[3.2] List and Tuples

January 5, 2023

1 Lists

Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls.

1.1 Sequence Operations

Because they are sequences, lists support all the sequence operations

[1]:
```
# A list of three different-type objects
L = [123, 'spam', 1.23]
# Number of items in the list
print(len(L))
```

```
3
```

we can index, slice, and so on

[2]:
```
# Indexing by position
print(L[0])
```

```
123
```

[3]:
```
# Slicing a list returns a new list
print(L[:-1])
```

```
[123, 'spam']
```

[4]:
```
# Concat/repeat make new lists too
print(L + [4, 5, 6])
```

```
[123, 'spam', 1.23, 4, 5, 6]
```

[5]:
```
print(L * 2)
```

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
```

[6]:
```
# We're not changing the original list
print(L)
```

```
[123, 'spam', 1.23]
```

1.2 Type-Specific Operations

Python's lists may be reminiscent of arrays in other languages, but they tend to be more powerful. * They have no fixed type constraint * They have no fixed size * They can grow and shrink on demand, in response to list-specific operations

[7]:
```
L.append('NI')     # Growing: add object at end of list
```

[8]:
```
print(L)
```

```
[123, 'spam', 1.23, 'NI']
```

[9]:
```
print(L.pop(2))     # Shrinking: delete an item in the middle
```

```
1.23
```

[10]:
```
print(L)
```

```
[123, 'spam', 'NI']
```

Common list operations: * append: insert an item at the end of the given list * insert: insert an item at an arbitrary position * extend: add multiple items at the end * remove: remove a given item by value * pop: pop method (or an equivalent del statement) removes an item at a given offset

Because lists are mutable, most list methods also change the list object in place, instead of creating a new one:

[11]:
```
M = ['bb', 'aa', 'cc']
M.sort()
print(M)
```

```
['aa', 'bb', 'cc']
```

[12]:
```
M.reverse()
```

[13]:
```
print(M)
```

```
['cc', 'bb', 'aa']
```

The list **sort** method here, for example, orders the list in ascending fashion by default, and **reverse** reverses it—in both cases, the methods modify the list directly.

1.3 Nesting

One nice feature of Python's core data types is that they support arbitrary nesting—we can nest them in any combination, and as deeply as we like. For example, we can have a list that contains a dictionary, which contains another list, and so on. One immediate application of this feature is to represent matrixes, or "multidimensional arrays" in Python.

[14]:
```
M = [[1, 2, 3],     # A 3 × 3 matrix, as nested lists
     [4, 5, 6],     # Code can span lines if bracketed
     [7, 8, 9]]
```

[15]:
```
print(M)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Here, we've coded a list that contains three other lists. The effect is to represent a 3×3 matrix of numbers. Such a structure can be accessed in a variety of ways:

[16]:
```
print(M[1])      #Get Row 2
```

```
[4, 5, 6]
```

[17]:
```
print(M[1][2])           # Get row 2, then get item 3 within the row
```

```
6
```

1.4 List Comprehension

Do you remember the basic mathematics?

- {x2 | x 2 N } gives squares of natural numbers.
- {x2 : x in {0 . . . 9}} gives squares of numbers with in the provided set, {0 . . . 9}.

List comprehension implements such well-known notations for sets. It is an elegant and concise way to define and create lists in Python, and off-course, it saves typing as well!

Syntax for the list comprehension is: "statement/expression" followed by a "for clause" with in "square brackets".

Let's learn with example while comparing for loop and list comprehension.

[18]:
```
# We have a list 'x'
x = [2,3,4,5]
```

What if we want to create a new list that contains squares of all the elements in list x? We can do this using a for loop as given in the code cell below (*please read the comments*).

[19]:
```
def list_of_squares():
    out = []      # empty list for squares
    for num in x:      # loop test
        out.append(num**2)      # taking squeares and appending them to the␣
 ↪empty list "out"
    print(out)      # using print to get the output
```

[20]:
```
#calling list_of_squares function
list_of_squares()
```

```
[4, 9, 16, 25]
```

Ok, we have accomplished the task to compute squares using for loop, however, the above task can be elegantly implemented using a list comprehension in a one line of code. **Simply take for statement and put it after what you want in result!**

[21]:
```
# So, this is going to be out first list comprehension to compute squares of␣
 ↪all ,!the elements in a given list!
print([num**2 for num in x])
```

```
[4, 9, 16, 25]
```

[22]:
```
# Another example using string -- notice the white space!
print([letters for letters in 'Hello World'])
```

```
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

[23]:
```
# one more example using range()
print([numbers**2 for numbers in range(2,10)])
```

```
[4, 9, 16, 25, 36, 49, 64, 81]
```

2 Tuples

The tuple object (pronounced "toople" or "tuhple," depending on whom you ask) is roughly like a list that cannot be changed—tuples are *sequences*, like lists, but they are immutable, like strings. Functionally, they're used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations:

Some working examples of range are given in the below code cels.

[24]:
```
T = (1, 2, 3, 4)      # A 4-item tuple
```

[25]:
```
print(len(T))      # Length
```

```
4
```

[26]:
```
print(T + (5, 6))      # Concatenation
```

```
(1, 2, 3, 4, 5, 6)
```

[27]:
```
print(T[0])      # Indexing, slicing, and more
```

```
1
```

Tuples also have type-specific callable methods as of Python 2.6 and 3.0, but not nearly as many as lists:

[28]:
```
print(T.index(4))      # Tuple methods: 4 appears at offset 3
```

```
3
```

[29]:
```
print(T.count(4))      # 4 appears once
```

```
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences (one-item tuples like the one here require a trailing comma):

[30]:
```
T[0] = 2     # Tuples are immutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16476\4101831216.py in <module>
----> 1 T[0] = 2     # Tuples are immutable

TypeError: 'tuple' object does not support item assignment
```

Like lists and dictionaries, tuples support mixed types and nesting, but they don't grow and shrink because they are immutable (the parentheses enclosing a tuple's items can usually be omitted, as done here):

[ ]:
```
T = 'spam', 3.0, [11, 22, 33]
```

[ ]:
```
print(T[1])
```

[ ]:
```
print(T[2][1])
```

[ ]:
```
T.append(4)
```

2.1 Why Tuples?

So, why have a type that is like a list, but supports fewer operations? Frankly, tuples are not generally used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in programs larger than those we'll write here.

3 Exercise

3.1 Rainfall Statistics

Design a program that lets the user enter the total rainfall for each of 12 months into a list. The program should calculate and display the total rainfall for the year, the average monthly rainfall, and the months with the highest and lowest amounts.

3.2 Driver's License Exam

The local driver's license office has asked you to create an application that grades the written portion of the driver's license exam. The exam has 20 multiple-choice questions. Here are the correct answers:

1. A	6. B	11. A	16. C
2. A	7. B	12. A	17. C
3. A	8. B	13. A	18. C
4. A	9. B	14. A	19. C

5. A	10. B	15. A	20. C

Your program should store these correct answers in a list. The program should read the student's answers for each of the 20 questions from a text file and store the answers in another list. (Create your own text file to test the application.) After the student's answers have been read from the file, the program should display a message indicating whether the student passed or failed the exam. (A student must correctly answer 15 of the 20 questions to pass the exam.) It should then display the total number of correctly answered questions, the total number of incorrectly answered questions, and a list showing the question numbers of the incorrectly answered questions.

3.3 Name Search

You are provided two dataset files:

- **GirlNames.txt**—This file contains a list of the 200 most popular names given to girls born in the United States from the year 2000 through 2009.
- **BoyNames.txt**—This file contains a list of the 200 most popular names given to boys born in the United States from the year 2000 through 2009.

Write a program that reads the contents of the two files into two separate lists. The user should be able to enter a boy's name, a girl's name, or both, and the application will display messages indicating whether the names were among the most popular.

3.4 Population Data

You are provided a dataset file named **USPopulation.txt**. The file contains the midyear population of the United States, in thousands, during the years 1950 through 1990. The first line in the file contains the population for 1950, the second line contains the population for 1951, and so forth.

Write a program that reads the file's contents into a list. The program should display the following data:

- The average annual change in population during the time period
- The year with the greatest increase in population during the time period
- The year with the smallest increase in population during the time period

3.5 World Series Champions

You are given a dataset file named **WorldSeriesWinners.txt**. This file contains a chronological list of the World Series winning teams from 1903 through 2009. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2009. Note that the World Series was not played in 1904 or 1994.) Write a program that lets the user enter the name of a team and then displays the number of times that team has won the World Series in the time period from 1903 through 2009.

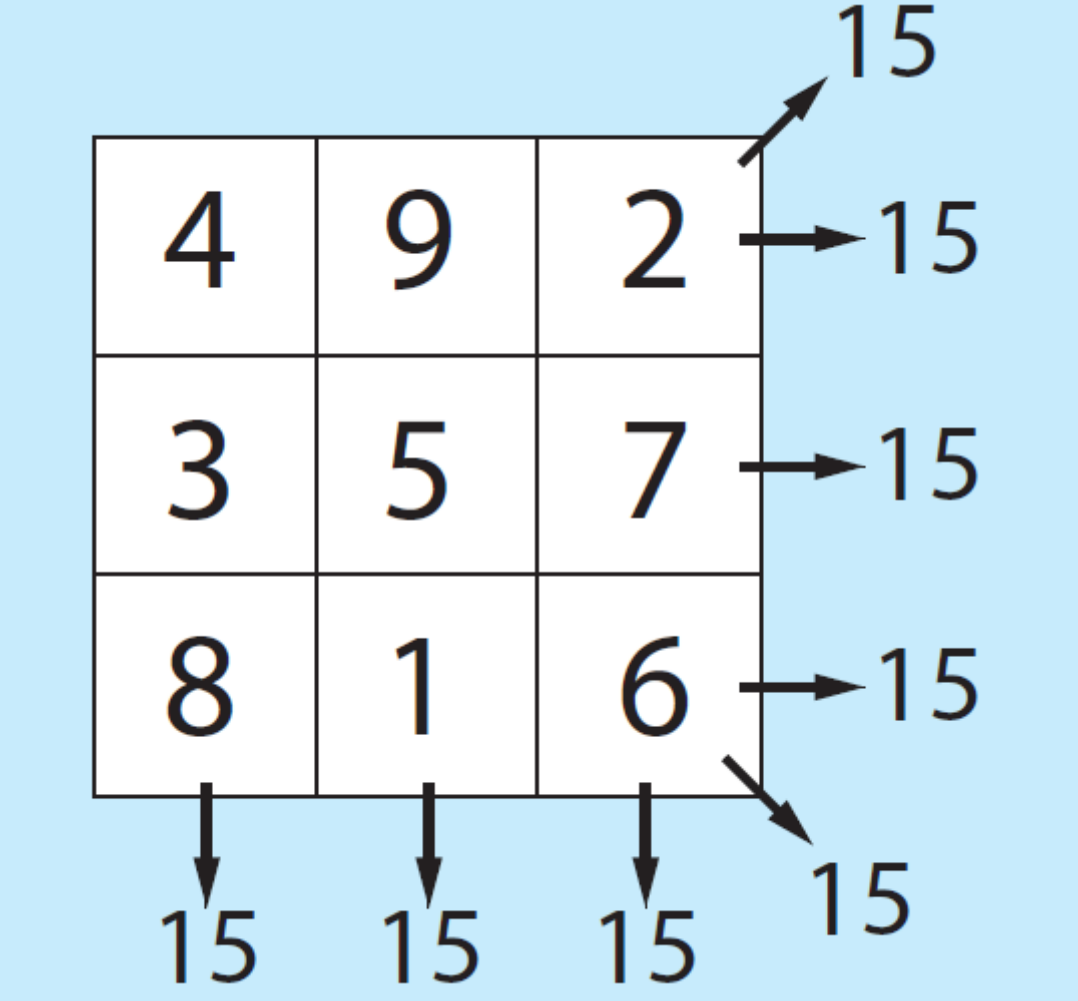
3.6 Lo Shu Magic Square

The Lo Shu Magic Square is a grid with 3 rows and 3 columns. The Lo Shu Magic Square has the following properties:

- The grid contains the numbers 1 through 9 exactly.
- The sum of each row, each column, and each diagonal all add up to the same number.

In a program you can simulate a magic square using a two-dimensional list. Write a function that accepts a two-dimensional list as an argument and determines whether the list is a Lo Shu Magic Square. Test the function in a program.

4	9	2
3	5	7
8	1	6


15
4
9
2
15
3
5
7
15
8
1
6
15
15
15
15
15

[ ]: