

CS 112 – Spring 2020 – Programming Assignment 3

Branching

Due Date: Sunday, February 16th, 11:59pm

The purpose of this assignment is to get more practice with boolean logic and branching.

See the “**assignment basics**” file for more detailed information about getting assistance, running the test file, grading, commenting, and many other extremely important things. Each assignment is governed by the rules in that document.

- https://piazza.com/class/profile/get_resource/k4wblafqtoj2cx/k5x3ennufm34cq

Needed file: the tester for this assignment

- **tester3.py** https://piazza.com/class/profile/get_resource/k4wblafqtoj2cx/k6har39pso85hy

Background

Selection statements (**if/elif/else** combinations) allow us to write code that can execute different statements based on the current values seen in a particular run of the program. We will use this to write a program that performs calculations and selectively reports on different properties of the calculated values.

Guidelines

- Think carefully about the order you will check different properties. You might want to write some pseudocode or perhaps draw a flowchart if this works better for you. Think first, then implement.
- Be careful what kinds of selection statements you use, and be sure to test your code with many examples. For instance, multiple **if** statements will not necessarily behave the same as a chain of **if-elif-elif**.
- When a specific test case isn't working, plug your code into the visualizer to watch what the code does, which lines run, which branches are taken.
- From built-in functions, you are allowed to call **abs()**, **int()**, **float()**, **str()** only.
- You are **not** allowed to **import** anything.
- You are **not** allowed to use loops, lists, sets, dictionaries and any feature that hasn't been covered in class yet
- Don't forget to review the “**assignment basics**” file
- Insert comments in the lines you deem necessary





















































	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
	1	2	3	4	5	6	7	8	9	10	11	12	13
Diamonds													
	14	15	16	17	18	19	20	21	22	23	24	25	26
Hearts													
	27	28	29	30	31	32	33	34	35	36	37	38	39
Spades													
	40	41	42	43	44	45	46	47	48	49	50	51	52

Table 2

Assumptions

You may assume that:

- The types of the values that are sent to the functions are the proper ones (card1 is an integer not a float, etc.), you don't have to validate them.
- The functions are going to be called with usable values (e.g. card1 is not negative, etc.), you don't have to validate them.
- All function parameters are guaranteed to be unique, you don't have to validate them either.

16, 1, 2

Scenario

You're working for a software company and your manager asks you to implement a series of functions that another team will use to build a **Three Card Poker** game application. Three Card Poker is similar to the traditional poker but it's played with 3 instead of 5 cards, which makes it simpler as it has fewer hands to consider. For more details about the game read this article:

https://en.wikipedia.org/wiki/Three_Card_Poker

Three Card Poker has the following six hands in descending order:

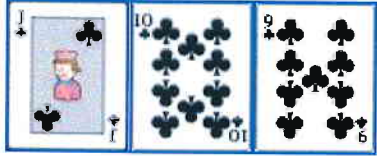
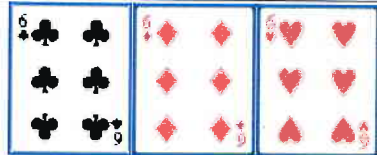
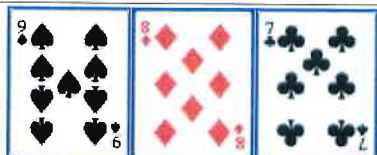
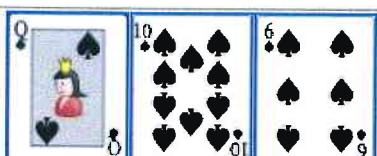
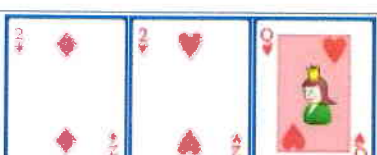

Rank	Description	Example
Straight flush	Three suited cards in sequence	
Three of a kind	Three cards of same rank	
Straight	Three cards in sequence	
Flush	Three suited cards	
Pair	Two cards of same rank	
High card	None of the above	

Table 1

Your task is to implement for each of these hands one function that takes three parameters (numbers that represent the three cards) and returns a boolean value, True or False, depending on whether the three cards make up the respective hand or not. We will use the Table 2 mapping to represent the standard 52-card deck with integers. Ace will always **rank low** (i.e. be the lowest card when considering a sequence for straight).

Testing

In this assignment testing will be done the same way as in the previous one. There will be **no user input or print** statements. You're given a number of tasks and for each of them you must implement one Python function. The tester will be calling your functions with certain arguments and will be examining the return values to decide the correctness of your code. Your functions **should not ask for user input and should not print anything**, just return a value based on the specification of the tasks.

Grading Rubric

Submitted correctly:	2 # see assignment basics file for file requirements!
Code is well commented:	8 # see assignment basics file for how to comment!
Tester calculations correct:	90 # see assignment basics file for how to test!
TOTAL:	100

Note: If your code does not run and crashes due to errors, it will receive **zero** points. Turning in running code is essential.

Return value: **True** or **False**

Examples:

flush(41, 40, 42) → **False** # straight flush

pair(card1, card2, card3)

Description: The function checks whether the three cards make up a pair hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

Return value: **True** or **False**

Examples:

pair(33, 46, 20) → **False** # three of a kind

high_card(card1, card2, card3)

Description: The function checks whether the three cards make up a high card hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

Return value: **True** or **False**

Examples:

high_card(41, 5, 21) → **True**

Helper functions – OPTIONAL

If you want to avoid repeating some computations again and again, you're allowed to create your own *helper* functions. This is **optional**, you don't need that to make your code run correctly. Some functions that you might find useful to implement are the following:

suit(card)

Description: Given a card number it returns the suit of the card as a string

value(card)

Description: Given a card number it returns the face value of the card as an int

sequence(card1, card2, card3)

Description: Given three card numbers it returns a boolean depending on whether the cards form a sequence or not

same_suit(card1, card2, card3)

Description: Given three card numbers it returns a boolean depending on whether the cards have the same suit or not

Functions

The signature of each function is provided below, do **not** make any changes to them otherwise the tester will not work properly. Keep in mind that you must **not** write a main body for your program. You should only implement these functions; it is the tester that will be calling and testing each one of them. The following are the functions you must implement:

straight_flush(card1, card2, card3)

Description: The function checks whether the three cards make up a straight flush hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

Return value: **True** or **False**

Examples:

straight_flush(2, 1, 3) → True

three_of_a_kind(card1, card2, card3)

Description: The function checks whether the three cards make up a three of a kind hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

Return value: **True** or **False**

Examples:

three_of_a_kind(14, 27, 1) → True

straight(card1, card2, card3)

Description: The function checks whether the three cards make up a straight hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

Return value: **True** or **False**

Examples:

straight(34, 33, 35) → False # straight flush

flush(card1, card2, card3)

Description: The function checks whether the three cards make up a flush hand, and only that.

Parameters: **card1** (int), **card2** (int), **card3** (int) are the three cards represented with the mapping provided in Table 2.

CS 112 – Spring 2020 – Programming Assignment 4

Loops

Due Date: Sunday, February 23rd, 11:59pm

The purpose of this assignment is to get practice with loops

See the “**assignment basics**” file for more detailed information about getting assistance, running the test file, grading, commenting, and many other extremely important things. Each assignment is governed by the rules in that document.

- https://piazza.com/class/profile/get_resource/k4wblafqtoj2cx/k5x3ennufm34cg

Needed file: the tester for this assignment

- **tester4.py** https://piazza.com/class/profile/get_resource/k4wblafqtoj2cx/k6rgib2nqsq3ar

Background

Loops allow us to run the same block of code multiple times in a row. When variables used in that block of code are changed between iterations, we can accumulate effects and process larger batches of data, such as the elements in a sequence (e.g. list, string, file, etc.). There are problems that selection statements alone are insufficient to solve; yet loops add enough power to tackle them. We will use loops to solve various problems.

Guidelines

- You **must** have at least one loop in each function.
 - You are **not** allowed to **import** anything.
 - From built-in functions, you are allowed to call **int()** and **range()** only.
 - You are **not** allowed to use strings, lists, tuples, sets, dictionaries
 - You are **not** allowed to use anything that hasn't been covered in class (e.g. list comprehension)
 - Don't forget to review the “**assignment basics**” file
 - Insert comments in the lines you deem necessary
-

Testing

In this assignment testing will be done the same way as in the previous one. There will be **no user input or print** statements. You're given a number of tasks and for each of them you must implement one Python function. The tester will be calling your functions with certain arguments and will be examining the return values to decide the correctness of your code. Your functions **should not ask for user input and should not print anything**, just return a value based on the specification of the tasks.

As for the *extra credit*, the testing is a bit different in that function. Some of the test cases we'll be using for grading it are omitted from the tester. This means that there might be errors in your code even if the tester is giving you no errors. You must do your own checks to make sure that you haven't missed anything and your code is correct. You do **not** need to modify the tester, just test on your own any way you like.

Grading Rubric

Submitted correctly:	2 # see assignment basics file for file requirements!
Code is well commented:	8 # see assignment basics file for how to comment!
Tester calculations correct:	90 # see assignment basics file for how to test!
Extra credit:	10 # 2 tests included in the tester, 8 tests are hidden
TOTAL:	110

Note: If your code does not run and crashes due to errors, it will receive **zero** points. Turning in running code is essential.

Functions

The signature of each function is provided below, do **not** make any changes to them otherwise the tester will not work properly. Keep in mind that you must **not** write a main body for your program. You should only implement these functions; it is the tester that will be calling and testing each one of them. The following are the functions you must implement:

count_multiples(num1, num2, N)

Description: Returns the number of multiples of **N** that exist between **num1** and **num2** inclusive.

Parameters: **num1** (int), **num2** (int), **N** (int). **num1** and **num2** can be in any order. **N** cannot be 0.

Return value: int

Examples:

<code>count_multiples(2, 13, 3)</code>	→	4
<code>count_multiples(17, -5, 4)</code>	→	6
<code>count_multiples(-10, -5, -3)</code>	→	2

skip_sum(num1, num2, N)

Description: Returns the sum of the integers between **num1** and **num2** inclusive but it skips every **Nth** number in this sequence.

Parameters: **num1** (int), **num2** (int), **N** (int). **num1** and **num2** can be in any order. **N** is always larger than 1.

Return value: int

Examples:

<code>skip_sum(2, 14, 3)</code>	→	70	# =2+3+5+6+8+9+11+12+14		(4,7,10,13 skipped
<code>skip_sum(4, -3, 2)</code>	→	0	# =-3-1+1+3		-2,0,2,4 skipped

cubic_root(num)

Description: Given a positive number **num**, it returns its cubic root only if it's a whole number. You're not allowed to use the ****** operator.

Parameters: **num** (int)

Return value: The cubic root as int otherwise **None**

Examples:

<code>cubic_root(8)</code>	→	2
<code>cubic_root(125)</code>	→	5
<code>cubic_root(10)</code>	→	None

2, 4,

4, 3, 2, 0

count_moves(start, end, step)

Description: It counts how many moves you need to get from **start** to **end** with a step of **step** by following two rules: 1) if a move lands you on a number ending with 3 or 5 your step is increased by 3 or 5 respectively, 2) if a move lands you on a number ending with 7 or 8 you're automatically transferred ahead by 7 or 8 positions respectively and this transfer doesn't count as a move neither as a landing on a new number. The last move can land either on **end** or any number larger than this.

Parameters: **start** (int), **end** (int), **step** (int) are all positive. **end** is larger or equal to **start**.

Return value: int

Examples:

count_moves(1, 30, 1) → 5 # 1 → 2 → 3 → 7 → 14 → 18 → 26 → 30
count_moves(2, 17, 2) → 4 # 2 → 4 → 6 → 8 ~ 16 → 18

max_dna(num)

Description: In a DNA sequence (e.g. AAAGTCTGAC) we use the letters A, C, G, T to represent the four nucleotide bases. But in order to make the storage more efficient we are going to represent a sequence with a number instead of a string. To do that, we'll use the following mapping: A → 1, C → 2, G → 3, T → 4. Therefore, the sequence AAAGTCTGAC will be represented with the integer 1113424312. Given a DNA sequence represented by **num**, this function returns its most frequent nucleotide base (assume there is no tie).

Parameters: **num** (int)

Return value: string

Examples:

max_dna(1113424312) → 'A'
max_dna(42133133243132) → 'G'

Extra credit (10 pts)

scrabble_number(num)

Description: Returns a scrabbled version of **num** by swapping the two digits of every consecutive pair of digits starting from the right.

Parameters: **num** (int)

Return value: int

Examples:

scrabble_number(123456) → 214365
scrabble_number(4739917) → 4379971

the tester contains only 2 out of 10 test cases that will be used for grading this function