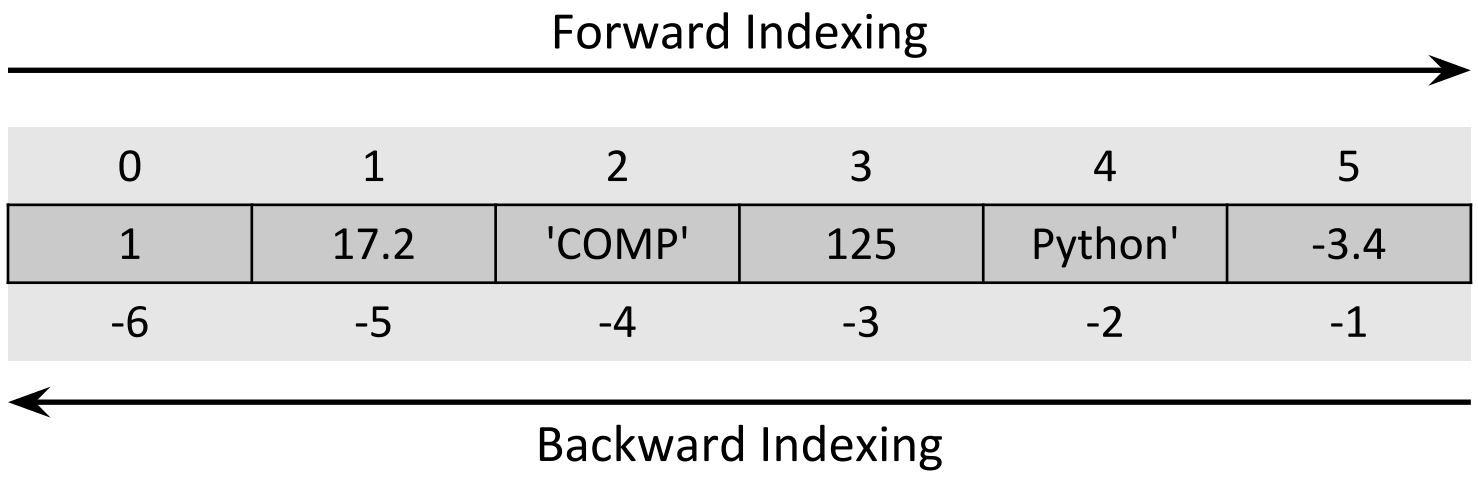
COMP 125 Programming with Python Lists



Mehmet Sayar

Koç University

Last Time: Console Programs

- Programs designed to be used with text input and output through a console (text terminal)

- Use `input()` to read in information from the user.
  - Make sure to convert the data to the correct type (from the string data type)!

- Use `print()` to display information for the user.
  - Make sure to convert the data to the correct type (from the int/float data type)

- Use a `while` loop to enable multiple runs of your program

Last Time: Strings

- A sequence of characters (letters, digits, symbols etc.)
- Use single (') or double quotes (") to define them
- Each character has an associated **index**
  - Indexing and slicing
- Strings can be **concatenated** using the + operator
- Strings are **immutable**

Last Time: Some String Operations

- `str.isupper()`, `str.islower()`
- `str.isalpha()`, `str.isdigit()`
- `str.upper()`, `str.lower()`

- Remember Python strings are immutable

Last Time: Type Conversion

- '123' is a `str`, `123` is an `int`
- Convert `int` to `str` to concatenate them
  - E.g. `print('Today is the' + str(18) + 'th of November')`
- Convert `str` to `int` to compare them (or use the `str` in math)
  - E.g. `int('123') == 123`
- More Examples

```
int('123') == 123
float('24.7') == 24.7
str(12345) == '12345'
str(20.19) == '20.19'
```

How can we store and organize data?

- **Collection**: A data structure used to store values as a single unit
- In most other programming languages, the most basic collection is an array which stores values of the same type.

- In Python, the most basic, and arguably one of the most useful, collection is the `List` data structure
- `List` is also a *sequence* (recall the `range` function and `strings`)

Sequences

- **Sequence**: an object that contains multiple items of data
  - The items are stored in sequence one after another

- Python provides different types of sequences, including lists and tuples
  - The difference between these is that a list is mutable, and a tuple is immutable

Lists

- A data type for storing values in a linear collection
- Python declaration:

```
[1, 2, 3, 4, 5]
```

- Using brackets ([ ]) to write in the code
- The list elements are separated with commas

Lists

- Can be used with multiple data types

```
['a', 'b', 'c', 'd', 'e']
[True, False, False]
```

- Can have varying number of items including 0 (empty list) and 1

```
[3.2]
[ ]
```

- Can include items of different data types

```
[ 1, 3.2, 'a', 'b', 'c', True]
```

List Indexing

- Indexing lists is like indexing strings, except with elements instead of characters

```
letters = ['a', 'b', 'c', 'd']
print(letters[0])
print(letters[3])
```

Output:

```
a
d
```

Length of Lists

- The **length** of a list is the number of items it contains, just like strings
- `len` function returns the length of any sequence such as a list

```
my_list = [ 1, 3.2, 'a', 'b', 'c', True]
len(my_list) → 6
```

- The index of last element is `len(list)-1`
- An `IndexError` exception is raised if an invalid index is used.
  - Can use the `len` function to prevent this exception when iterating over a list with a loop

Slicing

- Slice: a span of items that are taken from a sequence
- List slicing format: `list[start:end]`
- Spans a list containing copies of elements from `start` up to, **but not including**, end
- If start not specified, 0 is used for the start index
- If end not specified, len(list) is used for the end index
- Slicing expressions can include a step value and negative indexes relative to end of list

Slicing Lists

```
letters = ['a', 'b', 'c', 'd']
letters[2:4] → ['c', 'd']
letters[1:] → ['b', 'c', 'd']
letters[:2] → ['a', 'b']
letters[1:4:2] → ['b', 'd']
letters[1:3:2] → ['b']
letters[3:1] → []
letters[3:1:-1] → ['d', 'c']
```

More Slicing

```
lst = ['a', 'b', 'c', 'd', 'e’]
```

```
lst[::-1] → ['e', 'd', 'c', 'b', 'a’]
lst → ['a', 'b', 'c', 'd', 'e’]
```

```
lst[::2] → ['a', 'c', 'e']
```

(start_index, end_index, step)

Printing Lists

```
fruits = ['apple', 'banana', 'mango']
print(fruits)
```

Output

```
['apple', 'banana', 'mango']
```

Lists Are Mutable

- Mutable sequence: the items in the sequence can be changed
  - Lists are mutable, and so their elements can be changed

- An expression such as
  - `list[1] = new_value` can be used to assign a new value to a list element
  - Must use a valid index to prevent raising of an `IndexError` exception

Concatenating Lists

- Concatenate: join two things together (we have seen the string version)

- The + operator can be used to concatenate two lists
  - Cannot concatenate a list with another data type, such as a number

- The += augmented assignment operator can also be used to concatenate lists

Concatenating Lists

```
lst = [1, 2, 3, 4, 5]
lst += [6, 7]
lst → [1, 2, 3, 4, 5, 6, 7]

lst += 8 → TypeError
```

you can only use += for concatenating other lists!

Repetition Operator

- The * operator makes multiple copies of a list and joins them all together.

```
numbers = [0] * 5
numbers → [0, 0, 0, 0, 0]

numbers = [1, 2, 3] * 3
Numbers → [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Adding an Item to a List - append

- `append()`: adds a single element to the end of a list!

```
lst = [1, 2, 3, 4, 5]
lst.append(6)

lst → [1, 2, 3, 4, 5, 6]
```

Remove the Last Item of a List - pop

- `pop()`: removes the last element in a list and returns it

```
lst = [1, 2, 3, 4, 5]
last_elem = lst.pop()


last_elem → 5
lst → [1, 2, 3, 4]
```

Finding Out Whether an Item is in a List

- You can use the in operator to determine whether an item is contained in a list
  - General format: `item in list`
  - Returns `True` if the item is in the list, or `False` if it is not in the list

- Similarly you can use the `not` in operator to determine whether an item is not in a list

Finding Out Whether an Item is in a List

```
fruits = ['apple', 'banana', 'mango', 'kiwi']

'mango' in fruits
→ True

'broccoli' in fruits
→ False

'broccoli' not in fruits
→ True
```

Iterating over a List

- The for loop can be used to iterate over the items of a list

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
mango
```

Quick Exercise

- Write a function, `find_min()`, that returns the minimum float in a list
- Example:

```
find_min([2.3, 7.1, 10.6]) → 2.3
```

- Syntax reminder:

```
for elem in lst:
    # do something
```

Splitting Strings to Make Lists

- `split()` function splits strings where there are white spaces

```
s = 'I am comprised of words'
words = s.split()
```

```
words → ['I', 'am', 'comprised', 'of', 'words']
s → 'I am comprised of words
```

Splitting Strings to Make Lists

- `split(delimeter)` splits string where there is the delimiter

```
s = 'do,re,mi,fa,sol,la,ti'
notes = s.split(',')
notes → ['do', 're', 'mi', 'fa', 'sol', 'la', 'ti']

s = 'do, re, mi, fa, sol, la, ti'
notes = s.split(', ')
notes → ['do', 're', 'mi', 'fa', 'sol', 'la', 'ti']
```

Delimeter can be a general string

Range to List: Make a List of Ordered Numbers

- Recall the range function that creates an implicit sequence

```
numbers = list (range(5))
[0, 1, 2, 3, 4]
```

Type Conversion!

```
numbers = list (range(0, 10, 2))
[0, 2, 4, 6, 8]

numbers = list (range(10, -1, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Exercise

- Write a function that will take a string and a character that will return the words in the string starting with the given character:

```
def words_starting_with(sentence, character):
    …

words_starting_with('I love ice cream', 'i')
['I', 'ice']
```

Some List Methods

Table 7-1 A few of the list methods

Method	Description
`append(item)`	Adds *item* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *item* into the list at the specified *index*. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of *item* from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

List Methods and Useful Built-in Functions

- `append(item)`: used to add items to a list – item is appended to the end of the existing list
- `index(item)`: used to determine where an item is located in a list
  - Returns the index of the first element in the list containing `item`
  - Raises `ValueError` exception if `item` not in the list
- `insert(index, item)`: used to insert item at position index in the list
- `sort()`: used to sort the elements of the list in ascending order
- `remove(item)`: removes the first occurrence of `item` in the list
- reverse(): reverses the order of the elements in the list

List Methods and Useful Built-in Functions

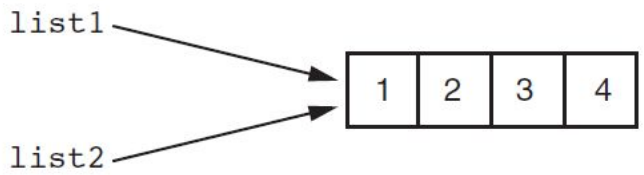
- `del` statement: removes an element from a specific index in a list
  - General format: `del list[i]`
  - The indices of the remaining part are updated!

- `min` and `max` functions: built-in functions that returns the item that has the lowest or highest value in a sequence
  - The sequence is passed as an argument

Copying Lists

- To make a copy of a list you must copy each element of the list
- Two methods to do this:
  - Creating a new empty list and using a for loop to add a copy of each element from the original list to the new list
  - Creating a new empty list and concatenating the old list to the new empty list
- `list1 = list2` does not work (or does not take a “copy”)!

Figure 7-4 `list1` and `list2` reference the same list



Processing Lists

- List elements can be used in calculations
- To calculate total of numeric values in a list use loop with accumulator variable
- E.g. to average numeric values in a list:
  - Calculate total of the values
  - Divide total of the values by `len(list)`
- List can be passed as an argument to a function
- List can be returned from a function
- Small exercise: Get a list of grades from a user via the console and find the average of them by dropping the lowest grade

Two Dimensional Lists

- Two-dimensional list: a list that contains other lists as its elements
  - “List of lists”
  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data
- To process data in a two-dimensional list need to use two indexes
- Typically use nested loops to process

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

	Column 0	Column 1	Column 2
Row 0	scores[0][0]	scores[0][1]	scores[0][2]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]

Exercise:
Write two-dimensional array filled with random numbers.

What about n-dimensions?