Data Processing and Analysis in Python
Lecture 6
Lists and Dictionaries

UNIVERSITY OF MARYLAND

ROBERT H. SMITH SCHOOL OF BUSINESS

DR. ADAM LEE

Lists

- A **list** allows the programmer to manipulate a sequence of data values (items or elements) of any types
  - Each item in a list has a unique index that specifies its position (from 0 to length – 1)
- Examples:
  - Shopping list for the grocery store
  - Roster for an athletic team
  - Guest list for a wedding
  - Recipe, which is a list of instructions
  - Text document, which is a list of lines
  - Names in a phone book

List Literals

- Examples:

```
>>> first = [1, 2, 3, 4]
>>> ['apples', 'oranges', 'cherries']
>>> [[11, 12], [21, 22]]
```

- When an element is an expression, its value is included in the list:

```
>>> import math
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
>>> [x + 1, x + 2, x + 3]
[3, 4, 5]
```

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

List Constructor

- Lists of integers can be built using **range()**:

```
>>> second = range(1, 5)
>>> second
range(1, 5)
>>> list(second)
[1, 2, 3, 4]
```

- The **list()** function can build a list from any iterable sequence of elements:

```
>>> third = list("Hi there!")
>>> third
['H', 'i', ' ' , 't', 'h', 'e', 'r', 'e', '!']
```

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

Basic Operators

- Equality (==):

```
>>> first == second
False
>>> first == list(second)
True
```

- Concatenation (+) :

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
```

- Subscript ([]) :

```
>>> first[2:4]
[3, 4]
```

Method Len and Operator In

- Length **len()**:

```
>>> len(first)
4
```

- **in** detects the presence of an element:

```
>>> 3 in first
True
>>> 0 in first
False
```

Replacing an Element in a List

- A list is **mutable**:
  - Elements can be inserted, removed, or replaced
  - The list itself maintains its identity, but its state—its length and its contents—can change
  - Subscript operator is used to replace an element:

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

Method Insert

- The method **insert()** expects an integer index and the new element as arguments

```
>>> example = [1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
>>> example.insert(50, 50)
>>> example
[1, 10, 2, 25, 50]
```

Methods Append and Extend

- The method **append()** expects just the new element as an argument
  - adds the new element to the end of the list
- The method **extend()** adds the elements the end of the list

```
>>> example = [1, 2]
>>> example.append(3)
>>> example
[1, 2, 3]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 3, 11, 12, 13]
>>> example + [14, 15]
[1, 2, 3, 11, 12, 13, 14, 15]
```

Method Pop

- The method **pop()** is used to remove an element at the end or at a given position

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop() # Remove the last element
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0) # Remove the first element
1
>>> example
[2, 10, 11, 12]
```

Searching a List

- **in** determines an element's presence or absence, but does not return position of element (if found)
- Use method **index()** to locate an element's position in a list
  - Raises an error when the target element is not found

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

Sorting a List

- A list's elements are always ordered by position, but you can impose a natural ordering on them
  - For example, in alphabetical order
- When the elements can be related by comparing them <, ==, and >, they can be sorted
- The method **sort()** mutates a list by arranging its elements in ascending order

```
>>> example = [4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

Aliasing and Side Effects

- Mutable property of lists leads to interesting phenomena:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> second
[10, 99, 30]
```

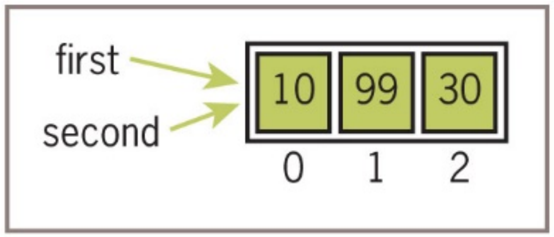

Figure 5-1 Two variables refer to the same list object

- first and second are aliases:
  - They refer to the exact same list object

Aliasing and Side Effects

- To prevent aliasing, create a new object and copy contents of original:

```
>>> third = first.copy()
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```

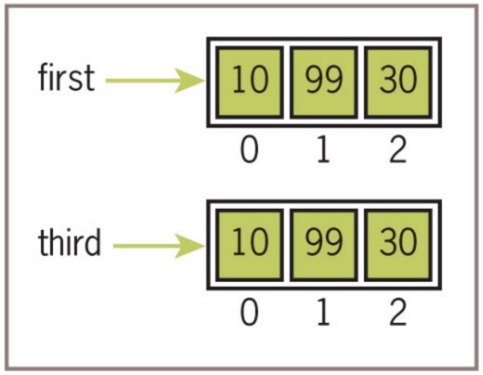

Figure 5-2 Two variables refer to different list objects

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

Equality: Object Identity and Structural Equivalence

- Programmers might need to see whether two variables refer to the exact same object or to different objects
  - Example, you might want to determine whether one variable is an alias for another
- The == operator returns True if the variables are aliases for the same object.
  - The first relation is called object identity
- Unfortunately, == also returns True if the contents of two different objects are the same
  - The second relation is called structural equivalence.
- The == operator has no way of distinguishing between these two types of relations.

Equality: Object Identity and Structural Equivalence

- Python's **is** operator can be used to test for object identity:

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first) # or first.copy()
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```

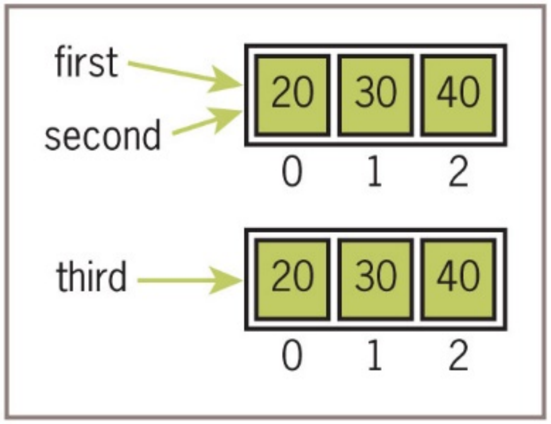

Figure 5-3 Three variables and two distinct list objects

Tuples

- A **tuple** resembles a list, but is immutable

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> fruits[0]
'apple'
>>> fruits[1]
'banana'
>>> fruits[1] = "orange"
TypeError: 'tuple' object does not support item
assignment
>>> fruits.count("apple")
1
>>> fruits.index("apple")
0
```

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

Sets

- A **set** is an unordered collection data type that is iterable, mutable and has no duplicate elements
- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set

```
>>> fruits = {"apple", "banana"}
>>> fruits.add("orange")
>>> fruits.issuperset({"apple"})
>>> fruits.issubset({"apple", "banana", "orange",
"peach"})
>>> fruits.union({"apple", "peach"})
>>> fruits.intersection({"apple", "peach"})
>>> fruits.difference({"apple", "peach"})
```

Dictionaries

- A **dict**/dictionary organizes information by association, not position
  - Example: When you use a dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn to the words beginning with "M"
- Data structures organized by association are also called tables or association lists
- In Python, a dictionary associates a set of keys with data values

Dictionary Literals

- A Python dictionary is written as a sequence of **key:value** pairs
  - Enclosed in curly braces { and }
  - Pairs, separated by commas, are sometimes called entries
  - A colon : separates a key and its value
- Examples:
  - A phone book: {'Ann':'301-476-3321', 'Bob':'240-351-7743'}
  - Personal information: {'Name':'Smith', 'Age':18}
  - An empty dictionary: {}
- Keys in a dictionary can be data of any immutable types, including other data structures
  - They are normally strings or integers

Adding Keys and Replacing Values

- Add a new key:value pair to a dictionary using []:
  <a dictionary>[<a key>] = <a value>
- Example:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name':'Sandy', 'occupation':'hacker'}
```

- Use [] also to replace a value at an existing key:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

Accessing Values

- Use **[]** to obtain the value associated with a key
  - If key is not present in dictionary, an error is raised

```
>>> info["name"]
'Sandy'
>>> info["job"]
KeyError: 'job'
```

- If the existence of a key is uncertain, test for it using the dictionary method **get()**

```
>>> if "job" in info:
 print(info["job"])
>>> info.get("job")
```

Removing Keys

- To delete an entry from a dictionary, remove its key using the method **pop()**
  - pop() expects a key and an optional default value as arguments

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name':'Sandy'}
```

Traversing a Dictionary

- To print all of the keys and their values:

```
>>> for key in info:
 print(key, ":", info[key])
```

- Alternative: Use the dictionary method **items()**
  - Entries are represented as tuples within the list

```
>>> grades = {90:'A', 80:'B', 70:'C'}
>>> list(grades)
[90, 80, 70]
>>> list(grades.items())
[(90, 'A'), (80, 'B'), (70, 'C')]
>>> for (key, value) in grades.items():
 print(key, ":", value)
```