Python: Lists

1405 Instructor: Ruiqing (Sam) Cao

Data Types: Scalar & Non-Scalar

Type	Examples	Description
Numeric	`10,3.14,-5`	Numbers such as integers (`int`) or decimals (`float`)
Boolean	`True,False`	Logical values, True or False (`bool`)
String	`"Hello",'world'`	Text data, enclosed in quotes (`str`)
List	`[1,2,3],["a","b"]`	A sequence of items that *can* be chang
Tuple	`(1,2,3),("a","b")`	A sequence of items that *cannot* be cha (`tuple`)
Dictionary	`{"key":"value"}`	A collection of key-value pairs (`dict`)
Set	`{1,2,3},{"a","b"}`	An unordered collection of unique item

Scalar Types

Ordered Non-Scalar

Unordered Non-Scalar

Ordered Types: List (`list`)

- A `list` is an **ordered, mutable** collection of elements

➢Think: a finite space hotel with N rooms numbered from 0, 1, 2,…, N-1 → *That's what a list is !*

More flexible than an array: can hold ***values of different types***

→ For example, `['ab',True,1.0]` is a valid `list`

Ordered Types: List (`list`)

- Lists are an extremely useful data structure in Python and has many powerful applications

➢e.g., ***list comprehension***: more advanced but very useful

- A `tuple` is similar to a `list`, but it is immutable, faster and expressed in ( ) instead of [ ]

→ For example, `('ab',True,1.0)` is a `tuple`

Create a List

- Creating an *empty list*: `li=[]` or `li=list()`

- Creating a *list with elements*: `li=[0,1,2]`

In Python, indexes *Start at 0* (not 1) !

Smartest people ranked:

12. You
11. Can't
10. Rank
9. Them
8. Because
7. People
6. Are
5. All
4. Smart
3. In
2. Different
1. Ways
0. *Programmers*

Index, Modify, & Slice a List

- *Indexing*: access an element in a list by referencing its position inside a pair of square brackets: `li[0]` and `li[-1]`

Think: find people in a hotel room by referencing the room number

➢ *Modify* a list at the position referenced, for instance `li[0]=5`

- *Slicing*: access a subsequence by referencing the begin position and end position (plus 1) separated by ":" inside a pair of square brackets: `li[1:4]` (note the end position is 4-1=3 not 4)

Your Turn: List Indexing & Slicing

```
X = ['Some', 1, 'data', 2, 'here']
```

`X[3]` ?

`X[-2]` ?

`X[:4]` ?

`X[4:]` ?

What do these expressions evaluate to?

Your Turn: List Indexing & Slicing

```
X = ['Some', 1, 'data', 2, 'here']
```

`X[3]` ?

2

`X[-2]` ?

2

`X[:4]` ?

['Some', 1, 'data', 2]

`X[4:]` ?

['here']

What do these expressions evaluate to?

Generalized Slicing

`li[start:stop:step]` returns a slice of the list `li`

`start`: The index to begin slicing (*inclusive*)

`stop`: The index where slicing stops (*exclusive*)

`step`: The step size, which determines the interval between indices and the direction of slicing

- Positive step: forward
- Negative step: backward

- Example: li = [0, 1, 2, 3, 4, 5]

li[1:4:2] →[1,3]

li[:2:-1] →[5, 4, 3]

Your Turn: List Generalized Slicing

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

`X[::-1]` ?

`X[1:3:2]` ?

`X[4:1:-2]` ?

`X[1::3]` ?

What do these expressions evaluate to?

Your Turn: List Generalized Slicing

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

`X[::-1]` ?

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

`X[1:3:2]` ?

`[2]`

`X[4:1:-2]` ?

`[5,3]`

`X[1::3]` ?

`[2,5]`

What do these expressions evaluate to?

Append an Element to a List

- *Adding an element* to the *end* of a list using the append( ) method

```
li = [1,2,3]
li.append(4)
```

→ **li** becomes **[1,2,3,4]**

- A *mutating* ***method***: operates on an object (method) and modifies it directly (mutating) without returning a value
- append( ) always adds the element to the ***end*** of the list
- Only one ***element*** is added at a time

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

→ **li** becomes **[1,2,3,[4,5]]**

Sort a List

- *Sorting a list* using the `sort()` method or the `sorted()` function
- The default sort method and function sort elements in ascending order (from smallest to largest), but you can use the argument `reverse=True` for descending order (from largest to smallest)

```
li = [2,1,3]
li.sort()
```
→ `li` becomes `[1,2,3]`

```
li = [2,1,3]
li.sort(reverse=True)
```
→ `li` becomes `[3,2,1]`

```
sorted([2,1,3])
```
→ returns new list `[1,2,3]`

```
sorted([2,1,3],reverse=True)
```
→ Returns new list `[3,2,1]`

Check Membership & Count Occurrences

- Check whether an object is an element of the list

```
item in li
```

➢ *Returns a Boolean type* **`True`** *or* **`False`**

- Count the number of times an item appears as element in the list

```
li.count(item)
```

➢ Returns an `int` type equal to the occurrences of `item` in `li`

Your Turn: Append, Sort, Membership

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

```
X.append(0) ? [tricky]
X ?
```

```
X.sort() ? [tricky]
X ?
```

```
sorted(X) ?
```

```
[1,2] in X ? [tricky]
```

What do these expressions evaluate to?

Your Turn: Append, Sort, Membership

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

```
X.append(0) ? None
X ? [3,4,5,1,2,0]
```

```
X.sort() ? None
X ? [1,2,3,4,5]
```

```
sorted(X) ?
[1,2,3,4,5]
```

```
[1,2] in X ?
False
```

What do these expressions evaluate to?

Iterate Through a List

- Sometimes you need to automate performing the same operation on every element of a list

→ common approach is to *iterate through a list using* `for` *loop*

```
for item in li:
    ...
```

- More advanced but simple (once learned) and very clean solution is to use *list comprehension*

Your Turn: Iterate

```
X = [3,4,5,1,2,0]
```

```
s = 0
for num in X:
    s+=num
print(s)
```

What does this code DO?

What does this code PRINT?

Your Turn: Iterate

```
X = [3,4,5,1,2,0]
```

```
s = 0
for num in X:
    s+=num
print(s)
```

Iterate through the list X, and add the number in the current iteration to variable s

15

What does this code DO?

What does this code PRINT?

Combine Lists

- Use the method `extend()` to add all items in `li2` to `li1` (note: the method *directly modifies* `li1` without returning an object)

```
li1.extend(li2)
```

- *Return the combined list as an object* without modifying the original lists

```
bothli = li1 + li2
```

```
bothli = [*li1, *li2]
```

Your Turn: Combine

```
X = [3,4,5]
Y = [1,2,0]
```

Write down AT LEAST TWO different ways to combine items in these two lists. The items in Y should appear after those in X.

Your Turn: Combine

```
X = [3,4,5]
Y = [1,2,0]
```

Write down AT LEAST TWO different ways to combine items in these two lists. The items in Y should appear after those in X.

```
X.extend(Y)
```

```
X = X+Y
```

Use Case: Lists & Tabular Data

- You can use a list to store *one row or one column of a tabular data set* without variable names

gvkey (int)	name (str)	year (int)	sale_bn (float)
33175	SPOTIFY	2019	7.0
11217	VOLVO	2019	46.0

For instance, consider the above table

- Represent the first row as a list: `[33175,'SPOTIFY',2019,7.0]`
- Represent the second column as a list: `['SPOTIFY','VOLVO']`

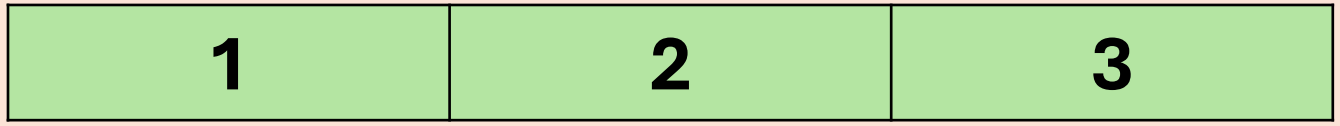
Exercises: Lists

1. Write a Python program that: Creates a list of 3 integers [1,-1,2]. Adds a new integer 3 to the end of the list. Sorts the list in descending order. Prints the modified list.

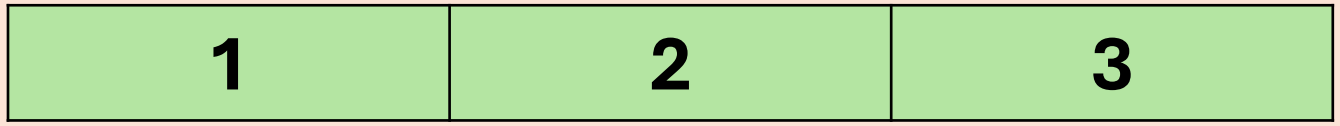
2. Write a Python program that: Creates a list `li = [1,2,3]`. Makes a slice of `li` using `sli = li[0:2]`. Change the third element of `sli` into 0. Prints the value of the original list `li` (did it change?) Makes a copy of `li` using `cli = li.copy()`. Change the second element of `cli` into `10`. Prints the value of the original list `li` (did it change?) Assigns it to a new list `new = li`. Changes the first element of `new` into `-9`. Prints the values of the original list `li` (did it change?) Explain the results.

Exercises: Lists

`li =`



`new =`



`cli =`


```
new = li
```

`new` refers to the same object as li, instead of being an (identical) copy of `li`

```
cli = li.copy()
```

`cli` becomes a copy of `li`, but not the same object as `li`

Immutable vs. Mutable Types

A key difference between immutable vs. mutable types is **how they are accessed in the memory**

- An immutable object is referenced by its entire value *(e.g., scalar)*
- A mutable object is referenced by its *address* (location in the memory), so *assigning it to a new object passes its address to the new object*, but doesn't pass its content or value **e.g., lists**

- Think: accessing an *immutable* is like being passed an apple🍏, while referencing a *mutable* is like being given the street address of a house🏚 (while the house itself and its residents can change)

Immutable vs. Mutable Types

- *Immutable* types: after an immutable type object is created, its values cannot be modified (i.e., any operation that modifies the object in fact creates a new object)

→ Not only scalar types, but also `str` and `tuple` are immutable

→ Immutable types cannot have *mutating* methods

- *Mutable* types: after a mutable type object is created, its values can be modified **without changing its identity** (i.e., you can add, remove, and change its elements without creating a new object)

→ `list`, `dict`, `set` are mutable

→ Mutable types *can* have *mutating* methods

Exercises: Immutable vs. Mutable Types

1. Write Python code for each of the following data types: numeric (`int` or `float`), `tuple`, `string`, `list`, `dictionary`, and `set`. Creates an object of each type. Assigns it to a new variable. Modifies the new object by changing its first element (if possible) or changing the entire object (if first element cannot be altered). Prints the original object.

2. Which of these types are immutable? Which are mutable? Explain how immutable and mutable types differ in in terms of whether the value of the original object changed in the above exercise.