CSE101 – Fall 2019
Programming Assignment #2

Due October 17 2019 by 11:59pm, KST. The assignment is worth 25 points.

Instructions

For each of the following problems, create an error-free Python program.

- Each program should be submitted in a separate Python file that follows a particular naming convention: Submit the answer for problem 1 as "Assign2Answer1.py" and for problem 2 as "Assign2Answer2.py" and so on.
- These programs should execute properly in PyCharm using the setup we created in lab.
- At the top of every file add your name and Stony Brook email address in a comment.
- Please provide at least 2 test cases for each problem. This means calling the function you have created with an example input, so that when I run the entire program, I see the output of your test cases in the console.
- Please continue to use the naming conventions in Python and programming style that was mentioned in Assignment 1.

Problems

Problem 1: (4 points)

A Mersenne prime is a prime number that is one less than a power of 2. For example, 131071 is a Mersenne prime because 131071 + 1 = 2^{17}. Write a function named `mersennes` that takes an integer as a parameter and returns a list of all Mersenne primes less than the specified value.

For example, this call makes a list of Mersenne primes less than 1000:

```
>>> mersennes(1000)
[3, 7, 31, 127]
```

Problem 2: (3 points)

Define a new function named `collatz` that takes one positive integer as a parameter and returns a list of numbers, starting with the initial number. Then use a while loop to add to the list using the following process:

- Let `n` be the value currently at the end of the list.
- If n is even, add to the list the number: `(n // 2)`
- If n is odd, add to the list the number: `(3*n + 1)`
- If the number 1 is added to the list, the function should return the entire list.

This problem is known as the Collatz conjecture and for any value of n, the list will eventually get to a 1 (though this hasn't been formally proven)

Here are some examples of output:

```
>>> collatz(5)
[5, 16, 8, 4, 2, 1]
>>> collatz(6)
[6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Problem 3: (3 points)

Create a function named `concat` that concatenates a varying number of input lists. You can define the function as `def concat(*argv)` to signify a parameter that has a *variable* number of arguments. That is, each time you call the function, you can use a different number of input parameters.

Lists should be concatenated in order of the arguments. Here are some examples:

```
>>> concat([1, 2, 3], [4, 5], [6, 7])
[1, 2, 3, 4, 5, 6, 7]
>>> concat([1], [2], [3], [4], [5], [6], [7])
[1, 2, 3, 4, 5, 6, 7]
>>> concat([6, 2], [1, 4])
[6, 2, 1, 4]
>>> concat([4, 4, 4, 4, 4])
[4, 4, 4, 4, 4]
```

Problem 4: (4 points)

Create a function `longestSubstring` that takes as input a string parameter and returns the longest substring that contains no consecutive duplicate characters as a list. If there are two or more substrings with the same length, then return them all as separate items in the list. If there is only one longest substring, return it as a list of length 1. Here are some examples:

```
>>>longsub('abcbba')

['abcb']

>>>longsub('abcdeffghijkkklmnopqr')

['klmnopqr']

>>>longsub('abcdeffghijkkklm')

['abcdef', 'fghijk']
```

Problem 5: (4 points)

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort (such as starting a new line). Refer to the ASCII table below:

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

In Python, you can use the function **`chr(num)`** to print the character corresponding to a specific ASCII number. For example:

```
>>>chr(65)

'A'
```

Similarly, a character can be converted to ASCII number using the function **`ord(character)`**

```
>>>ord('A')

65
```

Write a function named **`def asciiTriangle(numberOfLines)`** which will print an inverted triangle of ASCII characters for the specified number of lines. Check for the following condition: `numberOfLines` should be more than 0 and less than 14.

The start of the triangle should be the ASCII character '!' which can be printed using `chr(33)`. Each following character should be the next ASCII character in the table. The first line contains the same number of characters as the total number of lines, and each successive line has 1 fewer character.

Below are two examples of the expected output:

```
>>>asciiTriangle(6)
! " # $ % &
' ( ) * +
, - . /
0 1 2
3 4
5


>>>asciiTriangle(13)
! " # $ % & ' ( ) * + , -
. / 0 1 2 3 4 5 6 7 8 9
: ; < = > ? @ A B C D
E F G H I J K L M N
O P Q R S T U V W
X Y Z [ \ ] ^ _
` a b c d e f
g h i j k l
m n o p q
r s t u
v w x
y z
{
```

Problem 6: (7 points)

Tic-tac-toe is a two-player board game where each player takes turns marking spaces on a 3x3 grid. The first player marks with an 'X' and the other marks with an 'O'. The player who gets three X's or O's in a row (horizontal, vertical, or diagonal) wins. Otherwise if the board gets filled and neither player can win, it is a draw. For more information on Tic-tac-toe, check out the wiki page.

You will build the tic-tac-toe game in multiple steps. For the game board, you will use lists inside of a list to represent each row and column. Part of the game is already implemented in this Pythøn File. Download the file and open it in PyCharm.

Please refer to the code and code comments for more details. The code already has the initial board set up and most of the general structure in place. Your task is broken into two parts:

Part 1: Validate the user input on each move and update the board appropriately.

A. The user is asked for input on which row and column they want to select, which can be integers between 1 and 3, and must currently be a blank space. Find the "TODO Part 1a" line and write code in the `isBoardInputValid` function to validate if their input meets those criteria. Return `True` if it is a valid range and the corresponding board position is available (represented by `' '`, a blank string in the board). Otherwise the function should return `False`.
B. Now that you have validated the user's move, you will write code to update the board with the player's letter in their chosen position. Find the "TODO Part 1b" line and write code to update the selected board position with the user's input, setting the specified space equal to the player's letter. Once you have this working, you should be able to run the program and see the grid update with 'X's and 'O's when you choose valid positions.

   Remember that lists in Python start at index 0, though the user input is in the range from 1-3, so you will need to make sure to account for that difference in your solution.

Part 2: Now we can play the game, but we need to be able to determine the winner. Find the "TODO Part 2" line and implement the `isPlayerWinner(board, letter)` function. Return `True` if there are 3 of the passed in letter string (either 'X' or 'O') in a row (horizontal, vertical and diagonal orientations all count). Otherwise return `False`. (Hint: there should be 8 different win configurations).

Please note, you do not need to include test cases for this problem when you turn it in. However, it would be good programming practice to create test cases while you are working on the `isPlayerWinner` function so you can confirm you are accounting for all win configurations.

Once you do this, you have a working implementation of tic-tac-toe. Congratulations!