In-Depth 2022 Log: Learning Python

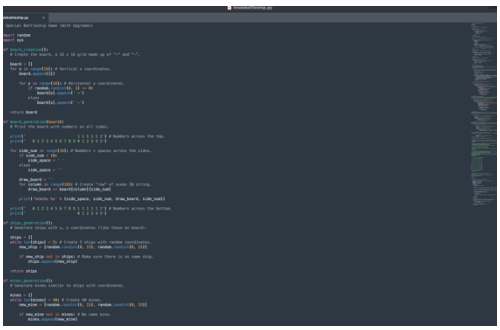
Table of Contents:

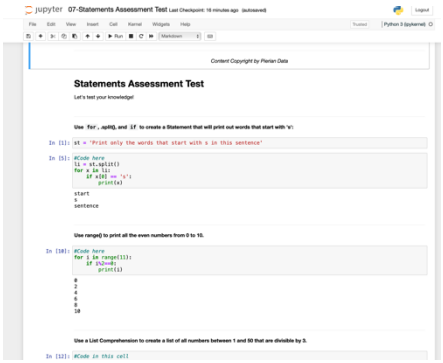
Monday, February 7, 2022

Hello! Welcome to my first log entry of this project. First off, I have not been recording every week thus far, as it has been mainly a review of information I have learned before. As I go deeper, I will record a log every week.

I must say that, as a customer, I have never been more satisfied with a purchase. The course I am enrolled in, by Jose Portilla, which includes over 22 hours of video, tests, homework, projects, and exercises, has been exceptionally informative (described below). As I predicted, I have *many* holes in my knowledge, but this course enlightened me in every way. Currently, I have completed Sections 5 of 23 (currently on #6), thus I am on schedule to finish by the end of March.

The first segment of the course teaches simply installing and running Python 3. While I have done this before, I have *never* heard of either the Anaconda Navigator or the Sublime Text Editor, both of which are excellent Python environments. Anaconda, as the lecturer describes, also includes many (common) resources, libraries, and environments normally not included which I will use later. Immediately, however, I am introduced to Jupyter Notebooks, a clean, web-based notebook environment (right). The instructor not only teaches his lessons in the notebook, but he has provided review notebooks for us to use too. We are also introduced to Sublime Text, a simple, desktop, Python editor (left).





The next two sections teach basic Python objects, data structures, and comparison operators, such as strings, numbers, lists, dictionaries, tuples, etc. Reviewing all this information has tripled my understanding of what these are and *how* they work, and I am much more

confident in applying/using them. Previously I had found dictionaries, lists, and tuples confusing, but now I fully explain what they are and how to use them. Take, for example, slicing, a way of retrieving a certain object from a number, string, or list, etc. Or key and value pairs inside dictionaries. In previous programs I have written, I barely understood what those meant, but now I can completely dissect and use these concepts. There were exercises throughout and a final assessment at the end, further reinforcing/proving my knowledge.

The final section I completed teaches statements, namely if/elif/else, for and while loops, useful operators, and list comprehensions. While there were some operators and "tricks" I had not learned previously, list comprehensions were the absolute highlight of the section. Essentially, list comprehensions condense a common operation (that would normally take multiple lines) in Python – taking the values of a list and appending them to a new one – into a single line. In the image on the right, the top section shows the operation, the bottom section shows what list comprehensions can do.

```
# WITHOUT LIST COMPREHENSIONS
mylist = []
for i in range(1, 51):
    if i%3==0:
        mylist.append(i)

# WITH LIST COMPREHENSIONS
mylist = [i for i in range(1,51) if i%3==0]
```

This concludes my first log entry. I am very excited to progress further into the course, where I will be delving into topics like object-oriented programming and writing milestone projects!

Monday, February 21, 2022

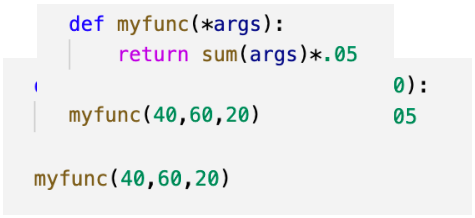
Welcome to my second log entry! In the past two weeks, I have continued with the Udemy course, still on track to finish half by the halfway mark. I covered two (nearly three) longer, major sections, namely methods and functions, object-oriented programming (OOP), and my first milestone project.

To start with, there are methods and functions. Functions are blocks of code that, when called, can be run, however many times is necessary. The basics are introduced, including the syntax, structure, parameters, and how to use functions with each other. Next, we learn how to perform different operations using functions, as functions open *many*, many more possibilities of what can be done using code. The top image shows a function, and the bottom image is calling a function (obviously functions will have *much*, much more code inside them). This section of the course included a significant amount of exercise problems because understanding how functions work is paramount. While functions are not new

```
def say_hello():
    print('hello')
```

```
say_hello()
```

to me, these served as excellent review and cleared up uncertainties I had, for example, how and when exactly parameters (arguments passed into the brackets) should be used.



From this point on, virtually everything taught in the course will be new to me! The latter half of the functions section teaches *args and **kwargs, which allow you to pass in an arbitrary number of arguments (*args) or keyword arguments (**kwargs) into a function, instead of needing to write each. For example, the image above shows a function that finds the sum of several numbers*0.05, requiring the use of *a*, *b*, *c*, etc., as variables. The image below uses the *args keyword to replace those, simplifying the function. Last, the section teaches map(), a way to iterate a function to a list of items, filter(), a function that filters objects in a list based on whether a function outputs True/False, and finally, lambda, simplified, one-time functions we can use without needing to properly create a function.

At long last, we reach OOP, one of my most anticipated parts of the course. First, an introduction: Objects are core to Python – they range from lists and integers to strings and tuples. Methods can be used to perform specific functions on these objects. OOP thus means creating "classes" of objects, assigning attributes to them, and creating "methods" (actions) we can perform on these classes of objects. Naturally, we are introduced to the *class* keyword, the __init__() special function, defining, then calling attributes, and creating methods (essentially functions related to these objects). In the image on the right, "Dog" is defined as a class. An unchanging attribute, species (biologically incorrect), is set as "mammal". The __init__() function is defined, and the attributes "breed" and "name" are set. The images below show that when an instance of "Dog" is created, and the breed and name are set as "Lab", and "Sam," respectively, when they are called, they return the corresponding result. Methods, as stated previously, work similarly; once they are defined, they can be called on the object to perform an action. Other topics introduced include class inheritance, polymorphism, and special methods.

```
class Dog:

    # Class Object Attribute
    species = 'mammal'

    def __init__(self,breed,name):
        self.breed = breed
        self.name = name
```

```
sam = Dog('Lab','Sam')
```

```
sam.name
```

```
'Sam'
```

```
sam.species
```

```
'mammal'
```

The first milestone project involves creating a simple Tic-Tac-Toe game using the knowledge I have gained so far (not including classes/OOP). I am nearly finished writing the program – it simply involves a couple functions, lists, loops, and creating logic with statements. There are also several OOP exercise problems that I will practice with to help solidify my understanding of the concept, which I will complete tomorrow.

Finally, during my meeting with my mentor, Bill Zhang, he suggested that I use Microsoft's Visual Studio Code in lieu of Sublime Text and Anaconda, the two Python environments I stated I was using last entry. VS Code offers a complete environment with all the features the other two had (e.g., Jupyter Notebooks), with additional features such as debugging, that they were missing.

This concludes my second log entry! I will bring updates next week.

Monday, February 28, 2022

As promised, here is this week's update on my Python journey – my third entry thus far. I have reached the halfway point of the online Udemy course, finishing object-oriented programming (OOP) exercises, the milestone project, modules and packages, as well as started errors and exception handling.

First, I completed OOP review homework. Like I had learned, I created classes with attributes and methods. For example, one problem was to create a *class* called "Line," where the user could pass in coordinates, then call a distance() and a slope() method to return a result. The images on the right show the "Line" class in action. There was then a challenge problem that involves similar concepts, where instead the goal is to create a bank account class. First, the user must be able to enter the owner's name and their starting balance (attributes). Then, when called using the print() function, relevant data (i.e., the owner and balance) should be printed for the user to see. Finally, for the methods, the user must be able to deposit and withdraw a balance, but not if the withdrawal is surpasses the available funds.

```
coordinate1 = (3,2)
coordinate2 = (8,10)

li = Line(coordinate1,coordinate2)
```

```
li.distance()
```

```
9.433981132056603
```

```
li.slope()
```

```
1.6
```

Next, I finished the first milestone project, which, as I stated, does not involve any OOP. It is a simple Tic-Tac-Toe game played between two players, involving a variety of functions – namely, one that prints the game board (using ASCII graphics made up of keyboard characters, not *real* graphics), another that asks for input, a function to updates the board after

```
print()
print('Welcome to Tic Tac Toe!')
print('Player 1 is "X", Player 2 is "O".')
print('The board is structured as a number pad:')
print('Reading 1-9, from left to right, top to bottom.')
input('Press ENTER to continue.')

while True:
    boardlist = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
    player1 = 'X'
    player2 = 'O'
    if randint(0,1) == 0:
        turn = player1
    else:
        turn = player2
    moves = []

    while True:
        print_board()
        choice = ask_input()
        take_move()

        if turn == player1:
            if check_win(player1):
                print_board()
                print()
                print('Congratulations! Player 1 has won!')
                print()
                break
            turn = player2
        else:
            if check_win(player2):
                print_board()
                print()
                print('Congratulations! Player 2 has won!')
                print()
                break
            turn = player1

        if len(moves) == 9:
            print_board()
            print()
            print('The game is a tie!')
            print()
            break

    print('Would you like to play again? (Yes/No)')
    if not input().lower().startswith('y'):
        break
```

```
     |     |
     |     |
     |     |
- - - - - - - - - -
     |     |
  X  |  O  |
     |     |
- - - - - - - - - -
     |     |
  O  |  O  |  X
     |     |

Player 1, what is your move? (1-9)
```

input, and lastly one that checks if a player has won. These are nicely tied up in a *while* loop that orchestrates the game itself (the image on the right). You can see when the functions are called (denoted by the "()" at the end – however, input(), print(), etc., are default methods) and logic I created using *if* and *else* statements. Below is an image of the game in action. The simplicity and readability of Python shines through in this example; it is almost like reading English.

The topic covered after OOP was modules and packages. Whether through brief glimpses, stock images, or advertisements, it's very likely that you have observed a professional programmer's IDE (integrated development environment). On the right, you may have seen the code (self-explanatory), but on the left, there might have been what appeared to be a file structure with folders. *Those* are modules and packages – allow me to illustrate. In the Tic-Tac-Toe script above, there is a randint() method a couple lines beneath the *while True* loop. Randint – a part of the default *random* module included with Python, which I had imported earlier – randomly generates an integer between two given numbers. Thus, modules are essentially *other* scripts with functions/methods inside of them, that, when imported, could be called in a separate (outside) program. Guess what? We can, in fact, create modules ourselves; programmers use modules (and packages, which are collections of modules) since projects are often too large to be held in a single script. In this section, __name__ and "__main__" were also discussed.

Finally, I covered errors and exception handling. Normally, when there is an error in Python – whether it is about syntax, wrong type (e.g., trying to add a string with an integer), etc. – the program stops completely. Errors and exceptions use the *try*, *except*, *else*, and *finally* keywords to allow the program to continue running. Like an *if else* statement, the code is placed indented under *try*. If an error occurs, the code under *except* is run; if there is none, the code under *else* is run. Finally, the code under *finally* is always run no matter what. These do not need to be used together, but the example on the right shows what happens in conjunction with a *while True* (infinite) loop. Note that *break* breaks out of the loop, *continue* skips any code underneath and executes back at the top, and int() tries to convert the string (received from the input) into an integer, but causes an error if the string is text, not digits.

```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            break
        finally:
            print("Finally, I executed!")
        print(val)
```

```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: four
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 3
Yep that's an integer!
Finally, I executed!
```

Thus concludes my third log entry! Information about my mentor meeting (and wisdom I have gained) can be found in Post #3. Next update will be on the following week.

Monday, March 28, 2022

Welcome to my 4th log entry! Today's progress report will include everything that I have done the week prior to and the two weeks during Spring Break. I was unable to provide an update early in March because I was working on a Milestone Project (and had not *learned* anything new) and progress slowed during the break itself. Thus, this update will be longer than usual. Finally, I am nearly done the course, although I *may* need until the end of the week.

Immediately after errors and exceptions handling, I was assigned Milestone Project #2 – with the goal, again, to create a game. This time, the requirements were to create single player Blackjack (against a dealer) *with* the application of my newly learned OOP; the challenge of this project. To start with, I created the Bank(), Card(), Hand(), and Deck() classes, which serve as the basis of the game and provide nearly all the functions/methods that I need for the game to work. I designed the *Bank* class (on the right) to manage the player's balance of chips, allowing him/her to wager chips (starting with 100) and updating their balance depending on if he/she won or lost. As per my friends' requests, I added a cheat code to instantly add 10,000 chips to their balance. Next, the *Card* class serves the simple function of being able to print the "name" of each whenever I want. Third, the *Hand* class (on the near right) has functions to process both the moves of the dealer and the player, with the additional ability to ask the player for a move. Finally, the *Deck* class generates the deck, setups the game, and prints the board for the player to see. Aside from classes, I also created a normal function that asks the player whether they want to play again - I end up needing this twice, which is why I created a function in the in the first place.

```
class Bank():
    def __init__(self, balance):
        self.balance = balance
        self.currentbet = None
        self.winner = None

    def __str__(self):
        return f'''
Your bank has {self.balance} chips.
'''

    def place_bet(self):
        while True:
            print('How many chips would you like to bet this hand?')
            try:
                temp = input()
                if str(temp) == '#GimmeMoney':
                    self.balance += 10000
                    print()
                    print(f'Your bank has {self.balance} chips.')
                temp = int(temp)
                if temp <= self.balance and temp > 0:
                    self.currentbet = temp
                    return self.currentbet
                else:
                    print()
                    print(f'Your bet is not within your balance of {self.balance} chips.')
                    print()
            except:
                print()

    def update_balance(self):
        if self.winner == 'Player':
            self.balance += self.currentbet
        elif self.winner == 'Dealer':
            self.balance -= self.currentbet
            if self.balance == 0:
                print()
                print('The entire game is over! You ran out of chips.')
```

```
class Hand():
    def __init__(self):
        self.valuecount = 0
        self.hand = []
        self.choice = None
        self.hit = True
        self.aces = 0

    def add_values(self):
        self.valuecount = 0
        for i in range(len(self.hand)):
            self.valuecount += values[self.hand[i][1]]
        temp = self.aces
        while self.valuecount > 21 and self.aces > 0:
            self.valuecount -= 10
            self.aces -= 1
        self.aces = temp

    def ask_for_choice(self):
        while True:
            print('Would you like to HIT or STAY?')
            temp = input().lower()
            if temp.startswith('h'):
                self.choice = 'Hit'
                break
            elif temp.startswith('s'):
                self.choice = 'Stay'
                break
            else:
                print()

    def input_processing(self, gamedeck):
        if self.choice == 'Hit':
            self.hand.append(gamedeck[0])
            if gamedeck[0][1] == 'Ace':
                self.aces += 1
            gamedeck.remove(gamedeck[0])
        else:
            self.hit = False
```

```
class Deck():
    def __init__(self):
        self.gamedeck = []

    def generate_deck(self):
        for x in suits:
            for y in ranks:
                self.gamedeck.append([x, y])
        shuffle(self.gamedeck)

    def setup(self, player, dealer):
        player.append(self.gamedeck[0])
        player.append(self.gamedeck[1])
        dealer.append(self.gamedeck[2])
        self.gamedeck.remove(self.gamedeck[0])
        self.gamedeck.remove(self.gamedeck[0])
        self.gamedeck.remove(self.gamedeck[0])

    def print_board(self, dealerhand, playerhand):
        print()
        print('--------------')
        print('Dealer\'s Hand:')
        print('--------------')
        for x in dealerhand:
            tempcard = Card(x[0], x[1])
            print(tempcard)
        print('--------------')
        print()
        print('--------------')
        print('Player\'s Hand:')
        print('--------------')
        for y in playerhand:
            tempcard = Card(y[0], y[1])
            print(tempcard)
        print('--------------')
        print()
```

Before continuing, here are the basic rules of Blackjack: It uses a normal,

56-card deck without Jokers. Cards worth a number *are* worth the number, while suits (Jack, Queen, King) equal 10. Aces, however, can be worth either 1 or 11, depending on the player's choice. The dealer deals each player (in this case, one) two cards, face up, dealing one to himself/herself. The player must add their values together, and, with the goal of *getting the number closest to 21*, they can either choose to HIT (get another card from the dealer), or STAY (sit on whatever they already have). If the player hits and goes over 21, they instantly bust and lose the hand. If the player hits and does not go over, they can either choose to hit or stay again (until they choose to stay or bust). Once the player has gone, the dealer does the same, except they must continuously hit until their values add up to 18 or over. If the dealer busts, the player wins. If neither the dealer nor the player bust, whoever has the greatest total wins.

```
### MAIN GAME LOOP ###
while True:
    print('''
Welcome to Blackjack!
Your goal is to beat the dealer
by achieving a closer sum to
21. If you go over, you bust!
The dealer will hit until their
total is 17 or greater.''')
    mybank = Bank(100)
    print(mybank)

    while True:
        # Place bet.
        mybank.place_bet()

        # Setup classes and variables.
        playerhand = Hand()
        dealerhand = Hand()
        mydeck = Deck()
        roundison = True

        # Setup deck.
        mydeck.generate_deck()
        mydeck.setup(playerhand.hand, dealerhand.hand)
        for x in playerhand.hand:
            if x[1] == 'Ace':
                playerhand.aces += 1
        for y in dealerhand.hand:
            if y[1] == 'Ace':
                dealerhand.aces += 1
        mydeck.print_board(dealerhand.hand, playerhand.hand)

        # Ask for move.
        while True:
            playerhand.ask_for_choice()
            playerhand.input_processing(mydeck.gamedeck)
            if playerhand.hit == False:
                playerhand.add_values()
                break
            mydeck.print_board(dealerhand.hand, playerhand.hand)
            playerhand.add_values()
            if playerhand.valuecount > 21:
                roundison = False
                break

        # If player busted.
        if roundison == False:
            print('You busted! The dealer wins.')
            mybank.winner = 'Dealer'
            mybank.update_balance()
            print(mybank)
        else:
            # Dealer turn.
            print('''
It is now the dealer's turn.''')
            dealerhand.add_values()
            dealerhand.choice = 'Hit'
            while dealerhand.valuecount < 17:
                dealerhand.input_processing(mydeck.gamedeck)
                dealerhand.add_values()
            mydeck.print_board(dealerhand.hand, playerhand.hand)

            # Analyze who won.
            if dealerhand.valuecount > 21:
                print('The dealer busted! You win.')
                mybank.winner = 'Player'
                mybank.update_balance()
                print(mybank)
            else:
                if dealerhand.valuecount > playerhand.valuecount:
                    print(f'''The dealer has won since his
total ({dealerhand.valuecount}) is greater than yours ({playerhand.valuecount})!''')
                    mybank.winner = 'Dealer'
                    mybank.update_balance()
                    print(mybank)
                elif dealerhand.valuecount < playerhand.valuecount:
                    print(f'''You have won since your
total ({playerhand.valuecount}) is greater than the dealer's ({dealerhand.valuecount})!''')
                    mybank.winner = 'Player'
                    mybank.update_balance()
                    print(mybank)
                else:
                    print(f'''The game is a push! Your
total values were both {playerhand.valuecount}!''')
                    mybank.winner = None
                    mybank.update_balance()
                    print(mybank)

        # If the bank is empty.
        if mybank.balance == 0:
            break

        # Next round.
        play_again('hand')

    # Next game.
    play_again('game')
```

The two images on the right illustrate the entire game loop of the Blackjack game. Inside the first *while True* loop, the game is introduced and the *Bank* class is setup. The player's balance is stated. Inside the next *while True* loop, the player asked how much they would like to wager for the round of play/hand. One can see that it uses the *Bank* class (created earlier as *mybank*) and is called with ".place_bet()," one of the methods shown earlier. The rest progresses similarly — I use the classes and their methods to create the player's hand and the dealer's hand, create the deck, setup Aces, ask the player for their move then process it, process the dealer's move (which is automated), and finally use several *if*, *elif*, and *else* loops to determine the winner. At the end, the chip balance is updated and the player is asked if they would like to play another game, and if they lost all their chips, if they would like to restart entirely. The image on the right shows sample output from my game.

```
--------------
Dealer's Hand:
--------------
Two of Spades
--------------

--------------
Player's Hand:
--------------
Four of Diamonds
Three of Diamonds
Five of Spades
--------------

Would you like to HIT or STAY?
```

After the project, the course covered Python decorators. Decorators are a way to modify the functionality of a pre-existing function with a "switch" without needing to rewrite the function entirely. In the image on the right, the function (the decorator) passes in the existing function ("func"), and then creates a function *inside* with the modifications (the print statements, in this case) above and below the passed-in function. At the bottom, *wrap_func()* is returned and the decorator is ready. To apply the decorator, we create a function we want to decorate, *func_needs_decorator()*, and simply write the "@" symbol with the name of the decorator on top of it. As you can see, when we call *func_needs_decorator()* afterwards, Python automatically applies the decorator onto it!

```
def new_decorator(func):

    def wrap_func():
        print("Code would be here, before executing the func")

        func()

        print("Code here will execute after the func()")

    return wrap_func
```

```
@new_decorator
def func_needs_decorator():
    print("This function is in need of a Decorator")
```

```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

The next topic was generators, another "advanced" way of doing something more efficiently. The image on the right shows a generator function that cubes a passed in range of numbers (*n*). While normally it would be necessary to append each result to a list, and then return the list, the *yield* keyword allows you to hold the cubed number and the function's state. In the second box, we see that afterwards we can simply print the yielded number, return back to the function, obtain the next yielded number in the sequence, etc. Without *yield*, *gencubes()* would not remember the cubed numbers at all and the following *for* loop would not print anything. Another way to illustrate this concept is with the image on the left. This example is even simpler;

```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

```
for x in gencubes(10):
    print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

```
def simple_gen():
    for x in range(3):
        yield x
```

```
# Assign simple_gen
g = simple_gen()
```

```
print(next(g))
```

```
0
```

```
print(next(g))
```

```
1
```

```
print(next(g))
```

```
2
```

simple_gen() yields the numbers 0, 1, and 2. The function is then assigned to "*g*". If we immediately printed the statement as *print(g)*, we would be told by Python that "g" is a generator object. However, when we use *next()*, as seen in the following boxes, the first number, 0, is printed. In the next box, with, *next()*, Python returns to the function in the previously left state, obtains the next number, and then prints it. Then, the same happens for 2. Although generators may at first seem not very useful, they can be *very* memory-saving in large programs. In fact, this operation is so common that Python offers a list-comprehension-like way (instead of writing an entire function, it can be done on a single line, with square brackets instead of parentheses; see Entry 1) of creating generators.

Finally, in a *lengthy* section, the course teaches several advanced modules. I will briefly summarize each. First, the Containers module offer specialized versions of basic container data types like dictionaries, lists, sets, tuples, etc. For example, *Counter()* is a way to get the number of each unique object you have in a list, in the form of a dictionary, and *namedtuple()* allows you to create tuples with class-like attributes. Second, the OS module allows you to move and edit files on your computer itself. An interesting method found in this module is *os.walk()*, which essentially allows you to see/view ("walk") through all the folders, subfolders, and files in a directory. Similarly, the *shutil* module allows you to perform similar functions such as unzipping files. Next, the *datetime* module offers unique ways to implement time into Python. What I find most interesting is the ability to insert dates and times (down to microseconds), and importantly, perform *arithmetic* between dates without any special syntax (for example, you can find the difference in minutes). Fourth, the Math module does what you would expect, allowing you to even perform trigonometry, logarithms, etc., and also get the values of irrational numbers such as *pi* and *e*. Fifth, the Regular Expressions module was *by far* the most interesting, offering the ability to search through text with not just for a keyword but for a complex pattern of digits/characters/letters. For example, you can look for any phone number with *re.search()*, as long as you know the pattern in which the digits are written. Finally, the *timeit* module allows you to time how long a program, script, or function takes to run at astonishing precision, although it most likely won't matter at this level.

The section concluded by bringing together many of the modules into a puzzle. The first step involves unzipping a file to get instructions (I unzipped it with the *shutil* module), which tells you the challenge: To find a hidden phone number inside a multitude of long text files full of nonsense - a task


```
import os
import re

for folder, sub_folders, files in os.walk('/Users/kheddie/Documents/Python2/Complete-Pytho
    for i in files:
        if i == '.DS_Store':
            pass
        else:
            temp = open(f'{folder}/{i}')
            opened = temp.read()
            num = re.search(r'\d{3}-\d{3}-\d{4}', opened)
            if num is not None:
                print()
                print(num.group())
                print(i)
                print()
```

that would take a long time to do manually. My solution (shown on the right) used the OS and Regular Expressions modules, specifically *os.walk()* and *re.search()*. The phone number that the program found was 719-266-2837, an Emergency Helpline phone number that plays you a Hall & Oates song.

Thus concludes this log entry! I am a few short sections away from completing the course, after which I will be able to begin the COVID-19 simulation!