On to Python 3...
“Hello, World”

- **C**

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
    printf("Hello, World!\n");
}
```

- **Java**

```java
public class Hello
{
    public static void main(String argv[])
    {
        System.out.println("Hello, World!");
    }
}
```

- **now in Python**

```python
print("Hello, World!")
```
void print_array(char* a[], int len)
{
    int i;
    for (i = 0; i < len; i++)
    {
        printf("%s\n", a[i]);
    }
}

C

Python

for element in list:
    print(element)

only indentations
no \{ \} blocks!

or even simpler:
print list

for ... in ...:
    ...

no C-style for-loops!

for (i = 0; i < 10; i++)
Reversing an Array

In Java:

```java
static int[] reverse_array(int a[])
{
    int [] temp = new int[ a.length ];
    for (int i = 0; i < len; i++)
    {
        temp [i] = a [a.length - i - 1];
    }
    return temp;
}
```

In Python:

```python
def rev(a):
    if a == []:  # singleton list
        return []
    else:
        return rev(a[1:]) + [a[0]]
```

or even simpler:

```python
a.reverse()  # built-in list-processing function
```

Java:

- No need to specify argument and return types!
- Python will figure it out. (Dynamically typed)
Quick-sort

```java
public void sort(int low, int high) {
    if (low >= high) return;
    int p = partition(low, high);
    sort(low, p);
    sort(p + 1, high);
}

int partition(int low, int high) {
    int pivot = a[low];
    int i = low - 1;
    int j = high + 1;
    while (i < j) {
        i++;
        while (a[i] < pivot) i++;
        j--;
        while (a[j] > pivot) j--;
        if (i < j) swap(i, j);
    }
    return j;
}

void swap(int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```python
def sort(a):
    if a == []:
        return []
    else:
        pivot = a[0]
        left = [x for x in a if x < pivot]
        right = [x for x in a[1:] if x >= pivot]
        return sort(left) + [pivot] + sort(right)
```

Python

```python
def sort(a):
    if a == []: return []
    else:
        pivot = a[0]
        left = [x for x in a if x < pivot]
        right = [x for x in a[1:] if x >= pivot]
        return sort(left) + [pivot] + sort(right)
```

Java

```python
{x | x ∈ a, x < pivot}
```

Java

**smaller semantic-gap!**

`how about return [sort(left)] + [pivot] + [sort(right)]` got an error??
Python is...

• a scripting language (strong in text-processing)
  • interpreted, like Perl, but much more elegant
• a very high-level language (closer to human semantics)
  • almost like pseudo-code!
• procedural (like C, Pascal, Basic, and many more)
• but also object-oriented (like C++ and Java)
• and even functional! (like ML/OCaml, LISP/Scheme, Haskell, etc.)
• from today, you should use Python for everything
  • not just for scripting, but for serious coding!
Let’s take a closer look...
Python Interpreter

• Three ways to run a Python program

1. Interactive
   • like DrJava
     >>>> for i in range(5):
     ...     print i,
     ...     
     0 1 2 3 4

2. (default) save to a file, say, foo.py
   • in command-line: python3 foo.py

3. add a special line pointing to the default interpreter
   • add #!/usr/bin/env python3 to the top of foo.py
   • make foo.py executable (chmod +x foo.py)
   • in the command-line: ./foo.py
Switching to Python3

• starting from this term, we’ll be using Python 3!
  • many libraries have dropped or will drop support of Python2
  • Python 3.x is not backward compatible with Python 2.x

• you can use python3 on school machine “flip”
  • you can ssh to access.engr.oregonstate.edu from home

• or you can install Python 3 on your own mac/windows
  • anaconda is highly recommended (esp. for deep learning)

<flip1:~> python3
Python 3.4.5 (default, Dec 11 2017, 14:22:24)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 

[<lhuang@Mac OS X:~>] which python3
/anaconda3/bin/python3
Major Differences b/w 2.x and 3.x

- `print(...)`
- lazy by default: `range` vs. `xrange`, `zip` vs. `izip`, ...
- `division: / vs. //`
Basic Python Syntax
Numbers and Strings

• like Java, Python has built-in (atomic) types
  • numbers (int, float), bool, string, list, etc.
  • numeric operators: + – * / ** %

```
>>> a = 5
>>> b = 3
>>> type (5)
<type 'int'>
>>> a += 4
>>> a
9

>>> 5/2
2.5
>>> 5/2.
2.5
>>> 5 // 2
2

>>> s = "hey"
>>> s + " guys"
'hey guys'
>>> len(s)
3
>>> s[0]
'h'
>>> s[-1]
'y'

no i++ or ++i
```
Assignments and Comparisons

```python
>>> a = b = 0
>>> a
0
>>> b
0

>>> a, b = 3, 5
>>> a + b
8
>>> (a, b) = (3, 5)
>>> a + b
8
>>> a, b = b, a
(swap)

>>> a = b = 0
>>> a == b
True
>>> type (3 == 5)
<type 'bool'>
>>> "my" == 'my'
True

>>> (1, 2) == (1, 2)
True
>>> 1, 2 == 1, 2
???
>>> (1, False, 2)
(1, False, 2)
>>> (1, 2) == 1, 2
???
```
for loops and range()

- **for** always iterates through a list or sequence

```python
>>> sum = 0
>>> for i in range(10):
...     sum += i
... print(sum)
45

>>> for word in ["welcome", "to", "python"]:  
...     print(word, end=' ')
... welcome to python
```

```java
Java 1.5
```
```java
    foreach (String word : words)
        System.out.println(word)
```

```python
>>> range(5), range(4,6), range(1,7,2)  
([0, 1, 2, 3, 4], [4, 5], [1, 3, 5])
```
while loops

- very similar to `while` in Java and C
- **but be careful**
  - `in` behaves differently in `for` and `while`
- `break` statement, same as in Java/C

```python
>>> a, b = 0, 1
>>> while b <= 5:
...       print(b)
...       a, b = b, a+b
... 1
1
1
2
3
5
```

- fibonacci series
Conditionals

```python
>>> if x < 10 and x >= 0:
...     print(x, "is a digit")
...
>>> False and False or True
True
>>> not True
False
```

```python
>>> if 4 > 5:
...     print("foo")
... else:
...     print("bar")
... bar
```

```python
>>> print("foo" if 4 > 5 else "bar")
... bar
```

C/Java

```c
printf( (4>5)? "foo" : "bar" );
```
if ... elif ... else

>>> a = "foo"

>>> if a in ["blue", "yellow", "red"]:
...     print(a + " is a color")
...     else:
...         if a in ["US", "China"]:
...             print(a + " is a country")
...         else:
...             print("I don't know what", a, "is!")
...     I don't know what foo is!

switch (a) {
    case "blue":
    case "yellow":
    case "red":
        print ...; break;
    case "US":
    case "China":
        print ...; break;
    else:
        print ...;
}
break, continue and else

- **break** and **continue** borrowed from C/Java
- Special **else** in loops
  - When loop terminated *normally* (i.e., not by **break**)
- Very handy in testing a set of properties

```python
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             break
...     else:
...         print(n, end=' ')

prime numbers
```

```c
for (n=2; n<10; n++) {
    good = true;
    for (x=2; x<n; x++)
        if (n % x == 0) {
            good = false;
            break;
        }
    if (good)
        printf("%d ", n);
}
```

#### C/Java

- C/Java if (x==n) || func(n)
Defining a Function  def

- no type declarations needed! **wow**!
- Python will figure it out at run-time
  - you get a run-time error for type violation
    - well, Python does not have a compile-error at all

```python
>>> def fact(n):
...     if n == 0:
...         return 1
...     else:
...         return n * fact(n-1)

>>> fact(4)
24
```
Fibonacci Revisited

>>> a, b = 0, 1
>>> while b <= 5:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5

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

costantly cleaner, but much slower!

>>> fib(5)
5
>>> fib(6)
8
Default Values

```python
>>> def add(a, L=[]):
...     return L + [a]
...
>>> add(1)
[1]

>>> add(1,1)
error!

>>> add(add(1))
[[1]]

>>> add(add(1), add(1))
???
[1, [1]]
```

lists are heterogenous!
Approaches to Typing

- **strongly typed**: types are strictly enforced, no implicit type conversion
- **weakly typed**: not strictly enforced
- **statically typed**: type-checking done at compile-time
- **dynamically typed**: types are inferred at runtime

<table>
<thead>
<tr>
<th></th>
<th>weak</th>
<th>strong</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>static</strong></td>
<td>C, C++</td>
<td>Java, Pascal</td>
</tr>
<tr>
<td><strong>dynamic</strong></td>
<td>Perl, VB</td>
<td>Python, OCaml</td>
</tr>
</tbody>
</table>
Lists

heterogeneous variable-sized array

\[ a = [1, 'python', [2, '4']] \]
Basic List Operations

- length, subscript, and slicing

```python
>>> a = [1, 'python', [2, '4']]
>>> len(a)
3
>>> a[2][1]
'4'
>>> a[3]
IndexError!
>>> a[-2]
[1, 'python']
>>> a[0:3:2]
[1, [2, '4']]
>>> a[1:2]
['python']
>>> a[0::2]
[1, [2, '4']]
+ , extend, +=, append

- extend (+=) and append mutates the list!

```python
>>> a = [1, 'python', [2, '4']]
>>> a + [2]
[1, 'python', [2, '4'], 2]
>>> a.extend([2, 3])
>>> a
[1, 'python', [2, '4'], 2, 3]
same as a += [2, 3]

>>> a.append('5')
>>> a
[1, 'python', [2, '4'], 2, 3, '5']
>>> a[2].append('xtra')
>>> a
[1, 'python', [2, '4', 'xtra'], 2, 3, '5']
```
• as in Java, comparing built-in types is by **value**

• by contrast, comparing objects is by **reference**

```python
>>> [1, '2'] == [1, '2']
True
>>> a = b = [1, '2']
>>> a == b
True
>>> a is b
True
>>> a[1] = 5
>>> a
[1, 5]
>>> a = 4
>>> b
[1, 5]
>>> a is b
True
>>> c = b [:]
>>> c
[1, 5]
>>> c == b
True
>>> c is b
False
>>> b[:0] = [2]
>>> b
[2, 1, 5]
>>> b[1:3]=[]
>>> b
[2]
```

slicing gets a shallow copy

insertion

deletion

a += b means a.extend(b) NOT a = a + b !!
List Comprehension

```python
>>> a = [1, 5, 2, 3, 4, 6]
>>> [x*2 for x in a]
[2, 10, 4, 6, 8, 12]

>>> [x for x in a if len([y for y in a if y < x]) == 3]
[4]

>>> a = range(2,10)
>>> [x*x for x in a if [y for y in a if y < x and (x % y == 0)] == []]
[4, 9, 25, 49]
```

- 4th smallest element
- Square of prime numbers
List Comprehensions

```python
>>> vec = [2, 4, 6]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

>>> [x, x**2 for x in vec]
SyntaxError: invalid syntax

>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]

>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]

>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]

>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

cross product

should use zip instead!

dot product
Strings

sequence of characters
Basic String Operations

- `join`, `split`, `strip`
- `upper()`, `lower()`

```python
>>> s = " this is a python course. 
" >>> words = s.split()
>>> words
['this', 'is', 'a', 'python', 'course. ']
>>> s.strip()
'this is a python course.'
>>> " ".join(words)
'this is a python course.'
>>> " ", "\njoin(words).split("; ")
['this', 'is', 'a', 'python', 'course.']
>>> s.upper()
' THIS IS A PYTHON COURSE. 
'

http://docs.python.org/lib/string-methods.html
Basic Search/Replace in String

```python
>>> "this is a course".find("is")
2
>>> "this is a course".find("is a")
5
>>> "this is a course".find("is at")
-1

>>> "this is a course".replace("is", "was")
'thwas was a course'
>>> "this is a course".replace(" is", " was")
'this was a course'
>>> "this is a course".replace("was", "were")
'this is a course'
```

these operations are much faster than regexps!
>>> print("%.2f%%" % 97.2363)
97.24%

>>> s = '%s has %03d quote types.' % ("Python", 2)
>>> print(s)
Python has 002 quote types.
### Sequence Types

- list, tuple, str; buffer, xrange, unicode

#### Examples

```python
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```
the tricky *

```python
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]

>>> [] * 3  # Empty list
[]

>>> [[]] * 3  # Nested list
[[], [], []]

>>> a = [3]
>>> b = a * 3
>>> b
[3, 3, 3]

>>> a[0] = 4
>>> b
[3, 3, 3]

>>> a = [[3]]
>>> b = a * 3
>>> b
[[3], [3], [3]]

>>> a[0][0] = 4
>>> b
[[3], [3], [3]]

>>> a[0] = 5
>>> b
[[4], [4], [4]]

>>> b[1] = 5
>>> b
[[4], 5, [4]]

>>> b[0] += [2]
>>> b
[[4, 2], 5, [4, 2]]

>>> " " * 3
" " "

>>> "_ " * 3
"_ _ _ "
```
**Pythonic Styles**

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

<table>
<thead>
<tr>
<th>Pythonic Style</th>
<th>Traditional Style</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for key in d.keys():</code></td>
<td><code>for key in d:</code></td>
</tr>
<tr>
<td><code>if d.has_key(key):</code></td>
<td><code>if key in d:</code></td>
</tr>
<tr>
<td><code>i = 0</code> <code>for x in a:</code> <code>...</code> <code>i += 1</code></td>
<td><code>for i, x in enumerate(a):</code></td>
</tr>
<tr>
<td><code>a[0:len(a) - i]</code></td>
<td><code>a[:-i]</code></td>
</tr>
<tr>
<td><code>for line in \[sys.stdin.readlines()\]</code></td>
<td><code>for line in sys.stdin:</code></td>
</tr>
<tr>
<td><code>for x in a:</code> <code>print(x, end=' ')</code> <code>print</code></td>
<td><code>print(&quot; &quot;.join(map(str, a)))</code></td>
</tr>
<tr>
<td><code>s = &quot;&quot;</code> <code>for i in range(lev):</code> <code>s += &quot; &quot;</code> <code>print(s)</code></td>
<td><code>print(&quot; &quot; * lev)</code></td>
</tr>
</tbody>
</table>
Tuples

immutable lists
Tuples and Equality

- caveat: singleton tuple
- `==`, `is`, `is not`

```
>>> (1, 'a')
(1, 'a')
>>> (1)
1
>>> [1]
[1]
>>> (1,)
(1,)
>>> [1,]
[1]
>>> (5) + (6)
11
>>> (5,) + (6,)
(5, 6)
```

```
>>> 1, 2 == 1, 2
(1, False, 2)
>>> (1, 2) == (1, 2)
True
>>> (1, 2) is (1, 2)
False
>>> "ab" is "ab"
True
>>> [1] is [1]
False
>>> 1 is 1
True
>>> True is True
True
```
Comparison

- between the same type: “lexicographical”
- between different types: arbitrary

- `cmp()`: three-way `<`, `>`, `==`

  - C: `strcmp(s, t)`, Java: `a.compareTo(b)`

```python
>>> (1, 'ab') < (1, 'ac')
True
>>> (1, ) < (1, 'ac')
True
>>> [1] < [1, 'ac']
True
>>> [1] == [1,] == [1.0]
True
>>> cmp ((1,), (1, 2))
-1
>>> cmp ((1,), (1, ))
0
>>> cmp ((1, 2), (1, ))
1
```
enumerate

```python
>>> words = ['this', 'is', 'python']
>>> i = 0
>>> for word in words:
...     i += 1
...     print(i, word)
...     print(i+1, word)
1 this
2 is
3 python

>>> for i, word in enumerate(words):
...     print(i+1, word)
...
```

*how to enumerate two lists/tuples simultaneously?*
>>> a = [1, 2]
>>> b = ['a', 'b']

>>> list(zip(a,b))
[(1, 'a'), (2, 'b')]

>>> list(zip(a,b,a))
[(1, 'a', 1), (2, 'b', 2)]

>>> list(zip([1], b))
[(1, 'a')]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [(x, y) for x in vec1 for y in vec2]
[(2, 4), (2, 3), (2, -9), (4, 4), (4, 3), (4, -9), (6, 4),
(6, 3), (6, -9)]

>>> [(vec1[i], vec2[i]) for i in range(len(vec1))]
[(2, 4), (4, 3), (6, -9)]

>>> sum([vec1[i]*vec2[i] for i in range(len(vec1))])
-34

>>> sum(x*y for (x,y) in zip(vec1, vec2))
-34

>>> sum(v[0]*v[1] for v in zip(vec1, vec2))
-34
how to implement zip?

binary zip: easy

```python
>>> def myzip(a,b):
...     if a == [] or b == []:
...         return []
...     return [(a[0], b[0])] + myzip(a[1:], b[1:])
...

>>> myzip([1,2], ['a','b'])
[(1, 'a'), (2, 'b')]
>>> myzip([1,2], ['b'])
[(1, 'b')]
```

how to deal with arbitrarily many arguments?
Dictionaries

(heterogeneous) hash maps
Constructing Dicts

- key : value pairs

```python
>>> d = {'a': 1, 'b': 2, 'c': 1}
>>> d['b']
2
>>> d['b'] = 3
>>> d['b']
3
>>> d['e']
KeyError!
>>> d.has_key('a')
True
>>> 'a' in d
True
>>> d.keys()
['a', 'c', 'b']
>>> d.values()
[1, 1, 3]
```
Other Constructions

• zipping, list comprehension, keyword argument

• dump to a list of tuples

```python
>>> d = {'a': 1, 'b': 2, 'c': 1}
>>> keys = ['b', 'c', 'a']
>>> values = [2, 1, 1]
>>> e = dict(zip(keys, values))
>>> d == e
True
>>> d.items()
[('a', 1), ('c', 1), ('b', 2)]

>>> f = dict([(x, x**2) for x in values])
>>> f
{1: 1, 2: 4}

>>> g = dict(a=1, b=2, c=1)
>>> g == d
True
```
**default values**

- **counting frequencies**

```python
>>> def incr(d, key):
...     if key not in d:
...         d[key] = 1
...     else:
...         d[key] += 1
...

>>> def incr(d, key):
...     d[key] = d.get(key, 0) + 1
...

>>> incr(d, 'z')
>>> d
{'a': 1, 'c': 1, 'b': 2, 'z': 1}
>>> incr(d, 'b')
>>> d
{'a': 1, 'c': 1, 'b': 3, 'z': 1}
```
defaultdict

• best feature introduced in Python 2.5

```python
>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d['a']
0
>>> d['b'] += 1
>>> d
{'a': 0, 'b': 1}

>>> d = defaultdict(list)
>>> d['b'] += [1]
>>> d
{'b': [1]}

>>> d = defaultdict(lambda : <expr>)
```
## Mapping Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>the number of items in <code>a</code></td>
</tr>
<tr>
<td><code>a[k]</code></td>
<td>the item of <code>a</code> with key <code>k</code></td>
</tr>
<tr>
<td><code>a[k] = v</code></td>
<td>set <code>a[k]</code> to <code>v</code></td>
</tr>
<tr>
<td><code>del a[k]</code></td>
<td>remove <code>a[k]</code> from <code>a</code></td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>remove all items from <code>a</code></td>
</tr>
<tr>
<td><code>a.copy()</code></td>
<td>a (shallow) copy of <code>a</code></td>
</tr>
<tr>
<td><code>a.has_key(k)</code></td>
<td>True if <code>a</code> has a key <code>k</code>, else <code>False</code></td>
</tr>
<tr>
<td><code>k in a</code></td>
<td>Equivalent to <code>a.has_key(k)</code></td>
</tr>
<tr>
<td><code>k not in a</code></td>
<td>Equivalent to <code>not a.has_key(k)</code></td>
</tr>
<tr>
<td><code>a.items()</code></td>
<td>a copy of <code>a</code>'s list of (key, value) pairs</td>
</tr>
<tr>
<td><code>a.values()</code></td>
<td>a copy of <code>a</code>'s list of values</td>
</tr>
<tr>
<td><code>a.get(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code></td>
</tr>
<tr>
<td><code>a.setdefault(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code> (also setting it)</td>
</tr>
<tr>
<td><code>a.pop(k[, x])</code></td>
<td><code>a[k]</code> if <code>k</code> in <code>a</code>, else <code>x</code> (and remove <code>k</code>)</td>
</tr>
</tbody>
</table>

*defaultdict behaves like setdefault, not get (following STL)*

[http://docs.python.org/lib/typesmapping.html](http://docs.python.org/lib/typesmapping.html)
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

```python
>>> 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])
>>> type({})
'dict'  # not set!
>>> 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.

```python
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td></td>
<td>cardinality of set s</td>
</tr>
<tr>
<td><code>x in s</code></td>
<td></td>
<td>test x for membership in s</td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td></td>
<td>test x for non-membership in s</td>
</tr>
<tr>
<td><code>s.issubset(t)</code></td>
<td><code>s &lt;= t</code></td>
<td>test whether every element in s is in t</td>
</tr>
<tr>
<td><code>s.issuperset(t)</code></td>
<td><code>s &gt;= t</code></td>
<td>test whether every element in t is in s</td>
</tr>
<tr>
<td><code>s.union(t)</code></td>
<td>`s</td>
<td>t`</td>
</tr>
<tr>
<td><code>s.intersection(t)</code></td>
<td><code>s &amp; t</code></td>
<td>new set with elements common to s and t</td>
</tr>
<tr>
<td><code>s.difference(t)</code></td>
<td><code>s - t</code></td>
<td>new set with elements in s but not in t</td>
</tr>
<tr>
<td><code>s.symmetric_difference(t)</code></td>
<td><code>s ^ t</code></td>
<td>new set with elements in either s or t but not both</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td></td>
<td>new set with a shallow copy of s</td>
</tr>
<tr>
<td><code>s.update(t)</code></td>
<td>`s</td>
<td>= t`</td>
</tr>
<tr>
<td><code>s.intersection_update(t)</code></td>
<td><code>s &amp;=</code> t</td>
<td>return set s keeping only elements also found in t</td>
</tr>
<tr>
<td><code>s.difference_update(t)</code></td>
<td><code>s -= t</code></td>
<td>return set s after removing elements found in t</td>
</tr>
<tr>
<td><code>s.symmetric_difference_update(t)</code></td>
<td><code>s ^= t</code></td>
<td>return set s with elements from s or t but not both</td>
</tr>
<tr>
<td><code>s.add(x)</code></td>
<td></td>
<td>add element x to set s</td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td></td>
<td>remove x from set s; raises KeyError if not present</td>
</tr>
<tr>
<td><code>s.discard(x)</code></td>
<td></td>
<td>removes x from set s if present</td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td></td>
<td>remove and return an arbitrary element from s; raises</td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td></td>
<td>remove all elements from set s</td>
</tr>
</tbody>
</table>
Basic import and I/O
import and I/O

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

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

```java
Java
import System;
for line in stdin:
    print(line.split())
```

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

```python
to read a line:
line = f.readline()
to read all the lines:
lines = f.readlines()
```
import and \_\_main\_\_

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

- handy for debugging

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

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

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

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

```python
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

\[
\begin{align*}
\text{map}(f, a) & \quad [f(x) \ for \ x \ in \ a] \\
\text{filter}(p, a) & \quad [x \ for \ x \ in \ a \ if \ p(x)]
\end{align*}
\]

\[
\begin{align*}
\text{map}(f, \text{filter}(p, a)) & \quad [f(x) \ for \ x \ in \ a \ if \ p(x)]
\end{align*}
\]

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

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

demo
• map/filter in one line for custom functions?
• “anonymous inline function”
• borrowed from LISP, Scheme, ML, OCaml

```python
>>> 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]
```
more on lambda

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

lazy evaluation

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

custom comparison

```python
>>> a = defaultdict(lambda : 5)
>>> a[1]
5
>>> a = defaultdict(lambda : defaultdict(int))
>>> a[1][‘b’]
0
```
demo
Basic Sorting

```python
>>> 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]

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

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

```python
>>> a = [5, 2, 3, 1, 4]
>>> sorted(a)
[1, 2, 3, 4, 5]
>>> a
[5, 2, 3, 1, 4]
```
Built-in and custom cmp

```python
>>> 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 = list(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
```
>>> 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)

sort by two keys
• you can use lambda for both custom cmp and key map

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

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

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

>>> deco.sort()

demo

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

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

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

- 3 ways: key mapping, custom cmp function, decoration
- decoration is most general, key mapping least general
- decoration is faster than key mapping & cmp function
  - decoration only needs $O(n)$ key mappings
  - other two need $O(n \log n)$ key mappings -- or $O(n^2)$ for insertsort
- real difference when key mapping is slow
- decoration is stable
Memoized Recursion v1

- Fibonacci revisited

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

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

```python
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

```python
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 not in fibs:
        fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
    return fibs[n]
```
Memoized Recursion v3

- Fibonacci revisited

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

```python
def fib(n, fibs={0:1, 1:1}):
    if n not in fibs:
        fibs[n] = fib(n-1) + fib(n-2)
    # print(n, fibs)
    return fibs[n]
```

```python
>>> 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!
```

-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

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

```python
def fib(n, fibs=None):
    if fibs is None:
        fibs = {0:1, 1:1}
    if n not in fibs:
        fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
    # print(n, fibs)
    return fibs[n]
```

```bash
>>> 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
```

this is so far the cleanest way to avoid this bug.
Mutable types are not hashable

- mutables: list, dict, set
- immutables: tuple, string, int, float, frozenset, ...
  - only recursively immutable objects are hashable
- your own class objects are hashable (but be careful...)

```python
>>> {{1}: 2}
TypeError: unhashable type: 'set'

>>> {{1:2}: 2}
TypeError: unhashable type: 'dict'

>>> {frozenset([1]): 2}
{frozenset([1]): 2}

>>> {frozenset([1, [2]]): 2}
TypeError: unhashable type: 'list'
```
**Pythonic Styles**

- *do not write ...*  
  - for key in d.keys():
  - if d.has_key(key):
  - i = 0
    - for x in a:
      - ...
      - i += 1
  - a[0:len(a) - i]
  - for line in sys.stdin.readlines():
  - for x in a:
    - print x,
    - print
  - s = ""
    - for i in range(lev):
      - s += " "
  - print s

- *when you can write ...*  
  - for key in d:
  - if key in d:
  - for i, x in enumerate(a):
  - a[:i]
  - for line in sys.stdin:
  - print " ".join(map(str, a))
  - print " " * lev