Why Python?

- Because it’s easy and great fun!
  - only 15 years old, yet very popular now
    - a wide-range of applications, esp. in AI and Web
  - extremely easy to learn
    - many schools haveshifted their intro-courses to Python
  - fast to write
    - much shorter code compared to C, C++, and Java
  - easy to read and maintain
    - more English-like syntax and a smaller semantic-gap
On to Python...
“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]);
    }
}

for element in list:
    print element

only indentations
no { ... } blocks!

or even simpler:

print list

C

Python

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

no C-style for-loops!

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

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;
}
```

Python

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

or even simpler:

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

Java

```java
no need to specify argument and return types!
```

Python

```python
(dynamically typed)
```
Quick-sort

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

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

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)
Basic Python Syntax
Numbers and Strings

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

```python
>>> a = 5
>>> b = 3
>>> type (5)
<type 'int'>
>>> a += 4
>>> a
9
no i++ or ++i

>>> c = 1.5
>>> 5/2
2
>>> 5/2.
2.5
>>> 5 ** 2
25

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

```python
>>> from __future__ import division
>>> 5/2
2.5
```

recommended!
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 = 1, False, 2
```
```
>>> a = b = 0
```
```
>>> a == b
True
```
```
>>> type (3 == 5)
<type 'bool'>
```
```
>>> "my" == 'my'
True
```
```
>>> 1, 2 == 1, 2
???
```
```
>>> (1, False, 2)
```
```
>>> (a, b) = (3, 5)
```
```
>>> a + b
8
```
```
>>> 1, 2 == 1, 2
True
```
```
>>> 1, 2 == 1, 2
???
```
```
>>> (1, False, 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
... 
welcome to python

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

**Java 1.5**

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

- for always iterates through a list or sequence
**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

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

>>> print "foo" if 4 > 5 else "bar"
... conditional expr since Python 2.5
>>> bar

C/Java  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!

C/Java

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,
... 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);
}
```

|| func(n)
Defining a Function  

- 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
```
>>> 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

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[0::2]
[1, [2, '4']]
>>> a[::-1]
[1, 'python', [2, '4']]
>>> a[:]
[1, 'python', [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']
```
Comparison and Reference

- 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
False
>>> c = b [:]
>>> a = b[1:3]=[]
>>> b
[2]
>>> a += b
>>> a
[1, 5]
>>> a is b
True
```

Slicing gets a shallow copy

- Insertion
- Deletion

```python
>>> a = []
>>> b = [1]
>>> id(a)
4299775728
>>> a += b
>>> id(a)
4299775728
>>> a = a + b
>>> id(a)
4299830840
```
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
Strings

sequence of characters
String Literals

- single quotes and double quotes; escape chars
- strings are immutable!

```python
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
SyntaxError!
>>> 'doesn\t'
'doesn\t'
>>> "doesn't"
'doesn't'
>>> "doesn"t"
SyntaxError!
>>> s = "a\nb"
'a\nb'
>>> print s
a
b
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> s = '"Isn\'t," she said.'
>>> s
"Isn't," she said.'
>>> print s
"Isn't," she said.
```

- single quotes and double quotes; escape chars
- strings are immutable!
Basic String Operations

• join, split, strip

• upper(), lower()

```python
>>> s = " this is a python course. \n"
>>> words = s.split()
>>> words
['this', 'is', 'a', 'python', 'course. ']
>>> s.strip()
'this is a python course.'
>>> " ".join(words)
'this is a python course.'
>>> " ; ".join(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.
### Pythonic Styles

- **do not write ...**

  - `for key in d.keys():`
  - `if d.has_key(key):`
  - `i = 0
    for x in a:
      ...
      i += 1`
  - `a[:len(a) - i]`
  - `for line in \n    sys.stdin.readlines():`
  - `for x in a:
    print x,`
  - `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`
Tuples

immutable lists
Tuples and Equality

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

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

```python
>>> a += (1, 2)  # new copy
>>> a += [1, 2]  # in-place

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

```
>>> a = [1, 2]
>>> b = ['a', 'b']

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

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

>>> zip ([1], b)
[(1, 'a')]}

>>> a = ['p', 'q']; b = [[2, 3], [5, 6]]
>>> for i, (x, [_, y]) in enumerate(zip(a, b)):
...     print i, x, y
...
0 p 3
1 q 6
Dictionaries

(heterogeneous) hash maps
Constructing Dicts

- key : value pairs

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

or

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

```java
import System;
```

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

File copy

Java

```java
import System.*;
```

```python
to read a line:
    line = f.readline()
to read all the lines:
    lines = f.readlines()
```

file copy

note this comma!
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())

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

foo.py
demo

interactive
Functional Programming
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]
Object-Oriented Programming
Classes

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return self.x ** 2 + self.y ** 2
```

- “self” is like “this” pointer in C++/Java/C#/PHP
- constructor `__init__(self, ...)`
- every (new-style) class is a subclass of Object like Java
  - we will only use new-style classes in this course

```python
>>> p = Point(3, 4)
>>> p.x
3
>>> p.norm()
25
```
Member variables

- each instance has its own hashmap for variables!
- you can add new fields on the fly (weird... but handy...)

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

>>> p = Point(5, 6)
>>> p.z = 7
>>> print p
(5, 6)
>>> p.z
7
>>> print p.w
AttributeError - no attribute 'w'
>>> p["x"] = 1
AttributeError - no attribute 'setitem'
```
More efficient: \_\_slots\_

- like C++/Java: fixed list of member variables
- class-global hash: all instances of this class share this hash
- can’t add new variables to an instance on the fly

```python
class Point(object):
    __slots__ = "x", "y"

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        "like toString() in Java"
        return "(%s, %s)" % (self.x, self.y)

>>> p = Point(5, 6)
>>> p.z = 7
AttributeError!
```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return self.x ** 2 + self.y ** 2

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

>>> P = Point(3, 4)
>>> p.__str__()
'(3, 4)'
>>> Point.__str__(p)
'(3, 4)'
>>> str(p)
'(3, 4)'
>>> print p
(3, 4)
```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

>>> p = Point(3,4)
>>> print p
(3, 4)
>>> p
<__main__.Point instance at 0x38be18>
```
```python
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    def __repr__(self):
        return self.__str__()

>>> p = Point(3,4)
>>> print p
(3, 4)
>>> p
(3, 4)
>>> repr(p)
(3, 4)

>>> p
<__main__.Point instance at 0x38be18>
```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

when __str__ is not defined, __repr__ is used
if __repr__ is not defined, Object.__repr__ is used
Special functions: cmp

```python
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

>>> p = Point(3,4)
>>> Point (3,4) == Point (3,4)
False
```

by default,
Python class object comparison is by pointer! define __cmp__!
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    def __cmp__(self, other):
        if self.x == other.x:
            return self.y - other.y
        return self.x - other.x

>>> cmp(Point(3,4), Point(4,3))
-1

if __eq__ is not defined, __cmp__ is used; if __cmp__ is not defined, Object.__cmp__ is used (by reference)

>>> p = Point(3,4)

>>> p
<Point instance at 0x38be18>

>>> Point (3,4) == Point (3,4)
False

>>> Point (3,4) == Point (3,4)
True
unique signature for each method

```python
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __init__(self, (x, y)):
        self.x, self.y = x, y
```

- no polymorphic functions (earlier defs will be shadowed)
  - ==> only one constructor (and no destructor)
- each function can have only one signature
  - because Python is dynamically typed
Inheritance

```python
class Point (object):
    ...
    def __str__(self):
        return str(self.x) + ', ' + str(self.y)
    ...

class Point3D (Point):
    "A 3D point"
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def __str__(self):
        return Point.__str__(self) + ', ' + str(self.z)

    def __cmp__(self, other):
        tmp = Point.__cmp__(self, other)
        return tmp if tmp != 0 else self.z - other.z
```

super-class, like C++
(multiple inheritance allowed)
__slots__ in inheritance

- like C++/Java: fixed list of member variables
- class-global hash: can’t add new field on the fly

```python
class Point(object):
    __slots__ = "x", "y"

    def __init__(self, x, y):
        self.x, self.y = x, y

class Point3D(Point):
    __slots__ = "z"

    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

>>> p = Point3D(5, 6, 7)
>>> p.z = 7
```
class Tree (object):
    __slots__ = "node", "children"
def __init__(self, node, children=[]):
    self.node = node
    self.children = children
def total(self):
    return self.node + sum([x.total() for x in self.children])
def pp(self, dep=0):
    print " |" * dep, self.node
    for child in self.children:
        child.pp(dep+1)
def __str__(self):
    return "(%s)" % " ".join(map(str, 
        [self.node] + self.children))

left = Tree(2)
right = Tree(3)

>>> t = Tree(1, [Tree(2), Tree(3)])
>>> t.total()
6

>>> t.pp()
1
   | 2
   | 3

>>> print t
(1 (2) (3))
Using map and unbound method

```python
class Tree (object):
    __slots__ = “node”, “children”
def __init__(self, node, children=[]):
    self.node = node
    self.children = children

def total(self):
    return self.node + sum(map(Tree.total, self.children))

def pp(self, dep=0):
    print “ |” * dep, self.node
    for child in self.children:
        child.pp(dep+1)

def __str__(self):
    return “(%s)” % “ “.join(map(str, 
    [self.node] + self.children))

left = Tree(2)
right = Tree(3)

>>> t = Tree(1, [Tree(2), Tree(3)])

>>> t.total()
6

>>> t.pp()
1
| 2
| 3

>>> print t
(1 (2) (3))
```

class.method(x, ...) is equivalent to:
x.method(...) if x is instance of class
postorder traversal

class Tree (object):
    __slots__ = “node”, “children”
def __init__(self, node, children=[]):
    self.node = node
    self.children = children
def total(self):
    return self.node + sum(map(Tree.total, self.children))
def pp(self, dep=0):  # preorder
    print “ ” * dep, self.node
    for child in self.children:
        child.pp(dep+1)
def __str__(self):  # preorder
    return “(%s)” % “ “.join(map(str, [self.node] + self.children))
def postorder(self):
    return “ “.join(map(Tree.postorder, self.children) + [str(self.node)])

left = Tree(2)
right = Tree(3)

>>> t = Tree(1, [Tree(2), Tree(3)])
>>> print t.postorder()
2 3 1

class.method(x, ...) is equivalent to:
x.method(...) if x is instance of class