Lecture 5 - Lists and `for` loops

Computer Programming

Robert D. Vincent and Samia Hilal

Marianopolis College

February 14, 2022

Lists

- We often want to store many closely-related pieces of data:
  - Daily rainfall
  - Monthly sales totals
  - Names of students
- Groups a sequence of elements under a single object.
- Each element is stored at a numbered location in the list.
- Sometimes called arrays, vectors, or *lists*.
- An example of a Python *collection* type.

List literals

- Enter a list of expressions inside square brackets:

```
counts = [21, 23, 27, 31, 30, 26, 20]
fruit = ['apple', 'banana', 'grape']
means = [9.1, 10.2, 8.9, 9.5]
is_prime = [False, False, True, True]
result = []          # Empty list
```

- The expressions are separated by commas.
- Lists can be *nested*:

```
[4, [1, 2], 5]
```

- List elements can have different types:

```
['apple', 30, 9.5]
```

Creating lists with `list()`

- The `list()` built-in function also creates lists:

```
myList = list()  # Another empty list.
```

- It can convert a string into a list:

```
>>> chars = list("Canada")
>>> print(chars)
['C', 'a', 'n', 'a', 'd', 'a']
```

- Or the `range()` built-in function:

```
>>> rng = range(5)
>>> lst = list(rng)
>>> print(lst)
[0, 1, 2, 3, 4]
```

Getting the list length

- The `len()` function returns the length of a list:

```
>>> L = [19, 21, 0, 10]
>>> len(L)
4
>>> len([])          # Empty list.
0
>>> M = ['a', ['b', 'c'], 'd']
>>> len(M)
3                    # This is correct. Why?
```

List indexing

- Lists are indexed almost exactly like strings:

```
>>> numbers = [1, 3, 5, 7, 9, 11, 13]
>>> numbers[0] # Indices start at 0
1
>>> numbers[1]
3
>>> numbers[6]
13
>>> numbers[-1]
13
```

- Each element on the list x is at an index from 0 to `len(x)-1`.

List slicing

- You can also take slices of lists:

```
>>> numbers = [1, 3, 5, 7, 9, 11, 13]
>>> numbers[4:] # Until end.
[9, 11, 13]
>>> numbers[:3] # From beginning.
[1, 3, 5]
>>> numbers[1::2]
[3, 7, 11]
>>> numbers[::3]
[1, 7, 13]
>>> numbers[0:6:2]
[1, 5, 9]
>>> numbers[::-1] # Reverses the List
[13, 11, 9, 7, 5, 3, 1]
```

Details of indexing

- Uses a single integer expression.
- The result is the *single element* at the location.
- Negative indices count from the end of the list.
- That location **must** already exist.

```
>>> L = [11, 21, 31, 41, 51]
>>> L[0]
11
>>> L[-1] # same as L[len(L)-1]
51
>>> L[-5] # same as L[len(L)-5]
11
>>> L[-6] # ERROR!
```

Details of slicing

- A slice expression contains up to three integer expressions separated by colons:
  [*start* : *end* : *step*]
- Runs from *start* up to but **not** including *end*.
- A slice of a list is *always* a list.
- The locations *need not* exist.
- Default values are used for missing expressions.
- Default values depend on the sign of the *step*:
  - $step > 0 \implies start = 0, end = \mathrm{len}(x)$
  - $step < 0 \implies start = -1, end = -\mathrm{len}(x) - 1$
  - *step* must not be zero!

List and String reverse slicing

- negative step with negative or default end

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[::-1] # Reverses the list
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> numbers[4::-1]
[5, 4, 3, 2, 1]
>>> numbers[-1:-5:-1]
[9, 8, 7, 6]
>>> string="123456789" # Same for Strings
>>> string[::-1]
"987654321"
>> string[-4::-1]
"654321"
>>> string[-1:-5:-1]
"9876"
```

Multidimensional lists

- It is often useful to create 2-dimensional or even N-dimensional arrays or matrices.
- We do this in Python by creating lists of lists.

```
>>> matrix = [ [1, 2, 3],  [4, 5, 6],
               [7, 8, 9] ]
>>> matrix[0]     # First row
[1, 2, 3]
>>> matrix[0][0] # First item on row 0.
1
>>> matrix[2][2] # Last item on last row.
9
>>> matrix[1][0]
4
```

Using matrices

```
# Process a 2-dimensional matrix.
# Version 1: while loops (l5ex0.py)
matrix = [ [1, 3, 8], [6, 5, 4],
           [7, 2, 9] ]
total = 0
row_idx = 0
while row_idx < len(matrix):
    col_idx = 0
    while col_idx < len(matrix[0]):
        total += matrix[row_idx][col_idx]
        col_idx += 1
    row_idx += 1
print(total) # Prints 45
```

Lists are mutable

- We can change elements in a list:

```
>>> numbers = [1, 3, 5, 7, 9, 11, 13]
>>> numbers[0]
1
>>> numbers[0] = 2 # New value at start.
>>> numbers
[2, 3, 5, 7, 9, 11, 13]
>>> numbers[-1] += 1  # Increment last.
>>> numbers
[2, 3, 5, 7, 9, 11, 14]
```

- We cannot do the same with strings:

```
>>> letters = 'ABCDEFG'
>>> letters[0] = 'X' # ERROR!
```

List operators

- Lists support some of the same operators as strings.
- The `in` operator can be used with lists:

```
>>> x = [5, 6, 7, 8]
>>> 5 in x
True
>>> 9 in x
False
```

- However, notice that `in` checks for a list element, not a sublist.

```
>>> [6, 7] in x
False
```

List operators

- Addition will concatenate lists:

```
>>> x = [1,2,3]
>>> y = [4,5,6]
>>> x + y
[1, 2, 3, 4, 5, 6]
>>> y + [9]
[4, 5, 6, 9]
```

- Multiplication with integers repeats a list:

```
>>> 3 * x
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [True] * 5
[True, True, True, True, True]
```

List comparison

- Lists can be compared:

```
>>> x = [1,2,3]
>>> y = [1,2,3]
>>> x == y
True
>>> x[1] = 4
>>> x == y
False
>>> x > y
True
```

- Performs lexicographic comparison:

```
>>> [1,2,3] > [1,1,300]
True
```

List methods: searching

- `index()` applies to lists as well as strings.

```
>>> x = ["a", "b", "c", "d", "a"]
>>> x.index("a")
0
>>> x.index("c")
2
>>> x.index("a", 2) # start at 2
4
>>> x.index("e")  # Throws an exception
```

- Note that `list()` does *not* include `rindex()`.

List methods: searching

- `count()` gives the number of elements with a particular value:

```
>>> array = ["a", "b", "c", "d", "a"]
>>> array.count("a")
2
>>> array.count("b")
1
>>> array.count("e")
0
```

List methods: insertion

- `append()` adds one element to the end of a list.

```
>>> array = [2, 4, 6, 8]
>>> array.append(10)
>>> print(array)
[2, 4, 6, 8, 10]
```

- `insert()` adds an element before the given position.

```
>>> array.insert(0, 1) # position, value
>>> array
[1, 2, 4, 6, 8, 10]
>>> array.insert(2, ['a', 'b'])
[1, 2, ['a', 'b'], 4, 6, 8, 10]
```

List methods: insertion

- `extend()` appends a list to another list:

```
>>> array = [2, 4, 6, 8]
>>> array.extend([10, 12, 14])
>>> print(array)
[2, 4, 6, 8, 10, 12, 14]
>>> array.extend(16) # ERROR!
```

- This will modify the list, whereas the '+' operator usually does not.

List methods: removal

- `remove()` removes an element by value:

```
>>> array = [2, 4, 6, 8, 10]
>>> array.remove(2)
>>> array
[4, 6, 8, 10]
```

- `pop()` removes an element by index:

```
>>> array = [2, 4, 6, 8, 10]
>>> array.pop(2)
6                # Returns the deleted value.
>>> array
[2, 4, 8, 10]
```

- `pop()` with no argument removes the last element.

List methods: removal

- The `del` operator can remove an item or a slice:

```
>>> array = [2, 4, 6, 8, 10]
>>> del array[0]
>>> print(array)
[4, 6, 8, 10]
>>> del array[-2:]
>>> print(array)
[4, 6]
```

Sorting a list

- `sort()` puts a list in *increasing* order.
- It modifies the list itself!

```
>>> array = [5, 3, 4, 1, 2]
>>> array.sort()
>>> print(array)
[1, 2, 3, 4, 5]
>>> veggies = ["kale", 'chard', "cabbage"]
>>>  veggies.sort()
>>> print(veggies)
['cabbage', 'chard', 'kale']
```

The `sorted()` function

- The `sorted()` function sorts the list without changing it:

```
>>> array = [5, 3, 4, 1, 2]
>>> sorted(array)
[1, 2, 3, 4, 5]
>>> print(array)
[5, 3, 4, 1, 2]    # Unchanged!
>>> veggies = ["kale", 'chard', "cabbage"]
>>> sorted(veggies)
['cabbage', 'chard', 'kale']
>>> print(veggies)
['kale', 'chard', 'cabbage']
```

split() and join()

- It is often useful to break a string into substrings:

```
>>> text = "To be or not to be."
>>> words = text.split()
>>> print(words)
['To', 'be', 'or', 'not', 'to', 'be.']
```

- The join() method reverses the process:

```
>>> ' '.join(words)
'To be or no to be.'
>>> '_'.join(words)
'To_be_or_not_to_be.'
```

- Both are methods of the str type!

`split()` details

- The `split()` method breaks a string into a list of substrings.
- By default, the string is split at each series of one or more spaces.

```
>>> text = " A  B C    D  E. "
>>> text.split()
['A', 'B', 'C', 'D', 'E.']
```

- Optionally we can provide an argument to `split()` that will use something other than spaces.

`join()` details

- The `join()` method combines a list of strings into a single string.
- The string object `join()` is called on is placed between each of the list elements.

```
>>> letters = ['A', 'B', 'C', 'D', 'E']
>>> ' '.join(letters) # single space.
'A B C D E'
>>> 'xyz'.join(letters)
'AxyzBxyzCxyzDxyzE'
>>> ''.join(letters)  # empty string
'ABCDE'
```

Traversing a list

- The Python `for` loop provides an easy way to repeat some code for every element in a list.
- l5ex1.py

```
array = [5, 3, 4, 0, 2]
for value in array:
    print(value, value ** 2)
```

This program will print:

```
5 25
3 9
4 16
0 0
2 4
```

The `for` loop

- The `for` loop has the form:

```
for name in expression :
    statement1
    statement2
    ...
```

- The *expression* can be any *iterable* type: string, list, or others we will see later.
- The *name* need not be previously assigned.
- The *name* will be assigned the value of each element in the expression.
- The body runs once for each element.

The `for` loop and *name*

- When the loop completes, the *name* will be bound to the final element in the expression.
- The name `word` will still be valid after this loop finishes: (Example: l5ex2.py)

```
strings = ["A", "B", "C", "D", "E", "F"]
letter_count = 0
for word in strings:
    print(word.lower())
    letter_count += len(word)
print("Last word is", word, "with",
      letter_count, "letters.")
```

Two examples

- Sum of a vector (l5ex6.py):

```
total = 0
scores = [89, 92, 99, 85, 95, 92]
for score in scores:
    total += score
print('Mean score', total / len(scores))
```

- Sum of a matrix: l5ex5.py compare with slide12:

```
matrix = [ [1, 3, 8], [6, 5, 4],
           [7, 2, 9] ]
total = 0
for row in matrix:
    for val in row:
        total += val
print(total) # Prints 45
```

The `for` loop and `range`

- The `range()` built-in function is often used with `for` loops.
- It creates an iterable, immutable object that represents a list of integers (l5ex3.py).

```
for index in range(5):
    print(index)
```

- This example will print:

```
0
1
2
3
4
```

Using `range()` with lists

- We often use `range()` to generate the indices of a list (l5ex4.py).

```
x = ['apple', 'peach',
     'banana', 'orange']
for i in range(len(x)):
    if x[i] == 'peach':
        x[i] = 'pear'
print(x)
```

- This example will print:

```
['apple', 'pear', 'banana', 'orange']
```

Advantages of mutability

- We can change elements on a list.
- We can add elements to a list, increasing its length:

```
>>> x = []
>>> x.append('maple')
>>> x.append('birch')
>>> print(x)
['maple', 'birch']
```

- We can remove elements from a list, decreasing its length:

```
>>> del x[-1]
>>> print(x)
['maple']
```

Pitfalls of mutability

- Because lists can be changed, an possible problem arises.
- Python assignment does not make a copy, so assigning one name to another creates an 'alias':

```
>>> x = [1,2,3]
>>> y = x          # y is an alias for x
>>> y[0] = 2
>>> x
[2, 2, 3]
```

- Need to make a copy to avoid this!

```
>>> y = x[:] # Forces a copy
```

Mutability and the `is` operator

- The normal == operator tests for equal values.
- A second operator, `is`, tests whether two names refer to the exact same object.

```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
>>> x == y
True
>>> y[0] = 2
>>> x
[2, 2, 3] # x is y!
```

Mutability and the `is` operator

- Two objects can be *equal* without being the *same object*:

```
>>> x = [1,2,3]
>>> y = x[:] # Force copy
>>> x is y
False
>>> x == y
True
>>> y[0] = 2
>>> x
[1, 2, 3]
>>> y
[2, 2, 3]
```

- `x is y` implies `x == y`, but not the reverse!

Tuples

- In Python, a *tuple* is an *immutable* list.
- Tuple literals are written like lists, using parentheses instead of brackets.
- `tuple()` converts other types to a tuple.
- Do not support methods that modify the object, e.g. `sort()`, `append()`, `pop()`.

```
>>> x = (1,2,3)
>>> y = (9,) # Extra comma - why?
>>> print(x + y)
(1, 2, 3, 9)
```

Useful Links

Python String Methods:

- https://docs.python.org/3/library/stdtypes.html#string-methods

Python List Methods:

- https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Summary

- A `list()` is a mutable sequence of objects.
  - Lists can contain any type, including other lists.
  - Mutability is useful but can lead to surprising results.
- A `tuple()` is an immutable sequence of objects.
- A `for` loop can be used to repeat a statement body once for each element in a list, tuple, or string.
- The `range()` built-in function creates an object that represents a sequence of integers.