Name: _____

Week 6: Examples

We've spent the last few weeks learning a lot of tools in Python. We also spent some time earlier learning about what an algorithm is, and how we can use algorithmic thinking to help us turn our ideas into code. Since then though, we've learned about a lot of new things, such as arrays, functions, and classes. How do we know when and how to use these? The answer is still algorithms! We just have to expand our thinking a little bit. We still want to find a series of steps that solve the problem, we just have easier ways to do some of the things in Python. We can keep track of lists of things in an array instead of needing multiple variables, we can take things we want to do multiple times and write them as functions, and we can take collections of variables and functions that are logically connected and bundle them up into classes. There aren't hard and fast rules for this, but hopefully some examples will help to give you an intuition.

Importing

First though, we're going to need to learn about one more thing: Importing. When we talked about classes, I mentioned that classes allowed us to break our code up into multiple files. This is nice because it lets us reuse code without copy+pasting it into our current file, but it also lets us use code that other people have written by importing their classes. Importing is really easy when you're either importing a class that's included in Python, or a class you wrote that's in the same file: You just type

```
import file_name
```

Once you've done that, you can access the functions and classes in the file with file_name.class_or_function_name. For example, the random file contains a method called randint that will choose a random integer between the two arguments passed to it. (Inclusive!) We would print a random number between 1 and 10 like this:

```
import random

int_number = random.randint(1, 10)
print(str(int_number))
```

There are a lot of other things we can import as well, including importing our own classes and functions, but random is the main one we'll be using for now. Any imported functions or classes

that we use in class will have an included definition of the function/class, so you don't need to worry about memorizing them.

Examples

Example 1: Rolling Dice

Okay, let's start our examples off! The first example we want is to come up with a way to roll dice. (Note that we're not going to print the value out at this point, just generate it.) This doesn't sound too complicated, since we have our random.randint() function. We can just make a variable for our result, and then generate a random number to represent our roll, like this:

```
import random

int_rand_num = random.randint(1, 6)
```

That works okay, but what if we want to roll five dice, a la Yahtzee? Well, if we know we're going to need to store multiple things, that sounds like an array! We'll still need a loop in order to add each die to the array though, and our loop will need to iterate five times, so our code would end up looking like this:

```
import random

die_array = []
int_i = 0
while int_i < 5:
    die_array.append(random.randint(1, 6))
```

Now we have all five of our values! What if we want to re-roll all dice less than 3? Well, we would need to look at each die, and re-roll any die that's less than 3. Doing something for each item sounds like a loop, and the number of time we'll want our loop to repeat will be equal to the size of our array. For each iteration of the loop, we'll access the current element of the array, and if it's less than 3, we'll re-roll it! We can re-use our counter variable int_i, as long as we remember to reset it to 0. Otherwise it would start out at 5! This would look like this:

```
import random

die_array = []
int_i = 0
while int_i < 5:
    die_array.append(random.randint(1, 6))
int_i = 0
```

```
while int_i < len(die_array):
    if die_array[int_i] < 3:
        die_array = random.randint(1, 6)
```

So far so good! But if you notice, we're having to repeat the random.rand_int(1, 6) part of our code. If we're repeating code, that's a good sign we should think about putting it in a function! So we could refactor our code to look like this:

```
import random

die_array = []
int_i = 0
while int_i < 5:
    die_array.append(roll())
int_i = 0
while int_i < len(die_array):
    if die_array[int_i] < 3:
        die_array = roll()

def roll():
    return random.randint(1, 6)
```

That's good, but we can do even better! If we think about it, our die values and the roll function are both related - they're both part of how we're simulating a die. If we think about the concept of a die, it contains both the number, representing which face we rolled, and the ability to roll the die. So since they're logically part of the same object, we can combine them into a class!

```
import random

class Die():

    def __init__(self):
        self.value = random.randint(1, 6)

    def roll(self):
        self.value = random.randint(1, 6)

die_array = []
int_i = 0
while int_i < 5:
    die = Die()
    die_array.append(die)
int_i = 0
while int_i < len(die_array):
    if die_array[int_i].value < 3:
        die_array[int_i].roll()
```

This may look a little more complicated, but that's only because this is a small example. In a small example, the overhead to create a new class exceeds the savings. In a large example, it's the other way around though! You'll also notice here that we're able to roll the die while it was still inside the array.

Patterns (ways of doing something in our program) worth noting:

- Using a counter (int_i) with our while loop, and using that counter to access each element of an array one at a time.
- Modifying class instances inside an array without having to explicitly pull them out and put them back in. (`die_array[int_i].roll()`)

Example 2: Blackjack

We saw poker in our lab, but making a whole game of poker is kinda complicated. Blackjack is way easier though, so let's do that! If you haven't seen blackjack before, it's a game played against a dealer, with each of you beginning with 2 cards. The goal is to get closer to 21 than the dealer by drawing more cards, but with the catch being that if you go over 21, you immediately lose. The dealer will draw until their hand is worth at least 17, and the player can decide when to draw, and when to hold. Face cards are worth 10, and (in our version) Aces are worth 11. Ties go to the dealer.

That sounds complicated! How can we start trying to make that into a program? Well, first let's think about what data we need to keep track of, and how we need to change it. This will correlate with our variables and functions. For data, we've got the player's total, and the dealer's total. For changes, we need to be able to draw cards and add them to the player's hand.

If you notice, we're talking about the player's hand as a conceptual object that we want to track and modify. That's exactly the sort of situation we have classes for! So we can make a Hand() class, and then have one instance for the player, and one for the dealer. Our hand class will need a total variable, and a draw() method, and it should start with two card in it when we create it. Given this information, let's look at what the class might look like in Python:

```
import random

class Hand():

    def __init__(self):
        self.total = 0
        self.draw()
        self.draw()

    def draw(self):
```

```
        card_values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]
        value_drawn = card_values[random.randint(0, len(card_values)-1)]
        self.total += value_drawn
```

That wasn't so hard! Things to note here are the fact that we call another class method in our __init__() function, and the fact that we have to use len(card_values)-1 because random.randint() includes the second parameter, and if we remember, the indexes of an array always go up to, but **don't** include, the length of the array, due to the fact we start counting at 0.

Now we have to try to actually make the game. Well, first off, we're definitely going to need two hands. Then, we'll need to let each player draw cards, one after the other. The player will draw first, until either the player decides to stop, or they go over 21, immediately losing. If the player didn't lose, the dealer will then draw until their total is over 17, possibly immediately losing if they go over 21. If the dealer didn't immediately lose, we will then compare the values of the players' hands, and the higher hand value wins, with ties going to the dealer. Let's try to write this out!

```
import random

class Hand():

    def __init__(self):
        self.total = 0
        self.draw()
        self.draw()

    def draw(self):
        card_values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]
        value_drawn = card_values[random.randint(0, len(card_values)-1)]
        self.total += value_drawn

def get_player_input():
    print("Your hand total is " + str(player.total))
    return input("Draw or hold?")

player = Hand()
dealer = Hand()

command = get_player_input()
while command == "draw" and player.total <= 21:
    player.draw()
    if player.total > 21:
        print("Your total is now " + str(player.total)))
        print("Player is over 21!")
    else:
```

```
        command = get_player_input()

if not player.total > 21:
    while dealer.total < 17:
        dealer.draw()
        print("Dealer's total is now " + str(dealer.total)))
        if player.total > 21:
            print("Dealer is over 21!")

if player.total > 21:
    print("Dealer wins!")
elif dealer.total > 21:
    print("Player wins!")
else:
    if player.total > dealer.total:
        print("Player wins!")
    else:
        print("Dealer wins!")
```

And there we go! We've made a simple blackjack game! Things to notice in this code:

- Because we wanted to print out multiple things when asking the user for input, we factored it out into a function. (get_player_input() is **not** inside the class)
- We wanted to loop as long as the player kept entering "draw", and we could have done this a couple of ways. We could have made a variable called `keep_going`, and started it off as True, then used that to control our loop, re-evaluating it at the end of each loop, for instance. Instead, we did what's called a "priming read", where we get input from the user once before starting our loop. This way, we already have input from the user when we get to our loop the first time!
- At the end, when we're comparing the player and the dealer's totals, we only check if `player.total` is larger than `dealer.total`. This is because we want to do the same thing if the dealer and the player tie, or if the dealer has more. So we can just use an `else`!

Example 3: Paper Company Customers

Let's imagine that we're working for a paper company, and that this paper company has a number of customers they need to keep track of. Each customer has a price at which they buy paper, and an amount of paper they purchase. The prices are different for each customer, since, e.g., a large company will get a volume discount. We want to keep track of how much each customer pays per ream and how much they buy, as well as what our total volume of paper sold is and what the total income we get from our customers is. How can we start thinking about this?

Well, right off the bat, we see that we have a logical “customer” object, so we probably want to make that a class. The class will need to keep track of the price per ream we charge the customer, as well as how much they buy, and have a method we can call to calculate how much they are paying us. Let’s make that class:

```
class Customer():

    def __init__(self, name, cost, amount):
        self.name = name
        self.cost = cost
        self.amount = amount

    def get_bill(self):
        return self.cost * self.amount
```

Not too complicated! We need an initial cost and amount for each customer, as well as their name, and we can calculate their bill with an easy method call.

We’ll also need some logic to use the class, creating some customers for us, and printing out the various costs and amounts for each customer, as well as our total volume and profit. Since we’ll be printing out the details of each customer, we’ll want to put the code to do that into a function. Since the function is something we effectively do to a customer, we probably want to put this function inside of our Customer class! This illustrates how sometimes we have to go back and update our classes later on, when we realize that there’s more code we should put inside it. Moving on, doing the printing for each customer sure sounds like a loop, and we can also use the loop to add up our total volume and profit. Let’s see how that would look:

```
class Customer():

    def __init__(self, name, cost, amount):
        self.name = name
        self.cost = cost
        self.amount = amount

    def get_bill(self):
        return self.cost * self.amount

    def print_details(self):
        print("Customer " + self.name + " buys " + str(self.amount) + " reams of
paper, at " + str(self.cost) + " per ream.")

white_pages = Customer("White Pages Phonebooks", 3.14, 10000)
prestige = Customer("Prestige Postal Company", 4.00, 4500)
harper_collins = Customer("Harper Collins", 4.25, 2000)
customers = [white_pages, prestige, harper_collins]

total_volume = 0
```

```
total_profit = 0
for int_i in range(0, len(customers)):
    customer = customers[int_i]
    customer.print_details()
    total_volume += customer.amount
    total_profit += customer.get_bill()
print("Our total volume is " + str(total_volume))
print("Our total profit is " + str(total_profit))
```

Things to note here:

- We didn't read in our data from the user! We could have though, and we might want to in the future, which is why we use `len(customers)` instead of hard coding a `3`.
- We use a for loop here, and we could have used it to loop directly over the values in our array, but using it this way is more like how for loops work in other languages - just a convenient way to do something a certain number of times. This will result in `int_i` going from 0 to len(customers) - 1. (Which is 0, 1, 2 in this case.) Those are exactly the indexes of our customers array, so we can use them to iterate over our array!
- We call our array the plural form of what's inside it, and use the singular form to reference a single element from the array when looping over it.
- We declare some counter variables before going into our loop, and then use them to keep track of our totals as we loop over the customers. We have to declare them before the loop, so that we can use them afterwards, and because if we declared them inside the loop they would be reset on each iteration!