ITI 1120

Lab # 6

More programming with lists

Arguably, this is the most important lab of this semester

Starting Lab 6

- Open a browser and log into Brightspace
- On the left hand side under Labs tab, find lab 6 material contained in lab6-students.pdf file
- Download that file to the Desktop.

Before starting, always make sure you are running Python 3

This slide is applicable to all labs, exercises, assignments … etc

ALWAYS MAKE SURE FIRST that you are running Python 3

That is, when you click on IDLE (or start python any other way) look at the first line that the Python shell displays. It should say Python 3 (and then some extra digits)

If you do not know how to do this, read the material provided with Lab 1. It explains it step by step

Task (to be completed at home)

I strongly encourage you to complete these two quizes at home. Leave the time in the lab for questions to TAs about quiz problems whose solution you do not understand.

Go to coursera webpage and log in.

1.Go to the following link to and complete the Quiz of Week 4. You will find the quiz at the bottom of the page
https://www.coursera.org/learn/learn-to-program/home/week/4

2.Go to the following link to and complete the Quiz of Week 5. You will find the quiz at the bottom of the page
https://www.coursera.org/learn/learn-to-program/home/week/5

- You can do each quiz more than once.

Programming Exercises (the most important lab)

The following exercises are easily the most important exercises in this whole semester. Solving these problems (by yourself preferably) should greatly increase your understanding of computational problem solving and programming.

Do as many as possible (preferably all) of the following 13 programming exercises from your textbook by Perkovic . 11 out of 13 are mandatory for this lab (your choice which 11) -- see the slides to come.

Introduction to Computing Using Python: An Application Development Focus, 2nd Edition, Ljubomir Perkovic

Sometimes the author uses a word "outputs". By that he means "prints" First recall from the next 4 slides, list (and few string) functions and methods that you will need.

List operators and functions

Like strings, lists can be manipulated with operators and functions

Usage	Explanation
`x in lst`	`x` is an item of `lst`
`x not in lst`	`x` is not an item of `lst`
`lst + lstB`	Concatenation of `lst` and `lstB`
`lst*n, n*lst`	Concatenation of `n` copies of `lst`
`lst[i]`	Item at index `i` of `lst`
`len(lst)`	Number of items in `lst`
`min(lst)`	Minimum item in `lst`
`max(lst)`	Maximum item in `lst`
`sum(lst)`	Sum of items in `lst`

```
>>> lst = [1, 2, 3]
>>> lstB = [0, 4]
>>> 4 in lst
False
>>> 4 not in lst
True
>>> lst + lstB
[1, 2, 3, 0, 4]
>>> 2*lst
[1, 2, 3, 1, 2, 3]
>>> lst[0]
1
>>> lst[1]
2
>>> lst[-1]
3
>>> len(lst)
3
>>> min(lst)
1
>>> max(lst)
3
>>> sum(lst)
6
>>> help(list
...
```

Lists methods

Usage	Explanation
`lst.append(item)`	adds `item` to the end of `lst`
`lst.count(item)`	returns the number of times `item` occurs in `lst`
`lst.index(item)`	Returns index of (first occurrence of) `item` in `lst`
`lst.pop()`	Removes and returns the last item in `lst`
`lst.remove(item)`	Removes (the first occurrence of) `item` from `lst`
`lst.reverse(item)`	Reverses the order of items in `lst`
`lst.sort(item)`	Sorts the items of `lst` in increasing order

Methods **`append()`**, **`remove()`**, **`reverse()`**, and **`sort()`** do not return any value; they, along with method `pop()`, modify list **`lst`**

```
>>> lst = [1, 2, 3]
>>> lst.append(7)
>>> lst.append(3)
>>> lst
[1, 2, 3, 7, 3]
>>> lst.count(3)
2
>>> lst.remove(2)
>>> lst
[1, 3, 7, 3]
>>> lst.reverse()
>>> lst
[3, 7, 3, 1]
>>> lst.index(3)
0
>>> lst.sort()
>>> lst
[1, 3, 3, 7]
>>> lst.remove(3)
>>> lst
[1, 3, 7]
>>> lst.pop()
7
>>> lst
[1, 3]
```

String operators

Usage	Explanation
`x in s`	`x` is a substring of `s`
`x not in s`	`x` is not a substring of `s`
`s + t`	Concatenation of `s` and `t`
`s * n, n * s`	Concatenation of `n` copies of `s`
`s[i]`	Character at index `i` of `s`
`len(s)`	(function) Length of string `s`

To view all **operators, use the** `help()` tool

```
>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(string[, encoding[, errors]]) -> str
...
```

```
>>> 'Hello, World!'
'Hello, World!'
>>> s = 'rock'
>>> t = 'climbing'
>>> s == 'rock'
True
>>> s != t
True
>>> s < t
False
>>> s > t
True
>>> s + t
'rockclimbing'
>>> s + ' ' + t
'rock climbing'
>>> 5 * s
'rockrockrockrockrock'
>>> 30 * '_'
'______________________________'
>>> 'o' in s
True
>>> 'o' in t
False
>>> 'bi' in t
True
>>> len(t)
8
```

String methods

Strings are immutable; none of the string methods modify string

link

Usage	Explanation
`s.capitalize()`	returns a copy of `s` with first character capitalized
`s.count(target)`	returns the number of occurences of `target` in `s`
`s.find(target)`	returns the index of the first occurrence of `target` in `s`
`s.lower()`	returns lowercase copy of `s`
`s.replace(old, new)`	returns copy of `s` with every occurrence of `old` replaced with `new`
`s.split(sep)`	returns list of substrings of `s`, delimited by `sep`
`s.strip()`	returns copy of `s` without leading and trailing whitespace
`s.upper()`	returns lowercase copy of `s`

5.16 Implement function `indexes()` that takes as input a word (as a string) and a one-character letter (as a string) and returns a list of indexes at which the letter occurs in the word.

```
>>> indexes('mississippi', 's')
[2, 3, 5, 6]
>>> indexes('mississippi', 'i')
[1, 4, 7, 10]
>>> indexes('mississippi', 'a')
[]
```

5.17 Write function `doubles()` that takes as input a list of integers and outputs the integers in the list that are exactly twice the previous integer in the list, one per line.

```
>>> doubles([3, 0, 1, 2, 3, 6, 2, 4, 5, 6, 5])
2
6
4
```

5.18 Implement function `four_letter()` that takes as input a list of words (i.e., strings) and returns the sublist of all four letter words in the list.

```
>>> four_letter(['dog', 'letter', 'stop', 'door', 'bus', 'dust'])
['stop', 'door', 'dust']
```

5.19 Write a function `inBoth()` that takes two lists and returns `True` if there is an item that is common to both lists and `False` otherwise.

```
>>> inBoth([3, 2, 5, 4, 7], [9, 0, 1, 3])
True
```

5.20 Write a function `intersect()` that takes two lists, each containing no duplicate values, and returns a list containing values that are present in both lists (i.e., the intersection of the two input lists).

```
>>> intersect([3, 5, 1, 7, 9], [4, 2, 6, 3, 9])
[3, 9]
```

5.21 Implement the function `pair()` that takes as input two lists of integers and one integer n and prints the pairs of integers, one from the first input list and the other from the second input list, that add up to n. Each pair should be printed.

```
>>> pair([2, 3, 4], [5, 7, 9, 12], 9)
2 7
4 5
```

5.22 Implement the function `pairSum()` that takes as input a list of distinct integers `lst` and an integer `n`, and prints the indexes of all pairs of values in `lst` that sum up to `n`.

```
>>> pairSum([7, 8, 5, 3, 4, 6], 11)
0 4
1 3
2 5
```

5.29 Write function `lastfirst()` that takes one argument—a list of strings of the format `<LastName, FirstName>`—and returns a list consisting two lists:

(a) A list of all the first names

(b) A list of all the last names

```
>>> lastfirst(['Gerber, Len', 'Fox, Kate', 'Dunn, Bob'])
[['Len', 'Kate', 'Bob'], ['Gerber', 'Fox', 'Dunn']]
```

5.31 Write a function `subsetSum()` that takes as input a list of positive numbers and a positive number `target`. Your function should return `True` if there are three numbers in the list that add up to `target`. For example, if the input list is `[5, 4, 10, 20, 15, 19]` and `target` is 38, then `True` should be returned since $4 + 15 + 19 = 38$. However, if the input list is the same but the target value is 10, then the returned value should be False because 10 is not the sum of any three numbers in the given list.

```
>>> subsetSum([5, 4, 10, 20, 15, 19], 38)
True
>>> subsetSum([5, 4, 10, 20, 15, 19], 10)
False
```

5.33 Implement a function `mystery()` that takes as input a positive integer n and answers this question: How many times can n be halved (using integer division) before reaching 1? This value should returned.

```
>>> mystery(4)
2
>>> mystery(11)
3
>>> mystery(25)
4
```

5.46 An inversion in a sequence is a pair of entries that are out of order. For example, the characters F and D form an inversion in string `'ABBFHDL'` because F appears before D; so do characters H and D. The total number of inversions in a sequence (i.e., the number of pairs that are out of order) is a measure of how *unsorted* the sequence is. The total number of inversions in `'ABBFHDL'` is 2. Implement function `inversions()` that takes a sequence (i.e., a string) of uppercase characters A through Z and returns the number of inversions in the sequence.

```
>>> inversions('ABBFHDL')
2
>>> inversions('ABCD')
0
>>> inversions('DCBA')
6
```

5.48 Let `list1` and `list2` be two lists of integers. We say that `list1` is a sublist of `list2` if the elements in `list1` appear in `list2` in the same order as they appear in `list1`, but not necessarily consecutively. For example, if `list1` is defined as

```
[15, 1, 100]
```

and `list2` is defined as

```
[20, 15, 30, 50, 1, 100]
```

then `list1` is a sublist of `list2` because the numbers in `list1` (15, 1, and 100) appear in `list2` in the same order. However, list

```
[15, 50, 20]
```

is not a sublist of `list2`.

Implement function `sublist()` that takes as input lists `list1` and `list2` and returns `True` if `list1` is a sublist of `list2`, and `False` otherwise.

```
>>> sublist([15, 1, 100], [20, 15, 30, 50, 1, 100])
True
>>> sublist([15, 50, 20], [20, 15, 30, 50, 1, 100])
False
```

maximum

5.37 Write function `mssl()` (~~minimum~~ sum sublist) that takes as input a list of integers. It then computes and returns the sum of the maximum sum sublist of the input list. The maximum sum sublist is a sublist (slice) of the input list whose sum of entries is largest. The empty sublist is defined to have sum 0. For example, the maximum sum sublist of the list

```
[4, -2, -8, 5, -2, 7, 7, 2, -6, 5]
```

is `[5, -2, 7, 7, 2]` and the sum of its entries is 19.

```
>>> l = [4, -2, -8, 5, -2, 7, 7, 2, -6, 5]
>>> mssl(l)
19
>>> mssl([3,4,5])
12
>>> mssl([-2,-3,-5])
0
```

In the last example, the maximum sum sublist is the empty sublist because all list items are negative.