Due Feb 23 by 11:59pm **Points** 100 **Available** after Feb 13 at 9:45pm

Home

Resources

Homework 4: Lists & Loops

Be sure to review the Developer's Notebook on Lists (in this module) before you attempt this homework.

Pro-Tip: Do NOT wait until the night before it's due to start this assignment. You'll need the time to think, experiment, and craft the code for this homework.

No Flowcharts or written Test Cases are needed for submission this week.

Written Component (20% of your homework score)

- Filename: `written.txt`

Please open a plaintext file (you can do this in IDLE or any plaintext editor you like, such as TextEdit or NotePad) and type out your answers to the questions below. You can type your answers into Python to confirm but answer the questions first!

Written #1

Each of the code snippets below would generate some kind of error. Identify whether each one is a...

- **NameError:** (https://docs.python.org/3/library/exceptions.html#NameError) A variable or function has been used before it has a value
- **TypeError** (https://docs.python.org/3/library/exceptions.html#TypeError): A function/operation is applied to a value of the wrong type
- **IndexError** (https://docs.python.org/3/library/exceptions.html#IndexError): A list or string subscript is out of range

1A

```
1  lst = [18, 0, -5]
2  lst[3] + 3
```

1B

```
1  lst = [4, 5, 6]
```

```
2  lst[0] + lst
```

1C

```
1  lst = [1, 2, 3]
2  lst[1] + "4"
```

Written #2

What do each of the following code snippets print to the terminal?

2A

```
1  nested = [[3, 4, 5], [8, 9, 10]]
2  for i in range(len(nested)):
3      for j in range(len(nested[i])):
           print(nested[i][j])
```

2B

```
1
2  nested = [[3, 4, 5], [8, 9, 10]]
3  for i in range(len(nested)):
4      print(nested[i])
```

Programming Component (80% of this HW)

Guidelines for all Programs in HW4:

- **You may not use any list functions other than *len(), pop(), extend()* and *append()*.**
- **List slices ARE allowed using the [:] operator if you want.**
- You may not import any additional modules other than the ones mentioned in the assignment. If you have questions on an import, ask.
- Remember to refer to our style guide when considering program structure and readability.

Programming #1 (25% of this HW)

- Filename: `runlength_decoder.py`
- Starter code for data: `hw4data.py` ⭳ (https://northeastern.instructure.com/courses/102990/files/13110946/download?download_frd=1)
- You must provide a function named `decode()` that takes a `list` of RLE-encoded values as its single parameter, and returns a `list` containing the decoded values with the "runs" expanded. If we were to use Python annotations, the function signature would look similar to this:
  `def decode(data : list) -> list:`

Run-length encoding (RLE) is a simple, "lossless" compression scheme in which "runs" of data. The same value in **consecutive** data elements are stored as a single occurrence of the data value, and a count of occurrences for the run.

For example, using Python lists, the initial list:

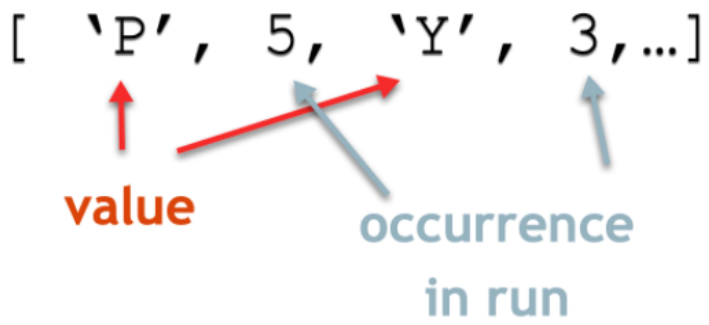
```
['P','P','P','P','P','Y','Y','Y','P','G','G']
```

would be encoded as:

```
['P', 5, 'Y', 3, 'P', 1, 'G', 2]
```

So, instead of using 11 "units" of space (if we consider the space for a character/int 1 unit), we only use 8 "units". In this small example, we don't achieve much of a savings (and indeed the smaller your overall data size, the less opportunity for any compression algorithm to save much space) but imagine if we had a *run* of 30 or 40 P's and 10 Y's. The space savings begin to add up.

Our RLE scheme for this assignment uses the format shown in the example above. Data is contained in a list such that the elements follow the pattern:



Notice the data contained in our list is **heterogeneous**. In other words, it's NOT all the same type. strings and integers are interleaved in the list. Be sure to consider this as you're designing your solution.

Do this:

- Write a program that decompresses the data included in the file starter file `hw4data.py`.
  IMPORTANT: **Import** `hw4data.py`, do **NOT** copy/paste the data! If you copy-paste and introduce a subtle data error due to "fat fingering" or some other mistake, your RLE program may not work correctly and you will lose points. The data is a Python file - import it as any other "outside asset" file (e.g. external functions) and proceed from there.
- There are six data values of increasing size and complexity: `DATA0, DATA1, …DATA5`. Your program must provide a function named decode(list) that takes a list of RLE-encoded values and returns a list containing the decoded values with the "runs" expanded. We will call your `decode()` function in our test suite, so name the function exactly as specified, and make sure it returns a Python list.

Optional:
We're using some fun compressed data, but it's all textual. If you want to actually see the original information, you can convert the decompressed (decoded) list into one single string value, then display that string on the screen using the `print()` statement. Again, this step is optional and for your personal enjoyment to view the data. Given the example data discussed above, the screen output would be something like this:

```
= RESTART: /Users/keithbagley/Dropb
mework/HW4/runlength_decoder.py
Decoding run-length encoded data


PPPPPYYYPGG
```

Notes

- WE do not want to see your decoded data printed out. IF you perform this optional step be sure to keep this separate from the core functionality you're submitting for your assignment.
  WARNING: Do NOT use your `decode()` function for optional actions described above. You are free to create other helper functions to convert and display the decompressed data. You will lose points if your decode() function does more than the specified decompression algorithm.

- You can assume “good” data and it conforms to the RLE scheme described without any missing elements.

What to submit:

- Your solution in a file named: `runlength_decoder.py`, which contains the function `decode()` as specified above.
- You may include other files with any “helper” functions you’ve written. Documentation/comments in runlength_decoder.py should indicate any additional helper files you’ve included

Programming #2 (30% of this HW)

- Filename: `cards.py`

In this program, you will create functions that use lists to create, shuffle, and deal cards to a number up to 4 “hands”.

The standard deck of playing cards contains 52 cards. Each card has one of four suits along with a value. Suits are normally spades, hearts, diamonds and clubs. The values are 2 through 10, Jack, Queen, King and Ace (we will ignore the “Joker”).

Each card will be represented using two characters in a string. The first character is the value of the card, with 2 through 9 represented directly. The other cars with letters: “T”, “J”, “Q”, “K”, and “A” represent the values 10, Jack, Queen, King, and Ace respectively.

The second character in the string represents the suit of the card. These are: “s” for spades, “h” for hearts, “d” for diamonds, and “c” for clubs.

Here’s an example:

Card	Abbreviation
Jack of spades	Js
Two of clubs	2c
Ten of diamonds	Td
Ace of hearts	Ah
Nine of spades	9s

In addition to any other functions you decide to write, you **MUST** provide the following functions that we can run with our test suite. The names, return type and parameter types must match (of course, you can name the parameters anything you wish).
Note: You are NOT providing a test suite, we are using ours after you submit your assignment. WE will be running our tests against your code. Make sure your function names & argument types match the descriptions below.

- `create_deck()`: Creates a deck of 52 cards using the abbreviations outlined in the requirements.
  Input parameters: None. **Return value**: a list of cards, not shuffled.
  If we specified this with Python annotations, the signature would be: `create_deck() -> list:`

- `shuffle(cards)`: Shuffles the 52 cards in a **new** list, such that the list of cards is ordered randomly, *while leaving the original list of 52 cards unchanged*. Shuffle returns the “shuffled deck” to the caller.
  Input parameters: a list of cards, not shuffled. **Return value**: a list of cards, shuffled. The original list is not modified (aka “mutated”)
  If we specified this with Python annotations, the signature would be: `shuffle(cards : list) -> list:`

- `deal(number_of_hands, number_of_cards, cards):` Takes the number of hands (players) (up to 4), number of cards per hand/player (up to 13), and the deck of cards to deal from. Your function should return a list containing all of the hands that were dealt. Each hand will be a represented as a list of cards. Therefore, this function will return a list of lists. As a side-effect, the cards dealt should be removed from deck passed as an input parameter to this function.
  Input parameters: number of hands -> int, number of cards -> int, cards -> list.

*** Note: It's perfectly fine to rename the "hands" parameter to "players" if you wish to, especially if that makes it more clear to you ***

Return value: a list of **hands/players**, where each hand is itself a list of cards dealt for a player.

If we specified this with Python annotations, the signature would be:

```
deal(number_of_hands : int, number_of_cards : int, cards : list) -> list:
```

For this function, assume PRECONDITION:

number of cards has been pre-validated, values allowed: 0..13;

number of hands has been pre-validated, values allowed: 1..4

Also keep in mind that deal() returns a nested list, not a single list!

Example tests we might run with your functions:

```
Original deck is:
['2s', '3s', '4s', '5s', '6s', '7s', '8s', '9s', 'Ts', 'Js', 'Qs', 'Ks', 'As', '2h
', '3h', '4h', '5h', '6h', '7h', '8h', '9h', 'Th', 'Jh', 'Qh', 'Kh', 'Ah', '2d', '
3d', '4d', '5d', '6d', '7d', '8d', '9d', 'Td', 'Jd', 'Qd', 'Kd', 'Ad', '2c', '3c',
'4c', '5c', '6c', '7c', '8c', '9c', 'Tc', 'Jc', 'Qc', 'Kc', 'Ac']

The shuffled deck is:
['4d', '3s', '4s', '5s', '6s', '7s', '8s', '9s', 'Ts', 'Js', 'Qs', 'Ks', 'As', '2h
', '3h', '4h', '5h', '6h', '7h', '8h', '9h', 'Th', 'Jh', 'Qh', 'Kh', 'Ah', '2d', '
3d', '2s', '5d', '6d', '7d', '8d', '9d', 'Td', 'Jd', 'Qd', 'Kd', 'Ad', '2c', '3c',
'4c', '5c', '6c', '7c', '8c', '9c', 'Tc', 'Jc', 'Qc', 'Kc', 'Ac']

Dealt one round of 4 players

[['4d', '6s'], ['3s', '7s'], ['4s', '8s'], ['5s', '9s']]

Cards left in deck
['Ts', 'Js', 'Qs', 'Ks', 'As', '2h', '3h', '4h', '5h', '6h', '7h', '8h', '9h', 'Th
', 'Jh', 'Qh', 'Kh', 'Ah', '2d', '3d', '2s', '5d', '6d', '7d', '8d', '9d', 'Td', '
Jd', 'Qd', 'Kd', 'Ad', '2c', '3c', '4c', '5c', '6c', '7c', '8c', '9c', 'Tc', 'Jc',
'Qc', 'Kc', 'Ac']
```

What to submit:

- Your solution in a file named: `cards.py`
- You may include other files with any “helper” functions you’ve written. Documentation/comments in `cards.py` should indicate any additional helper files you’ve included

Programming #3 (25% of this HW)

- Filename: `upc.py`

As you learned from our class and readings, the modulo operator can be useful in surprising ways. For this assignment, you'll use it to validate UPC barcodes. In most stores today, almost every item you purchase has a Universal Product Code (UPC). The UPC is typically printed on the product and is

read by a barcode scanner.

Most of these digits are chosen to specify the manufacturer, product type, country, etc., but one of them is set aside to be the “check digit.” After all the other numbers are set, the check digit is chosen such that a calculation we apply to the numbers will end up being a multiple of 10.

Why are UPCs designed this way? Because if you’re a cashier typing entering the information manually, the system can easily notify you of any mistakes in the data entry.

For UPC numbers, the validation algorithm proceeds on the number from **right** to left:

- Digits in even positions, including zero: no change
- Digits in odd positions: *3
- Sum the results
- Check: If the barcode has a valid UPC number, this result is a multiple of 10.

Example:

The UPC number above is 0 7 3 8 5 4 0 0 8 0 8 9.

We apply the algorithm (the number above is written from left to right, but the algorithm goes right to left, so we say 9 is at position 0, 8 is at position 1, etc.):

```
9 + 3·8 + 0 + 3·8 + 0 + 3·0 + 4 + 3·5 + 8 + 3·3 + 7 + 3·0
=     9 + 24 + 24 + 4 + 15 + 8 + 9 + 7
=    100
```

And we’re done! We’ve verified that this, in fact, a valid UPC number.

Do This:

Write a Python function that determines whether a given list of integers represents a valid UPC. Use a loop to apply the algorithm described here. UPC codes can be different lengths, so don’t assume it’ll always be exactly 12 digits.

Special case: If the input is less than 2 digits long, or if all the digits have value 0, the upc code is not valid.

Your function must meet the following specifications:

Name: `is_valid_upc(list_of_integers)`
Input: `List` of integers, a possible UPC number
Returns: `Boolean`, indicating whether the given input is valid or not

If we specified this with Python annotations, the signature would be: `is_valid_upc(list_of_integers : list) -> bool:`