An Introduction to python

Vincent Voelz

Temple University

September 28, 2015
Preliminaries
  General Information
  Installation

Digging into python
  The interpreter
  Strings
  Lists, tuples, and dictionaries
  Looping and Control Flow

Writing python scripts
  Modules
  Functions
  Classes
  Scripting example: mc.py
What is python?

- Python is an interpreted language (like MATLAB)
- Often thought of as a *scripting* language (like perl)
- It’s written in C
- It’s object oriented
Why python?

- quickly becoming a "standard" in scientific computing
- shallow learning curve
  - can use like a calculator
  - simply and easily get ideas implemented
- yet very powerful
  - highly extensible and object-oriented
  - Interface to numerical and scientific computing libraries
  - many applications have python 'handles'
Resources for Learning Python

- http://docs.python.org/tutorial/
- http://www.learnpython.org/
- O’Reilly Books
Installing python

- Most current and stable version: 2.7
- Install a binary
  - Recommended: Anaconda Python Distribution
    https://www.continuum.io/downloads
  - Macpython (pythonmac.org)
- Install from source
  - Use a package installer
  - Mac: MacPorts (port), fink; Linux: rpm, yum
  - Last resort: http://www.python.org/download/releases/2.7.2/
We will use the (UNIX) command line

- Mac: open the Terminal app
- PC: Cygwin (http://www.cygwin.com/)
- Linux: you're already there
Options for editing source code

- Command-line editors
  - vi
  - emacs
  - pico

- Integrated Developer Environment (IDE)
  - IDLE on Mac: >>>> import idlelib.idle
  - IDLE on PC: comes with python installation at python.org
  - WingIDE free trial (http://wingware.com/)
  - Xcode on Mac: open myscript.py

- Text editor apps
  - Mac: Textedit, BBEdit, TextMate, etc.
  - PC: ?
Setting environment variables (UNIX)

- The python executable needs to be in your PATH. In .bash_profile:

  ```
  # anaconda python
  export PATH="/Users/vv/anaconda/bin:$PATH"
  ```
invoke the python interpreter

vvs-MacBook-Pro:~ vv$ python
Python 2.7.10 |Anaconda 2.1.0 (x86_64)| (default, May 28 2015, 17:04:42)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org

CTRL-D to exit
The interpreter as a calculator

- The interpreter is a lot like MATLAB or Mathematica
  >>> 2+3
  5
  >>> 2/3
  0
  >>> 2./3.
  0.6666666666666666
  >>> 4.3**2
  18.49
  >>> 5 % 3  # 5 modulo 3
  2
  >>> _ + 5  # _ is the last printed expression
  7

- Note that comments are anything after a #
The interpreter as a calculator

- Line continuations with backslash
  >>> 1 + 2 + 3 \
  ... + 4 + 5
  15

- Line concatenation (rarely used – bad form!) with ;
  >>> 1 + 2 + 3 ; 4 +5
  6
  9
The interpreter as a calculator

- **Data types**
  ```python
  >>> type(3)
  <type 'int'>
  >>> type(4.5)
  <type 'float'>
  >>> type("hello")
  <type 'str'>
  >>> type(True)
  <type 'bool'>
  >>> type(4.+3.j)
  <type 'complex'>
  >>> type(999999999)
  <type 'int'>
  >>> type(999999999999)
  <type 'long'>
  ```
The interpreter as a calculator

- Converting different types

  ```
  >>> bool(1)
  True
  >>> bool(0)
  False
  >>> float("66.5")
  66.5
  >>> int(4.4)
  4
  >>> str(5)
  '5'
  >>> "5"
  '5'
  ```

- Note that " and ' are interchangeable, as long as they are paired correctly. This is useful for nested quotes in strings, for example.
The interpreter as a calculator

Logical operations

```python
g>>> 2 > 3
False
g>>> 4 <= 6
True
g>>> "r" < "s"
True
g>>> 3 != 3
False
g>>> not 3 == 3
False
g>>> (2<3) and (4<5)
True
g>>> True or False
True
```
Assigning variables

- Variables are case-sensitive
  ```python
  >>> Ted = 2; ted = 5
  >>> Ted, ted
  (2, 5)
  ```

- Dynamic typing
  ```python
  >>> a = 3
  >>> a = "three"  # was an int, now a string
  ```

- All assignments are pointers to objects
  ```python
  >>> b = "four"
  >>> a = b  # a now points to "four"
  >>> b = "six"  # a new object "six" is created, which now b points to
  >>> a  # a still points to "four"
  'four'
  ```
Strings

- String operations
  ```python
  >>> name = "Larry" + 'Moe'  # concatenate two strings
  >>> name
  'LarryMoe'
  >>> name*5  # repeat a string 5 times
  'LarryMoeLarryMoeLarryMoeLarryMoeLarryMoe'
  ```

- String formatting
  ```python
  >>> f = "Hello\nwor\tld."  # \n is a newline, \t is a tab
  >>> print f
  Hello
  wor ld.
  >>> """Triple-quotes denote a verbatim string
  ... ... with newlines
  ... ... included.""
  'Triple-quotes denote a verbatim string\n... with newlines\n... included.'
  ```
Indexing strings

- Strings can be indexed and sliced...

  ```python
  >>> w = "word"
  >>> w[0]
  'w'
  >>> w[3]
  'd'
  >>> w[0:3]
  'wor'
  ```

- ... but cannot be changed like C character arrays

  ```python
  >>> w[1] = '4'
  Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: 'str' object does not support item assignment
  >>> w[0] + '4' + w[2:]
  'w4rd'
  ```
Built-in string functions

- **Strings are objects that have their own methods**
  
  ```python
  >>> "apple banana".split()  # returns a list of substrings
  ["apple", "banana"]
  >>> "apple banana".capitalize()
  'Apple banana'
  >>> "KACHOW!".lower()
  'kachow!'
  >>> "    string    ".strip()
  'string'
  >>> "hello".replace("ll","ckuvabanj")
  'heckuvabanjo'
  ```

- **The length of a string**
  
  ```python
  >>> len("abcedfg")
  7
  ```

- **Testing for substrings**
  
  ```python
  >>> "cde" in "abcedfg"
  True
  ```
Special formatting characters are used, like C’s printf():

```python
>>> print "%s is the first letter in %s"%(‘V’,’Volvo’)
V is the first letter in Volvo
>>> print "$%3.2f for %d eggs?"%(4.99,6.0)
$4.99 for 6 eggs?
>>> print "%e"%4.99  # exponential notation
4.990000e+00
>>> print "Eight spaces for this number: %8d."%2
Eight spaces for this number: 2.
>>> print "Eight spaces, but left-justified: %-8d."%2
Eight spaces, but left-justified: 2 .
```
Lists

- Lists are iterable container objects
- They are very flexible and powerful

```python
>>> f = [1, 2, 3]
>>> f[2]
3
>>> g = [True, "dog", 47.0]
>>> h = [1, [3, 4], 3]
>>> h[-1]
3
>>> h[0::2]
[1, 3]
```

# indexing starts at zero
# lists can be a different types
# lists can be nested
# negative indices start from the last entry
# skip every two entries
Generating Lists

- **the built-in function `range()`**
  ```python
  >>> range(4)  # generate a 4-entry list, starting with 0
  [0, 1, 2, 3]
  >>> range(1,5,2)  # returns a list from 1 to 4, skipping every two
  [1, 3]
  ```

- **list comprehensions**
  ```python
  >>> [2*i for i in range(5)]
  [0, 2, 4, 6, 8]
  >>> [[2*i for i in range(k)] for k in range(5)]
  [[], [0], [0, 2], [0, 2, 4], [0, 2, 4, 6]]
  ```

- **filtering by conditions**
  ```python
  >>> [s for s in ['blue', 'red', 'green', 'yellow'] if 'l' in s]
  ['blue', 'yellow']
  ```
List operations and functions

- List manipulations

```python
>>> f = []
>>> f.append(3)
>>> f
[3]
>>> f.remove(3)
>>> f
[]
>>> g = [0, 1, 2, 3, 4]
>>> g.insert(2, "hi!")  # insert at position 2
>>> g
[0, 1, 'hi!', 2, 3, 4]
>>> del g[3:5]  # delete positions 3 and 4
>>> g
[0, 1, 'hi!', 4]
```
More list manipulations

```python
>>> f = ['a', 'b', 'd'] + [3,4]  # joining two lists
>>> f
['a', 'b', 'd', 3, 4]
>>> f.pop()  # pops off the last element and returns its value
4
>>> f.pop(0)  # pops off the 0th element and returns its value
'a'
>>> f
['b', 'd', 3]
>>> f.extend([5,6,7])
>>> f
['b', 'd', 3, 5, 6, 7]
>>> f.sort()
>>> f
[3, 5, 6, 7, 'b', 'd']
>>> f.reverse()
>>> f
['d', 'b', 7, 6, 5, 3]
```
Tuples

- Tuples are like lists, but are immutable (as static objects they are faster to load and operate on).

```python
>>> () # an empty tuple
()
>>> f = (1, 2, 3)
>>> f[2]
3
>>> f[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- Converting between lists and tuples:

```python
>>> list(f)
[1, 2, 3]
>>> tuple(f)
(1, 2, 3)
```
List assignments and name binding

- Assigning variables to lists
  ```python
  >>> a = [1,2,3]  \# a new list is created in memory; a points to it
  >>> b = a  \# now b points to it too
  >>> a[1] = 99
  >>> b  \# thus, b shows changes to this list too
  [1, 99, 3]
  ```

- If we want to make a copy of a list that won’t change, we need to use the copy module. (More on modules later).
  ```python
  >>> import copy
  >>> b = copy.copy(a)
  >>> a[1] = 44
  >>> b
  [1, 99, 3]
  ```
List assignments and name binding

- `copy.copy()` only copies one-level deep. If we want to copy nested lists, for example, we need `copy.deepcopy()`

```python
>>> import copy
>>> a = [1,2,[3,4]]
>>> b = copy.copy(a)
>>> c = copy.deepcopy(a)
>>> a[2][1] = 5
>>> a,b,c
([1, 2, [3, 5]], [1, 2, [3, 5]], [1, 2, [3, 4]])
```

- Aside from issues like this that you need to be away of, python generally handles memory issues well, with automatic garbage collection. If needed, though, you can explicitly deallocate objects using `del()`:

```python
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```
Dictionaries

- Dictionaries are like "hash tables"; they specify a mapping from keys to values:

  ```python
  >>> d = {}  # empty dictionary object
  >>> d[’Frank’] = 78
  >>> d[’Joe’] = 42
  >>> d
  {’Frank’: 78, ’Joe’: 42}
  ```

- you can’t add dictionaries, or use lists as keys

  ```python
  >>> f = {’Julie’: 41}
  >>> d + f
  Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: unsupported operand type(s) for +: ’dict’ and ’dict’
  >>> f[[3,4]] = 6
  Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: unhashable type: ’list’
  ```
Dictionaries as a lookup table

```python
>>> d = {'Frank': 78, 'Joe': 42}
>>> d.keys()
['Frank', 'Joe']
>>> d.values()
[78, 42]
>>> d.has_key('Sue')
False
>>> for k, v in d.items():
...     print k, v
...
Frank 78
Joe 42
```
In python, the indentation level specifies the nest loop structure

```python
>>> for i in range(3):
...     print i
...     print i

File "<stdin>", line 3
    print i
    ^
IndentationError: unexpected indent
```

spaces are very important!

Python ”style guide” recommends 4 spaces per indent.
Conditions and Looping

- **for loops**
  ```python
  >>> l = [(1, 2), (3, 4), (5, 6)]
  >>> for (a, b) in l:
  ...   print a + b
  ...   <hit return>
  3
  7
  11
  ```

- **while loops**
  ```python
  >>> i = 0
  >>> while i < 3:
  ...   print i
  ...   i += 1
  ...
  0
  1
  2
  ```
Conditions and Looping

- **if, elif and else**

  ```python
  >>> for i in range(5):
  ...     print i
  ...     if i > 2:
  ...         break
  ...     elif i == 0:
  ...         print "yahoo!", # trailing comma prevents newline
  ...     else:
  ...         continue
  ... 0
  yahoo! 1
  2
  3
  ```
try and except

- Errors can be tested for and exceptions thrown (instead of crashing your program)

```python
>>> try:
...     import matplotlib
... except:
...     pass
```

- Forcing particular exceptions to be raised

```python
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```
A simple script

For the most part, a python script is just a list of commands, stored in a separate file, to be read and executed by the interpreter.

Consider the contents of a file `hello.py`:

```python
#!/usr/bin/env python
print "Hello world."
```

We could run the script using

```
$ python hello.py
```

Or, we can make the file executable using

```
$ chmod +x hello.py
$ ./hello.py
```
Like any programming language, we want the ability to modularize our code into separate objects and subroutines.

Python already provides much of this in the form of standard modules, much like the Standard Template Library in C++.

There is a namespace associated with modules that mirrors the hierarchy of script files constituting the module.

In order to find non-standard modules, the path needs to be specified in your $PYTHONPATH environment variable:

```bash
$ export PYTHONPATH=$PYTHONPATH:/Users/vince/scripts
```
Modules

- Importing a module
  ```python
  >>> import os
  ```
- Submodules specified by dot notation
  ```python
  >>> os.curdir
  '.'
  >>> os.path.abspath('./')
  '/Users/vince/projects'
  ```
- Python provides documentation of objects via `help()`
  ```python
  >>> help(os)
  ```
Tweaking the module namespace

▶ Importing a submodule into the topmost namespace

```python
>>> import os.path as path
>>> path.exists('./')
True
```

▶ Importing all submodules

```python
>>> from os import *
>>> path.abspath('./')
'/Users/vince/projects'
```

▶ You need to be careful using import * to avoid namespace confusion
The \texttt{dir()} function

\begin{itemize}
  \item To see the names defined by a given module:
    
    \begin{verbatim}
    >>> import os
    >>> dir(os)
    ['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR' ... \\
    \end{verbatim}
  \item Without arguments, \texttt{dir()} lists the names you have defined currently:
    
    \begin{verbatim}
    >>> dir()
    ['__builtins__', '__doc__', '__name__', '__package__', 'os']
    \end{verbatim}
  \item Python provides ample documentation of objects via \texttt{help()}:
    
    \begin{verbatim}
    >>> help(os)
    \end{verbatim}
\end{itemize}
The sys module

- Among other things, sys keeps track of the paths in your $PYTHONPATH
  >>> import sys
  >>> print sys.path
  ['', '/Users/vince/projects', '/Users/vince/scripts']
  >>> sys.path.append('/my-new-project/scripts')

- sys.argv stores command line arguments
  >>> sys.argv
  ['']

- Exiting a python script
  >>> sys.exit()
$
Packages

Packages are collections of modules with a hierarchical file structures. Example:

```
sound/
  __init__.py
  formats/
      __init__.py
      wavread.py
      wavwrite.py
      aifread.py
      aiffwrite.py
      auread.py
      auwrite.py
      ...
  effects/
      __init__.py
      echo.py
      surround.py
      reverse.py
      ...
```

Top-level package
Initialize the sound package
Subpackage for file format conversions
Subpackage for sound effects

You can import individual modules like so:

```python
>>> import sound.effects.echo
```
You can write your own packages, provided that there is a file __init__.py in each subdirectory that contains a line explicitly specifying the names of all of submodules.

For example, sounds/effects/__init__.py must contain the line:
__all__ = ["echo", "surround", "reverse"]
The file potentials.py defines some functions

```python
#!/usr/bin/env python

def LJ(r, sigma=0.2, epsilon=0.20):
    """Returns the energy of a Lennard-Jones potential.
    UNITS
    r Angstrom
    sigma Angstrom
    epsilon kcal/mol.""

    result = 4.*epsilon*( (sigma/r)**12 - (sigma/r)**6 )
    return result

def harmonic(r, kspring=1.0):
    """Returns the energy of a harmonic oscillator with spring constant kspring.
    UNITS
    r Angstrom
    kspring kcal/mol/Angstrom**2""

    result = 0.5 * kspring * (r**2)
    return result
```
Functions

- running this script simply loads them into memory and exits
  $ python potentials.py
  $
- but can use the functions interactively
  >>> import potentials
  >>> potentials.LJ
  <function LJ at 0x2c69f0>
  >>> potentials.LJ(3)
  -0.0054794417442387755
  >>> potentials.LJ(3, sigma=5)  # careful!
  0.0
  >>> potentials.LJ(3, sigma=5.)
  1751.8407499609561
- or from other scripts that import the functions
Classes

- Even though Python is a "scripting" language, it is fully object-oriented.
- **Classes** are the main objects in Python.
  - For those unacquainted with object-oriented programming (OOP): a *class* is compartmentalized data structure that bundles together stored data (i.e. *attributes*) with functions (i.e. *methods*) that operate on this data.
  - The more programming you do, the more you realize that OOP is the most natural way to write modular code.
Even though python is a "scripting" language, it is fully object-oriented.

**Classes** are the main objects in python.

- For those unacquainted with object-oriented programming (OOP): a *class* is compartmentalized data structure that bundles together stored data (i.e. *attributes*) with functions (i.e. *methods*) that operate on this data.
- The more programming you do, the more you realize that OOP is the most natural way to write modular code.
Example: mc.py

- mc.py contains a definition of a `System` class that can perform Monte Carlo simulation on a set of 1-dimensional particles.
Let’s look at the code

```python
#!/usr/bin/env python

import potentials
import random, math
import copy

class System(object):
    """An object for calculating 1D Monte Carlo trajectories."""

    def __init__(self, nparticles=2, spread=1.0, kT=0.5959):
        """Initialize the class.
PARAMETERS
nparticles  - the number of particles
spread      - the width of Gaussian spread for initial particle positions (Angstroms).
kT          - thermal energy (k_B times the temperature of the system), in kcal/mol."

        self.nparticles = nparticles
        self.spread    = spread
        self.kT        = kT # in kcal/mol

        self.positions = [] # a list of 1D particle positions

...
Example: mc.py - page 2

```python
# set initial positions randomly
for i in range(self.nparticles):
    self.positions.append(random.normalvariate(0.0, spread))

# make a list of (pointers to) energy functions that must be calculated
self.energyFunctions = []
self.energyFunctions.append(potentials.LJ)
# self.energyFunctions.append(potentials.harmonic)

self.energy = self.computeEnergy(self.positions)

# MC statistics
self.attempted = 0.  # keep these floats, so we can divide to get acc. ratio
self.accepted = 0.

def computeEnergy(self, positions):
    """Returns the energy of the system.""

    energy = 0.0
    for fn in self.energyFunctions:
        for i in range(self.nparticles):
            for j in range(i+1, self.nparticles):  # this avoids double-counting
                r = abs(positions[i] - positions[j])
                energy += fn(r)

    return energy
```

Vincent Voelz

An Introduction to python
Example: mc.py - page 3

def simulate(self, nsteps, dsigma=0.1, verbose=True):
    """Simulate nsteps of Monte Carlo dynamics, with a move set
    of random Gaussian perturbations of width dsigma. Returns (energy, positions)."""

    for step in range(nsteps):
        accept = False
        new_positions = copy.copy(self.positions)

        # perturb all particles' positions
        for i in range(self.nparticles):
            new_positions[i] = self.positions[i] + dsigma*random.normalvariate(0.0, dsigma)
        new_energy = self.computeEnergy(new_positions)

        # accept according to the Metropolis Criterion
        if new_energy < self.energy:
            accept = True
        elif random.random() < math.exp(-(new_energy - self.energy)/self.kT):
            accept = True

        if accept:
            self.positions = new_positions
### Main ###

```python
if __name__ == '__main__':
    import sys

    usage = '''Usage: python run_mc.py NPARTICLES NITERATIONS
    Performs NITERATIONS of Monte Carlo for a 1D system with NPARTICLES.'''

    if len(sys.argv) < 3:
        print usage
        sys.exit(1)

    N = int(sys.argv[1])  # the number of particles
    niters = int(sys.argv[2])  # the number of iterations

    UseMatplotlib = True
    try:
        import matplotlib
        import pylab
    except:
        UseMatplotlib = False
```
s = System(nparticles=N)

steps_per_iter = 1
energies = []
trajectories = [ [] for p in range(N)]
for t in range(niters):
    e, c = s.simulate(steps_per_iter, verbose=False)
    energies.append(e)
    for i in range(N):
        trajectories[i].append(c[i])

# print the trajectories
print '#iteration\tE\tpositions'
for t in range(niters):
    print t, energies[t],
    for i in range(N):
        print trajectories[i][t],
    print

if UseMatplotlib:
    # plot the trajectories
    pylab.figure()
    for i in range(N):
        pylab.plot(range(niters), trajectories[i])
    pylab.hold(True)
    pylab.show()
Example: mc.py

- Let's create an *instance* of the System object:

```python
>>> from mc import *
>>> s = System()
>>> s
Energy: -0.000007 Positions: 1.685422 0.300590
```

- The `__init__()` method is automatically called when a class object is created to initializes all the data.

- Additionally, the `__repr__()` method provides a string representation of the object.

- Without this, printing the object would produce something like this:

```python
>>> s
<mc.System object at 0x42eed0>
```
Example: mc.py

- All of the text strings following class/method/function declarations are called *documentation strings*
- They make your code readable, but they also provide documentation. Example: `>>> help(s)`

Help on System in module mc object:

class System(_builtin_.object):
    | An object for calculating 1D Monte Carlo trajectories.
    |
    | Methods defined here:
    |
    | __init__(self, nparticles=2, spread=1.0, kT=0.5959)
    | Initialize the class.
    | PARAMETERS
    | nparticles   - the number of particles
    | spread       - the width of Gaussian spread for initial particle positions (Angstroms).
    | kT           - thermal energy (k_B times the temperature of the system), in kcal/mol.
    |
    ...

Vincent Voelz
An Introduction to python
Class methods are different than functions!
Note that the self argument (a pointer to the class object) is needed for class methods, e.g.

```python
def simulate(self, nsteps, dsigma=0.1, verbose=True):
```

When calling the method function, however, the argument is only implied

```python
>>> s = System()
>>> s.simulate(10)
```
Derived Classes

- Specialized classes can be derived from more generic classes
- For example, `System` is a derivation of the generic ”object” class

```python
class System(object):
```
- In general,

```python
class DerivedClass(BaseClass):
```
- The derived class inherits the methods of the base class, unless it is overridden.
Private attributes and methods in classes

- Python doesn’t encourage "private" class variables (like C++), but there is a convention for treating attributes and methods as non-public parts of an API (application programming interface).

- Any name prefixed by an underscore is considered "private":

```python
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

__update = update  # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Vincent Voelz  An Introduction to python
mc.py is a also a script that can take arguments

```python
if __name__ == "__main__":
    import sys

    usage = """Usage: python run_mc.py NPARTICLES NITERATIONS
    Performs NITERATIONS of Monte Carlo for a 1D system with NPARTICLE"""

    if len(sys.argv) < 3:
        print usage
        sys.exit(1)

    N = int(sys.argv[1])  # the number of particles
    niters = int(sys.argv[2])  # the number of iterations
...
```
Give it too few arguments and it now barks at you

$ python mc.py
Usage: python mc.py NPARTICLES NITERATIONS
    Performs NITERATIONS of Monte Carlo for a 1D system with NPARTICLES.

$ python run_mc.py 3 10
#iteration positions
0 -0.0901481322992 0.584149150592 0.345001543989
1 -0.0901481322992 0.584149150592 0.345001543989
2 -0.100521272738 0.584149150592 0.345001543989
3 -0.100521272738 0.584149150592 0.345001543989
4 -0.100521272738 0.584149150592 0.345001543989
5 -0.115506087953 0.584149150592 0.345001543989
6 -0.115506087953 0.584149150592 0.345001543989
7 -0.115506087953 0.584149150592 0.345001543989
8 -0.115506087953 0.584149150592 0.345001543989
9 -0.12884262928 0.584149150592 0.345001543989
To combine an executable script and modular code in one file, `mc.py` uses this plan:

```python
#!/usr/bin/env python

# Functions
def myfunc(blah):
    ...

# Classes
class my class(object):
    ...

### Main ###
if __name__ == "__main__":
    # execute script commands
```
How can we improve the System class?

- Could make separate MCMoveSet classes so we can modularize different move sets
- How about file writer classes for reading/writing trajectories?
- Other ideas?
Reading from and writing to files

- To open a file handle for writing
  ```
  >>> fout = open("junk.txt",'w')
  >>> fout.write("What\na bunch of\njunk!\n")
  >>> fout.close()
  ```

- Reading by line
  ```
  >>> fin = open("mc.py","r")
  >>> fin.readline() # returns the first line
  
  
  
  #!/usr/bin/env python
  >>> lines = fin.readlines() # returns list of remaining lines
  >>> fin.close()
  ```
The pickle module

- Some data objects are cumbersome to write to file
- The pickle module saves text-encoded instructions for re-instantiating any object
  ```python
  >>> f = {3:"basson", "sally":True}
  >>> fout = open('f.pkl', 'w')
  >>> import pickle
  >>> pickle.dump(f, fout)
  >>> fout.close()
  ```
- The output file looks strange, but it has all the info python needs to rebuild our dictionary
  ```bash
  $ cat f.pkl
  (dp0
  I3
  S’basson’
  p1
  sS’sally’
  p2
  I01
  s.
  ```

Vincent Voelz
An Introduction to python
Scripting UNIX commands

- `os.system` is really useful for scripting commands in UNIX shells

  ```python
  >>> import os
  >>> os.system('ls -lasrt | tail -1')
  8 -rw-r--r-- 1 vincentvoelz staff 40 Jan 19 02:37 f.pkl
  >>> os.mkdir('newdir')
  >>> os.system('ls -lasrt | tail -2')
  0 drwxr-xr-x 2 vincentvoelz staff 68 Jan 19 02:53 newdir
  0 drwxr-xr-x 17 vincentvoelz staff 578 Jan 19 02:53 .
  0
  ```

- **Related**: the commands module gives you direct control over standard input and output
Many other useful standard modules

- Standard module index: http://docs.python.org/modindex.html
- `time` functions for program timing and returning the current time/date in various formats
- `tempfile` automatic creation and deletion of temporary files
- `glob` wildcard-type file expressions (e.g., "*.txt")
- `shutil` high-level file operations (e.g., copying, moving)
- `gzip`, `bz2`, `zipfile`, `tarfile` reading/writing compressed file formats
- `socket` low-level networking
- `popen2` executing other processes, capturing output
- `urllib` downloading data from the internet and telnet protocols
Next time: Scientific Computing in Python

- numpy and scipy
- matplotlib
- pytables
- scikits
- ... all available through the Enthought Python Distribution