

Advanced Data Structures

- Lists
- Tuples
- Sets
- Dictionaries



Introduction to lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = [100, 200, 300, 400, 500]
```

```
>>> a
```

```
[100, 200, 300, 400, 500]
```

Can be accessed with index values like arrays in C/C++/Java.

```
>>> a[0]
```

```
100
```

Access the 2nd element from the end of the list

```
>>> a[-2]
```

```
400
```

Slicing: Extract values from the list within a given index range.

For eg. from the 2nd element of the given list to the last but one element.

```
>>> a[1:-1]
[200, 300, 400]
```

Some special cases

`listobject[:x]`

- If x is a positive value, then x number of entries from the beginning of the list would be sliced

```
>>a[:3]
```

```
[100, 200, 300]
```

- If x is a negative value, for eg -2, then all the entries from the list except the last 2 would be sliced

```
>>> a[:-2]
```

```
[100, 200, 300]
```

Some special cases(contd...)

`listobject[x:]`

- If x is positive, all the entries except for the first x entries would be sliced

```
>>> a[2:]  
[300, 400, 500]
```

- If x is negative, for eg -2, then 2 entries from the end of the list would be sliced

```
>>> a[-2:]  
[400, 500]
```



Assignment to slices is also possible, and this can even change the size of the list or clear it entirely

```
>>> a[0:2] = [1, 12]
>>> a
[1, 12, 300, 400, 500]
```

Removing a slice

```
>>> a[0:2] = []
>>> a
[300, 400, 500]
```

Assignment to slices (special case)

You can empty the whole list with

```
>>> a[:] = []
```

```
>>> a  
[]
```

Nesting lists

```
>>> q=[2,3]
>>> p=[1,q,4]
>>> p
[1, [2, 3], 4]
```

```
>>> p[1]
[2, 3]
```

Methods of the list data type

- `list.append(x)`

Add an item to the end of the list

- `list.extend(L)`

Extend the list by appending all the items in the given list



- `list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`

- `list.remove(x)`

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

- `list.index(x)`

Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

- `list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list

- `list.count(x)`

Return the number of times `x` appears in the list.

- `list.sort()`

- `list.reverse()`



- Another way to remove entries from a list is the use of the del statement

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Tuples

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5) # Tuples may
    be nested
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4,
    5))
```

- Construct a single element tuple

```
>>> singleton = 'hello',  
# note trailing comma  
>>> singleton  
( 'hello', )
```

The statement `t = 12345, 54321, 'hello!'`
is an example of tuple packing



- Sequence unpacking

Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence

```
>>> x, y, z = t
```



Sets

A set is an unordered collection with no duplicate elements. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.



```
>>> basket = ['apple', 'orange', 'apple',  
              'pear', 'orange', 'banana']  
>>> fruit = set(basket)  
>>> fruit  
set(['orange', 'pear', 'apple', 'banana'])
```

```
# fast membership testing
```

```
>>> 'orange' in fruit  
True  
>>> 'crabgrass' in fruit  
False
```

Set operations on unique letters from two words

```
>>> a = set('abracadabra')
```

```
>>> b = set('alacazam')
```

```
# unique letters in a
```

```
>>> a
```

```
set(['a', 'r', 'b', 'c', 'd'])
```

```
# letters in a but not in b
```

```
>>> a - b
```

```
set(['r', 'd', 'b'])
```

```
# letters in either a or b
```

```
>>> a | b
```

```
set(['a', 'c', 'r', 'd', 'b', 'm', 'z',  
    'l'])
```

```
# letters in both a and b
```

```
>>> a & b
```

```
set(['a', 'c'])
```

```
# letters in a or b but not both
```

```
>>> a ^ b
```

```
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Dictionaries

Unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary)

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack':
 4098}
```

The `keys()` method of the dictionary data type returns a list of the keys.

```
>>> tel.keys()  
['sape', 'jack', 'guido']
```

Both keys and values can either be strings or numbers.

- Build a dictionary from key-value pairs stored as tuples

```
>>> dict([('sape', 4139), ('guido',  
    4127), ('jack', 4098)])  
{'sape': 4139, 'jack': 4098, 'guido':  
    4127}
```

Note the dict() constructor

Looping techniques

Use the `iteritems()` method to loop through dictionaries

```
>>> knights = {'gallahad': 'the pure',  
               'robin': 'the brave'}  
>>> for k, v in knights.iteritems():  
...     print k, v  
...  
gallahad the pure  
robin the brave
```

Looping over two sequences at the same time

Use the zip() method

```
>>> a = ['foo', 'bar']
>>> b = [42, 0]
>>> for i,j in zip(a,b):
...     print "%s:%d" % (i,j)
...
foo:42
bar:0
```

File objects

- Quick introduction to file handling

`open()` returns a file object, and is most commonly used with two arguments: `'open(filename, mode)'`

```
>>> f = open('temp.txt', 'w')
```

```
>>> f
```

```
<open file 'temp.txt', mode 'w' at  
0xb7d4c5c0>
```

- 'r'

The file must already exist, and it is opened in read-only mode.

- 'w'

The file is opened in write-only mode. The file is truncated and overwritten if it already exists, or created if it does not exist.



- 'a'

The file is opened in write-only mode. The file is kept intact if it already exists, and the data you write is appended to what's already in the file. The file is created if it does not exist.

- 'r+'

The file must already exist and is opened for both reading and writing, so all methods of *f* can be called .



- 'w+'

The file is opened for both reading and writing, so all methods of *f* can be called. The file is truncated and overwritten if it already exists, or created if it does not exist.

- 'a+'

The file is opened for both reading and writing, so all methods of *f* can be called. The file is kept intact if it already exists, and the data you write is appended to what's already in the file. The file is created if it does not exist.

- Reading from a file object

```
f.read(size)
```

The `size` is an optional argument, which when specified reads that many bytes at most. If nothing is specified, it reads the contents of the whole file and returns it as a string.



```
f.readline()
```

Reads a line from the file, returns it as a string and moves on to the next line.

```
f.readlines()
```

This returns a list of all the lines in the file. Can be used to loop through the contents of a file.

For eg

```
>>>for line in f.readlines():  
...     print line  
...
```

- Writing to a file

```
f.write(string)
```

It's simple!

- Closing the file object

```
f.close()
```

Close the file and free up system resources being used by the file object.

A few important string functions

- `s.split()`

Splits a string consisting of words separated by whitespaces and returns a list of the words.

- `s.rstrip()`

Remove whitespaces from the leading end or the right side

- `s.lstrip()`



Exceptions

Errors detected during execution are called exceptions and are not unconditionally fatal.



Handling exceptions

Overall structure....

```
try:
    #code that may cause runtime
    #error
except [errortype]:
    #handle the error
[else:]
    #do something if no error raised
[finally:]
    #irrespective of error being
    #raised or not do something
```

```
>>> while True:
...     try:
...         x = int(raw_input("enter a
number: "))
...         break
...     except ValueError:
...         print "Oops!"
... 
```



- Raising exceptions

```
>>> raise NameError, 'HiThere'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
NameError: HiThere
```

Try this: Accept two numbers, raise a ValueError when division by 0 is attempted and handle it too
