Sets

identity maps, unordered collection

Sets

- [] for lists, () for tuples, {} for dicts, and {} for sets (2.7)
- construction from lists, tuples, dicts (keys), and strs
- in, not in, add, remove

```
>>> a = {1, 2}
a
>> set([1, 2])
>>> a = set((1,2))
>>> a
set([1, 2])
>>> b = set([1,2])
>>> a == b
True
>>> c = set({1:'a', 2:'b'})
>>> c
set([1, 2])
```

```
>>> a = set([])
>>> 1 in a
False
>>> a.add(1)
>>> a.add('b')
>>> a
set([1, 'b'])
>>> a.remove(1)
>>> a
set(['b'])
```

Set Operations

- union, intersection, difference, is_subset, etc..

demo

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
set(['r', 'd', 'b'])
>>> a | b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b
set(['a', 'c'])
>>> a ^ b
set(['r', 'd', 'b', 'm', 'z', 'l'])
>>> a |= b
>>> a
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
```

set and frozenset type

Operation	Equivalent	Result
`len(s)`		cardinality of set *s*
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*
`s.update(t)`	`s |= t`	return set *s* with elements added from *t*
`s.intersection_update(t)`	`s &= t`	return set *s* keeping only elements also found in *t*
`s.difference_update(t)`	`s -= t`	return set *s* after removing elements found in *t*
`s.symmetric_difference_update(t)`	`s ^= t`	return set *s* with elements from *s* or *t* but not both
`s.add(x)`		add element *x* to set *s*
`s.remove(x)`		remove *x* from set *s*; raises KeyError if not present
`s.discard(x)`		removes *x* from set *s* if present
`s.pop()`		remove and return an arbitrary element from *s*; rais
`s.clear()`		remove all elements from set *s*

Basic `import` and I/O

`import` and I/O

- similar to `import` in Java
- File I/O much easier than Java

demo

```
import sys
for line in sys.stdin:
   print line.split()
```

or

```
from sys import *
for line in stdin:
   print line.split()
```

```
import System;
```

Java

```
import System.*;
```

```
>>> f = open("my.in", "rt")
>>> g = open("my.out", "wt")
>>> for line in f:
...     print >> g, line,
... g.close()
```

file copy

to read a line:

```
line = f.readline()
```

to read all the lines:

```
lines = f.readlines()
```

note this comma!

`import` and `__main__`

- multiple source files (modules)
  - C: `#include "my.h"`
  - Java: `import My`
- demo
- handy for debugging

foo.py

demo

```
def pp(a):
   print " ".join(a)

if __name__ == "__main__":
   from sys import *
   a = stdin.readline()
   pp (a.split())
```

interactive

```
>>> import foo
>>> pp([1,2,3])
1 2 3
```

Quiz

- Palindromes

  `abcba`

  - read in a string from standard input, and print `True` if it is a palindrome, print `False` if otherwise

```
def palindrome(s):
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and palindrome(s[1:-1])

if __name__ == '__main__':

    import sys
    s = sys.stdin.readline().strip()
    print palindrome(s)
```

Functional Programming

map and filter

- intuition: function as data
- we have already seen functional programming a lot!
  - list comprehension, custom comparison function

```
map(f, a)
filter(p, a)
```

```
[ f(x) for x in a ]
[ x for x in a if p(x) ]
```

```
map(f, filter(p, a))
```

```
[ f(x) for x in a if p(x) ]
```

```
>>> map(int, ['1','2'])
[1, 2]
>>> " ".join(map(str, [1,2]))
1 2
```

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> filter(is_even, [-1, 0])
[0]
```

demo

lambda

- map/filter in one line for custom functions?
  - "anonymous inline function"
- borrowed from LISP, Scheme, ML, OCaml

```
>>> f = lambda x: x*2
>>> f(1)
2
>>> map (lambda x: x**2, [1, 2])
[1, 4]
>>> filter (lambda x: x > 0, [-1, 1])
[1]
>>> g = lambda x,y : x+y
>>> g(5,6)
11
>>> map (lambda (x,y): x+y, [(1,2), (3,4)])
[3, 7]
```

demo

more on lambda

```
>>> f = lambda : "good!"
>>> f
<function <lambda> at 0x381730>
>>> f()
'good!'
```

lazy evaluation

```
>>> a = [5, 1, 2, 6, 4]
>>> a.sort(lambda x,y : y - x)
>>> a
[6, 5, 4, 2, 1]
```

custom comparison

```
>>> a = defaultdict(lambda : 5)
>>> a[1]
5
>>> a = defaultdict(lambda : defaultdict(int))
>>> a[1]['b']
0
```

demo

Basic Sorting

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> print a
[1, 2, 3, 4, 5]

>>> a = [5, 2, 3, 1, 4]
>>> a.sort(reverse=True)
>>> a
[5, 4, 3, 2, 1]

>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a.reverse()
>>> a
[5, 4, 3, 2, 1]
```

sort() is in-place,
but sorted() returns new copy

```
>>> a = [5, 2, 3, 1, 4]
>>> sorted(a)
[1, 2, 3, 4, 5]
>>> a
[5, 2, 3, 1, 4]
```

Built-in and custom cmp

```
>>> a = [5, 2, 3, 1, 4]
>>> def mycmp(a, b):
        return b-a

>>> sorted(a, mycmp)
[5, 4, 3, 2, 1]

>>> sorted(a, lambda x,y: y-x)
[5, 4, 3, 2, 1]

>>> a = zip([1,2,3], [6,4,5])
>>> a.sort(lambda (_,y1), (__, y2): y1-y2)
>>> a
[(2, 4), (3, 5), (1, 6)]
>>> a.sort(lambda (_,y1), (_, y2): y1-y2)
SyntaxError: duplicate argument '_' in function definition
```

demo

Sorting by Keys or Key mappings

```
>>> a = "This is a test string from Andrew".split()
>>> a.sort(key=str.lower)
>>> a
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

>>> import operator
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3), ('b', 1)]

>>> L.sort(key=operator.itemgetter(1))
>>> L
[('d', 1), ('b', 1), ('c', 2), ('b', 3), ('a', 4)]

>>> sorted(L, key=operator.itemgetter(1, 0))
[('b', 1), ('d', 1), ('c', 2), ('b', 3), ('a', 4)]

>>> operator.itemgetter(1,0)((1, 2, 3))
(2, 1)
```

demo

sort by two keys

lambda for key mappings

- you can use lambda for both custom cmp and key map

```
>>> a = "This is a test string from Andrew".split()
>>> a.sort(lambda x, y: cmp(x.lower(), y.lower()))
>>> a
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

>>> a.sort(key=lambda x: x.lower())

>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3), ('b', 1)]

>>> L.sort(key=lambda (_, y): y)
>>> L
[('d', 1), ('b', 1), ('c', 2), ('b', 3), ('a', 4)]

>>> sorted(L, key=lambda (x, y): (y, x))
[('b', 1), ('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Decorate-Sort-Undecorate

```
>>> words = "This is a test string from Andrew.".split()

>>> deco = [ (word.lower(), i, word) for i, word in \
... enumerate(words) ]

>>> deco.sort()

>>> new_words = [ word for _, _, word in deco ]

>>> print new_words
['a', 'Andrew.', 'from', 'is', 'string', 'test', 'This']
```

demo

- Most General
- Faster than custom cmp (or custom key map) -- why?
- stable sort (by supplying index)

Memoized Recursion v1

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib (n-1) + fib (n-2)
```

```
fibs = {0:1, 1:1}
def fib(n):
    if n in fibs:
        return fibs[n]
    fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]
```

can we get rid of the global variable?

Memoized Recursion v2

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib (n-1) + fib (n-2)
```

```
def fib(n, fibs={0:1, 1:1}):
    if n in fibs:
        return fibs[n]
    fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
    return fibs[n]
```

Memoized Recursion v3

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
>>> fib(3)
1 {1: 1}
0 {0: 1, 1: 1}
2 {0: 1, 1: 1, 2: 2}
3 {0: 1, 1: 1, 2: 2, 3: 3}
3
>>> fib(2)
2
>>> print fibs
Error!
```

draw the tree!

```
def fib(n, fibs={0:1, 1:1}):
    if n in fibs:
        return fibs[n]
    fibs[n] = fib(n-1) + fib(n-2)
#   print n, fibs
    return fibs[n]
```

the **fibs** variable has a weird closure!! feature or bug?

most people think it's a bug, but Python inventor argues it's a feature.

Memoized Recursion v4

- Fibonacci revisited

```
def fib(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return b
```

```
>>> fib(4)
{0: 1, 1: 1, 2: 2}
{0: 1, 1: 1, 2: 2, 3: 3}
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5}
5
>>> fib(3)
{0: 1, 1: 1, 2: 2}
{0: 1, 1: 1, 2: 2, 3: 3}
3
```

```
def fib(n, fibs=None):
    if fibs is None:
        fibs = {0:1, 1:1}
    if n in fibs:
        return fibs[n]
    fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
#   print n, fibs
    return fibs[n]
```

this is so far the cleanest way to avoid the bug.

Implementation

- lists, tuples, and strings
  - random access: O(1)
  - insertion/deletion/in: O(n)
- dict
  - in/random access: almost O(1)
  - insertion/deletion: almost O(1)
  - but no linear ordering!

Pythonic Styles

- do not write ... when you can write ...

`for key in d.keys():`	`for key in d:`
`if d.has_key(key):`	`if key in d:`

`i = 0`
`for x in a:`
`    ...`
`    i += 1`
	`for i, x in enumerate(a):`

`a[0:len(a) - i]`	`a[:-i]`
`for line in \` `    sys.stdin.readlines():`	`for line in sys.stdin:`

`for x in a:`
`    print x,`
`print`
	`print " ".join(map(str, a))`

`s = ""`
`for i in range(lev):`
`    s += " "`
`print s`
	`print " " * lev`