
Severance, Charles

http://hdl.handle.net/2027.42/64959
Unless otherwise noted, the content of this course material is licensed under a Creative Commons Attribution 3.0 License. 
http://creativecommons.org/licenses/by/3.0/.

Copyright © 2009, Charles Severance.

You assume all responsibility for use and potential liability associated with any use of the material. Material contains copyrighted content, used in accordance with U.S. law. Copyright holders of content included in this material should contact open.michigan@umich.edu with any questions, corrections, or clarifications regarding the use of content. The Regents of the University of Michigan do not license the use of third party content posted to this site unless such a license is specifically granted in connection with particular content. Users of content are responsible for their compliance with applicable law. Mention of specific products in this material solely represents the opinion of the speaker and does not represent an endorsement by the University of Michigan. For more information about how to cite these materials visit http://michigan.educommons.net/about/terms-of-use.

Any medical information in this material is intended to inform and educate and is not a tool for self-diagnosis or a replacement for medical evaluation, advice, diagnosis or treatment by a healthcare professional. You should speak to your physician or make an appointment to be seen if you have questions or concerns about this information or your medical condition. Viewer discretion is advised: Material may contain medical images that may be disturbing to some viewers.
What is **not** a “Collection”

- Most of our variables have one value in them - when we put a new value in the variable - the old value is over written

```
$ python
Python 2.5.2 (r252:60911, Feb 22 2008, 07:57:53)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
>>> x = 2
>>> x = 4
>>> print x
4
```
What is a Collection?

- A collection is nice because we can put more than one value in them and carry them all around in one convenient package.

- We have a bunch of values in a single “variable”

- We do this by having more than one place “in” the variable.

- We have ways of finding the different places in the variable

(Luggage) CC: BY-SA: xajondee (Flickr) http://creativecommons.org/licenses/by-sa/2.0/deed.en
A Story of Two Collections..

- **List**
  - A linear collection of values that stay in order

- **Dictionary**
  - A “bag” of values, each with its own label

(Pringle's Can) CC:BY-NC Roadsidepictures (flickr) http://creativecommons.org/licenses/by-nc/2.0/deed.en
(Pringles) CC:BY-NC Cartel82 (flickr) http://creativecommons.org/licenses/by-nc/2.0/deed.en
(Chips) CC:BY-NC-SA Bunchofpants (flickr) http://creativecommons.org/licenses/by-nc-sa/2.0/deed.en
(Bag) CC:BY-NC-SA Monkeyc.net (flickr) http://creativecommons.org/licenses/by-nc-sa/2.0/deed.en
The Python List Object
The grades variable will have a list of values.

Append some values to the list.

Add up the values in the list using the \texttt{sum()} function.

What is in the list?

Figure the average...
What is in `grades`?

Make a copy of the entire `grades` list.

Change the second new grade (starts at [0])

The original `grades` are unchanged.
Looking in Lists...

• We use square brackets to look up which element in the list we are interested in.

• grades[2] translates to “grades sub 2”

• Kind of like in math $x_2$

```python
>>> print grades
[100, 97, 100]

>>> print grades[0]
100

>>> print grades[1]
97

>>> print grades[2]
100
```
Why lists start at zero?

- Initially it does not make sense that the first element of a list is stored at the zeroth position
- grades[0]
- Math Convention - Number line
- Computer performance - don’t have to subtract 1 in the computer all the time

Elevators in Europe!

(elevator) CC:BY marstheinfomage (flickr)
http://creativecommons.org/licenses/by-nc/2.0/deed.en
Fun With Lists

• Python has many features that allow us to do things to an entire list in a single statement

• Lists are powerful objects
```python
>>> lst = [21, 14, 4, 3, 12, 18]
>>> print lst
[21, 14, 4, 3, 12, 18]
>>> print 18 in lst
True
>>> print 24 in lst
False
>>> lst.append(50)
>>> print lst
[21, 14, 4, 3, 12, 18, 50]
>>> lst.remove(4)
>>> print lst
[21, 14, 3, 12, 18, 50]
>>> print lst.index(18)
4
>>> lst.reverse()
>>> print lst
[50, 18, 12, 3, 14, 21]
>>> lst.sort()
>>> print lst
[3, 12, 14, 18, 21, 50]
>>> del lst[2]
>>> print lst[3, 12, 18, 21, 33]
```

More functions for lists

```python
>>> a = [ 1, 2, 3 ]
>>> print max(a)
3
>>> print min(a)
1
>>> print len(a)
3
>>> print sum(a)
6
>>> 

http://docs.python.org/lib/built-in-funcs.html
```python
>>> print Ist
[3, 12, 14, 18, 21, 33]
>>> for xval in Ist:
    ...    print xval
... 3
... 12
... 14
... 18
... 21
... 33
```
List Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;list&gt;.append(x)</td>
<td>Add element x to end of list.</td>
</tr>
<tr>
<td>&lt;list&gt;.sort()</td>
<td>Sort (order) the list. A comparison function may be passed as parameter.</td>
</tr>
<tr>
<td>&lt;list&gt;.reverse()</td>
<td>Reverse the list.</td>
</tr>
<tr>
<td>&lt;list&gt;.index(x)</td>
<td>Returns index of first occurrence of x.</td>
</tr>
<tr>
<td>&lt;list&gt;.insert(i,x)</td>
<td>Insert x into list at index i.</td>
</tr>
<tr>
<td>&lt;list&gt;.count(x)</td>
<td>Returns the number of occurrences of x in list.</td>
</tr>
<tr>
<td>&lt;list&gt;.remove(x)</td>
<td>Deletes the first occurrence of x in list.</td>
</tr>
<tr>
<td>&lt;list&gt;.pop(i)</td>
<td>Deletes the ith element of the list and returns its value.</td>
</tr>
</tbody>
</table>
Quick Peek: Object Oriented
What “is” a List Anyways?

- A list is a **special** kind of variable
- Regular variables - integer
- Contain some data
- Smart variables - string, list
- Contain some data and **capabilities**

```python
>>> i = 2
>>> i = i + 1
>>> x = [1, 2, 3]
>>> print x
[1, 2, 3]
>>> x.reverse()
>>> print x
[3, 2, 1]
```

When we combine data + capabilities - we call this an “**object**”
One way to find out **Capabilities**

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;list&gt;.append(x)</code></td>
<td>Add element x to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.sort()</code></td>
<td>Sort (order) the list. A comparison function may be passed as parameter.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.reverse()</code></td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(x)</code></td>
<td>Returns index of first occurrence of x.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i,x)</code></td>
<td>Insert x into list at index i.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(x)</code></td>
<td>Returns the number of occurrences of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.remove(x)</code></td>
<td>Deletes the first occurrence of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.pop(i)</code></td>
<td>Deletes the ith element of the list and returns its value.</td>
</tr>
</tbody>
</table>

Buy a book and read it and carry it around with you.
The `dir()` command lists capabilities

- Ignore the ones with underscores - these are used by Python itself
- The rest are real operations that the object can perform
- It is like `type()` - it tells us something *about* a variable

```python
def x = list()
def type(x)
<type 'list'>
def dir(x)
['__add__', '__class__', '__contains__',
  '__delattr__', '__delitem__',
  '__delslice__', '__doc__',
  '__eq__', '__setitem__', '__setslice__',
  '__str__', 'append', 'count', 'extend',
  'insert', 'index', 'pop', 'remove', 'reverse', 'sort']```
Try `dir()` with a String

```python
>>> y = "Hello there"
>>> dir(y)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
 '__lt__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__',
 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind',
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```
What does $x = \text{list()}$ mean?

• These are called “constructors” - they make an empty list, str, or dictionary

• We can make a “fully formed empty” object and then add data to it using capabilities (aka methods)
Object Oriented Summary

• Variables (Objects) contain data and capabilities
• The `dir()` function asks Python to list capabilities
• We call object capabilities “methods”
• We can construct fresh, empty objects using constructors like `list()`
• Everything in Python (even constants) are objects
Python Dictionaries

http://en.wikipedia.org/wiki/Associative_array
Dictionaries

- Dictionaries are Python’s most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / Php
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net

http://en.wikipedia.org/wiki/Associative_array
(Bag) CC:BY-NC-SA Monkeyc.net (flickr)
http://creativecommons.org/licenses/by-nc-sa/2.0/deed.en
Dictionaries

• Lists label their entries based on the position in the list

• Dictionaries are like bags - no order

• So we mark the things we put in the dictionary with a “tag”

```python
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print purse
{'money': 12, 'tissues': 75, 'candy': 3}
>>> print purse['candy']
3
>>> purse['candy'] = purse['candy'] + 2
>>> print purse
{'money': 12, 'tissues': 75, 'candy': 5}
```
```python
>>> purse = dict()

>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75

>>> print purse
{'money': 12, 'tissues': 75, 'candy': 3}

>>> print purse['candy']
3

>>> purse['candy'] = purse['candy'] + 2

>>> print purse
{'money': 12, 'tissues': 75, 'candy': 5}
```
Lookup in Lists and Dictionaries

- **Dictionaries** are like **Lists** except that they use **keys** instead of **numbers** to look up **values**

```python
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print lst
[21, 183]
>>> lst[0] = 23
>>> print lst
[23, 183]
```

```python
>>> ddd = dict()
>>> ddd["age"] = 21
>>> ddd["course"] = 182
>>> print ddd
{'course': 182, 'age': 21}
>>> ddd["age"] = 23
>>> print ddd
{'course': 182, 'age': 23}
```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print lst
[21, 183]
>>> lst[0] = 23;
>>> print lst
[23, 183]

>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print ddd
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print ddd
{'course': 182, 'age': 23}
# Dictionary Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dict&gt;.has_key(&lt;key&gt;)</code></td>
<td>Returns true if dictionary contains the specified key, false if it doesn’t.</td>
</tr>
<tr>
<td><code>&lt;key&gt; in &lt;dict&gt;</code></td>
<td>Same as has_key</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.keys()</code></td>
<td>Returns a list of the keys.</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.values()</code></td>
<td>Returns a list of the values.</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.items()</code></td>
<td>Returns a list of tuples (key, value) representing the key-value pairs.</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.get(&lt;key&gt;, &lt;default&gt;)</code></td>
<td>If key is not in the dictionary, returns default; otherwise returns the value for key.</td>
</tr>
<tr>
<td><code>del &lt;dict&gt;[&lt;key&gt;]</code></td>
<td>Delete the specified entry.</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.clear()</code></td>
<td>Delete all entries.</td>
</tr>
</tbody>
</table>
Dictionary Literals (Constants)

- Dictionary literals use curly braces and have a list of key : value pairs
- You can make an empty dictionary using empty curly braces

```python
>>> jjj = { 'chuck' : 1, 'fred' : 42, 'jan': 100}
>>> print jjj
{'jan': 100, 'chuck': 1, 'fred': 42}
>>> ooo = {}
>>> print ooo
{}
```
Dictionary Patterns

- One common use of dictionary is counting how often we “see” something

```python
>>> ccc = dict()
>>> ccc["csev"] = 1
>>> ccc["cwen"] = 1
>>> print ccc
{'csev': 1, 'cwen': 1}
>>> ccc["cwen"] = ccc["cwen"] + 1
>>> print ccc
{'csev': 1, 'cwen': 2}
```
Dictionary Patterns

• It is an **error** to reference a key which is not in the dictionary

• We can use the **in** operator to see if a key is in the dictionary

```python
>>> ccc = dict()
>>> print ccc["csev"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'csev'
>>> print "csev" in ccc
False
```
ccc = dict()
if "csev" in ccc:
    print "Yes"
else:
    print "No"
ccc["csev"] = 20
if "csev" in ccc:
    print "Yes"
else:
    print "No"
Dictionary Counting

• Since it is an error to reference a key which is not in the dictionary

• We can use the dictionary get() operation and supply a default value if the key does not exist to avoid the error and get our count started.

```python
>>> ccc = dict()
>>> print ccc.get("csev", 0)
0
>>> ccc["csev"] = ccc.get("csev",0) + 1
>>> print ccc
{'csev': 1}
```
What `get()` effectively does...

- The `get()` method basically does an implicit if checking to see if the key exists in the dictionary and if the key is not there - return the default value.

- The main purpose of `get()` is to save typing this four line pattern over and over.

```python
# Example

d = dict()
x = d.get("fred", 0)
```

```python
# Example

d = dict()
if "fred" in d:
    x = d["fred"]
else:
    x = 0
```
Retrieving lists of **Keys** and **Values**

- You can get a list of **keys**, **values** or **items (both)** from a dictionary

```python
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print jjj.keys()
['jan', 'chuck', 'fred']
>>> print jjj.values()
[100, 1, 42]
>>> print jjj.items()
[('jan', 100), ('chuck', 1), ('fred', 42)]
```
• We loop through the key-value pairs in a dictionary using *two* iteration variables

• Each iteration, the first variable is the key and the second variable is the corresponding value

```python
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for aaa,bbb in jjj.items() :
...    print aaa, bbb
...
jan 100
chuck 1
fred 42
```
Dictionary Maximum Loop

```
$ cat dictmax.py
jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
print j jj

maxcount = None
for person, count in j jj.items() :
    if maxcount == None or count > maxcount :
        maxcount = count
        maxperson = person

print maxperson, maxcount
```

```
$ python dictmax.py
{'jan': 100, 'chuck': 1, 'fred': 42}
jan 100
```

None is a special value in Python. It is like the “absense” of a value. Like “nothing” or “empty”.
Dictionaries are not Ordered

• Dictionaries use a Computer Science technique called “hashing” to make them very fast and efficient

• However hashing makes it so that dictionaries are not sorted and they are not sortable

• Lists and sequences maintain their order and a list can be sorted - but not a dictionary

http://en.wikipedia.org/wiki/Hash_function
Dictionaries are not Ordered

>>> dict = { "a" : 123, "b" : 400, "c" : 50 }
>>> print dict
{'a': 123, 'c': 50, 'b': 400}

Dictionaries have no order and cannot be sorted. Lists have order and can be sorted.

>>> lst = dict()
>>> lst.append("one")
>>> lst.append("and")
>>> lst.append("two")
>>> print lst
['one', 'and', 'two']

>>> lst.sort()
>>> print lst
['and', 'one', 'two']

http://en.wikipedia.org/wiki/Hash_function
Summary: Two Collections

• **List**
  - A linear collection of values that stay in order

• **Dictionary**
  - A “bag” of values, each with its own label / tag
What do we use these for?

- **Lists** - Like a Spreadsheet - with columns of stuff to be summed, sorted. Also when pulling strings apart - like `string.split()`

- **Dictionaries** - For keeping track of (keyword,value) pairs in memory with very fast lookup. It is like a small in-memory database. Also used to communicate with databases and web content.