyFolder/MySubFolder/OtherDataFile.txt'`

This can go on for any number of levels of subfolders. Just add a slash after every folder name, eventually placing the name of the file at the end.

## Reading text from and writing to a file

(Download: [fileReadWrite.py](#) from class site)

## Building reusable File I/O functions

We now have enough information to build three reusable functions, `fileExists`, `readFile`, and `writeFile`. Once we have these functions, reading and writing data files becomes very easy.

Open the Python file: `FileReadWrite.py`

[Show and explain reusable functions]:

```
import os    #import the operating system module

def fileExists(filePath):
    exists = os.path.exists(filePath)
    return exists

def readFile(filePath):
    if not fileExists(filePath):
        print('The file, ' + filePath + ' does not exist - cannot read it.')
        return ''

    fileHandle = open(filePath, 'r')
    data = fileHandle.read()
    fileHandle.close()
    return data

def writeFile(filePath, textToWrite):
    fileHandle = open(filePath, 'w')
    fileHandle.write(textToWrite)
    fileHandle.close()
```

These three functions are all you need to read text from and write text to a file. But rather than copy all this code, I have provided it for you in a file

named `FileReadWrite.py`, which is available for download from the Files section of the class site.

To write text to a file, you call: `writeFile` and provide the path of a file to write to, and a string or string variable that contains the text to write.

To read from a file, you call: `readFile` and provide the path of a file to read from. It returns the entire context of the file as a string, and you typically assign the result to a string variable.

## The Python “os” module

Before attempting to read from a file, there is one more thing you need to do. Obviously, you cannot read from a file that doesn’t exist, so you need to check that the file is there before you attempt to read from it.

In the same way that we import the `random` module, there is a module that you can import, that can tell us lots of information about the operating system. It is called the `os` module. To use it, you first import it:

```
import os
```

For now, we’re only interested in one operation: the one that can tell us if a file exists. Here’s how you use it:

```
exists = os.path.exists(filePath) #returns a Boolean
```

We’ve built a reusable function around it:

```
def fileExists(filePath):  
    exists = os.path.exists(filePath)  
    return exists
```

Once we have written this function, we don’t need to remember the details of the `os` module, (i.e., that you have to use `os.path.exists`). Instead, we have built a simple function with a nice clean name: `fileExists`. We can re-use this in any of our projects.

++++

Note: For anyone who is into Unix, or who wants to write the equivalent of shell scripts (for automation), the `os` module is very important. If you don't care about these things, just ignore the following.

The `os` module allows you to do many Unix commands as Python statements. For example, here are just a small listing of the things that the `os` module allows you to do:

```
os.listdir
os.mkdir
os.rename
os.walk
os.getcwd
os.chmod
```

## Importing our own modules

Since we have these three functions already written, if we want to re-use this code, we could copy and paste the code into any program that wants to do File I/O. However, consider what happens if we find a bug in our `FileReadWrite.py` file, or if we want to add more functions to help us read files in different ways. In either of these cases, we would have to go back into every Python source file that incorporated these three functions, and modify the code there to fix the bug and/or add functionality. There is a better way.

We have seen how we use the `import` statement to make a built-in Python package available to our program. For example importing the `random` package is done with this statement:

```
import random
```

When we import a package this way, we have to explicitly specify the name of the package when we make a call to a function in that package. For example, when we want to get a random number, we write:

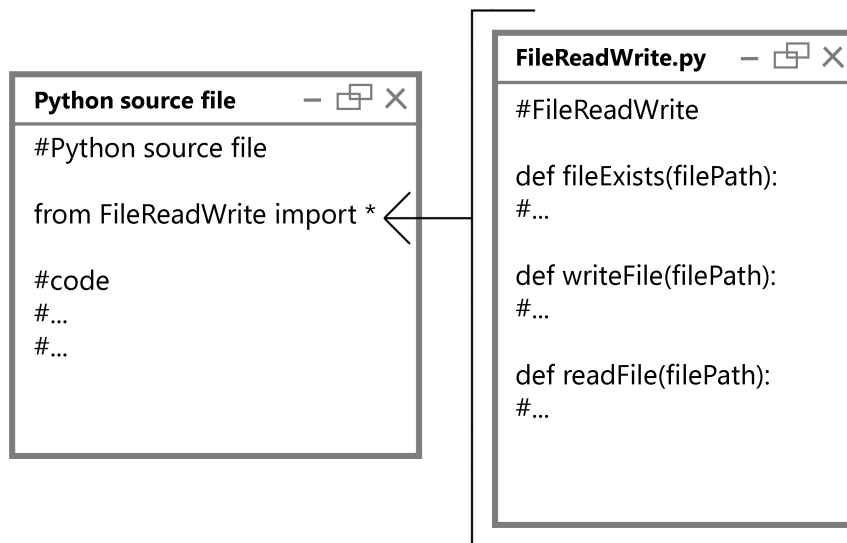
```
value = random.randrange(0, 10)
```

Since we have our reusable functions in `FileReadWrite.py`, the ideal thing to do is to bring in those routines into any new program we write. We do that with the import statement – similar to the way we tell Python to import other pre-written packages. However, we'll use a different syntax to just import those functions:

```
from <filename> import *
```

Using our `FileReadWrite` file, we'll do this:

```
from FileReadWrite import *
```



the file is imported as though you had written the code right there. Therefore, when you do this type of import, you do NOT need to use the name of the imported file when referencing functions inside it. You can just call `readFile`, `writeFile`, and `fileExists` directly.

This allows programmers to break up large programs into a number of more manageable reusable files. Any program you write that imports a file this way, must be in the same folder as the file you are importing.

## Simple test program

Let's work through an example of writing to and reading from a file.

Create a new Python file (in the same folder as your FileReadWrite.py) and write the following:

```
from FileReadWrite import *  
  
DATA_FILE_PATH = 'TestData.txt'  
myString = 'abcdefghijkl'  
writeFile(DATA_FILE_PATH, myString)
```

Run it. Find TestData.txt in the same folder. Should contain abcdefghijkl

Now we'll add code to read the string back in.

```
newString = readFile(DATA_FILE_PATH)  
print('Read in: ', newString)
```

Should print: abcdefghijkl

## Saving data to a file and reading it back in

Given these three reusable functions, we can now build a sample main program. Here is the idea of the program:

We want to write a program that will count how many times it has been run. To do this, the program will attempt to read a file that contains a number – the number signifies how many times it has been run. Every time the program runs, it should read the file, add 1 to the number, then rewrite the file.

Immediately we would run into a problem: the first time we run the program, there is no file. So, we need to check for the existence of the file right up front.

++++

Pseudo-code:

```
if the file does not exist
    Create a file with '1' in it
else
    Read the content of the file into a variable
    Add 1 to the variable
    Write out value to the file
```

And now we will take that pseudo code and turn it into real Python code using our three re-usable functions. Notice that we are using a constant to represent the name of the file.

```
DATA_FILE_PATH = 'CountData.txt'
if not fileExists(DATA_FILE_PATH):
    # File was not found
    print('First time - creating data file.')
    writeFile(DATA_FILE_PATH, '1')

else:
    count = readFile(DATA_FILE_PATH)
    print('Found the file, data read was: ', count)
    count = int(count)
    count = count + 1
    textToWrite = str(count)
    print('Writing to data file: ', textToWrite)
    writeFile(DATA_FILE_PATH, textToWrite)
```

[Remember to delete 'CountData.txt' before running first time]

The first time the program runs, the data file will not exist. Therefore, we set a variable called count to 1 to say this is the first time we are running this program. The code at the bottom takes the variable, converts it to a string, and writes it out.

Every subsequent time we run the program, the file will exist. So, the else clause will run. In code of the else, we read in the contents of the file (which



will just be number as text), we convert what we read back into an integer, then increment it to add one to the number of times we have run the program. Finally, we write out the new value.

## Building an adding game

Now, let's try to use these concepts in a real program. First, we'll build a small game. To make this simple, we'll build a simple Adding Game. The basic idea will be this. We'll start by building a game where we ask the user to add two numbers and see if they answer it correctly. Then we'll expand the program to keep score. Then we'll expand it further to write out the score when we exit the program, and read the score back in when we start up the program again. Let's start building the game.

## Simple game example of File I/O:

Let's use the File I/O in a real program. I'll give you a starting point of an "Adding Game" program.

```
import random

score = 0

# Main loop
while True:
    firstNumber = random.randrange(0, 11)
    secondNumber = random.randrange(0, 11)
    correctAnswer = firstNumber + secondNumber

    question = 'What is: ' + str(firstNumber) + ' + ' + str(secondNumber) + '? '
    userAnswer = input(question)
    if userAnswer == '':
        break

    userAnswer = int(userAnswer)
```

```

    if userAnswer == correctAnswer:
        print('Yes, you got it!')
        score = score + 2

    else:
        print('No, sorry, the correct answer was: ', correctAnswer)
        score = score - 1

    print('Your current score is: ', score)
    print

print('Thanks for playing')

```

Now we'll modify the code to:

- At the end of the game, write out the current score to a file
- At the start up of the game, check if the file exists
- If the file does NOT exist
  - print a welcome message
  - set the score to zero
- otherwise (file does exist)
  - read the score file
  - set score to int version of what was in the file
  - print a welcome back message, with the current score

## Writing/reading multiple pieces of data to and from a file

Next, we want to modify this code to make the game persist across runs. That is, we want to save the state, the data of the game to a file, so that when the user comes back later, he or she can pick up where they left off.

In fact, we'll want to keep track of, and write out and read back four pieces of information:

- User name
- score
- number of problems tried
- number of problems answered correctly

In order to be able to write out and read back multiple pieces of information, we need two more built-in functions: split and join.

## The join function

Let's start with join. The data that we want to write to a file must be one long string of text. If we want to write out multiple pieces of data, we need to take all the data that we want to save, and build a string out of all of them. We'll do this in two steps:

- 1) Take all data we want to save (each piece as a string), create a list containing the data.
- 2) Convert the data to a string using join.

The join operation takes a list and concatenates all elements to create a string, where each piece of data is separated by a character of your choice. Join has an odd syntax:

```
++++
```

```
<separatorChar>.join(<list>)
```

Takes a list (of strings), and creates a new string by concatenating all elements of the list separated by the given character.

```
++++Example:
```

```
myList = ['abc', 'de', 'fghi', 'jk']
```

```
# Use comma as separator
myString = ','.join(myList)

print(myString)

# prints: abc,de,fg,hi,jk
```

## The split function

The other built-in function to talk about is called: split.

The split function takes a string that has some separator character, and turns it into a list.

++++

```
<list> = <string>.split(<separatorChar>)
```

++++Example:

```
myList = 'abc,def,ghi,jkl,23,-123.45'
```

```
# Look for comma as a separator char
myList = myString.split(',')

print(myList)
```

```
# Prints ['abc', 'def', 'ghi', 'jkl', '123', '-123.45']
```

Since split is an operation that you do on a string, we can use after we read in data from a file, and separate out the pieces of data that are in the string we read in.

Split and Join perform complimentary or opposite actions. Join takes in a list and produces a string. Split takes in a string and produces a list.

## Final version of the game

Now, let's modify the game further by keeping track of 3 more pieces of data. In addition to the score, as one piece of data, let's find out the user's name. We'll also remember how many problems the user has seen and how many problems the user has answered correctly.

Let's start by modifying the program.

At the top of the program, we'll ask the user their name, and we'll initialize two more variables:

```
userName = input('Please enter your name: ')
nProblems = 0
nCorrect = 0
```

And we'll add some code to increment the number of problems and the number correct at the appropriate places:

```
... nProblems = nProblems + 1

... if userAnswer == correctAnswer:
    nCorrect = nCorrect + 1
```

Now we are finally ready to add code to save all the user's information to a file. At the top of the code, we add two lines. First, because we intend to read and write files, we add this import statement:

```
from FileReadWrite import * # to bring in all read/write code
```

then we'll define a constant for the name of the data file:

```
DATA_FILE_PATH = 'AddingGameInfo.txt'
```

When the user chooses to quit the program (by just pressing Return/Enter), we'll add some code to write the information we want to remember, out to a file. As a format for the content of the file, we'll use the following:

```
<name>,<score>,<nProblems>,<nCorrect>
```

```
# First, create a list of this information:
```

```
# Must convert each number data to a string
```

```
infoList = [userName, str(score), \
            str(nProblems), str(nCorrect)]
```

```
# Use join to convert the list to comma delimited string:
```

```
infoString = ','.join(infoList)
```

```
# Lastly, write it all out:
```

```
writeFile(DATA_FILE_PATH, infoString)
```

When we run the program, it first asks us for our name. Then we answer as many problems as we want. When we quit the program, the code writes out our information to a file. If we find the file and open it in some text editor, we see something like this:

```
Joe Schmoe,29,15,14
```

The last step is to add code at the beginning of the program to look for the saved data file. If there is no such data file, then we can assume that this is the first time the user is running the program. If the data file exists, we

need to read it, extract the information, and set the program's variables from appropriately.

```
if not(fileExists(DATA_FILE_PATH)):
    name = input('You must be new here, please enter your name: ')
    score = 0
    nProblems = 0
    nCorrect = 0

    print('When you are done answering problems, just press RETURN/ENTER
and your score will be saved')
    print("OK, let's get started", name, " ...")
    print()

else:
    savedDataString = readFile(DATA_FILE_PATH) #read the whole file into a
variable
    savedDataList = savedDataString.split(',') # turn that into a list
    name = savedDataList[0]
    score = savedDataList[1]
    score = int(score)
    nProblems = savedDataList[2]
    nProblems = int(nProblems)
    nCorrect = int(savedDataList[3])

    print('Welcome back', name, 'nice to see you again! ')
    print('Your current score is: ', score)
    print()
```

## Writing and reading a file, one line at a time

(If time permits)

If you open FileReadWrite.py and scroll down to the bottom, you will see that there are three additional functions there: `openFile`, `readALine`, and `closeFile`. These functions are there to allow you to read in files that have multiple lines of data in them.

For example, assume you have some file named MultiLineFile.txt that contains:

This is line 1

This is line 2

Some data,12,-415.23

The file could contain as many lines as you need to represent any data that you want save and can have any format that you want to use to save the data. To read this example file, we could have a program like this:

```
DATA_FILE_PATH = 'MultiLineData.txt'
myFileHandle = openFile(DATA_FILE_PATH)
data1 = readALine(myFileHandle)
print(data1)
data2 = readALine(myFileHandle)
print(data1)
data3 = readALine(myFileHandle)
print(data3)
```

# Add code here to split up data3 into separate pieces of information



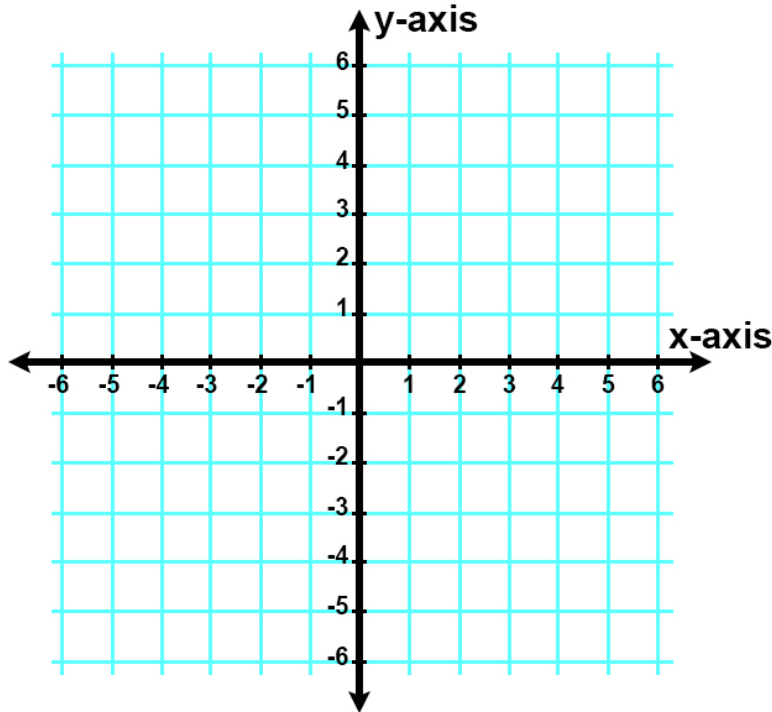
# Object-Oriented Programming

We have been working in what is called “procedural programming” where you have many functions, and you pass in data to a function, the function does it’s job, and potentially returns an answer. In the time remaining, I will try to give you a quick introduction to a concept called Object-Oriented Programming. This is a very complex topic, and in fact, this is the topic of another course that I teach here. I’m also writing a whole book on this topic (available soon from No Starch Press). Object-oriented programming is a new way of thinking about how to write code than what we have done so far in this course. So, I will give you a quick preview of that course.

In order to give you a good feel for what object-oriented programming is about, I want to introduce an external package called “Pygame”. It is an extension to Python that allows you to do the types of things that we typically associate with computers these days. That is, be able to create a window, use the mouse as a pointing device, draw graphics on the screen, play sounds, etc. If you remember from day one, I showed a 4 in a row game that was developed using this environment. To understand how this all works together, I first need to explain how the computer screen works in a graphical environment.

## Screen coordinate system

The first deals with the coordinate system of the screen. You are probably familiar with “Cartesian coordinates” in a grid like this:



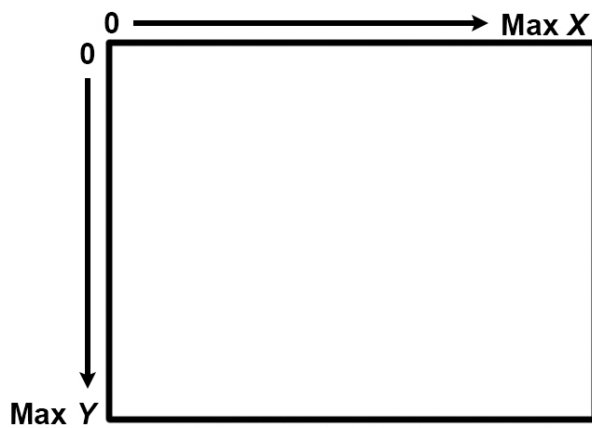
Any point in a Cartesian grid can be located by specifying its X and Y coordinates (in that order). The “origin” is the point specified as (0, 0), and is found in the center of the grid.

Computer screen coordinates work in a similar way. To locate any pixel on the screen, we still specify an X and a Y coordinate. However, there are a few key differences:

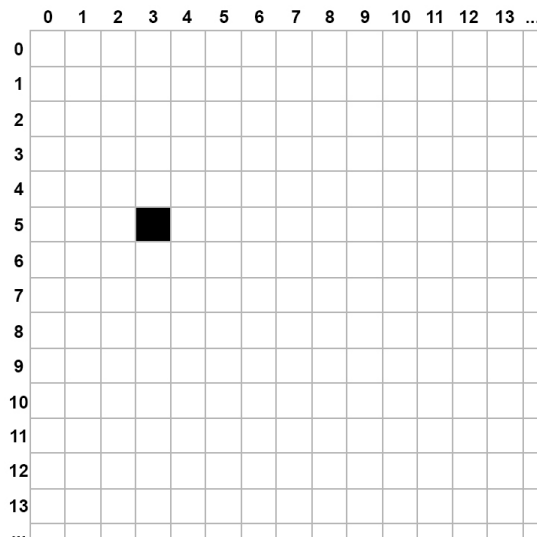
- 1) With screen coordinates, the X and Y values are always integers. Each (X, Y) pair specifies a single pixel on the screen.
- 2) The origin, the (0, 0) point, is in the upper left-hand corner.
- 3) The Y axis is reversed. With screen coordinates, The Y value starts at zero at the top of the screen, and the value of Y increases as you go down.
- 4) In a Pygame program, the X and Y coordinates are always specified as relative to the upper left-hand corner of the application’s window, not to the screen. (Yes, there are screen coordinates also. However, within an application, since coordinates are always relative to the upper left-hand corner of the window, the user can move the window anywhere on the screen and that will not affect the program.)

5) As pixel coordinates, every X Y pair represents a single colored picture element on the screen (as opposed to Cartesian coordinates where a specific point has zero width and zero height).

The window of an application is made up of a given width (X values) and a given height (Y values). When we first start up Pygame, we need to specify the width and height of the window we want to create.

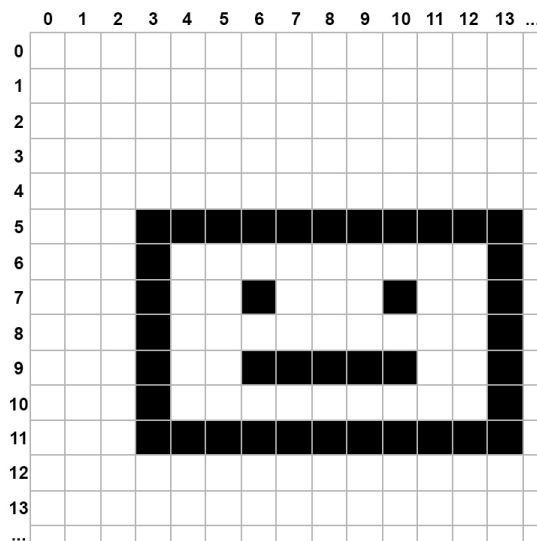


Within the window, we can address any pixel by using its X and Y coordinates.



In the graphic above, there is a black pixel at position (3, 5). That is an X value of 3 (actually the 4th column across since screen and window coordinates start at 0), and a Y value of 5 (actually the 6th row down). Each pixel in a window is commonly referred to as a point.

Whenever we want to show a graphical image in a window, we will need to specify the coordinates as an (X, Y) pair. The coordinates are used as the upper left-hand corner of where the image should be placed.



This image is drawn specifying its location as (3, 5). That means that its upper left corner is positioned where its X location is column 3 and its Y location is row 5.

Here is a simple program that shows a ball bouncing around a window.  
[Show ball bouncing program]

Here is the code of that program:

```
# pygame demo 4(a), One image, bounce around the window using X, Y  
coords
```

```
# 1 - Import packages  
import pygame
```

```

from pygame.locals import *
import sys
import random

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
BALL_WIDTH_HEIGHT = 100
N_PIXELS_PER_FRAME = 3

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sounds, etc.
ball = pygame.image.load('images/ball.png')

# 5 - Initialize variables
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT
ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
xSpeed = N_PIXELS_PER_FRAME
ySpeed = N_PIXELS_PER_FRAME

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # check if the event is the X button
        if event.type == pygame.QUIT:
            # if it is quit the game
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions

```

```

if (ballX < 0) or (ballX > MAX_WIDTH):
    xSpeed = -xSpeed # reverse X direction

if (ballY < 0) or (ballY > MAX_HEIGHT):
    ySpeed = -ySpeed # reverse Y direction

# update the location of the ball, using the speed in two directions
ballX = ballX + xSpeed
ballY = ballY + ySpeed

# 9 - Clear the screen before drawing it again
window.fill(BLACK)

# 10 - Draw the screen elements
window.blit(ball, (ballX, ballY))

# 11 - Update the screen
pygame.display.update()

# 12 - Slow things down a bit
clock.tick(FRAMES_PER_SECOND) # make PyGame wait the correct
amount

```

There is a lot of code there and we don't have time to go through all the steps. Instead, I'll point out that there are two variables: `ballX` and `ballY` that indicate the current location (upper left hand corner) of the image. There are also two more variables, `xSpeed` and `ySpeed` that contain values for the speed in the x direction and the speed in the y direction.

These variables are set up at the top of the program, then the program goes into an infinite loop. At the end of the infinite loop, the program "draw" all screen elements. In this program, it clears the entire window, then draws the ball at a location based on the values of `ballX` and `ballY`.

Each time through the loop, the program checks to see if the ball has gone too far right or left, or has gone too low or high. In any of these cases, it negates the appropriate speed. Then it simply adds the `xSpeed` to the `ballX` value and adds the `ySpeed` to the `ballY` value. That way, the ball appears in a different location each time through the loop. The result is that you see an

animation of the ball appearing at different locations, and it bounces off the walls.

This works fine for one ball. However, what if you wanted to have two balls or three, or more, For each ball you want to put on the screen, you would have to have a copy of each of the ballX and ballY variables, and have a copy of the xSpeed and ySpeed variables. This would become unmanageable very quickly.

Since the data of each ball and the code that operates on each ball is obviously very closely tied together, we would want a way to bind them together in a single unit. This is what an object is:

Definition: Data plus code that acts on that data over time.

Let's show you how we can rewrite this code to create an object that represents the data and code of the ball. The idea is to split the code into a smaller main code that runs the main loop, and put all the code and data dealing with the ball in a separate file. The ball is really made up of three parts:

- 1) Creating a ball (loading the image, choosing a starting location and speed)
- 2) Updating the ball every "frame"
- 3) Drawing the ball in the window

The following is the definition of a "class" that will contain all the code and data that is needed to represent the data of a ball (ballX, ballY, xSpeed, ySpeed, etc. and the operations that a ball can do (create, update, and draw).

```
import pygame
from pygame.locals import *
import random
```

```
# BALL CLASS
class Ball():
```

```
    def __init__(self, window, windowWidth, windowHeight):
```

```

self.window = window # remember the window, so we can draw later
self.windowWidth = windowWidth
self.windowHeight = windowHeight
self.ballImage = pygame.image.load("images/ball.png")
self.rect = self.ballImage.get_rect()

# Pick a random starting position
self.rect.left = random.randrange(0, self.windowWidth - self.rect.width)
self.rect.top = random.randrange(0, self.windowHeight -
self.rect.height)

# Choose a random speed in both the x and y directions
self.xSpeed = random.randrange(1, 4)
self.ySpeed = random.randrange(1, 4)

def update(self):
    # check for hitting a wall. If so, change that direction
    if (self.rect.left < 0) or (self.rect.right > self.windowWidth):
        self.xSpeed = -self.xSpeed

    if (self.rect.top < 0) or (self.rect.bottom > self.windowHeight):
        self.ySpeed = -self.ySpeed

    # update the left and top of the ball rect, based on the speed in two
directions
    self.rect.left = self.rect.left + self.xSpeed
    self.rect.top = self.rect.top + self.ySpeed

def draw(self):
    self.window.blit(self.ballImage, self.rect)

```

The main code now is much simplified, and now looks like this:

```

# pygame demo 6(a) using Ball class, bounce one ball

# 1 - Import packages
import pygame
from pygame.locals import *
import sys

```



```
import random
from Ball import * # bring in the Ball class code

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock() # set the speed (frames per second)

# 4 - Load assets: image(s), sounds, etc.

# 5 - Initialize variables
oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions
    oBall.update() # tell the ball to update itself

    # 9 - Clear the screen before drawing it again
    window.fill(BLACK)

    # 10 - Draw the screen elements
    oBall.draw() # tell the ball to draw itself

    # 11 - Update the screen
    pygame.display.update()
```

```
# 12 - Slow things down a bit  
clock.tick(FRAMES_PER_SECOND) # make PyGame wait the correct  
amount
```

Let's first run it to see it working. From the user's point of view, there is no change. The ball bounces exactly the same way. However, from an "architecture" point of view, things are very different. There is a clear separation between the functionality of the main loop, and the representation of the ball. The main code creates a ball object (saved in the variable `oBall`) from the `Ball` class, and every time through the loop, we call `oBall.update` and `oBall.draw` to update and draw the ball. All the data of the ball is kept inside the ball object.

Why is this important? Because this allows us to solve the problem of having multiple balls. Lets say that we want to have three balls instead of just one. To do that, instead of creating a single ball object, we'll create and maintain a list of ball objects. The code in the `Ball` class does NOT change at all. Instead, the main code has a slight change to create a list of ball objects, then in the main loop, we call the `oBall.update` and `oBall.draw` on each one. Each ball maintains it's own copy of the data. Each ball knows its own `ballX` and `ballY`, and it's own `xSpeed` and `ySpeed`.

In our main program we'll start by creating three ball objects. When we go through the main loop, we also loop through all ball objects telling each one to update itself, then draw itself.

The really cool thing about this is that now that we have things set up this way, we can modify a single constant and instead of creating 3 balls, we can create 10 balls, or one thousand balls.

Now creating a 100 balls and animating them is pretty impressive, but it might not be clear what the real usefulness of OOP is. Well, imagine a simple graphical user interface. Here is an example of a program that just consists of three buttons, A, B, and C. rather than use a procedural programming approach, I've built a single "Button" class, and made three button objects. Each runs the exact same code, but because the data in each button is different, each button recognizes mouse down, mouse up, and "rollovers" on

itself. And when a button is clicked (down and up on the same button), it is easy to tell which button has been clicked.

Graphical user interface (GUI) widgets like buttons, radio buttons, checkboxes, input text fields, output text fields, etc, are all real-world examples of objects. However, any time you have a collection of data and code that works on that data, that is a good opportunity to write a class and create an object from that class.

In my upcoming book/class, we will get into the details of writing your own classes, and writing in an object-oriented style!