AI – Python lab 2

AI – 10/11 Sept, 2020

Your second Python lab is to solve all the level 2 problems under the Python tab at CodingBat.com. That means tabs Warmup-2, Logic-2, String-2, and List-2. The complete list of problems appears below.

This lab is expected to be assigned on the second day of class. It is similar to the first lab, only the problems are more difficult, and the criteria are also more stringent. In particular, the goal is to be able to write each definition with a single return statement. Certain characters / constructs are not allowed:

Semicolons	Tabs	Imports
Triple quoted strings	): except at end of line	

and `else` is strongly discouraged. See the end of this doc for scoring details.

Specific problems:

Warmup-2

`string_times`	`front_times`	`string_bits`
`string_splosion`	`last2`	`array_count9`
`array_front9`	`array123`	`string_match`

Logic-2

`make_bricks`	`lone_sume`	`lucky_sum`
`no_teen_sum`	`round_sum`	`close_far`
`make_chocolate`		

String-2

`double_char`	`count_hi`	`cat_dog`
`count_code`	`end_other`	`xyz_there`

List-2

`count_evens`	`big_diff`	`centered_average`
`sum_13`	`sum67`	`has22`

In this class, we will use the following conventions:
Indent levels should be either 2, 3, or 4 spaces and do not employ tabs (one can be difficult to discern and 8 leads to code that is too wide).

Function definitions should have a blank line preceding them and following them. This will help your instructor spot problems.

Slices (corresponding to ranges) of lists or strings

Slices come in the form of [#], [#:#], [#:#:#] appearing at the end of a list or string (eg. `myList[1:]`) and specify the indices for a sublist, working similarly to how a range works. In most cases when the # is omitted, python will put in a value that makes the most sense.

A single # ([#]) picks out the item at the given index. In the case of a string, you get a new string of length 1. When there are one or two colons, then you get a new list. The first number acts as the starting index, the second one indicates the ending position (as with ranges), while the third indicates the step size. Strings, for our purposes, can be viewed as a list of characters. However, strings are immutable so that you cannot update a single character of a string – you would have to make a copy of the string with the one character altered.

When omitted, Python defaults the slice specs to the most reasonable number. If the step size is positive (the default being 1, of course), then the default starting index is 0 and the default ending index is the length of the string. If the step size is negative, the default starting index is -1 (that is to say, the end of the string), while the default ending index implies the position just left of the start of the string. Note that a negative index counts from the right side of the string or list so that -1 means the final character or list item and -2 indicates the index of the penultimate element.

Some examples follow. When reviewing the examples below, it makes sense to cover up the left column to see if you can determine what it is, prior to seeing it.

`myList[:5]`	The first 5 elements of the list
`myList[1::2]`	The elements of the list which have an odd numbered index
`myList[::-1]`	Reverses the list
`myList[-3:]`	Last three items of the list
`myList[:-4:-1]`	Last three items of the list in reverse order.
`myList[-3:][::-1]`	Poor man's version of the last three items of the list in reverse order
`myList[:-3]`	All but the last three items of the list
`myList[-4::-1]`	All but the last three items of the list in reverse order
`myList[-2:0:-1]`	All but the first and last item in the list, in reverse order

`myList[1:5:3] = ["newItem1", "newItem2"]`	Replaces two items
`s = [*"foobar"]; s[1::2] = "ABC"`	Replaces three chars

`myList[:]`	Shallow copy the list (or string). If the list has only strings, numbers, and Booleans, then the copy is complete. However, if the list contains sublists, then pointers will be copied so one doesn't wind up with a true copy of the original list. This will be explored in a future section.
`*myList`	Dereferences a string, list, or set.

`*myList` never appears by itself (and would generate a syntax error). It is always enclosed in something, typically a list or set. One may think of it as listing all the elements of the data structure it is applied to with commas between them. Thus, `[*"foo"]` is `["f", "o", "o"]`, a list of characters (strings of length 1), which is different from `"foo"`. `[*myList]` is equivalent to `myList[:]`. `[*mySet]` is a way of converting a set to a list whereas `{*mySet}` makes a copy of the set.

`myStr[pos]` is the same as `myStr[pos:pos+1]` as a single char is a string of length 1, but `myList[pos]` is a list item whereas `myList[pos:pos+1]` is a list containing a single item.

The workhorse data structures that will be used throughout this class are lists, sets, and dictionaries. Lists (sometimes referred to as arrays) are (ordered) sequences of items where the numbering starts at 0. Sets are unordered items, so there is no notion of order (this is more than a theoretical idea. If you print out a set multiple times, the elements may be printed in different order). Dictionaries are key:value associations where the keys are numbers or strings (or tuples), and the values are arbitrary python items. The keys of a dictionary constitute a set. Finally, tuples are like lists only they are immutable, so they are slightly more efficient for python. We will have little reason in this class to concern ourselves about tuples.

`str(myArg)`	will try valiantly to coerce a string representation from myArg. Typically used to convert a number to a string, but may also be used to cast a list to a string. In practise, `"{}".format(myArg)` is preferred as it has more flexibility, and `f"{myArg}"` is the modern variation.
`"glue".join(myList)`	can be used to create a single string from a list of strings. If one wants a nice comma separated list, `", ".join(myLstOfStr)` could do the trick, while `"\n".join(myLstOfStr)` will put them all on separate lines. `"".join([*myStr])` would recover the original string.
`len(myStruct)`	returns the number of items in myStruct, which could be a string, list, set, dictionary, or tuple.
`zip(list1, list2)`	packages up the two lists so that you get a sequence of tuples where the 1st tuple comes from the first element of each list, the 2nd tuple comes from the 2nd element of each list, etc. Not often used.

Booleans

Python, Javascript, PHP, and some other languages have an expanded notion of how to work with Booleans. In particular, they employ lazy Boolean evaluation, coercion of Booleans to integers (when needed), and a more expansive version of when a variable is True or False.

When making a Boolean test, items that are considered to be 0 or empty will fail the test, whereas any other items will pass the test. Items that will fail a Boolean test are:
`False`, `0`, `""`, `[]`, `{}`, `set()`, `tuple()`

However, `[0]`, `{0}`, and `tuple([0])` will not fail a Boolean test because those items do have something in them, even if it is only a zero.

A Boolean value that is True is considered to have a numerical value of 1, if needed, while a Boolean value of False is considered to be 0. The "if needed" arises when you indicate to python that you want the value treated numerically. Thus,

`(a>2)`	gives a True or False.
`(a>2)+(b>2)`	counts how many of a and b are greater than 2. The plus mandates numbers.
`n*(n%2>0)`	yields n if n is odd, and yields 0 if n is even. The asterisk mandates numbers.

Boolean negation is done by means of `not`. Therefore, `not "Fred" or "Ginger" => (not "Fred") or "Ginger" => False or "Ginger" => "Ginger"`. See next section for details on this calculation.

Lazy Boolean evaluation

Lazy Boolean evaluation does not refer to Python being inefficient or slow. Actually, it is the other way around. If one has `expressionA and expressionB`, then it is pointless to evaluate `expressionB` if `expressionA` is not equivalent to True. Where this really makes a difference is when `expressionB` is a complicated function. The same applies when evaluating `expressionA or expressionB` when `expressionA` is equivalent to True. In that case it is pointless to evaluate `expressionB`.

But wait, there's more. Python won't coerce expressions to Booleans if it doesn't need to, and it doesn't need to in the above cases. In particular, the Boolean value of `expressionA and expressionB` and is determined completely from the final expression that python must evaluate before determining the truth of the entire expression. That is to say, the final expression evaluated is equivalent to the Boolean value of the entire expression. Therefore, rather than coercing the final expression evaluated to a Boolean, python will just return the final expression.

`"cat" and "dog"`	returns `"dog"`
`"cats" or "dogs"`	returns `"cats"`
`if aRay: doSomething`	do something if `aRay` not empty (ie. `if len(aRay)>0`)

`if aRay and max(aRay)>0:`
`  doSomething`
	if `aRay` has a positive value then do something. More formally, if `aRay` is not empty and it has a positive value then do something.

`def fact(n):`
`   return 1*(n<2) or n*fact(n-1)`
	If `n<2` then return 1; else return `n*fact(n-1).` The `1*` is to coerce the Boolean to a number.

`if any(lstOfBools):`
`  doSomething`
	if any item in lstOfBools is equivalent to True, do something

`if all(lstOfBools):`
`  doSomething`
	if all items in lstOfBools correspond to True, do something

Comprehension

What will `x` be, depending on the type of data structure that `myDataStruct` is, in the following?

```
for x in myDataStruct:
```

`myDataSturct` type	`x`
range	number
string	char
list	list item
set	set item
dictionary	key

It is sometimes useful to have both the index and the value for list items, and for these cases `enumerate` can be useful.

```
for idx,val in enumerate(myList): doSomething
```

will set both `idx` and `val` on each pass. It is like getting two variables for the price of one. You can enumerate any of the types in the above table and `val` will be as shown above for `x`. In other words, for a dictionary, you will get a key. For a set, you will get an element, but because it is a set, the order is not unique and may vary from run to run. Hence, enumerate is mostly useless for dictionaries and sets.

It is possible and usual to have a loop inside of a list, set, or dictionary. This is referred to as a comprehension, or a list comprehension, set comprehension, or dictionary comprehension according to what is being created. The loop comes second, while how you'd like to utilize the list comes first. For example:

Examples:

`newSet = {s+"!" for s in setOfStrings}`	Derive a new set from a set of strings, where each new string has an exclamation point suffixed.
`mySum = sum([n for n in range(101)])`	Add the numbers from 1 to 100

For the second example it is possible to shorten it:

```
mySum = sum(n for n in range(101))
mySum = sum([*range(101)])
```

This method shown in the above two lines also applies to `min` and `max`.

Comprehensions support nested loops and conditionals. For example, if one were to want to add up those numbers from 1 through 999 which are less than 5 mod 100, comprehensions offer a few straightforward ways:

Double loops:
```
mySum = sum(x+y for x in range(0,1000,100) for y in range(5))
```

Conditional:
```
mySum = sum(x for x in range(1000) if x%100<5)
```

If one has multiple loops with conditionals, any conditionals come last.

Comprehensions don't natively allow temporary variables, but one can use either the python 3.8.5 construct of := or there is an artificial way to create one by creating a list with a single element and setting a variable to run through it such as
```
… for x in [mySingleton] …
```
but the setup overhead is so great that it is rarely worth the effort, and the := construct is shorter.

Finally, we'll put everything together with a last example: given a lstOfInts find the number of adjacent monotonic increasing subsequences that it contains. For example,
[38, 17, 12, 17, 26, 2, 40, 5, 5, 6, 9, 1] contains the following adjacent monotonic increasing subsequences: [38], [17], [12, 17, 26], [2, 40], [5, 5, 6, 9], and [1]. That is to say, 6 subsequences.

You might take some moments to think about how to solve this problem.

Here's a solution. First, it is the number of bigger to smaller transitions that are important. Therefore, one can set up a comparison of adjacent pairs:

```
[(lstOfInts[i]>v) for i,v in enumerate(lstOfInts[1:])]
```

The technique shown in the comprehension using the slice is quite useful when analyzing adjacent pairs of elements. Note also that while the slice starts from 1, once the slice is performed it will start from 0. In other words, the enumeration enumerates the right neighbor, while the index, `i`, accounts for the left neighbor. Now all that is needed is to know the number of transitions from higher to lower:

```
1+sum((lstOfInts[i]>v) for i,v in enumerate(lstOfInts[1:]))
```

An alternate, slightly shorter way, is to use zip:

```
1+sum(z[0]>z[1] for z in zip(lstOfInts,lstOfInts[1:]))
```

Scoring

Each passing test receives 3/LineCt points. Thus, if each test is one line long with an `else` in it, then 84 points would result. If a successful test has more than one line, the number of lines is indicated. If a successful test has only one line, then there are three possible indications (`$`, `^`, `=`, or `1`), described below. `A=>10, B=>11,...,a=>36,b=>37,...`

If a test shows `"$"`, it receives 4/7 additional points (for 25/7 total points on that problem)

If a test shows `"="` or `"1"`, it receives no additional points (for 3 total points). This happens if there is an `else` in the solution or the solution (`=`) or the solution is long, despite being one line (`1`).

If a test shows `"^"`, the additional points it receives is given by 4*(2-L/P)/7, where L is the character length of the offered solution and P is the character length of the grader's solution.

Meanings of other symbols are as follows:

`-` No function submitted for this test	`*` Script error
`0` No script error, but test did not pass	`@` Timeout on this test
`.` Test encountered after a timeout	

The character length of a solution is computed by:

A) Excluding all comments and extra whitespace
eg. `return (3 + 7) # final answer => return(3+7)`

B) Counting each parameter to a function as a single character.
eg. `def fact(num): return (num>1 and num*fact(num-1)) or 1` would automatically be converted by the grader to the equivalent
`def fact(n):return(n>1 and n*fact(n-1))or 1`

C) Any temporary variables that may be used (eg. in comprehensions) are not reduced in length.

D) The grader does not remove unnecessary parens. Thus,
`def fact(n):return(n>1 and n*fact(n-1))or 1` is not reduced to
`def fact(n):return n>1 and n*fact(n-1)or 1`.

E) The length is computed from after the first colon, and 7 is subtracted, accounting for the `return` and one character. Thus, the length of
`def fact(num): return (num>1 and num*fact(num-1)) or 1` comes in at 24.

The longest solution in Lab 2 can be done in 75, while the others can each be done in under 50.