ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Python for Finance

Enthought, Inc.
www.enthought.com

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin, TX 78701

www.enthought.com

Q3-2013

Python for Finance

Enthought, Inc.
www.enthought.com

ENTHOUGHT

Python for Finance

Enthought, Inc.
www.enthought.com

ENTHOUGHT

What Is Python?

ONE LINER

Python is an interpreted programming language that allows you to do almost anything possible with a compiled language (C/C++/Fortran) without requiring all the complexity.

PYTHON HIGHLIGHTS

- **Automatic garbage collection**
- **Dynamic typing**
- **Interpreted and interactive**
- **Object-oriented**
- **"Batteries Included"**
- **Free**
- **Portable**
- **Easy to Learn and Use**
- **Truly Modular**

ENTHOUGHT

Who is using Python?

WALL STREET

Banks and hedge funds rely on Python for their high speed trading systems, data analysis, and visualization.

HOLLYWOOD

Digital animation and special effects:

- Industrial Light and Magic
- Imageworks
- Tippett Studios
- Disney
- Dreamworks

PETROLEUM INDUSTRY

Geophysics and exploration tools:

- ConocoPhillips
- Shell

GOOGLE

One of top three languages used at Google along with C++ and Java. Guido van Rossum worked there.

YOUTUBE

Highly scalable web application for delivering video content.

REDHAT

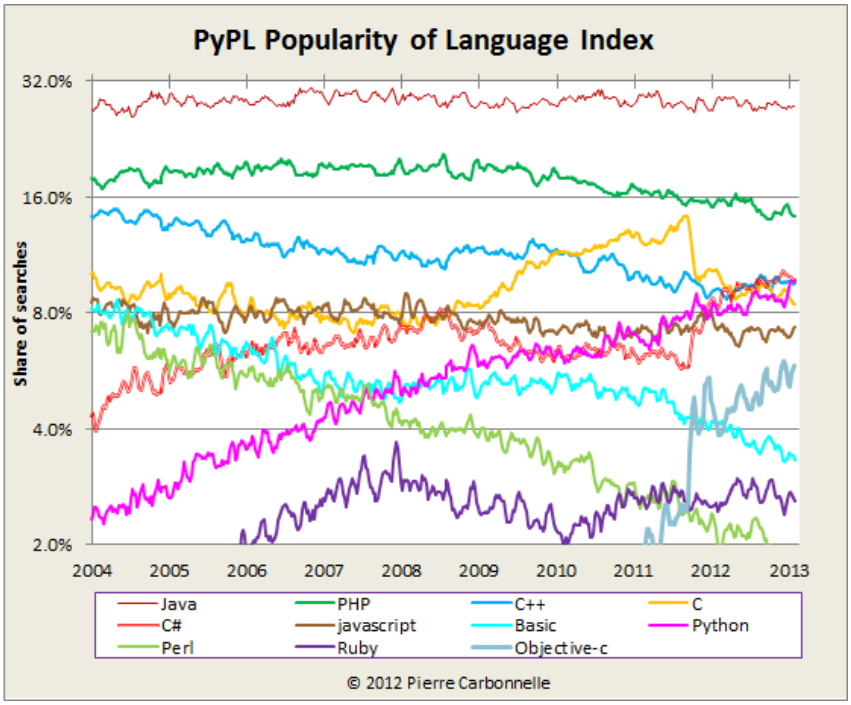
Anaconda, the Redhat Linux installer program, is written in Python.

PROCTER & GAMBLE

Fluid dynamics simulation tools.

ENTHOUGHT

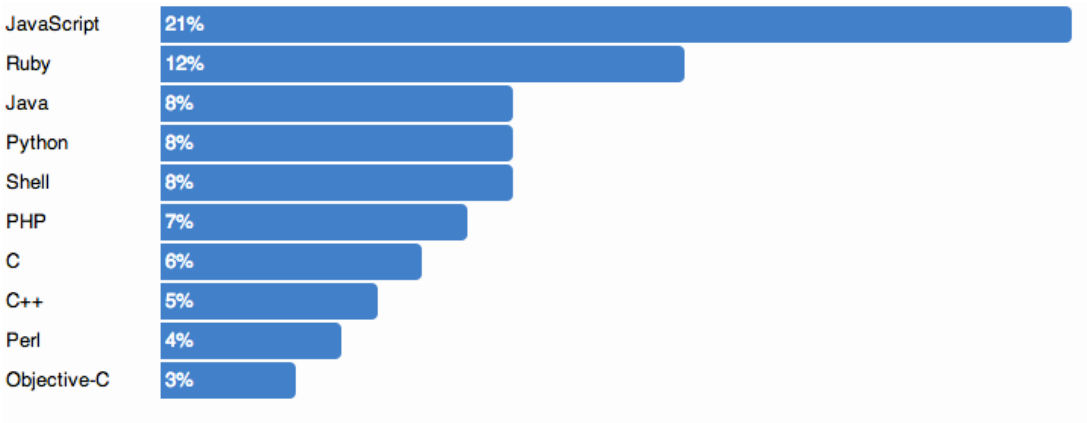
PYPL PopularitY of Programming Language index



* https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language (CC by 3.0 license)

ENTHOUGHT

GitHub Top Languages 2013



ENTHOUGHT

IPython

An enhanced
interactive Python shell

ENTHOUGHT

Starting IPython

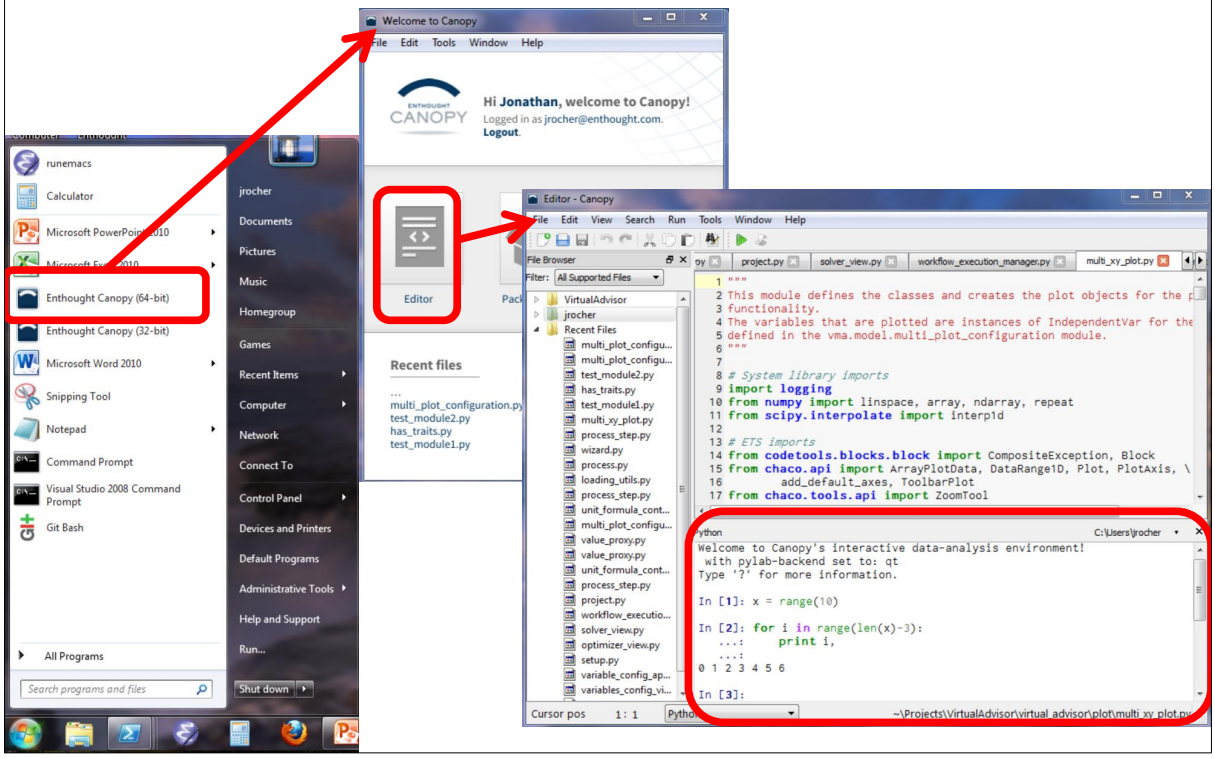
IPython can be started:

- w/ the pylab icon
- by typing `ipython --pylab` in any terminal/command prompt


```
jrocher — Python — 75×19
Last login: Sun Aug  4 20:03:00 on ttys005
Admins-MacBook-Pro-2:~ jrocher$ ipython --pylab
Enthought Python Distribution -- www.enthought.com

Python 2.7.3 | 64-bit | (default, Jun 14 2013, 18:17:36)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX
For more information, type 'help(pylab)'.

In [1]:
```


ENTHOUGHT

Starting IPython in Canopy

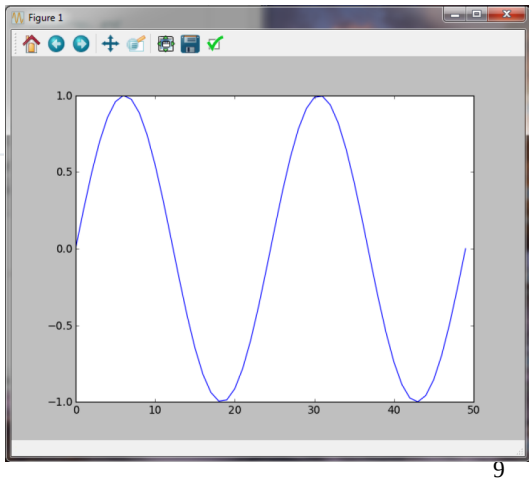


ENTHOUGHT

PyLab: Interactive Python Environment

Python

```
In [4]: x = linspace(0, 4*pi)

In [5]: plot(sin(x))
Out[5]: [<matplotlib.lines.Line2D at 0x84909e8>]

In [6]: print sin(3*pi/4)
0.707106781187

In [7]: sin(3*pi/4) == sqrt(2)/2
Out[7]: True

In [8]:
```



The PyLab mode in IPython handles some gory details behind the scenes. It allows both the Python command interpreter (above) and the GUI plot window (right) to coexist. This involves a bit of multi-threaded magic.

PyLab also imports some handy functions into the command interpreter for user convenience.

ENTHOUGHT

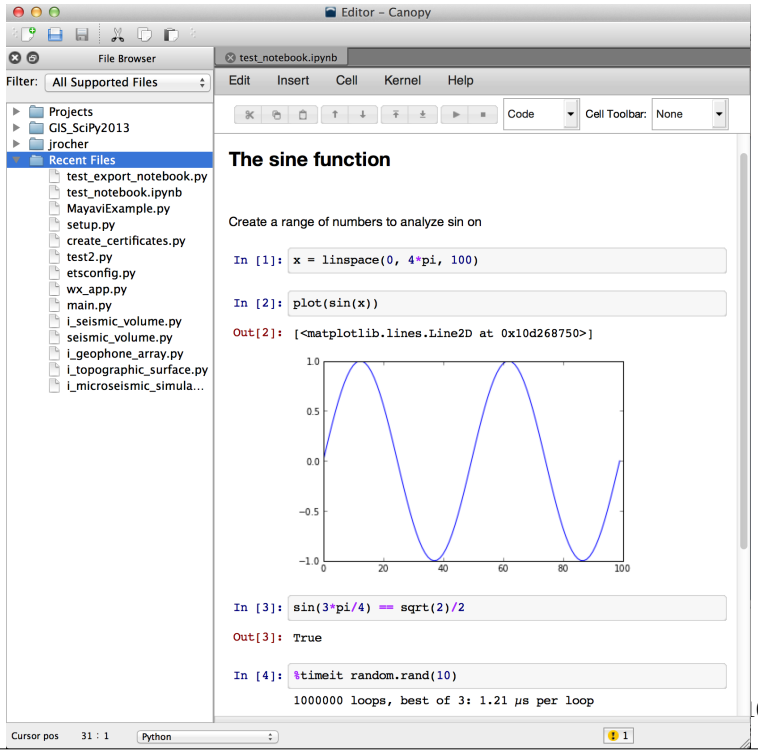
IPython notebook: exportable session

Contains in 1 file the inputs, the outputs, figures, plain text (markdown), titles and more...

- `.ipynb` files.
- Create a new one in Canopy by selecting:

File > New >

IPython notebook



ENTHOUGHT

IPython

STANDARD PYTHON

```
In [1]: a=1

In [2]: a
Out[2]: 1
```

AVAILABLE VARIABLES

```
In [3]: b = [1,2,3]

# List available variables.
In [4]: %whos
Variable   Type    Data/Info
---------------------------
a          int     1
b          list    n=3
```

RESET

```
# Remove user defined variables.
In [5]: %reset
In [6]: %whos
Interactive namespace is empty.
```

"%reset" also removes the names imported by PyLab, such as the plot command.

```
In [7]: plot
NameError: name 'plot' is not
defined

# Reload pylab.
In [8]: %pylab
Welcome to pylab,…
```


ENTHOUGHT

Directory Navigation in IPython

```
# Change directory (note Unix style forward slashes!)
In [9]: cd c:/python_class/Demos/speed_of_light
c:\python_class\Demos\speed_of_light
```

Tab completion helps you find and type directory and file names.

```
# List directory contents (Unix style, not "dir").
In [10]: ls
 Volume in drive C has no label.
 Volume Serial Number is 5417-593D
 Directory of c:\python_class\Demos\speed_of_light
09/01/2008  02:53 PM  <DIR>          .
09/01/2008  02:53 PM  <DIR>          ..
09/01/2008  02:48 PM         1,188 exercise_speed_of_light.txt
09/01/2008  02:48 PM     2,682,023 measurement_description.pdf
09/01/2008  02:48 PM       187,087 newcomb_experiment.pdf
09/01/2008  02:48 PM         1,312 speed_of_light.dat
09/01/2008  02:48 PM         1,436 speed_of_light.py
09/01/2008  02:48 PM         1,232 speed_of_light2.py
               6 File(s)      2,874,278 bytes
               2 Dir(s)  11,997,437,952 bytes free
```

ENTHOUGHT

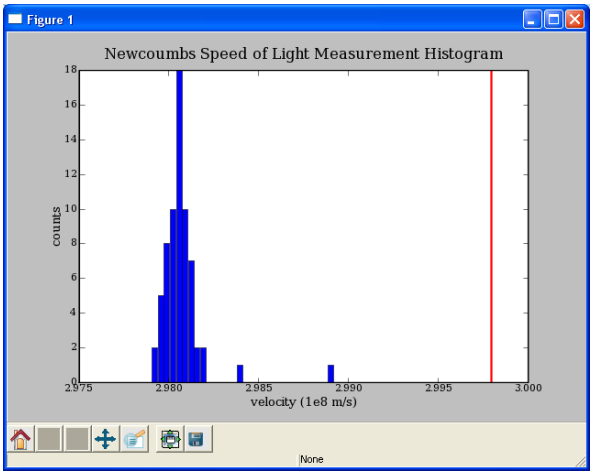
Directory Bookmarks

```
# Print working directory name (Unix style, not "cd").
In [11]: pwd
c:\python_class\Demos\speed_of_light

# Bookmark the demo and exercise directories, so we can return
# to them easily.
In [12]: cd ..
c:\python_class\Demos

In [13]: %bookmark demo
In [14]: cd ../Exercises
c:\python_class\Exercises

In [15]: %bookmark exer
In [16]: %bookmark -l
demo -> c:\python_class\Demos
exer -> c:\python_class\Exercises

In [17]: cd demo
(bookmark:demo) -> c:\python_class\Demos
```


ENTHOUGHT

Running Scripts in IPython

```
# tab completion
In [11]: %run speed_of_li
speed_of_light.dat  speed_of_light.py

# execute a python file
In [11]: %run speed_of_light.py
```



ENTHOUGHT

Function Info

HELP USING ?

```
# Follow a command with '?' to print its documentation.

In [19]: len?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function len>
Namespace:      Python builtin
Docstring:
    len(object) -> integer

Return the number of items of a sequence or mapping.
```


ENTHOUGHT

Function Info

SHOW SOURCE CODE USING ??

```
# Follow a command with '??' to print its source code.
In [43]: squeeze??
def squeeze(a):
    """Remove single-dimensional entries from the shape of a.
    Examples
    --------
    >>> x = array([[[1,1,1],[2,2,2],[3,3,3]]])
    >>> x.shape
    (1, 3, 3)
    >>> squeeze(x).shape
    (3, 3)
    """
    try:
        squeeze = a.squeeze
    except AttributeError:
        return _wrapit(a, 'squeeze')
    return squeeze()
```

?? can't show the source code for "extension" functions that are implemented in C.

ENTHOUGHT

IPython History

HISTORY COMMAND

```
# list previous commands. Use
# 'magic' % because 'hist' is
# histogram function in pylab
In [3]: %hist
a=1
a
```

INPUT HISTORY

```
# list string from prompt[2]
In [4]: _i2
Out[4]: 'a\n'
```

OUTPUT HISTORY

```
# grab previous result
In [5]: _
Out[5]: 'a\n'

# grab result from prompt[2]
In [6]: _2
Out[6]: 1
```

The up and down arrows scroll through your ipython input history.

ENTHOUGHT

Reading Simple Tracebacks

ERROR ADDING AN INTEGER TO A STRING

```
In [9]: 1 + "hello"
---------------------
TypeError       Traceback (most recent call last)
C:\...<ipython-input…> in <module>()
----> 1  1 + "hello"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Location and code where error occurred.

The "type" of error that occurred.

Short message about why it occurred.

ERROR TRYING TO ADD A NON-EXISTENT VARIABLE

```
# Again we fail when adding two variables, but note that the
# traceback tells us we have a completely different problem.
# In this case, our variable doesn't exist, so the operation fails.
In [10]: undefined_var + 1
…
NameError: name 'undefined_var' is not defined
```

ENTHOUGHT

Language Introduction

ENTHOUGHT

Interactive Calculator

```
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variable's type
>>> type(a)
<type 'int'>
# an arbitrarily long integer
>>> a = 12345678901234567890
>>> a
12345678901234567890L
>>> type(a)
<type 'long'>
# Remove 'a' from the 'namespace'
>>> del a
>>> a
NameError: name 'a' is not
defined
```

```
# real numbers
>>> b = 1.4 + 2.3
>>> b
3.6999999999999997
# "prettier" version.
>>> print b
3.7
>>> type(b)
<type 'float'>
# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j)
```

The four numeric types in Python on 32-bit architectures are:

`integer` **(4 byte)**
`long integer` **(any precision)**
`float` **(8 byte like C's double)**
`complex` **(16 byte)**

The numpy module, which we will see later, supports a larger number of numeric types.

ENTHOUGHT

More Interactive Calculation

ARITHMETIC OPERATIONS

```
>>> 1+2-(3*4/6)**5+(7%5)
-27
```

SIMPLE MATH FUNCTIONS

```
>>> abs(-3)
3
>>> max(0, min(10, -1, 4, 3))
0
>>> round(2.718281828)
3.0
```

OVERWRITING FUNCTIONS (!)

```
# don't do this
>>> max = 100

# ...some time later...
>>> x = max(4, 5)
TypeError: 'int' object is not
callable
```

TYPE CONVERSION

```
>>> int(2.718281828)
2
>>> float(2)
2.0
>>> 1+2.0
3.0
```

ALTERNATIVE NOTATIONS

```
>>> 0xFF
255
>>> 023  # octal!
19
>>> 6e3
6000.0
```

IN-PLACE OPERATIONS

```
>>> b = 2.5
>>> b += 0.5      # b = b + 0.5
>>> b
3.0
# Also -=, *=, /=, etc.
```


ENTHOUGHT

Logical expressions, `bool` data type

COMPARISON OPERATORS

```
# <, >, <=, >=, ==, !=
>>> 1 >= 2
False
>>> 1 + 1 == 2
True
>>> 2**3 != 3**2
True
# Chained comparisons
>>> 1 < 10 < 100
True
```

`bool` DATA TYPE

```
>>> q = 1 > 0
>>> q
True
>>> type(q)
<type 'bool'>
```

`and` OPERATOR

```
>>> 1 > 0 and 5 == 5
True
# If first operand is false,
# the second is not evaluated.
>>> 1 < 0 and max(0,1,2) > 1
False
```

`or` OPERATOR

```
>>> a = 50
>>> a < 10 or a > 90
False
# If first operand is true,
# the second is not evaluated.
>>> a = 0
>>> a < 10 or a > 90
True
```

`not` OPERATOR

```
>>> not 10 <= a <= 90
True
```

ENTHOUGHT

Strings

CREATING STRINGS

```
# using double quotes
>>> s = "hello world"
>>> print s
hello world
# single quotes also work
>>> s = 'hello world'
>>> print s
hello world
```

STRING OPERATIONS

```
# concatenating two strings
>>> "hello " + "world"
'hello world'

# repeating a string
>>> "hello " * 3
'hello hello hello '
```

STRING LENGTH

```
>>> s = "12345"
>>> len(s)
5
```

SPLIT/JOIN STRINGS

```
# split space-delimited words
>>> s = "hello world"
>>> wrd_lst = s.split()
>>> print wrd_lst
['hello', 'world']

# join words back together
# with a space in between
>>> space = ' '
>>> space.join(wrd_lst)
'hello world'
```


ENTHOUGHT

A few string methods and functions

REPLACING TEXT

```
>>> s = "hello world"
>>> s.replace('world','Mars')
'hello Mars'
```

CONVERT TO UPPER CASE

```
>>> s.upper()
'HELLO WORLD'
```

REMOVE WHITESPACE

```
>>> s = "\t    hello    \n"
>>> s.strip()
'hello'
```

NUMBERS TO STRINGS

```
>>> str(1.1 + 2.2)
'3.3'
>>> repr(1.1 + 2.2)
'3.3000000000000003'
>>> str(1)
'1'
>>> hex(255)
'0xFF'
>>> oct(19)
'023'
```

STRINGS TO NUMBERS

```
>>> int('23')
23
>>> int('FF', 16)
255
>>> float('23')
23.0
```

ENTHOUGHT

Available string methods

```
# list available methods on a string
>>> dir(s)
[…
'capitalize',
'center',
'count',
'decode',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'index',
'isalnum',
'isalpha',
'isdigit',
'islower',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```


ENTHOUGHT

Multi-line Strings

TRIPLE QUOTES

```
# strings in triple quotes
# retain line breaks
>>> a = """hello
... world"""
>>> print a
hello
world
```

LINE CONTINUATION

```
# The \ character means line
# continuation.  Be careful
# because it must be the last
# character on line.
>>> a = "hello " \
...     "world"
>>> print a
hello world
```

MULTI-LINE WITH PARENTHESES

```
# group strings using
# parentheses
>>> a = ("hello "
...      "world")
>>> print a
hello world
```

NEW LINE CHARACTER

```
# including the new line
>>> a = "hello\nworld"
>>> print a
hello
world
```

ENTHOUGHT

String Formatting

`str.format(*args, **kwargs)`

The format() method replaces the ***replacement fields*** in the string with the values given as arguments. Any other text in the string remains unchanged. For example:

```
>>> '{0:2d}-{1}: {name}, ${price:.2f}'.format(7, 19, name='SC1', price=3.4)
' 7-19: SC1, $3.40'
```

Optional

Replacement field: {<field_name> :<format_spec> }

- Delimited by curly brackets. (Use {{ and }} to include curly brackets in the output.)
- If **`field_name`** is an integer, it refers to a position in the positional arguments:
  ```
  >>> '{0} is greater than {1}'.format(100, 50)
  '100 is greater than 50'
  ```
- If **`field_name`** is a name, it refers to a *keyword argument*:
  ```
  >>> '{last}, {first}'.format(first='Ellen', last='Ripley')
  'Ripley, Ellen'
  ```


ENTHOUGHT

String Formatting - Format Spec

```
>>> 'price: ${0:=-7.2f}'.format(3.4)
'price: $   3.40'
```

The *format spec* is a sequence of characters including:

- the *alignment* option,
- the *sign* option,
- the *width* (and *.precision*) option
- the *type code*.

ALIGNMENT OPTION

Char	Meaning
<	Left aligned.
>	Right aligned.
=	(For numeric types only.) Pad after the sign but before the digits (e.g. +000000120).
^	Center within the available space.

If an alignment character is given, it may be preceded by a *fill character*.

SIGN OPTION

For numbers only.

Char	Meaning
+	Include a sign for positive and negative number.
-	Indicate sign for negative numbers only (default)
space	Include a leading space for positive numbers.

STRING TYPE CODES

Type	Meaning
s	String. This is the default, and may be omitted.

INTEGER TYPE CODES

Type	Meaning
b	Binary format.
c	Character; converts int to unicode char.
d	Decimal integer (base 10).
o	Octal (base 8).
x	Hex (base 16), lower case.
X	Hex (base 16), upper case.
n	Number; same as 'd', but uses current locale.
None	Same as 'd'.

FLOATING POINT TYPE CODES

Type	Meaning
e	Scientific notation.
E	Scientific notation, with upper case 'E'.
f	Fixed point.
F	Fixed point; same as 'f'.
g	General format.
G	General format; same as 'g', with upper case 'E' when necessary.
n	Number; same as 'g', but uses current locale.
%	Percentage. Multiplies by 100 and displays with 'f', followed by a percent sign.
None	Same as 'g'.

ENTHOUGHT

String Formatting - Examples

```
# Basic string formatting
>>> print '{} {} {}'.format('a', 'b', 'c')
a b c

# Numbered fields refer to the position of the arguments
>>> print '{2} {1} {0}'.format('a', 'b', 'c')
c b a

# Named fields refer to keyword arguments
>>> print '{color} {n} {x}'.format(n=10, x=1.5, color='blue')
blue 10 1.5

# Positional and keyword arguments can be combined
>>> print '{color} {0} {x} {1}'.format(10, 'foo', x=1.5, color='blue')
blue 10 1.5 foo

# Precision and padding
>>> from math import pi
>>> print '{0:10} {1:10d} {c:10.2f}'.format('foo', 5, c=2*pi)
foo                5       6.28
```


ENTHOUGHT

String Formatting - Examples cont.

```
# Fixed point format (and a named keyword argument).
>>> print '[{x:5.0f}]  [{x:5.1f}]  [{x:5.2f}]'.format(x=12.3456)
[   12]  [ 12.3]  [12.35]

# Sign options.
>>> '{:-f}; {:-f}'.format(3.14, -3.14)       # Default
'3.140000; -3.140000'
>>> '{:+f}; {:+f}'.format(3.14, -3.14)       # Use '+'
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)       # Use ' '
' 3.140000; -3.140000'

# Alignment (and using a numbered positional argument).
>>> print '[{0:<10s}]  [{0:>10s}]  [{0:^10s}]'.format('PYTHON')
[PYTHON    ]  [    PYTHON]  [  PYTHON  ]
# Alignment with fill character.
>>> print '[{0:*<10s}]  [{0:*>10s}]  [{0:*^10s}]'.format('PYTHON')
[PYTHON****]  [****PYTHON]  [**PYTHON**]

# Different bases for an integer (hex, decimal, octal, binary).
>>> '{0:X} {0:d} {0:o} {0:b}'.format(254)
'FE 254 376 11111110'
```

ENTHOUGHT

Formatting with %

FORMAT OPERATOR %

```
# the % operator formats values
# to strings using C conventions.
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> t = "%s %f, %d" % (s,x,y)
>>> print t
some numbers: 1.340000, 2

>>> y = -2.1
>>> print "%f\n%f" % (x,y)
1.340000
-2.100000

>>> print "% f\n% f" % (x,y)
 1.340000
-2.100000

>>> print "%4.2f" % x
1.34
```

CONVERSION CODES

Conversion	Meaning
d or i	Signed integer decimal
o	Unsigned octal
u	Unsigned decimal
x	Unsigned hexadecimal (lowercase)
X	Unsigned hexadecimal (uppercase)
e	Floating point exponential format (lowercase)
E	Floating point exponential format (uppercase)
F or f	Floating point decimal format
G or g	Floating point format or exponential
c	Single character
r	Converts object using repr()
s	Converts object using str()

CONVERSION FLAGS

Flag	Meaning
0	The conversion will be zero padded for numeric values.
-	The converted value is left adjusted (overrides the "0" conversion if both are given).
<space>	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

ENTHOUGHT

List objects

LIST CREATION WITH BRACKETS

```
>>> a = [10,11,12,13,14]
>>> print a
[10, 11, 12, 13, 14]
```

CONCATENATING LIST

```
# simply use the + operator
>>> [10, 11] + [12, 13]
[10, 11, 12, 13]
```

REPEATING ELEMENTS IN LISTS

```
# the multiply operator
# does the trick
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

range(start, stop, step)

```
# the range function is helpful
# for creating a sequence
>>> range(5)
[0, 1, 2, 3, 4]

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

>>> range(2,7,2)
[2, 4, 6]
```

ENTHOUGHT

Indexing

RETRIEVING AN ELEMENT

```
# list
# indices: 0  1  2  3  4
  >>>  a = [10,11,12,13,14]
  >>>  a[0]
  10
```

SETTING AN ELEMENT

```
>>> a[1] = 21
>>> print a
[10, 21, 12, 13, 14]
```

OUT OF BOUNDS

```
>>> a[10]
Traceback (innermost last):
File "<interactive input>",line 1,in ?
IndexError: list index out of range
```

NEGATIVE INDICES

```
# negative indices count
# backward from the end of
# the list
#
# indices: -5 -4 -3 -2 -1
    >>> a = [10,11,12,13,14]

    >>> a[-1]
    14
    >>> a[-2]
    13
```

The first element in an array has `index=0` as in C. ***Take note Fortran programmers!***

ENTHOUGHT

More on list objects

LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,
# string, and another list
>>> a = [10,'eleven',[12,13]]
>>> a[1]
'eleven'
>>> a[2]
[12, 13]

# use multiple indices to
# retrieve elements from
# nested lists
>>> a[2][0]
12
```

Prior to version 2.5, Python was limited to sequences with ~2 billion elements. Python 2.5 can handle up to 2^{63} elements.

LENGTH OF A LIST

```
>>> len(a)
3
```

DELETING OBJECT FROM LIST

```
# use the del keyword
>>> del a[2]
>>> a
[10,'eleven']
```

DOES THE LIST CONTAIN x ?

```
# use in or not in
>>> a = [10,11,12,13,14]
>>> 13 in a
True
>>> 13 not in a
False
```

ENTHOUGHT

Slicing

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element *is not included.*
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING LISTS

```
# indices:
#       -5 -4 -3 -2 -1
#        0  1  2  3  4
>>> a = [10,11,12,13,14]
# [10,11,12,13,14]
>>> a[1:3]
[11, 12]

# negative indices work also
>>> a[1:-2]
[11, 12]
>>> a[-4:3]
[11, 12]
```

OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
[10, 11, 12]
# grab last two elements
>>> a[-2:]
[13, 14]
# every other element
>>> a[::2]
[10, 12, 14]
```


ENTHOUGHT

A few methods for list objects

some_list.append(x)

Add the element x to the end of the list some_list.

some_list.count(x)

Count the number of times x occurs in the list.

some_list.extend(sequence)

Concatenate sequence onto this list.

some_list.index(x)

Return the index of the first occurrence of x in the list.

some_list.insert(index, x)

Insert x before the specified index.

some_list.pop(index)

Return the element at the specified index. Also, remove it from the list.

some_list.remove(x)

Delete the first occurrence of x from the list.

some_list.reverse( )

Reverse the order of elements in the list.

some_list.sort(cmp)

By default, sort the elements in ascending order. If a compare function is given, use it to sort the list.

ENTHOUGHT

List methods in action

```
>>> a = [10,21,23,11,24]

# add an element to the list
>>> a.append(11)
>>> print a
[10,21,23,11,24,11]
# how many 11s are there?
>>> a.count(11)
2
# extend with another list
>>> a.extend([5,4])
>>> print a
[10,21,23,11,24,11,5,4]
# where does 11 first occur?
>>> a.index(11)
3
# insert 100 at index 2?
>>> a.insert(2, 100)
>>> print a
[10,21,100,23,11,24,11,5,4]

# pop the item at index=3
>>> a.pop(3)
23
# remove the first 11
>>> a.remove(11)
>>> print a
[10,21,100,24,11,5,4]
# sort the list (in-place)
# Note: use sorted(a) to
#       return a new list.
>>> a.sort()
>>> print a
[4,5,10,11,21,24,100]
# reverse the list
>>> a.reverse()
>>> print a
[100,24,21,11,10,5,4]
```


ENTHOUGHT

Mutable vs. Immutable

MUTABLE OBJECTS

```
# Mutable objects, such as
# lists, can be changed
# in place.

# insert new values into list
>>> a = [10,11,12,13,14]
>>> a[1:3] = [5,6]
>>> print a
[10, 5, 6, 13, 14]
```

The cStringIO module treats strings like a file buffer and allows insertions. It's useful when working with large strings or when speed is paramount.

IMMUTABLE OBJECTS

```
# Immutable objects, such as
# integers and strings,
# cannot be changed in place.

# try inserting values into
# a string
>>> s = 'abcde'
>>> s[1:3] = 'xy'
Traceback (innermost last):
File "<interactive input>",line 1,in ?
TypeError: object doesn't support
           slice assignment

# here's how to do it
>>> s = s[:1] + 'xy' + s[3:]
>>> print s
'axyde'
```

ENTHOUGHT

Tuple – Immutable Sequence

TUPLE CREATION

```
>>> a = (10,11,12,13,14)
>>> print a
(10, 11, 12, 13, 14)
```

PARENTHESES ARE OPTIONAL

```
>>> a = 10,11,12,13,14
>>> print a
(10, 11, 12, 13, 14)
```

LENGTH-1 TUPLE

```
>>> (10,)
(10,)
>>> (10)
10
```

(10) is not a tuple, but an integer with parentheses.

TUPLES ARE IMMUTABLE

```
# create a list
>>> a = range(10,15)
[10, 11, 12, 13, 14]

# cast the list to a tuple
>>> b = tuple(a)
>>> print b
(10, 11, 12, 13, 14)

# try inserting a value
>>> b[3] = 23
TypeError: 'tuple' object doesn't
support item assignment
```


ENTHOUGHT

Dictionaries

Dictionaries store *key/value* pairs. Indexing a dictionary by a *key* returns the *value* associated with it. The *key* must be immutable.

DICTIONARY EXAMPLE

```
# Create an empty dictionary using curly brackets.
>>> record = {}
# Each indexed assignment creates a new key/value pair.
>>> record['first'] = 'Jmes'
>>> record['last'] = 'Maxwell'
>>> record['born'] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
# Create another dictionary with initial entries.
>>> new_record = {'first': 'James', 'middle':'Clerk'}
# Now update the first dictionary with values from the new one.
>>> record.update(new_record)
>>> print record
{'first': 'James', 'middle': 'Clerk', 'last':'Maxwell', 'born':
1831}
```

ENTHOUGHT

Accessing and deleting keys and values

ACCESS USING INDEX NOTATION

```
>>> print record['first']
James
```

ACCESS WITH get(key, default)

The get() method returns the value associated with a key; the optional second argument is the return value if the key is not in the dictionary.

```
>>> record.get('born',0)
1831
>>> record.get('home', 'TBD')
'TBD'
>>> record['home']
KeyError: ...
```

REMOVE AN ENTRY WITH DEL

```
>>> del record['middle']
>>> record
{'born': 1831, 'first':
'James', 'last': 'Maxwell'}
```

REMOVE WITH pop(key, default)

pop() removes the key from the dictionary and returns the value; the optional second argument is the return value if the key is not in the dictionary.

```
>>> record.pop('born', 0)
1831
>>> record
{'first': 'James', 'last':
'Maxwell'}
>>> record.pop('born', 0)
0
```


ENTHOUGHT

A few dictionary methods

some_dict.clear( )

Remove all key/value pairs from the dictionary, some_dict.

some_dict.copy( )

Create a copy of the dictionary

x in some_dict

Test whether the dictionary contains the key x.

some_dict.keys( )

Return a list of all the keys in the dictionary.

some_dict.values( )

Return a list of all the values in the dictionary.

some_dict.items( )

Return a list of all the key/value pairs in the dictionary.

ENTHOUGHT

Dictionary methods in action

```
# dict of animals:count pairs
>>> barn = {'cows': 1,
...         'dogs': 5,
...         'cats': 3}

# test for chickens
>>> 'chickens' in barn
False

# get a list of all keys
>>> barn.keys()
['cats','dogs','cows']

# get a list of all values
>>> barn.values()
[3, 5, 1]
```

```
# return key/value tuples
>>> barn.items()
[('cats', 3), ('dogs', 5),
 ('cows', 1)]

# How many cats?
>>> barn['cats']
3


# Change the number of cats.
>>> barn['cats'] = 10
>>> barn['cats']
10

# Add some sheep.
>>> barn['sheep'] = 5
>>> barn['sheep']
5
```


ENTHOUGHT

Set objects

DEFINITION

A set is an *unordered* collection of *unique*, *immutable* objects.

CONSTRUCTION

```
# an empty set
>>> s = set()
# convert a sequence to set
>>> t = set([1,2,3,1])
# note removal of duplicates
>>> t
set([1, 2, 3])
```

ADD ELEMENTS

```
>>> t.add(5)
set([1, 2, 3, 5])
>>> t.update([5,6,7])
set([1, 2, 3, 5, 6, 7])
```

REMOVE ELEMENTS

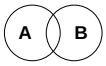
```
# Remove an element. Raise
# exception if not in the set.
>>> t.remove(1)
>>> t
set([2, 3, 5, 6, 7])
# Remove and return an
# arbitrary element from the set.
>>> t.pop()
2
# Remove element from list
# if it exists.
>>> t.discard(3)
>>> t
set([5, 6, 7])
# Otherwise, do nothing.
>>> t.discard(20)
>>> t
set([5, 6, 7])
```

ENTHOUGHT

Set Operations

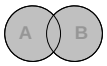
OVERLAPPING SETS

```
>>> a = set([1,2,3,4])
>>> b = set([3,4,5,6])
```



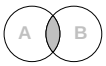
UNION

```
>>> a.union(b)
set([1, 2, 3, 4, 5, 6])
>>> a | b
set([1, 2, 3, 4, 5, 6])
```



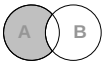
INTERSECTION

```
>>> a.intersection(b)
set([3, 4])
>>> a & b
set([3, 4])
```

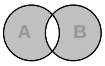


DIFFERENCE

```
>>> a.difference(b)
set([1, 2])
>>> a - b
set([1, 2])
```



SYMMETRIC DIFFERENCE




```
>>> a.symmetric_difference(b)
set([1, 2, 5, 6])
>>> a ^ b #xor
set([1, 2, 5, 6])
```


ENTHOUGHT

Set Containment

OVERLAPPING SETS

```
>>> a = set([1,2,3])
>>> b = set([1,2])
```

ISSUBSET

```
# Is b fully contained in a?
>>> b.issubset(a)
True
>>> b <= a
True
```

ISSUPERSET

```
# Does a fully contain b?
>>> a.issuperset(b)
True
>>> a >= b
True
```

There is also an immutable type of set called a frozenset (analogous to a tuple) which can be used as a dictionary key or an element of a set.

ENTHOUGHT

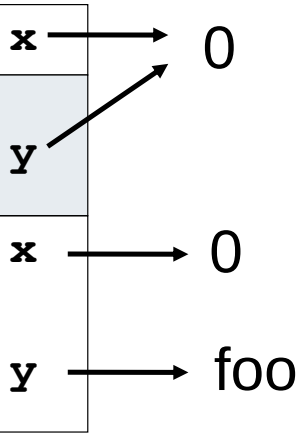
Assignment of “simple” object

Assignment creates object references.

```
>>> x = 0
```

```
# This causes x and y to point
# to the same value.
>>> y = x
```

```
# Re-assigning y to a new value
# decouples the two variables.
>>> y = "foo"
>>> print x
0
```



ENTHOUGHT

Assignment of Container object

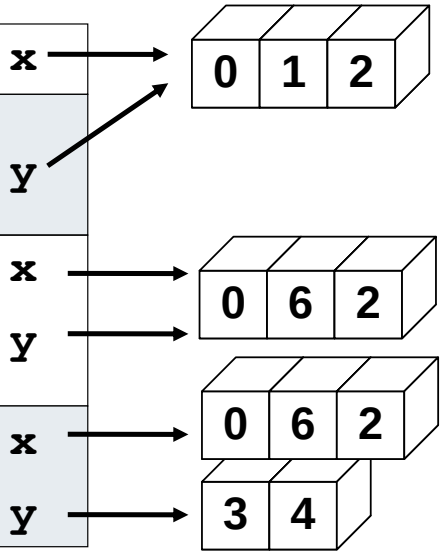
Assignment creates object references.

```
>>> x = [0, 1, 2]
```

```
# This causes x and y to point
# to the same list.
>>> y = x
```

```
# A change to y also changes x.
>>> y[1] = 6
>>> print x
[0, 6, 2]
```

```
# Re-assigning y to a new list
# decouples the two variables.
>>> y = [3, 4]
```



ENTHOUGHT

If statements

if/elif/else provides conditional execution of code blocks.

IF STATEMENT FORMAT

```
if <condition>:
    <statement 1>
    <statement 2>
elif <condition>:
    <statements>
else:
    <statements>
```

IF EXAMPLE

```
# a simple if statement
>>> x = 10
>>> if x > 0:
...     print 'Hey!'
...     print 'x > 0'
... elif x == 0:
...     print 'x is 0'
... else:
...     print 'x is negative'
... < hit return >
Hey!
x > 0
```


ENTHOUGHT

Test Values

- zero, **None**, "", and empty objects are treated as `False`.
- All other objects are treated as `True`.

EMPTY OBJECTS

```
# empty objects test as false
>>> x = []
>>> if x:
...     print 1
... else:
...     print 0
... < hit return >
0
```

It often pays to be explicit. If you are testing for an empty list, then test for:

```
if len(x) == 0:
    ...
```

This is clearer to future readers of your code. It also can avoid bugs where x==None may be passed in and unexpectedly go down this path.

ENTHOUGHT

While loops

while loops iterate until a condition is met

```
while <condition>:
    <statements>
```

WHILE LOOP

```
# the condition tested is
# whether lst is empty
>>> lst = range(3)
>>> while lst:
...     print lst
...     lst = lst[1:]
... < hit return >
[0, 1, 2]
[1, 2]
[2]
```

BREAKING OUT OF A LOOP

```
# breaking from an infinite
# loop
>>> i = 0
>>> while True:
...     if i < 3:
...         print i,
...     else:
...         break
...     i = i + 1
... < hit return >
0 1 2
```


ENTHOUGHT

For loops

for loops iterate over a sequence of objects

```
for <loop_var> in <sequence>:
    <statements>
```

TYPICAL SCENARIO

```
>>> for item in range(5):
...     print item,
... < hit return >
0 1 2 3 4

# For a large range, xrange()
# is faster and more memory
# efficient.
>>> for item in xrange(10**6):
...     print item,
... < hit return >
0 1 2 3 4 5 6 7 8 9 10 11 ...
```

LOOPING OVER A STRING

```
>>> for item in 'abcde':
...     print item,
... < hit return >
a b c d e
```

LOOPING OVER A LIST

```
>>> animals=['dogs','cats',
...          'bears']
>>> accum = ''
>>> for animal in animals:
...     accum += animal + ' '
... < hit return >
>>> print accum
dogs cats bears
```

ENTHOUGHT

List Comprehension

LIST TRANSFORM WITH LOOP

```
# element by element transform of
# a list by applying an
# expression to each element
>>> a = [10,21,23,11,24]
>>> results=[]
>>> for val in a:
...     results.append(val+1)
>>> results
[11, 22, 24, 12, 25]
```

FILTER-TRANSFORM WITH LOOP

```
# transform only elements that
# meet a criteria
>>> a = [10,21,23,11,24]
>>> results=[]
>>> for val in a:
...     if val>15:
...         results.append(val+1)
>>> results
[22, 24, 25]
```

LIST COMPREHENSION

```
# list comprehensions provide
# a concise syntax for this sort
# of element by element
# transformation
>>> a = [10,21,23,11,24]
>>> [val+1 for val in a]
[11, 22, 24, 12, 25]
```

LIST COMPREHENSION WITH FILTER

```
>>> a = [10,21,23,11,24]
>>> [val+1 for val in a if val>15]
[22, 24, 25]
```

Consider using a list comprehension whenever you need to transform one sequence to another.

ENTHOUGHT

Generator Expressions

Generator expressions are like list comprehensions without the brackets:

LIST COMPREHENSION

```
>>> set([str(i) for i in range(5)])
set(['1', '0', '3', '2', '4'])

# tests on sequences
>>> any([i >= 3 for i in range(5)])
True
>>> all([i >= 3 for i in range(5)])
False

# summation
>>> sum([i**2 for i in range(5)])
30
```

GENERATOR EXPRESSION

```
>>> set(str(i) for i in range(5))
set(['1', '0', '3', '2', '4'])

# tests on sequences
>>> any(i >= 3 for i in range(5))
True
>>> all(i >= 3 for i in range(5))
False

# summation
>>> sum(i**2 for i in range(5))
30
```

ENTHOUGHT

Multiple assignments

FROM TUPLE TO TUPLE

```
# creating a tuple without ()
>>> d = 1,2,3
>>> d
(1, 2, 3)


# multiple assignments
>>> a,b,c = 1,2,3
>>> print b
2

# multiple assignments from a
# tuple
>>> a,b,c = d
>>> print b
2
```

FROM LIST TO TUPLE

```
# also works for lists
>>> a,b,c = [1,2,3]
>>> print b
2
```


ENTHOUGHT

Looping Patterns

MULTIPLE LOOP VARIABLES

```
# Looping through a sequence of
# tuples allows multiple
# variables to be assigned.
>>> pairs = [(0,'a'),(1,'b'),
...          (2,'c')]
>>> for index, value in pairs:
...     print index, value
0 a
1 b
2 c
```

ZIP

```
# zip 2 or more sequences
# into a list of tuples
>>> x = [0, 1, 2]
>>> y = ['a', 'b', 'c']
>>> zip(x,y)
[(0,'a'), (1,'b'), (2,'c')]
>>> for index, value in zip(x,y):
...     print index, value
0 a
1 b
2 c
```

ENUMERATE

```
# enumerate -> index, item.
>>> y = ['a', 'b', 'c']
>>> for index, value in enumerate(y):
...     print index, value
0 a
1 b
2 c
```

REVERSED

```
>>> z = [(0,'a'),(1,'b'),(2,'c')]
for index, value in reversed(z):
...     print index, value
2 c
1 b
0 a
```

ENTHOUGHT

Looping over a dictionary

```
>>> d = {'a':1, 'b':2, 'c':3}
```

DEFAULT LOOPING (KEYS)

```
>>> for key in d:
...     print key, d[key]
a 1
c 3
b 2
```

LOOPING OVER VALUES

```
>>> for val in d.values():
...     print val
1
3
2
```

LOOPING OVER KEYS (EXPLICIT)

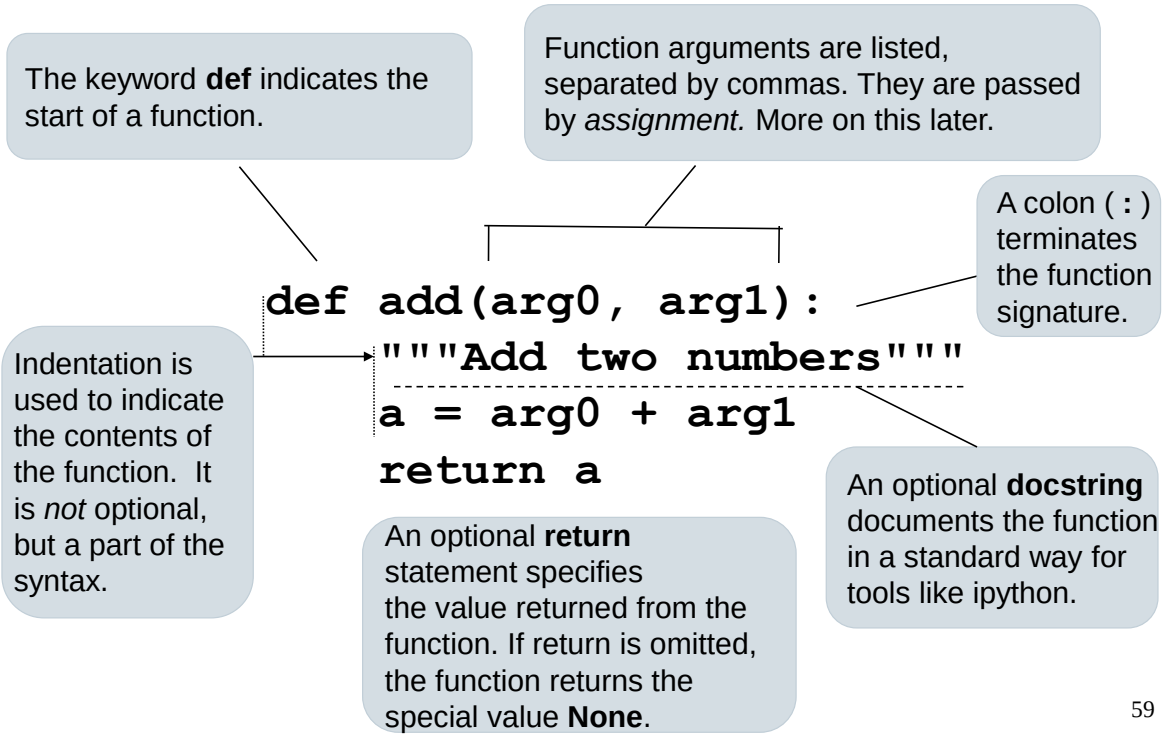
```
>>> for key in d.keys():
...     print key, d[key]
a 1
c 3
b 2
```

LOOPING OVER ITEMS

```
>>> for key, val in d.items():
...     print key, val
a 1
c 3
b 2
```


ENTHOUGHT

Anatomy of a function



ENTHOUGHT

Our new function in action

```
# We'll create our function
# on the fly in the
# interpreter.
>>> def add(x,y):
...     a = x + y
...     return a

# Test it out with numbers.
>>> val_1 = 2
>>> val_2 = 3
>>> add(val_1,val_2)
5
```

```
# How about strings?
>>> val_1 = 'foo'
>>> val_2 = 'bar'
>>> add(val_1,val_2)
'foobar'


# Functions can be assigned
# to variables.
>>> func = add
>>> func(val_1, val_2)
'foobar'
```

```
# How about numbers and strings?
>>> add('abc',1)
Traceback (innermost last):
File "<interactive input>", line 1, in ?
File "<interactive input>", line 2, in add
TypeError: cannot add type "int" to string
```


ENTHOUGHT

Function Calling Conventions

POSITIONAL ARGUMENTS

```
# The "standard" calling
# convention we know and love.
>>> def add(x, y):
...     return x + y

>>> add(2, 3)
5
```

KEYWORD ARGUMENTS

```
# specify argument names
>>> add(x=2, y=3)
5
# or even a mixture if you are
# careful with order
>>> add(2, y=3)
5
```

DEFAULT VALUES

```
# Arguments can be
# assigned default values.
>>> def quad(x,a=1,b=1,c=0):
...   return a*x**2 + b*x + c

# Use defaults for a, b and c.
>>> quad(2.0)
6.0

# Set b=3. Defaults for a & c.
>>> quad(2.0, b=3)
10.0

# Keyword arguments can be
# passed in out of order.
>>> quad(2.0, c=1, a=3, b=2)
17.0
```

ENTHOUGHT

Function Calling Conventions

VARIABLE NUMBER OF ARGS

```
# Pass in any number of
# arguments. Extra arguments
# are put in the tuple args.
>>> def foo(x, y, *args):
...     print x, y, args

>>> foo(2, 3, 'hello', 4)
2 3 ('hello', 4)
```

VARIABLE KEYWORD ARGS

```
# Extra keyword arguments
# are put into the dict kw.

>>> def bar(x, y=1, **kw):
...     print x, y, kw

>>> bar(1, y=2, a=1, b=2)
1 2 {'a': 1, 'b': 2}
```


ENTHOUGHT

Function Calling Conventions

THE 'ANYTHING' SIGNATURE

```
# This signature takes any
# number of positional and
# keyword arguments.
>>> def foo(*args, **kw):
...     print args, kw

>>> foo(2, 3, x='hello', y=4)
(2, 3) {'x': 'hello', 'y': 4}
```

MULTIPLE FUNCTION RETURNS

```
# To return multiple values
# from a function, we return
# a tuple containing those
# values. This is a common
# use of multiple (tuple)
# assignment.
>>> def functions(x):
...     y1 = x**2 + x
...     y2 = x**3 + x**2 + 2*x
...     return y1, y2

>>> a, b = functions(c)
```

ENTHOUGHT

Expanding Function Arguments

POSITIONAL ARGUMENT EXPANSION

```
>>> def add(x, y):
...     return x + y

# '*' in a function call
# converts a sequence into the
# arguments to a function.
>>> vars = [1,2]
>>> add(*vars)
3
```

KEYWORD ARGUMENT EXPANSION

```
>>> def bar(x, y=1, **kw):
...     print x, y, kw

# '**' expands a
# dictionary into keyword
# arguments for a function.
vars = {'y':3, 'z':4}
>>> bar(1, **vars)
1, 3, {'z': 4}
```


ENTHOUGHT

Modules

EX1.PY

```
# ex1.py

PI = 3.1416

def sum(lst):
    tot = lst[0]
    for value in lst[1:]:
        tot = tot + value
    return tot

w = [0,1,2,3]
print sum(w), PI
```

FROM SHELL

```
[ej@bull ej]$ python ex1.py
6 3.1416
```

FROM INTERPRETER

```
# load and execute the module
>>> import ex1
6 3.1416
# get/set a module variable
>>> print ex1.PI
3.1416
>>> ex1.PI = 3.14159
>>> print ex1.PI
3.14159
# call a module function
>>> t = [2,3,4]
>>> ex1.sum(t)
9
```

ENTHOUGHT

Modules *cont.*

INTERPRETER

```
# load and execute the module
>>> import ex1
6 3.1416
< edit file >
# import module again
>>> import ex1
# nothing happens!!!


# Use reload to force a
# previously imported library
# to be reloaded.
>>> reload(ex1)
10 3.14159
```

EDITED EX1.PY

```
# ex1.py version 2

PI = 3.14159

def sum(lst):
    tot = 0
    for value in lst:
        tot = tot + value
    return tot

w = [0,1,2,3,4]
print sum(w), PI
```


ENTHOUGHT

Modules *cont. 2*

A Python file can be used as a script, or as a module, or both.

EX2.PY

```
# An example module that can
# be run as a script.
PI = 3.1416

def sum(lst):
    """ Sum the values in a
        list.
    """
    tot = 0
    for value in lst:
        tot = tot + value
    return tot
```

```
def add(x,y):
    " Add two values."
    a = x + y
    return a

def test():
    w = [0,1,2,3]
    assert( sum(w) == 6)
    print 'test passed'


# This code runs only if this
# module is the main program.
if __name__ == '__main__':
    test()
```

ENTHOUGHT

`import` Variations

BASIC IMPORTS

```
# The most basic import
>>> import ex2
>>> ex2.PI
3.1416
```

ALIASING A NAME

```
# Use an 'alias'
>>> import ex2 as e2
>>> e2.PI
3.1416
```

IMPORTING SPECIFIC SYMBOLS

```
# Select specific names to
# bring into the local
# namespace.
>>> from ex2 import add, PI
>>> PI
3.1416
>>> add(2,3)
5
```

IMPORTING *EVERYTHING*

```
# Pull *everything* into the
# local namespace.
>>> from ex2 import *
>>> PI
3.1416
>>> add(3,4.5)
7.5
```


ENTHOUGHT

Modules *cont. 3*

PACKAGES

Often a library will contain several modules. These may be organized in a hierarchical directory structure, and imported using "dotted module names". The first and the intermediate names (if any) are called "packages".

Example:

```
>>> from foo.bar import func
>>> from foo.baz import zap
```

`bar.py` and **`baz.py`** are modules in the *package* **`foo`**.

FILE STRUCTURE

```
foo/
    __init__.py
    bar.py (defines func)
    baz.py (defines zap)
```

- The directory **`foo`** must be in the python search path
- The file **`__init__.py`** indicates that foo is a package. It may be an empty file.
- In the simplest case:
  package = directory
  module = python file

ENTHOUGHT

Standard Modules

Python has a large library of standard modules ("batteries included"):

re - regular expressions

copy – shallow and deep copy operations

datetime - time and date objects

math, cmath - real and complex math

decimal, fraction - arbitrary precision decimal and rational number objects

os, os.path, shutil - filesystem operations

sqlite3 - internal SQLite database

gzip, bz2, zipfile, tarfile – compression and archiving formats

csv, netrc – file format handling

xml – various modules for handling XML

htmllib – an HTML parser

httplib, ftplib, poplib, socket, etc. – modules for standard internet protocols

cmd – support for command interpreters

pdb – Python interactive debugger

profile, cProfile, timeit – Python profilers

collections, heapq, bisect – standard CS algorithms and data structures

mmap – memory-mapped files

threading, Queue – threading support

multiprocessing – process based 'threading'

subprocess – executing external commands

pickle, cPickle – object serialization

struct – interpret bytes as packed binary data

and many more… To see the content of one:

>>> dir(module_name**)**

ENTHOUGHT

Setting up PYTHONPATH

PYTHONPATH is an environment variable (or set of registry entries on Windows) that lists the directories Python searches for modules.

WINDOWS

- Right-click on My Computer
- Click Properties
- Click Advanced Tab
- Click Environment Variables Button at the bottom of the Advanced Tab
  - Click New to create PYTHONPATH or
  - Click Edit to change existing PYTHONPATH
- Changes take effect in the next Command Prompt or IPython session.

UNIX: .cshrc

```
!! NOTE: The following should !!
!! all be on one line         !!

setenv PYTHONPATH
    $PYTHONPATH:$HOME/your_modules
```

UNIX: .bashrc

```
PYTHONPATH=$PYTHONPATH:$HOME/your
_modules
export PYTHONPATH
```

ENTHOUGHT

Reading files

FILE INPUT EXAMPLE

```
>>> results = []
>>> f = open('c:\\rcs.txt','r')
# Read all the lines.
>>> lines = f.readlines()
>>> f.close()
# Discard the header.
>>> lines = lines[1:]

>>> for line in lines:
...     # split line into fields
...     fields = line.split()
...     # convert text to numbers
...     freq = float(fields[0])
...     vv = float(fields[1])
...     hh = float(fields[2])
...     # group & append to results
...     all = [freq,vv,hh]
...     results.append(all)
... < hit return >
```

PRINTING THE RESULTS

```
>>> for i in results: print i
[100.0, -20.30…, -31.20…]
[200.0, -22.70…, -33.60…]
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)   vv (dB)   hh (dB)
  100         -20.3     -31.2
  200         -22.7     -33.6
```

See demo/reading_files directory for code.

ENTHOUGHT

More compact version

ITERATING ON A FILE AND LIST COMPREHENSIONS

```
>>> results = []
>>> f = open('c:\\rcs.txt','r')
>>> f.readline()
'#freq (MHz)  vv (dB)  hh (dB)\n'
>>> for line in f:
...     all = [float(val) for val in line.split()]
...     results.append(all)
... < hit return >
>>> for i in results:
...     print i
... < hit return >
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)   vv (dB)   hh (dB)
  100         -20.3     -31.2
  200         -22.7     -33.6
```

ENTHOUGHT

Writing files

FILE OUTPUT

```
>>> # Mode 'w': create new file:
>>> f = open('a.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
>>> open('a.txt').read()
'Hello, world!'
>>> # Use the 'with' statement:
>>> with open('a.txt', 'w') as f:
....     f.write('Wow!')
....
>>> open('a.txt').read()
'Wow!'
>>> # Mode 'a': append to file:
>>> with open('a.txt', 'a') as f:
....     f.write(' Boo.')
....
>>> open('a.txt').read()
'Wow! Boo.'
```

REDIRECTED PRINT

```
>>> # Redirect output of a
>>> # print statement:
>>> f = open('a.txt', 'w')
>>> print >> f, "Here I am."
>>> f.close()
>>> open('a.txt').read()
'Here I am.\n'
```

WRITE AND READ

```
>>> f = open('a.txt', 'w+')
>>> print >> f, 12, 34, 56
>>> f.seek(3)
>>> f.read(2)
'34'
>>> f.close()
```


ENTHOUGHT

Particle Class Example

SIMPLE PARTICLE CLASS

```
>>> class Particle(object):
...     # Initializer method
...     def __init__(self, m, v):
...         self.mass = m  # Assign attribute values of new object
...         self.velocity = v
...     # Method for calculating object momentum
...     def momentum(self):
...         return self.mass * self.velocity
...     # A "magic" method defines object's string representation.
...     # Evaluating the repr recovers the Particle object.
...     def __repr__(self):
...         return "Particle({0}, {1})".format(
...             repr(self.mass), repr(self.velocity))
```

EXAMPLE

```
>>> a = Particle(3.2,4.1)
>>> a.momentum()
13.119999999999999
>>> a
Particle(3.2, 4.1)
```

ENTHOUGHT

Overloading Addition

SIMPLE PARTICLE CLASS

Classes can override many behaviors using special method names — including numeric behavior.

```
...     def __add__(self, other)
...         if not isinstance(other, Particle):
...             return NotImplemented
...         mnew = self.mass + other.mass
...         vnew = (self.momentum() + other.momentum())/mnew
...         return Particle(mnew, vnew)
```

EXAMPLE (cont.)

```
>>> b = Particle(8.6,10.2)
>>> b, a
((m:8.6, v:10.2), (m:3.2, v:4.1))
>>> c = a+b
>>> c
(m:11.8, v:8.5)
>>> print c.momentum()
100.84
```


ENTHOUGHT

Sorting

IN-PLACE VS NEW LIST

```
# sorting
>>> x = ['s','o','r','t']
# new list
>>> sorted(x)
['o', 'r', 's', 't']
# in-place
>>> x.sort()
>>> x
['o', 'r', 's', 't']

# reversing
# new list (note explicit 'list')
>>> x = ['b', 'a', 'c', 'k']
>>> list(reversed(x))
['k', 'c', 'a', 'b']
# in-place
>>> x.reverse()
>>> x
['k', 'c', 'a', 'b']
```

CUSTOM SORTS

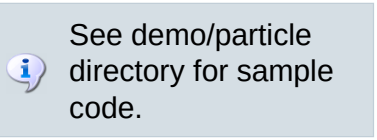
```
# define a key function to
# transform values before
# comparing in a sort
>>> def ignore_case(x):
...     return x.lower()

>>> x = ['S','o','r','T']
>>> sorted(x)
['S', 'T', 'o', 'r']
>>> sorted(x, key=ignore_case)
['o', 'r', 'S', 'T']
```

ENTHOUGHT

Sorting

SORTING CLASS INSTANCES

```
# comparison functions for a variety of particle values
>>> def by_mass(x):
...     return x.mass
>>> def by_momentum(x):
...     return x.momentum()
>>> def by_kinetic_energy(x):
...     return 0.5*x.mass*x.velocity**2

# sorting particles in a list by their various properties
>>> from particle import Particle
>>> x = [Particle(1.2,3.4), Particle(2.1,2.3), Particle(4.6,.7)]
>>> sorted(x, key=by_mass)
[(m:1.2, v:3.4), (m:2.1, v:2.3), (m:4.6, v:0.7)]

>>> sorted(x, key=by_momentum)
[(m:4.6, v:0.7), (m:1.2, v:3.4), (m:2.1, v:2.3)]

>>> sorted(x, key=by_kinetic_energy)
[(m:4.6, v:0.7), (m:2.1, v:2.3), (m:1.2, v:3.4)]
```



See demo/particle directory for sample code.

ENTHOUGHT

Excellent source of help

http://www.python.org/doc

ENTHOUGHT

Advanced Python Topics

ENTHOUGHT

General Scoping Rules

Variables are looked up from namespaces in the following order:

- Local Function Scope
- Enclosing Function Scope
- Global Scope
- Builtin Scope

ENTHOUGHT

Scoping Rules

LOCAL SCOPE

```
# a, b, c and d are all local
# variables in the function.
def foo(a,b):
    c = 1
    d = a + b + c
```

GLOBAL SCOPE

```
# If a requested variable is not
# found in the local scope, the
# "global" or module level scope
# is searched.

# global module level variable
c = 1

def foo(a,b):
    d = a + b + c
```

MODIFYING GLOBALS

```
# Modifying globals from within
# a local scope requires the
# global keyword.

c = 1

def foo():
    global c
    c = 2

>>> foo()
>>> print c
2
```


ENTHOUGHT

Scoping Rules

BUILTIN SCOPE

```
# If not found in the local or
# global scope, the "builtin"
# scope is searched.  'len' is
# a builtin function.

def list_length(a):
    return len(a)

>>> a = [1,2,3]
>>> print list_length(a)
3

# The "builtin" functions are
# found in the special
# __builtin__ module
>>> import __builtin__
>>> __builtin__.len
<built-in function len>
```

ENTHOUGHT

Class Scoping Rules

CLASS SCOPING

```
# global variable
var = 0

class MyClass(object):
    # class variable
    var = 1

    def access_class_c(self):
        print 'class var:', self.var

    def access_global_c(self):
        print 'global var:', var

    def write_class_c(self):
        # Modify the class variable.
        MyClass.var = 2
        print 'class var:', self.var

    def write_instance_c(self):
        # Create an instance variable.
        self.var = 3
        print 'instance var:', self.var
```

EXAMPLES

```
>>> obj = MyClass()
>>> obj.access_class_c()
class var: 1
>>> obj.access_global_c()
global var: 0
>>> obj.write_class_c()
class var: 2
>>> obj.write_instance_c()
instance var: 3
```

See demo/class_scoping.py.

ENTHOUGHT

Lexical Scoping Rules

NESTED SCOPE

```
# Nested functions have access
# to the enclosing function's
# variables.

def outer():
    a = 1
    def inner():
        print "a =", a
    inner()

>>> outer()
a = 1
```

See lexical_scoping.py demo.

FUNCTION CLOSURE

```
# The inner() function has access
# to the outer variables that it
# uses for its entire lifetime.

def outer():
    a = 1
    def inner():
        return a
    return inner #Note: returns
                 #function object!

>>> func = outer()
>>> print 'a (1):', func()
a (1): 1
```

ENTHOUGHT

Partial Function Application (or Curries)

PARTIAL

```
# functools.partial "freezes"
# one or more arguments of a
# function to create a simplified
# signature and/or modify default
# arguments.

# By default, int uses base 10
# conversion from string to int.
>>> int('10010')
10010

# Create function to convert
# base 2 strings to ints.
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo('10010')
18
```


ENTHOUGHT

Object Oriented Programming In Python

ENTHOUGHT

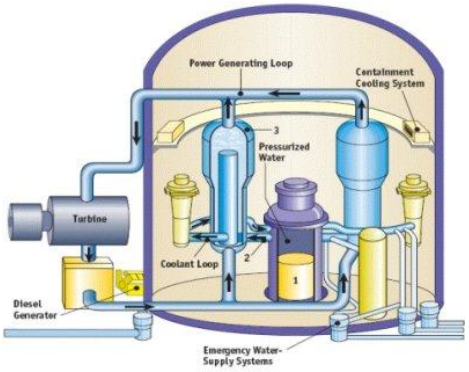
Modeling a Power Plant

ATTRIBUTES

```
# Data that describes the
# modeled "object"

name: Comanche Peak
maximum_output: 2.3 Gigawatts
current_output:  ?  Gigawatts
status: normal
night_watchman: Homer Simpson (!)
...
```

BEHAVIORS (METHODS)

```
# tasks or operations that the
# model does.
start_up()
shut_down()
emergency_shutdown()
...
```



ENTHOUGHT

Creating Objects

Object oriented programming unifies (encapsulates) attributes and behavior within a single definition, called a Class.

Instances, or Objects, of a class are specific manifestations of that class.

INSTANTIATING CLASS INSTANCES (OBJECTS)

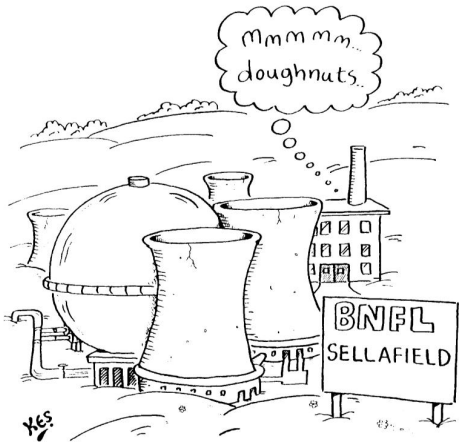
Instance Class

```
>>> plant_1 = PowerPlant(name='Comanche Peak')

# Create a 2nd instance from the same class
>>> plant_2 = PowerPlant(name='Susquehanna')
```

ENTHOUGHT

Working with Objects

```
# Our befuddled friend.
>>> homer = Person(first='Homer', last='Simpson')

# Create the power plant he is in charge of...
>>> plant = PowerPlant(name='Comanche Peak', maximum_output=2.3,
                       night_watchman=homer)

# Start the plant using the 'start_up' method.
>>> plant.start_up()

# Homer does his thing.
>>> homer.eat('donut')
>>> homer.take_nap()

# Check the status (an attribute) of
# the plant. No Bueno.
>>> if plant.status != 'normal':
...     print 'status:', plant.status
...     homer.speak('Doh!')
...     plant.emergency_shutdown()
status: meltdown
Homer says 'Doh!'
```



ENTHOUGHT

Class Definition

```
class PowerPlant(object):

    # Constructor method
    def __init__(self, # self is ALWAYS the first argument.
                 name='', maximum_output=0.0, night_watchman=None):
        # assign passed-in arguments to new object
        self.name = name
        self.maximum_output = maximum_output
        self.night_watchman = night_watchman
        # Intialize other attributes.
        self.current_output = 0.0
        self.status = 'normal'

    # class methods
    def start_up(self):
        self.start_reactor_cooling_pump()
        self.reduce_boric_acid_concentration()
        self.remove_control_rods()

    def shut_down(self):
        ...
    <other methods>
```

ENTHOUGHT

Object Creation Process

CREATING A CLASS OBJECT

```
# What does Python do when it sees this line of code?
>>> plant = PowerPlant(name='Comanche Peak', maximum_output=2.3,
                       night_watchman=homer)
```

UNDER THE COVERS…

```
# The first step is to call the PowerPlant class' "magic"
# __new__ method to create the object.  (This is a secret…)
new_object = PowerPlant.__new__(PowerPlant,
                                name='Comanche Peak',
                                maximum_output=2.3,
                                night_watchman=homer)
# The second step is to call the magic "constructor" method, __init__.
# This is the method you will need to write...
PowerPlant.__init__(new_object,
                    name='Comanche Peak',
                    maximum_output=2.3,
                    night_watchman=homer)
# Python hands this newly created object back to you.
plant = new_object
```


ENTHOUGHT

Particle Class Example

SIMPLE PARTICLE CLASS

```
>>> class Particle(object):
...     # Initializer method
...     def __init__(self, m, v):
...         self.mass = m  # Assign attribute values of new object
...         self.velocity = v
...     # Method for calculating object momentum
...     def momentum(self):
...         return self.mass * self.velocity
...     # A "magic" method defines object's string representation.
...     # Evaluating the repr recovers the Particle object.
...     def __repr__(self):
...         return "Particle({0}, {1})".format(
...             repr(self.mass), repr(self.velocity))
```

EXAMPLE

```
>>> a = Particle(3.2,4.1)
>>> a.momentum()
13.119999999999999
>>> a
Particle(3.2, 4.1)
```

Overloading Addition

SIMPLE PARTICLE CLASS

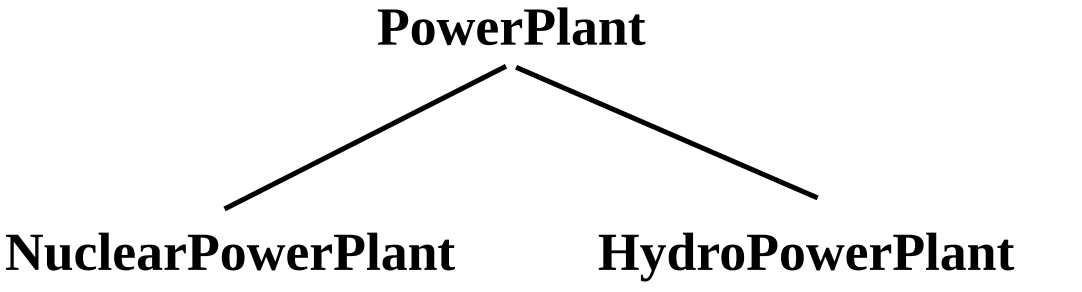
Classes can override many behaviors using special method names — including numeric behavior.

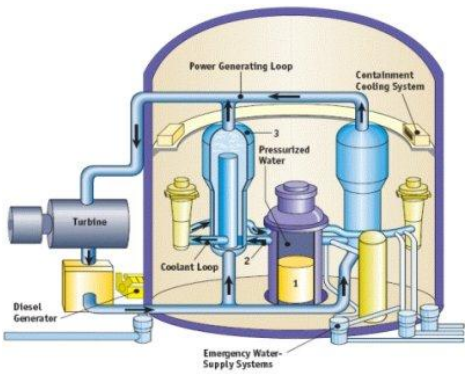
```
...     def __add__(self, other)
...         if not isinstance(other, Particle):
...             return NotImplemented
...         mnew = self.mass + other.mass
...         vnew = (self.momentum() + other.momentum())/mnew
...         return Particle(mnew, vnew)
```

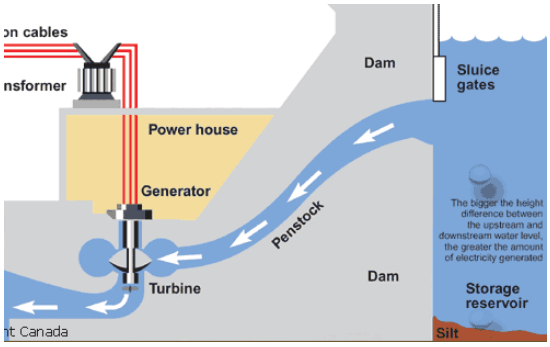
EXAMPLE (cont.)

```
>>> b = Particle(8.6,10.2)
>>> b, a
((m:8.6, v:10.2), (m:3.2, v:4.1))
>>> c = a+b
>>> c
(m:11.8, v:8.5)
>>> print c.momentum()
100.84
```


Inheritance







ENTHOUGHT

Base Class

```
class PowerPlant(object):

    # Constructor method
    def __init__(self, # "self" is ALWAYS the first argument.
                 name='', maximum_output=0.0, night_watchman=None):
        # assign passed-in arguments to new object
        self.name = name
        self.maximum_output = maximum_output
        self.night_watchman = night_watchman
        # Initialize other attributes.
        self.current_output = 0.0
        self.status = 'normal'

    # Default implementation for methods
    def start_up(self):
        self.current_output = self.maximum_output

    def shut_down(self):
        self.current_output = 0.0

    ...
```


ENTHOUGHT

Sub-Classing

```
# Derive 'specialized' classes from the PowerPlant base class.
# They will 'inherit' the methods of the base class.
class NuclearPowerPlant(PowerPlant):

    # over-ride methods that need custom behavior.
    def start_up(self):
        self.start_reactor_cooling_pump()
        self.reduce_boric_acid_concentration()
        self.remove_control_rods()

    def shut_down(self):
        ...
```

ENTHOUGHT

Using Super

```
# Use 'super' to call the "super-class" (parent class) methods.
class HydroPowerPlant(PowerPlant):

    def __init__(self, name='', river_name='',
                 maximum_output=0.0, night_watchman=None):

        # Use 'super' to call an over-ridden method from the
        # base class.
        super(HydroPowerPlant, self).__init__(name=name,
                    maximum_output=maximum_output,
                    night_watchman=night_watchman)
        self.river_name = river_name

    # over-ride methods that need custom behavior.
    def start_up(self):
        self.open_sluice_gate()
        self.unlock_turbine_shaft()
        self.remove_control_rods()
```


ENTHOUGHT

Interchangeable Classes

```
# The same code will work without modification for a
# NuclearPowerPlant or a HydroPowerPlant.
>>> homer = Person(first='Homer', last='Simpson')

# Create the power plant he is in charge of...
>>> plant = HydroPowerPlant(name='Comanche Peak', maximum_output=2.3,
                            night_watchman=homer)

# Start the plant using the 'start_up' method.
>>> plant.start_up()

# Homer does his thing.
>>> homer.eat('donut')
>>> homer.take_nap()

# Check the status (an attribute) of the plant. No Bueno.
>>> if plant.status != 'normal':
...     print 'status:', plant.status
...     homer.speak('Doh!')
...     plant.emergency_shutdown()
status: Fish stuck in impeller.
Homer says 'Doh!'
```

ENTHOUGHT

Error and Exception Handling

ENTHOUGHT

Basic Exception Handling

ERROR ON LOG OF ZERO

```
>>> import math
>>> math.log10(10.0)
1.0
>>> math.log10(0.0)
Traceback (innermost last):
ValueError: math domain error
```

CATCHING ERROR AND CONTINUING

```
>>> a = 0.0
>>> try:
...     r = math.log10(a)
... except ValueError:
...     print 'Warning: overflow occurred. Value set to -100.0'
...     # set value to -100.0 and continue
...     r = -100.0
Warning: overflow occurred. Value set to -100.0
>>> print r
-100.0
```

ENTHOUGHT

Exceptions

ACCESS TO THE EXCEPTION OBJECT

```
try:
    a = value_bag[key]
    r = math.log10(a)
except KeyError as error:
    print "Variable '%s' not found" % error.args
```

Exception Object

MULTIPLE EXCEPTION CLAUSES

```
try:
    a = value_bag[key]
    r = math.log10(a)
except KeyError as error:
    print "Variable '%s' not found" % error.args
except ValueError:
    print 'Warning: overflow occurred. Value set to -100.0'
    # set value to -100.0 and continue
    r = -100.0
```


ENTHOUGHT

Exceptions

HANDLING MULTIPLE EXCEPTIONS IN ONE CODE BLOCK

```
try:
    a = value_bag[key]
    r = math.log10(a)
except (KeyError, ValueError) as error:
    print "an error occurred", error
```

CATCH ALL EXCEPTIONS

```
# If the Exception exception type is
# specified, almost all exceptions
# are caught. Note: Don't do this in
# libraries because it can "mask"
# unexpected exceptions, which
# should be passed to calling code.
try:
    employee = directory_lookup(name)
except Exception as e:
    print "An error occurred: %s" % e
```

You can also leave off the exception type completely and catch all exceptions, including SystemExit and KeyboardInterrupt. This can make it hard to stop execution of a program, so this is almost never a good idea.

ENTHOUGHT

Exceptions – Finally Clause

TRY, FINALLY

```
# The finally clause always executes.  Use it for code that
# needs to "clean up" a resource whether the code executed
# successfully or not.

try:
    # We don't want others to use the radio_station object
    # while it is on the air.
    radio_station.on_air = True
    radio_station.broadcast("Pinball Wizard")
except KeyError as error:
    print "Could not find song '%s'" % error.args
finally:
    # If an exception occurs, or the song finishes,
    # the station is taken off air.
    radio_station.on_air = False
```


ENTHOUGHT

Exceptions – Else Clause

TRY, ELSE

```
# The else clause only executes if an exception *does not*
# occur.

# An example from Python's standard tutorial.
#
# Print out the line count for all the file names passed in
# on the command line.
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

ENTHOUGHT

Raising Exceptions

RAISE

```
# Use the raise statement to raise an exception from your code.

def percent_range_check(percent):
  if not 0.0 <= percent <= 100.0:
    msg = "Percent value should be between 0 and 100"
    # This is the standard, Python 3.0 compatible
    # way to raise exceptions.
    raise ValueError(msg)

    # The following also works, but is deprecated.  It will not work
    # in Python 3.0
    # raise ValueError, msg

>>> percent_range_check(101.0)
ValueError: Percent value should be between 0 and 100
```


ENTHOUGHT

Error Messages

RAISE

```
# Use the raise statement to raise an exception from your code.

def percent_range_check(percent):
  if not 0.0 <= percent <= 100.0:
    msg = "Percent value should be between 0 and 100"
    raise ValueError(msg)

>>> percent_range_check(101.0)
ValueError: Percent value should be between 0 and 100
```

INFORMATIVE ERROR MESSAGES

```
# If possible, print out the offending value in error messages.
def percent_range_check2(percent):
  if not 0.0 <= percent <= 100.0:
    msg = "Percent (%3.2f) is not between 0 and 100" % percent
    raise ValueError(msg)

>>> percent_range_check2(101.0)
ValueError: Percent(101.00) is not between 0 and 100
```

ENTHOUGHT

Warnings

WARNINGS

```
# Warnings alert users without halting execution.
import warnings

def percent_range_warning(percent):
  if not 0.0 <= percent <= 100.0:
    msg = "Percent(%3.2f) is not between 0 and 100" % percent
    warnings.warn(msg, RuntimeWarning)

# Warnings are different than exceptions because they don't halt execution.
>>> percent_range_warning(101.0)
RuntimeWarning: Percent(101.00) is not between 0 and 100

# You can choose to ignore certain types of warnings globally.
# allowable actions: error, ignore, always, default, once, module
>>> warnings.filterwarnings(action="ignore", category=RuntimeWarning)
>>> percent_range_warning(101.0)
<no output>
```


ENTHOUGHT

Exception and Warnings Example

```
import warnings

def weighted_average(values, weights):
    """
    Return the average of all the values weighted by the weights array.
    Both values and weights are numpy arrays and must be the same length.
    The sum of the weights must be 1.0.
    """

    #ensure weights sum to (nearly) 1.0
    weights_total = sum(weights)
    if not allclose(weights_total, 1.0, atol=1e-8):
        msg = "The sum of the weights should usually be 1.0." \
              "Instead, it was '%f'" % weights_total
        warnings.warn(msg)

    # Provide useful error message for unequal array lengths.
    if len(weights) != len(values):
        msg = "The values (len=%d) and weights (len=%d) arrays" \
              "must have the same lengths." % (len(values), len(weights))
        raise ValueError(msg)

    # weighted average calculation
    avg = sum(values * weights) / weights_total

    return avg
```

ENTHOUGHT

Defining New Exceptions

EXCEPTION CLASSES

```
class LengthMismatchError(ValueError):
  """ Sequence of wrong length was used """
  pass

def check_for_length_mismatch(a, b):
  """ Compare the lengths of sequences a and b.  If they are different,
      raise a LengthMismatchError.
  """
  if len(a) != len(b):
    msg = "The two sequences do not have the same length " \
          "(%d!=%d)." % (len(a), len(b))
    raise LengthMismatchError(msg)

>>> check_for_length_mismatch([1,2],[1,2,3])
LengthMismatchError: The two sequences do not have the same length (2!=3).
```

CATCHING BASE EXCEPTIONS

```
# Catching either LengthMismatchError, or ValueError will catch our exception.
>>> try:
...   check_for_length_mismatch([1,2],[1,2,3])
... except ValueError:
...   print "exception caught"
exception caught
```


ENTHOUGHT

Robust File IO Error Handling

TYPICAL FILE IO TRY BLOCKS

```
 try:
     file = open(file_name, "rb")
     try:
         # Move into the file header and read the "name" field.
         file.seek(NAME_OFFSET)
         name = file.read(12)

         # Other file manipulation...

     finally:
         # Ensure File is closed, even if an exception occurs.
         file.close()
except IOError as err:
    # Handle file IO Errors that occur during opening the file or
    # reading data from the file here.  Use err.errno to determine
    # the type of IOError if necessary.

    # Using the logging system to report unhandled exceptions.
    logging.exception("unexpected IOError")
```

ENTHOUGHT

Using ‘with’ for File IO Error Handling

TYPICAL FILE IO TRY BLOCKS

```
# Enable the "with" statement feature within this module.
# Not necessary for Python >= 2.6.
from __future__ import with_statement

try:
     with open("myfile.txt", "rb") as file:
         # Move into the file header and read the “name” field.
         file.seek(NAME_OFFSET)
         name = file.read(12)

         # Other file manipulation...
except IOError as err:
    # Handle file IO Errors that occur during opening the file or
    # reading data from the file here.  Use err.errno to determine
    # the type of IOError if necessary.

    # Using the logging system to report unhandled exceptions.
    logging.exception("unexpected IOError")
```


ENTHOUGHT

Iterators, generators and coroutines

ENTHOUGHT

Iterators

Iterators are anything that can be looped over. They shield users from implementation details of looping over an object. Classes that have **`__iter__`** and **`next`** methods can be used as iterators.

CLASS DEFINTION

```
class FootballRushIterator(object):
  def __init__(self, count=5):
    self.count = count

  def __iter__(self):
    # Prepare an iterable object.
    self._counter = 1
    return self

  def next(self):
    cnt = self._counter
    if cnt <= self.count:
      # Return the next element
      res = str(cnt) + " mississippi"
      self._counter += 1
      return res
   else:
     # Or signal the end of iteration.
     raise StopIteration
```

USING ITERATORS

```
>>> rusher = FootballRushIterator()
>>> for count in rusher:
...     print count
1 mississippi
2 mississippi
3 mississippi
4 mississippi
5 mississippi
```

(WITH GENERATOR)

```
def get_rusher(count=5):
    for i in range(count):
        yield "%d mississippi" % (i+1,)

>>> for count in get_rusher():
...     print count
```


ENTHOUGHT

Generators

Generators are a special kind of function that contain a ***`yield`*** statement. These functions are really **iterator factories**. The returned iterator can be looped over or its next() method can be called directly to return elements from the sequence.

GENERATOR BASICS

```
def simple_generator():
    print 'first'
    yield 1
    print 'second'
    yield 2

>>> sequence = simple_generator()
>>> sequence.next()
first
1
>>> sequence.next()
second
2
>>> sequence.next()
Traceback (most recent call last):
 File "<stdin>" line 1, in ?
StopIteration
```

USING GENERATORS

```
# Create a generator that returns the
# first N elements of the Fibonacci
# sequence.
def fib(N=10):
    a, b = 0, 1
    for count in range(N):
        yield b
        a, b = b, a+b

>>> fib_gen = fib(6)
>>> for value in fib_gen:
    print value
1
1
2
3
5
8
```

ENTHOUGHT

Generators --> CoRoutines

Generators can receive values as well using the return value of a **`yield`** expression: they are called **coroutines**.

COROUTINES BASICS

```
def receiving_generator():
    while True:
        val = (yield)
        print "Received", val

>>> sequence = receiving_generator()

# First next() call take to the first
# yield
>>> sequence.next()
>>> sequence.next()
Received None
>>> sequence.send(1)
Received 1
>>> sequence.send("hello")
Received hello

>>> sequence.close()
```

EMITTING AND RECEIVING

```
def emiting_receiving_generator():
    i = -1
    while True:
        i += 1
        val = (yield i)
        print "Received", val

>>> sequence = receiving_generator()

>>> sequence.next()
0
>>> sequence.next()
Received None
1
>>> sequence.send("hello")
Received hello
2
>>> sequence.close()
```


ENTHOUGHT

datetime – Dates and Times

```
>>> from datetime import date, time
```

DATE OBJECT

```
# date(year, month, day)
# date in Gregorian calendar,
# assuming its permanence
>>> d1 = date(2007, 9, 25)
>>> d2 = date(2008, 9, 25)
>>> d1.strftime('%A %m/%d/%y')
'Tuesday 09/25/07'

# difference is timedelta
>>> print d2 - d1
366 days, 0:00:00
>>> (d2-d1).days
366
>>> print date.today()
2008-09-24
```

TIME OBJECT

```
# time(hour, min, sec, us)
# local time of day
# always 24 hrs per day
>>> t1 = time(15, 38)
>>> t2 = time(18)
>>> t1.strftime('%I:%M %p')
'03:38 PM'

# difference is not supported
>>> print t2 - t1
Traceback ...
TypeError: unsupported operand ...
# use datetime objects for
# difference operation.
```

ENTHOUGHT

datetime – Dates and Times

```
>>> from datetime import datetime, timedelta
```

DATETIME OBJECT

```
# datetime(year, month, day,
           hr, min, sec, us)
# combination of date and time
>>> d1 = datetime.now()
>>> print d1
2008-09-24 14:20:30.978207
>>> d2 = d1 + timedelta(30)
>>> d2.strftime('%A %m/%d/%y')
'Friday 10/24/08'

# creating datetime from
# a format string
>>> datetime.strptime('2/10/01',
                      '%m/%d/%y')
datetime.datetime(2001, 2, 10, 0, 0)
```

DATETIME FORMAT STRING

Directive	Meaning
%a (%A)	Abbrev. (full) weekday name.
%w	Weekday number [0(Sun),6]
%b (%B)	Abbrev. (full) month name
%d	Day of month [01,31]
%H (%I)	Hour [00,23] ([01,12])
%j	Day of the year [001,366]
%m	Month [01,12]
%M	Minute [0,59]
%p	AM or PM
%S	Second [00,61]
%U (%W)	Week number of the year [00,53] Sunday (Monday) as first day of week.
%y (%Y)	Year without (with) century [00,99]

ENTHOUGHT

Connecting to Databases

ENTHOUGHT

DB-API 2.0

PEP (Python Enhancement Proposal) 249 detailed version 2.0 of the DB-API to promote consistency between Python front-ends to data-bases.

Database	Module or Modules for Python 2.X
Oracle	cx_Oracle, DCOracle2, mxODBC, pyodbc
PostgreSQL	psycopg2, PyGreSQL, pyPgSQL, mxODBC, pyodbc, pg8000
MySQL	MySQLdb, mxODBC, pyodbc, myconnpy
Sqlite	sqlite3 **(included in standard library)**
Microsoft SQL Server	adodbapi, pymssql, mssql, mxODBC, pyodbc
Ingres	ingresdbi
IBM DB2	Ibm_db, PyDB2, ceODBC, mxODBC, pyodbc
Sybase ASE	Sybase, mxODBC
Sybase SQL Anywhere	mxODBC, sqlanydb
SAP DB	sdb.dbapi, sapdbapi, mxODBC, sdb.sql, sapdb
Informix	InformixDB, mxODBC,
Firebird	KInterbasdb

ENTHOUGHT

DB-API 2.0

Connect to the database

```
import <somedbmodule> as db
# Connect to the data-source
# Extra arguments typically
# include user and password
conn = db.connect(<dsn>,…)
```

Get a cursor object

```
c = conn.cursor()
```

Execute SQL statements

```
c.execute("create table stocks(date
  text, trans text, symbol text, qty real,
  price real)")

c.execute("insert into stocks values
  ('2006-01-05','BUY','RHAT',100,35.14)")

conn.commit()
```

Execute SQL Statements (cont.)

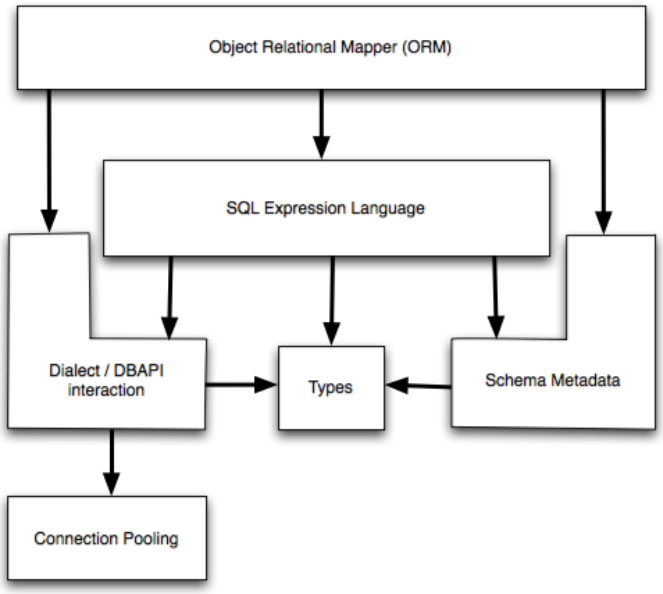
```
t = ('2006-04-06', 'SELL',
     'IBM', 500, 53.00)
c.execute("insert into stocks values
  (?,?,?,?,?)", t)
# some modules use a different
# place-holder instead of ?
# see db.paramstyle

t = ('IBM',)
c.execute("select * from stocks where
  symbol=? order by price", t)
for row in c:
    print row
single = c.fetchone()
list_of_lists = c.fetchall()
```

Close cursor and connection

```
c.close()
conn.close()
```

ENTHOUGHT

An ORM: SQLAlchemy

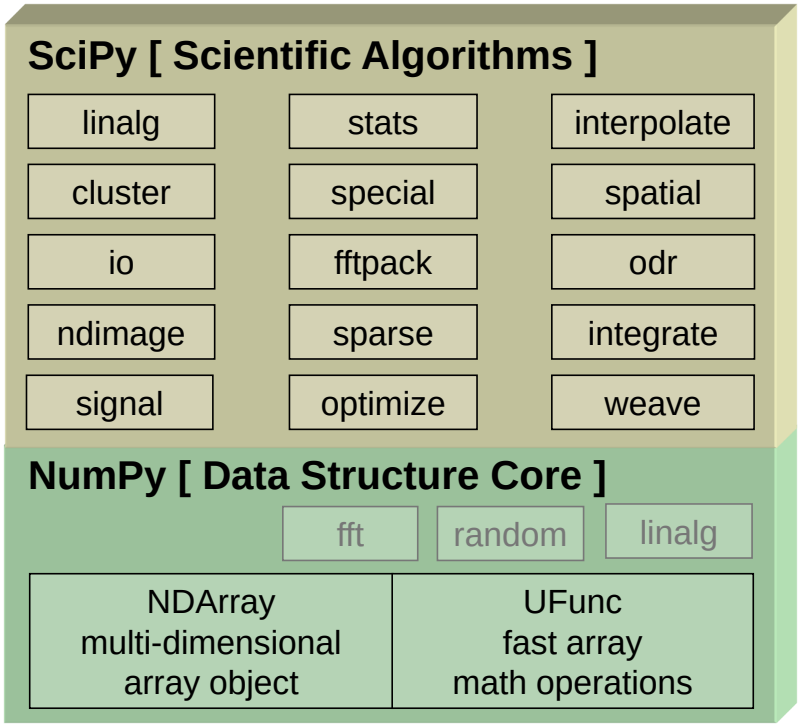


ENTHOUGHT

NumPy

ENTHOUGHT

NumPy and SciPy



ENTHOUGHT

NumPy

- Website: http://numpy.scipy.org/

- Offers Matlab-ish capabilities within Python
- NumPy replaces Numeric and Numarray
- Developed by Travis Oliphant
- >100 “committers” to the project (github.com)
- NumPy 1.0 released October, 2006
- NumPy 1.7.1 released April, 2013
- ~45K downloads/month from SourceForge

This does not count:

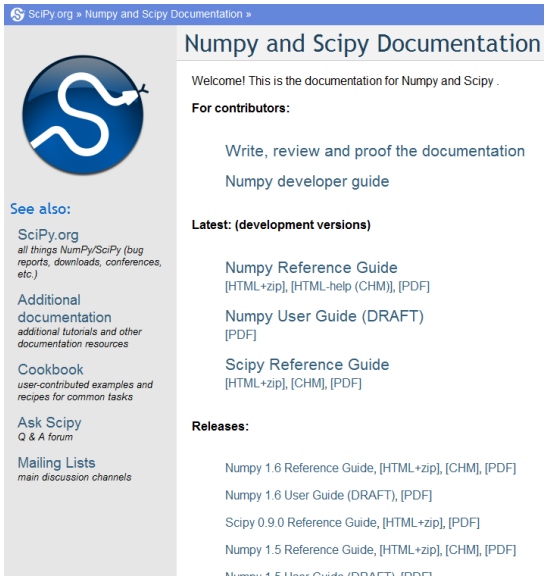
- Linux distributions that include NumPy
- Enthought distributions that include NumPy

ENTHOUGHT

Helpful Sites

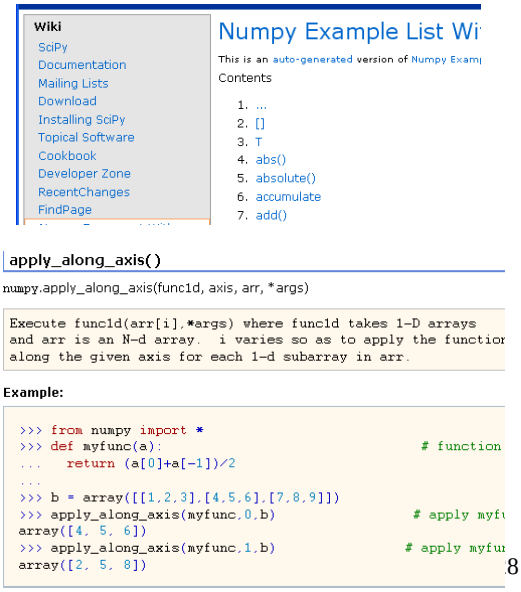
SCIPY DOCUMENTATION PAGE

http://docs.scipy.org/doc



NUMPY EXAMPLES

http://www.scipy.org/Numpy_Example_List_With_Doc



ENTHOUGHT

Getting Started

IMPORT NUMPY

```
In [1]: from numpy import *
In [2]: __version__
Out[2]: 1.6.0
```

or

```
In [1]: from numpy import \
   array, ...
```

Often at the command line, it is handy to import everything from NumPy into the command shell.

However, if you are writing scripts, it is easier for others to read and debug in the future if you use explicit imports.

USING IPYTHON -PYLAB

```
C:\> ipython --pylab
In [1]: array([1,2,3])
Out[1]: array([1, 2, 3])
```

IPython has a 'pylab' mode where it imports all of NumPy, Matplotlib, and SciPy into the namespace for you as a convenience. It also enables threading for showing plots.

While IPython is used for all the demos, '**>>>**' is used on future slides instead of '**In [1]:**' to save space.

ENTHOUGHT

Array Operations

SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
>>> a * b
array([ 2, 6, 12, 20])
>>> a ** b
array([  1,  8,  81,  1024])
```

NumPy defines these constants:
pi = 3.14159265359
e = 2.71828182846

MATH FUNCTIONS

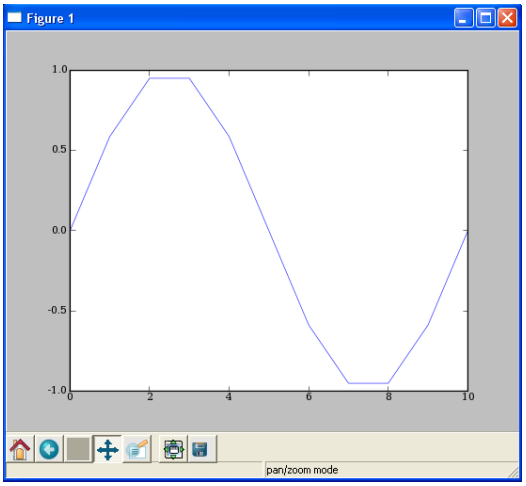
```
# create array from 0 to 10
>>> x = arange(11.)

# multiply entire array by
# scalar value
>>> c = (2*pi)/10.
>>> c
0.62831853071795862
>>> c*x
array([ 0.,0.628,…,6.283])

# in-place operations
>>> x *= c
>>> x
array([ 0.,0.628,…,6.283])

# apply functions to array
>>> y = sin(x)
```


ENTHOUGHT

Plotting Arrays

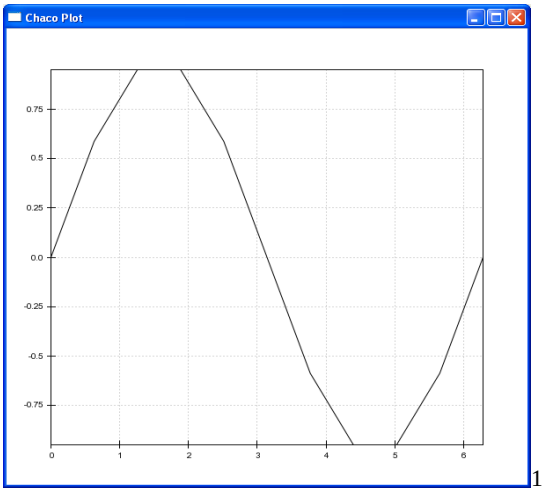
MATPLOTLIB

```
>>> plot(x,y)
```



CHACO SHELL

```
>>> from chaco import shell
>>> shell.plot(x,y)
```



ENTHOUGHT

Matplotlib Basics (an interlude)

ENTHOUGHT

http://matplotlib.org/

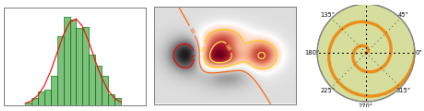
matplotlib

home | search | examples | gallery | docs » modules | index

intro

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.

matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory



For example, using "ipython -pylab" to provide an interactive environment, to generate 10,000 gaussian random numbers and plot a histogram with 100 bins, you simply need to type

```
x = randn(10000)
hist(x, 100)
```

For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users. The pylab mode provides all of the pyplot plotting functions listed below, as well as non-plotting functions from numpy and matplotlib.mlab.

plotting commands

Function	Description
acorr	plot the autocorrelation function

News

Please donate to support matplotlib development.

matplotlib 1.0.1 is available for download. See what's new and tips on installing

Sandro Tosi has a new book Matplotlib for python developers also at amazon.

Build websites like matplotlib's, with sphinx and extensions for mpl plots, math, inheritance diagrams -- try the sampledoc tutorial.

Videos

Watch the SciPy 2009 intro and advanced matplotlib tutorials

Watch a talk about matplotlib presented at NIPS 08 Workshop *MLOSS* and one presented at ChiPy.

Toolkits

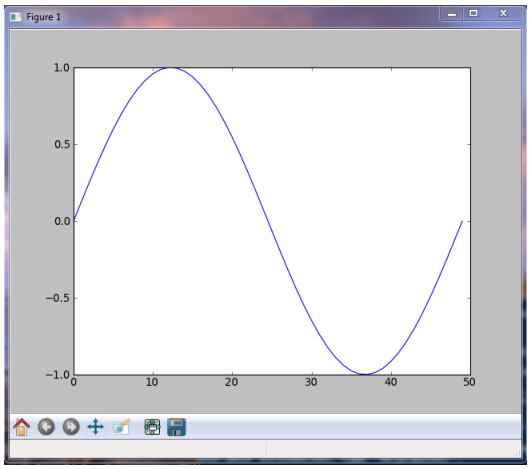
There are several matplotlib addon toolkits, including the projection and mapping toolkit

ENTHOUGHT

Line Plots

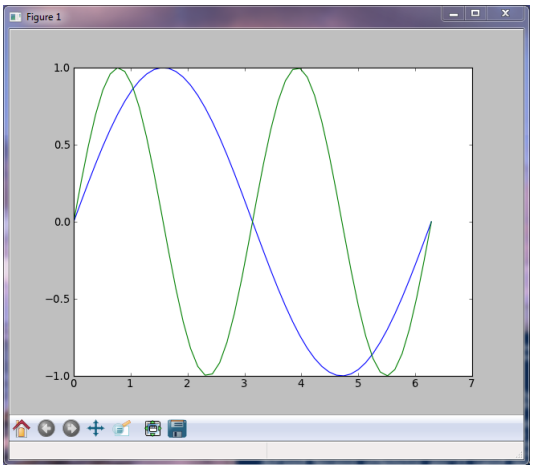
PLOT AGAINST INDICES

```
>>> x = linspace(0,2*pi,50)
>>> plot(sin(x))
```



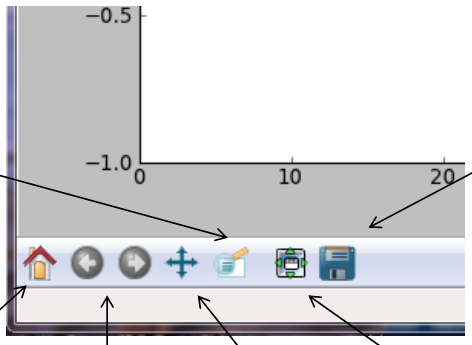
MULTIPLE DATA SETS

```
>>> plot(x, sin(x),
...      x, sin(2*x))
```



ENTHOUGHT

Matplotlib Menu Bar



Zoom tool

Save to file (png, svg, eps, pdf, …)

Back to original view

Previous/ next views

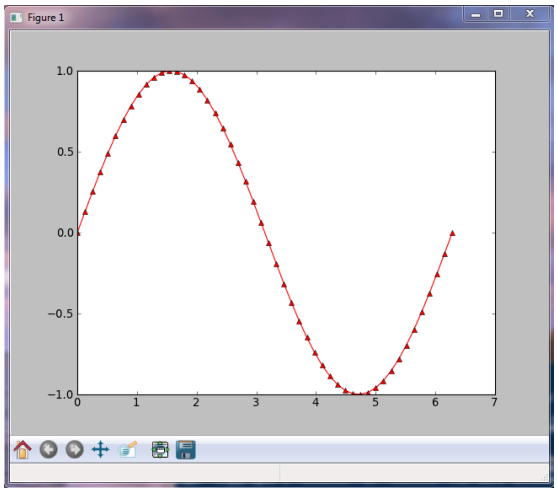
Pan tool

Plot size parameter within window

Line Plots

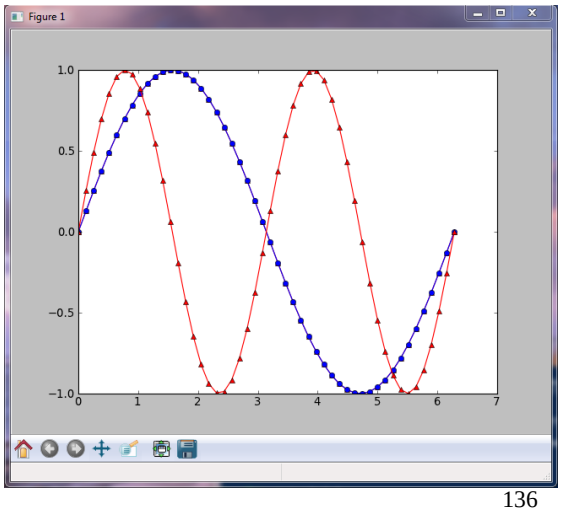
LINE FORMATTING

```
# red, dot-dash, triangles
>>> plot(x, sin(x), 'r-^')
```



MULTIPLE PLOT GROUPS

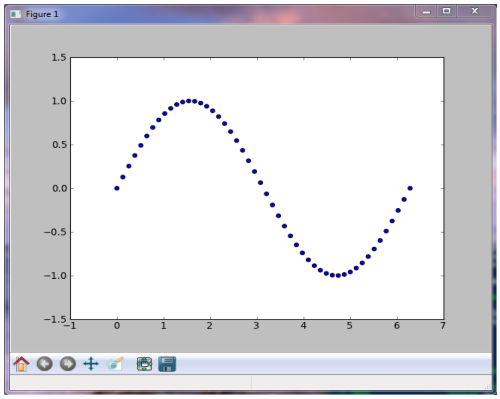
```
>>> plot(x, sin(x), 'b-o',
...      x, sin(2*x), 'r-^')
```



Scatter Plots

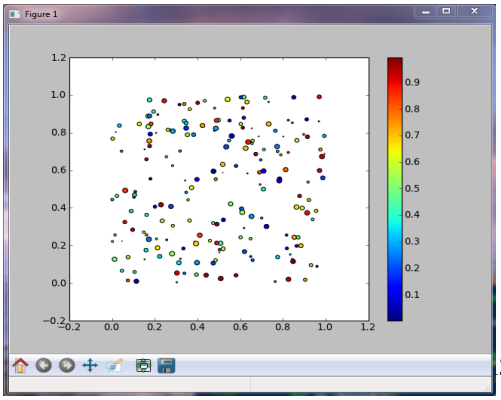
SIMPLE SCATTER PLOT

```
>>> x = linspace(0,2*pi,50)
>>> y = sin(x)
>>> scatter(x, y)
```



COLORMAPPED SCATTER

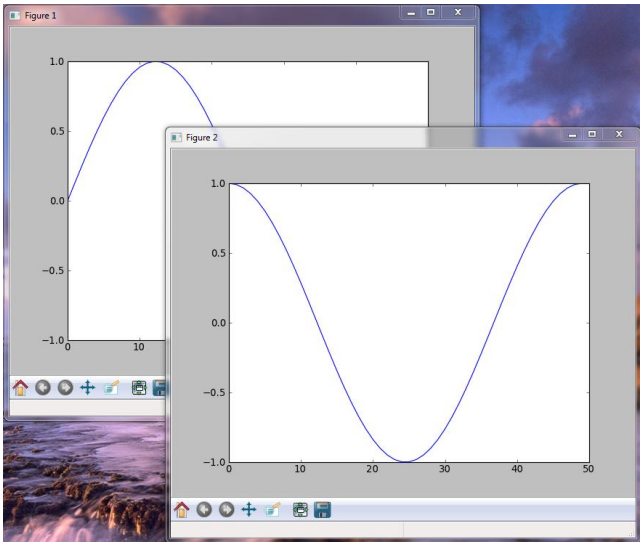
```
# marker size/color set with data
>>> x = rand(200)
>>> y = rand(200)
>>> size = rand(200)*30
>>> color = rand(200)
>>> scatter(x, y, size, color)
>>> colorbar()
```



ENTHOUGHT

Multiple Figures

```
>>> t = linspace(0,2*pi,50)
>>> x = sin(t)
>>> y = cos(t)

# Now create a figure
>>> figure()
>>> plot(x)

# Now create a new figure.
>>> figure()
>>> plot(y)
```



ENTHOUGHT

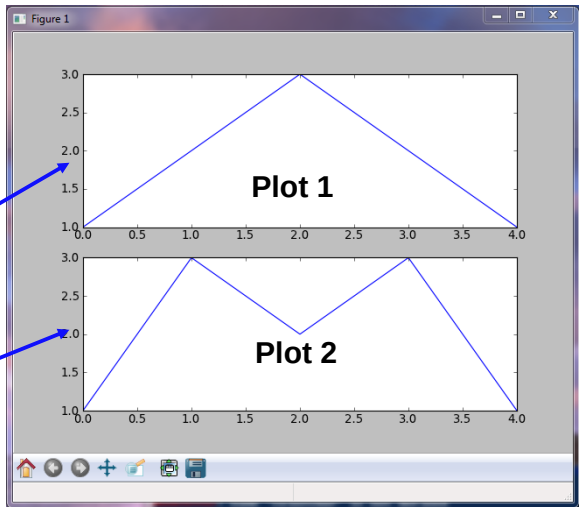
Multiple Plots Using `subplot`

```
>>> x = array([1,2,3,2,1])
>>> y = array([1,3,2,3,1])

# To divide the plotting area
>>> subplot(2, 1, 1)
>>> plot(x)

# Now activate a new plot
# area.
>>> subplot(2, 1, 2)
>>> plot(y)
```

columns

rows

active plot



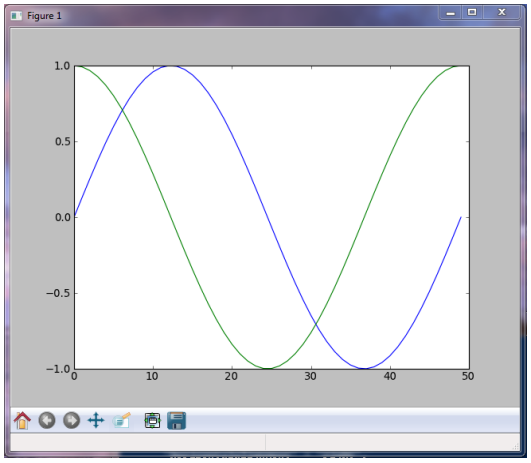
If this is used in a python script, a call to the function show() is required.

ENTHOUGHT

Adding Lines to a Plot

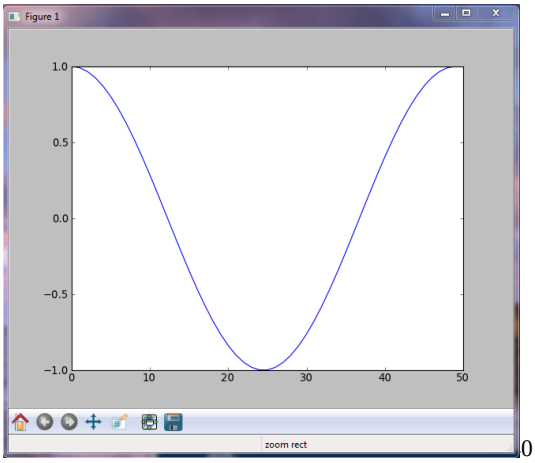
MULTIPLE PLOTS

```
# By default, previous lines
# are "held" on a plot.
>>> plot(sin(x))
>>> plot(cos(x))
```



ERASING OLD PLOTS

```
# Set hold(False) to erase
# old lines
>>> plot(sin(x))
>>> hold(False)
>>> plot(cos(x))
```

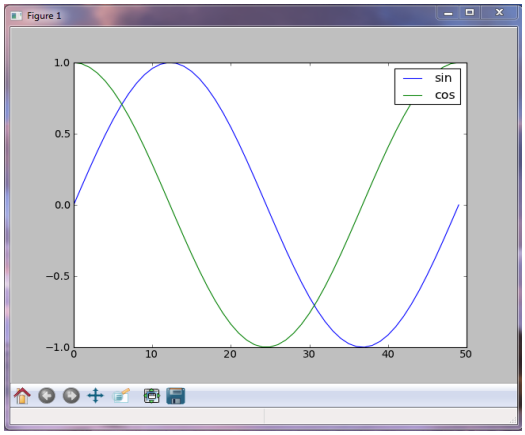


ENTHOUGHT

Legend

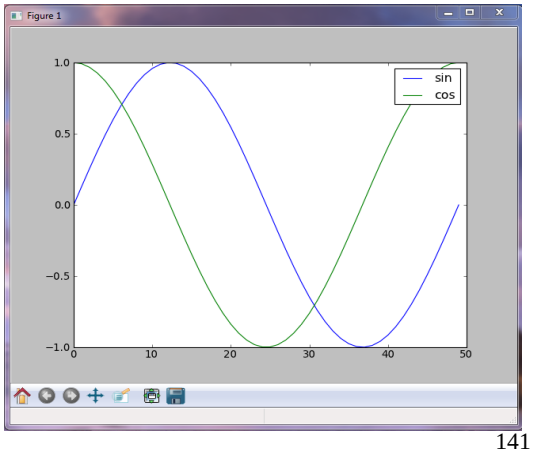
LEGEND LABELS WITH PLOT

```
# Add labels in plot command.
>>> plot(sin(x), label='sin')
>>> plot(cos(x), label='cos')
>>> legend()
```



LABELING WITH LEGEND

```
# Or as a list in legend().
>>> plot(sin(x))
>>> plot(cos(x))
>>> legend(['sin', 'cos'])
```

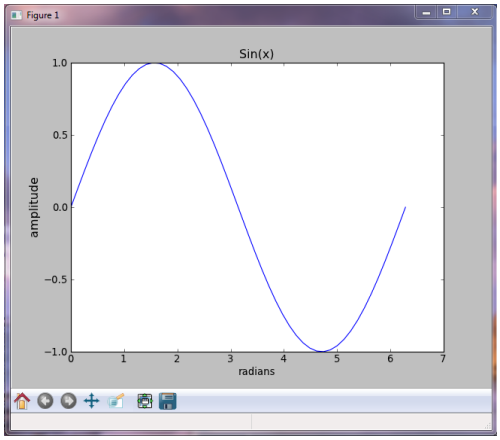


ENTHOUGHT

Titles and Grid

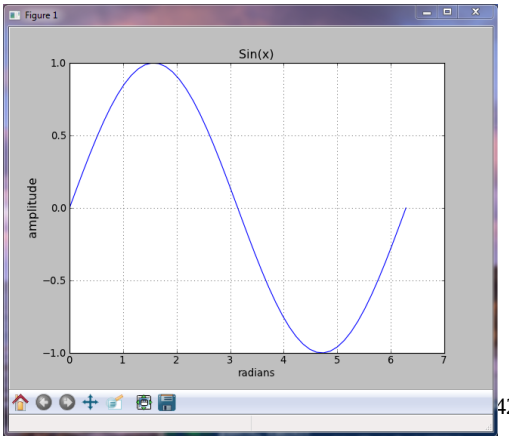
TITLES AND AXIS LABELS

```
>>> plot(x, sin(x))
>>> xlabel('radians')
# Keywords set text properties.
>>> ylabel('amplitude',
...         fontsize='large')
>>> title('Sin(x)')
```



PLOT GRID

```
# Display gridlines in plot
>>> grid()
```

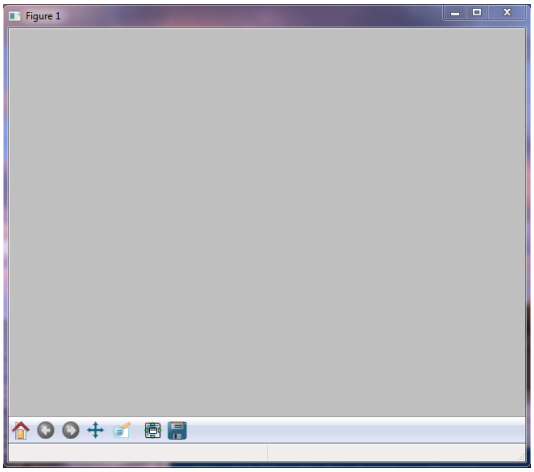


ENTHOUGHT

Clearing and Closing Plots

CLEARING A FIGURE

```
>>> plot(x, sin(x))
# clf will clear the current
# plot (figure).
>>> clf()
```



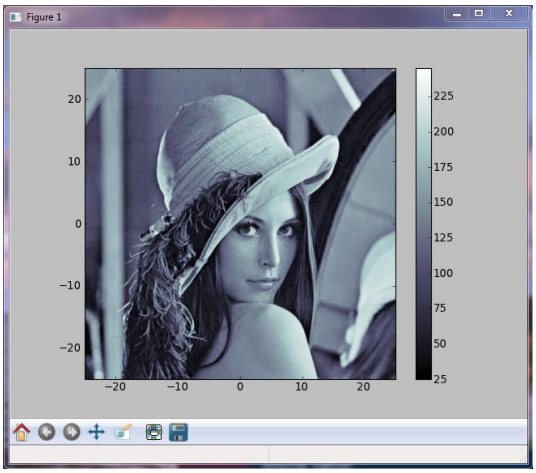
CLOSING PLOT WINDOWS

```
# close() will close the
# currently active plot window.
>>> close()

# close('all') closes all the
# plot windows.
>>> close('all')
```

ENTHOUGHT

Image Display

```
# Get the Lena image from scipy.
>>> from scipy.misc import lena
>>> img = lena()

# Display image with the jet
# colormap, and setting
# x and y extents of the plot.
>>> imshow(img,
...         extent=[-25,25,-25,25],
...         cmap = cm.bone)

# Add a colorbar to the display.
>>> colorbar()
```



ENTHOUGHT

Plotting from Scripts

INTERACTIVE MODE

```
# In IPython, plots show up
# as soon as a plot command
# is called.
>>> figure()
>>> plot(sin(x))
>>> figure()
>>> plot(cos(x))
```

NON-INTERACTIVE MODE

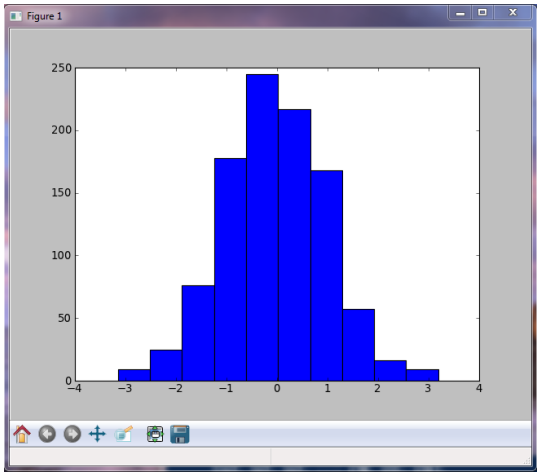
```
# script.py
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.
figure()
plot(sin(x))
figure()
plot(cos(x))

# Plots will not appear until
# this command is issued.
show()
```

ENTHOUGHT

Histograms

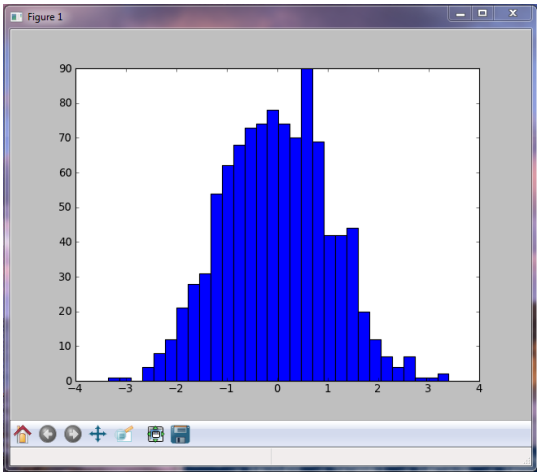
HISTOGRAM

```
# plot histogram
# defaults to 10 bins
>>> hist(randn(1000))
```



HISTOGRAM 2

```
# change the number of bins
>>> hist(randn(1000), 30)
```

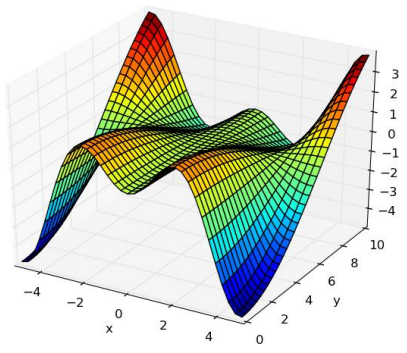


ENTHOUGHT

3D Plots with Matplotlib

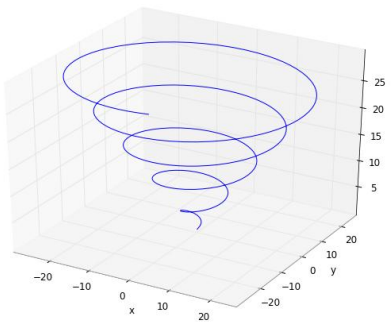
SURFACE PLOT

```
>>> from mpl_toolkits.mplot3d import
Axes3D
>>> x, y = mgrid[-5:5:35j, 0:10:35j]
>>> z = x*sin(x)*cos(0.25*y)
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot_surface(x, y, z,
...     rstride=1, cstride=1,
...     cmap=cm.jet)
>>> xlabel('x'); ylabel('y')
```



PARAMETRIC CURVE

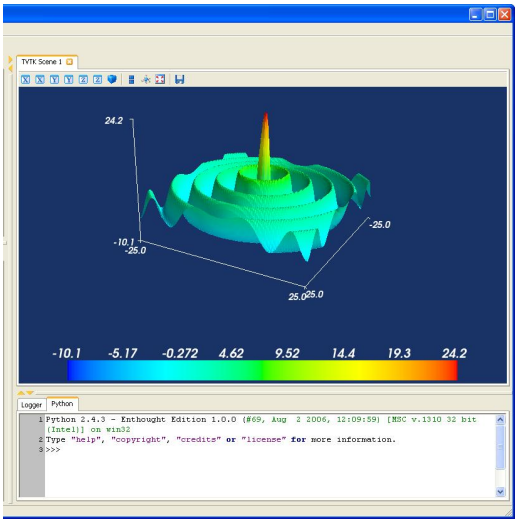
```
>>> from mpl_toolkits.mplot3d import
Axes3D
>>> t = linspace(0, 30, 1000)
>>> x, y, z = [t*cos(t), t*sin(t), t]
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot(x, y, z)
>>> xlabel('x')
>>> ylabel('y')
```



ENTHOUGHT

Surface Plots with `mlab`

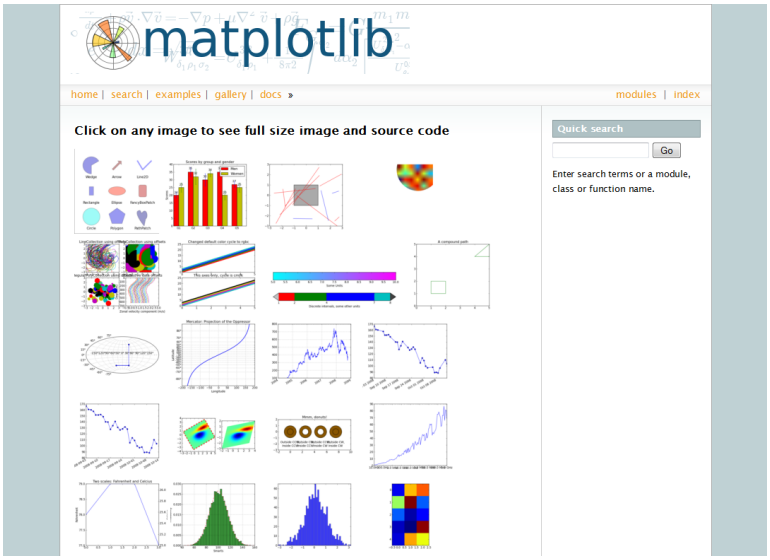
```
# Create 2D array where values
# are radial distance from
# the center of array.
>>> from numpy import mgrid
>>> from scipy import special
>>> x,y = mgrid[-25:25:100j,
...             -25:25:100j]
>>> r = sqrt(x**2+y**2)
# Calculate Bessel function of
# each point in array and scale.
>>> s = special.j0(r)*25

# Display surface plot.
>>> from mayavi import mlab
>>> mlab.surf(x,y,s)
>>> mlab.scalarbar()
>>> mlab.axes()
```



ENTHOUGHT

More Details

- Simple examples with increasing difficulty: http://matplotlib.org/examples/index.html
- Gallery (huge): http://matplotlib.org/gallery.html



ENTHOUGHT

Continuing NumPy…

ENTHOUGHT

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
numpy.ndarray
```

NUMERIC 'TYPE' OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

BYTES PER ELEMENT

```
>>> a.itemsize
4
```

ARRAY SHAPE

```
# Shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

ARRAY SIZE

```
# Size reports the entire
# number of elements in an
# array.
>>> a.size
4
>>> size(a)
4
```

ENTHOUGHT

Introducing NumPy Arrays

BYTES OF MEMORY USED

```
# Return the number of bytes
# used by the data portion of
# the array.
 >>> a.nbytes
16
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```


ENTHOUGHT

Setting Array Elements

ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```

FILL

```
# set all values in an array
>>> a.fill(0)
>>> a
array([0, 0, 0, 0])

# this also works, but may
# be slower
>>> a[:] = 1
>>> a
array([1, 1, 1, 1])
```

BEWARE OF TYPE COERCION

```
>>> a.dtype
dtype('int32')

# assigning a float into
# an int32 array truncates
# the decimal part
>>> a[0] = 10.6
>>> a
array([10, 1, 2, 3])

# fill has the same behavior
>>> a.fill(-4.8)
>>> a
array([-4, -4, -4, -4])
```

ENTHOUGHT

Slicing

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element is **not** included.
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING ARRAYS

```
# indices:    0  1  2  3  4
>>> a = array([10,11,12,13,14])
# [10,11,12,13,14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
array([10, 11, 12])
# grab last two elements
>>> a[-2:]
array([13, 14])
# every other element
>>> a[::2]
array([10, 12, 14])
```


ENTHOUGHT

Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

SHAPE = (ROWS,COLUMNS)

```
>>> a.shape
(2, 4)
```

ELEMENT COUNT

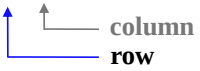
```
>>> a.size
8
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
2
```

GET/SET ELEMENTS

```
>>> a[1,3]
13
```




```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

ADDRESS SECOND (ONETH) ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, -1])
```

ENTHOUGHT

Arrays from/to ASCII files

BASIC PATTERN

```
# Read data into a list of lists,
# and THEN convert to an array.
file = open('myfile.txt')

# Create a list for all the data.
data = []

for line in file:
  # Read each row of data into a
  # list of floats.
  fields = line.split()
  row_data = [float(x) for x
                          in fields]
  # And add this row to the
  # entire data set.
  data.append(row_data)

# Finally, convert the "list of
# lists" into a 2D array.
data = array(data)
file.close()
```

ARRAYS FROM/TO TXT FILES

Data.txt

```
-- BEGINNING OF THE FILE
% Day,  Month,  Year, Skip, Avg Power
01, 01, 2000, x876, 13 % crazy day!
% we don't have Jan 03rd
04, 01, 2000, xfed, 55
```

```
# loadtxt() automatically generates
# an array from the txt file
arr = loadtxt('Data.txt', skiprows=1,
   dtype=int, delimiter=",",
   usecols = (0,1,2,4),
   comments = "%")

# Save an array into a txt file
savetxt('filename', arr)
```


ENTHOUGHT

Arrays to/from Files

OTHER FILE FORMATS

Many file formats are supported in various packages:

File format	Package name(s)	Functions
txt	numpy	loadtxt, savetxt, genfromtxt, fromfile, tofile
csv	csv	reader, writer
Matlab	scipy.io	loadmat, savemat
hdf	pytables, h5py	
NetCDF	netCDF4, scipy.io.netcdf	netCDF4.Dataset, scipy.io.netcdf.netcdf_file

This includes many industry specific formats:

File format	Package name	Comments
wav	scipy.io.wavfile	Audio files
LAS/SEG-Y	Scipy cookbook, EPD	Data files in Geophysics
jpeg, png, …	PIL, scipy.misc.pilutil	Common image formats
fits	pyfits	Image files in Astronomy

ENTHOUGHT

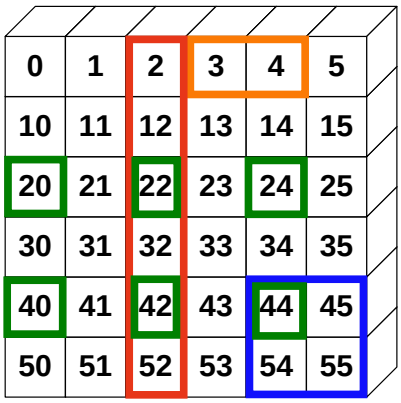
Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



ENTHOUGHT

Slices Are References

Slices are references to memory in the original array.

Changing values in a slice also changes the original array.

```
>>> a = array((0,1,2,3,4))

# create a slice containing only the
# last element of a
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 0,  1, 10, 3, 4])
```

ENTHOUGHT

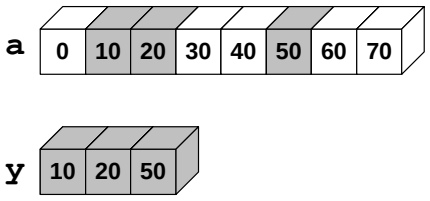
Fancy Indexing

INDEXING BY POSITION

```
>>> a = arange(0,80,10)


# fancy indexing
>>> indices = [1, 2, -3]
>>> y = a[indices]
>>> print y
[10 20 50]
```

INDEXING WITH BOOLEANS

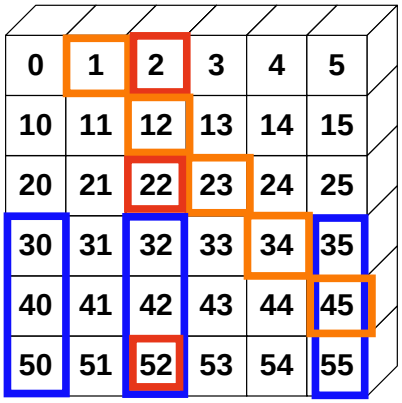
```
# manual creation of masks
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)

# conditional creation of masks
>>> mask2 = a < 30

# fancy indexing
>>> y = a[mask]
>>> print y
[10 20 50]
```



ENTHOUGHT

Fancy Indexing in 2-D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```

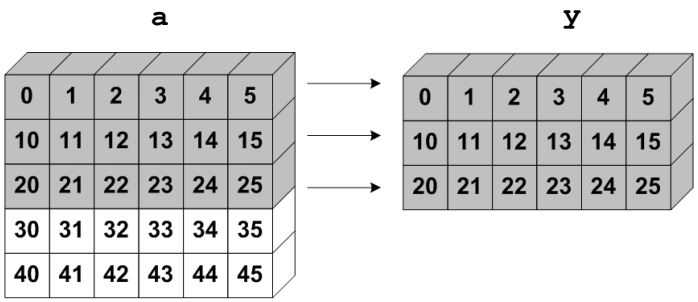


Unlike slicing, fancy indexing creates copies instead of a view into original array.

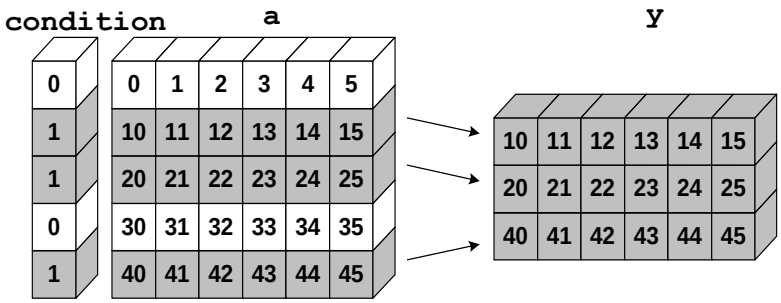
ENTHOUGHT

“Incomplete” Indexing

```
>>> y = a[:3]
```




```
>>> y = a[condition]
```

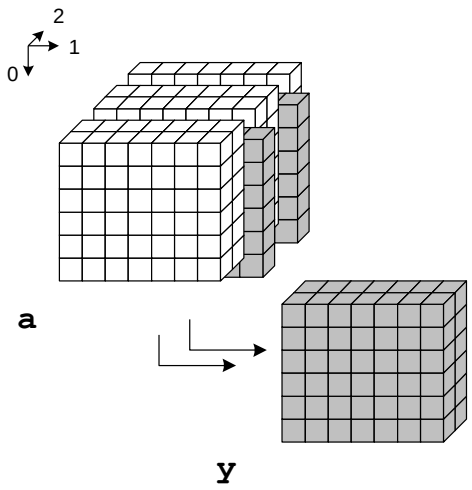


ENTHOUGHT

3D Example

MULTIDIMENSIONAL

```
# retrieve two slices from a
# 3D cube via indexing
>>> y = a[:,:,[2,-2]]
```



ENTHOUGHT

Where

1 DIMENSION

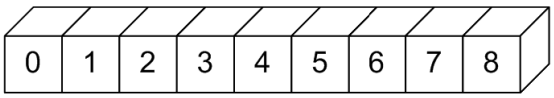
```
# find the indices in array
# where expression is True
>>> a = array([0, 12, 5, 20])
>>> a > 10
array([False, True, False,
       True], dtype=bool)

# Note: it returns a tuple!
>>> where(a > 10)
(array([1, 3]),)
```

n DIMENSIONS

```
# In general, the tuple
# returned is the index of the
# element satisfying the
# condition in each dimension.
>>> a = array([[0, 12, 5, 20],
               [1, 2, 11, 15]])
>>> loc = where(a > 10)
>>> loc
(array([0, 0, 1, 1]),
array([1, 3, 2, 3]))

# Result can be used in
# various ways:
>>> a[loc]
array([12, 20, 11, 15])
```

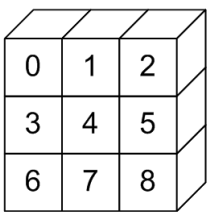

ENTHOUGHT

Array Data Structure



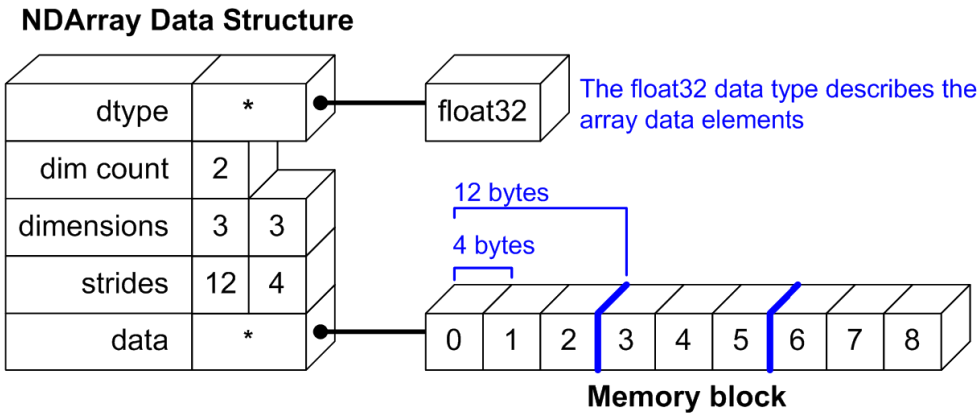
Memory block

Python View:

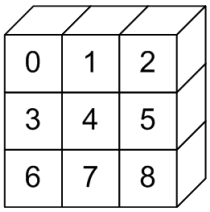


ENTHOUGHT

Array Data Structure



Python View:

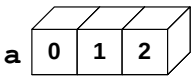


ENTHOUGHT

Indexing with newaxis

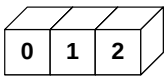
`newaxis` is a special index that inserts a new axis in the array at the specified location.

Each **`newaxis`** increases the array's dimensionality by 1.



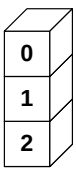
1 X 3

```
>>> shape(a)
(3,)
>>> y = a[newaxis,:]
>>> shape(y)
(1, 3)
```



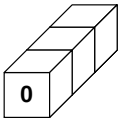
3 X 1

```
>>> y = a[:,newaxis]
>>> shape(y)
(3, 1)
```



1 X 1 X 3

```
> y = a[newaxis, newaxis, :]
> shape(y)
(1, 1, 3)
```



ENTHOUGHT

"Flattening" Arrays

a.flatten()

`a.flatten()` converts a multi-dimensional array into a 1-D array. The new array is a *copy* of the original data.

```
# Create a 2D array
>>> a = array([[0,1],
               [2,3]])

# Flatten out elements to 1D
>>> b = a.flatten()
>>> b
array([0,1,2,3])

# Changing b does not change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a          <-- no change
array([[0, 1],
       [2, 3]])
```

a.flat

`a.flat` is an *attribute* that returns an iterator object that accesses the data in the multi-dimensional array data as a 1-D array. It *references* the original memory.

```
>>> a.flat
<numpy.flatiter obj...>
>>> a.flat[:]
array(0,1,2,3)

>>> b = a.flat
>>> b[0] = 10
>>> a          <-- changed!
array([[10, 1],
       [ 2, 3]])
```


ENTHOUGHT

"(Un)raveling" Arrays

a.ravel()

`a.ravel()` is the same as **`a.flatten()`**, but returns a *reference (or view)* of the array if possible (i.e., the memory is contiguous). Otherwise the new array copies the data.

```
# create a 2-D array
>>> a = array([[0,1],
               [2,3]])

# flatten out elements to 1-D
>>> b = a.ravel()
>>> b
array([0,1,2,3])

# changing b does change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a          <-- changed!
array([[10, 1],
       [ 2, 3]])
```

a.ravel() MAKES A COPY

```
# create a 2-D array
>>> a = array([[0,1],
               [2,3]])

# transpose array so memory
# layout is no longer contiguous
>>> aa = a.transpose()
>>> aa
array([[0, 2],
       [1, 3]])
# ravel creates a copy of data
>>> b = aa.ravel()
array([0,2,1,3])
# changing b doesn't change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[0, 1],
       [2, 3]])
```

ENTHOUGHT

Reshaping Arrays

SHAPE

```
>>> a = arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)


# reshape array in-place to
# 2x3
>>> a.shape = (2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```

RESHAPE

```
# return a new array with a
# different shape
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# reshape cannot change the
# number of elements in an
# array
>>> a.reshape(4,2)
ValueError: total size of new
array must be unchanged
```


ENTHOUGHT

Transpose

TRANSPOSE

```
>>> a = array([[0,1,2],
...            [3,4,5]])
>>> a.shape
(2,3)
# Transpose swaps the order
# of axes. For 2-D this
# swaps rows and columns.
>>> a.transpose()
array([[0, 3],
       [1, 4],
       [2, 5]])

# The .T attribute is
# equivalent to transpose().
>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

TRANSPOSE RETURNS VIEWS

```
>>> b = a.T

# Changes to b alter a.
>>> b[0,1] = 30
>>> a
array([[ 0,  1,  2],
       [30,  4,  5]])
```

TRANSPOSE AND STRIDES

```
# Transpose does not move
# values around in memory. It
# only changes the order of
# "strides" in the array
>>> a.strides
(12, 4)

>>> a.T.strides
(4, 12)
```

ENTHOUGHT

Squeeze

SQUEEZE

```
>>> a = array([[1,2,3],
...            [4,5,6]])
>>> a.shape
(2,3)

# insert an "extra" dimension
>>> a.shape = (2,1,3)
>>> a
array([[[0, 1, 2]],
       [[3, 4, 5]]])

# squeeze removes any
# dimension with length==1
>>> a = a.squeeze()
>>> a.shape
(2,3)
```


ENTHOUGHT

Diagonals

DIAGONAL

```
>>> a = array([[11,21,31],
...            [12,22,32],
...            [13,23,33]])


# Extract the diagonal from
# an array.
>>> a.diagonal()
array([11, 22, 33])

# Use offset to move off the
# main diagonal (offset can
# be negative).
>>> a.diagonal(offset=1)
array([21, 32])
```

DIAGONALS WITH INDEXING

```
# "Fancy" indexing also works.
>>> i = [0,1,2]
>>> a[i, i]
array([11, 22, 33])

# Indexing can also be used
# to set diagonal values…
>>> a[i, i] = 2
>>> i2 = array([0,1])
# upper diagonal
>>> a[i2, i2+1] = 1
# lower diagonal
>>> a[i2+1, i2]= -1
>>> a
array([[ 2,  1, 31],
       [-1,  2,  1],
       [13, -1,  2]])
```

ENTHOUGHT

Complex Numbers

COMPLEX ARRAY ATTRIBUTES

```
>>> a = array([1+1j, 2, 3, 4])
array([1.+1.j, 2.+0.j, 3.+0.j,
       4.+0.j])
>>> a.dtype
dtype('complex128')

# real and imaginary parts
>>> a.real
array([ 1.,  2.,  3.,  4.])
>>> a.imag
array([ 1.,  0.,  0.,  0.])

# set imaginary part to a
# different set of values
>>> a.imag = (1,2,3,4)
>>> a
array([1.+1.j, 2.+2.j, 3.+3.j,
       4.+4.j])
```

CONJUGATION

```
>>> a.conj()
array([1.-1.j, 2.-2.j, 3.-3.j,
       4.-4.j])
```

FLOAT (AND OTHER) ARRAYS

```
>>> a = array([0., 1, 2, 3])

# .real and .imag attributes
# are available
>>> a.real
array([ 0., 1., 2., 3.])
>>> a.imag
array([ 0., 0., 0., 0.])

# but .imag is read-only
>>> a.imag = (1,2,3,4)
TypeError: array does not
have imaginary part to set
```


ENTHOUGHT

Array Constructor Examples

FLOATING POINT ARRAYS

```
# Default to double precision
>>> a = array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

REDUCING PRECISION

```
>>> a = array([0,1.,2,3],
...           dtype=float32)
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

UNSIGNED INTEGER BYTE

```
>>> a = array([0,1,2,3],
...           dtype=uint8)
>>> a.dtype
dtype('uint8')
>>> a.nbytes
4
```

ARRAY FROM BINARY DATA

```
# frombuffer or fromfile
# to create an array from
# binary data.
>>> a = frombuffer('foo',
...           dtype=uint8)
>>> a
array([102, 111, 111])
# Reverse operation
>>> a.tofile('foo.dat')
```

ENTHOUGHT

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	`bool`	Elements are 1 byte in size.
Integer	`int8, int16, int32, int64, int128, int`	int defaults to the size of long in C for the platform.
Unsigned Integer	`uint8, uint16, uint32, uint64, uint128, uint`	uint defaults to the size of unsigned long in C for the platform.
Float	`float16, float32, float64, float,longfloat,`	float is always a double precision floating point value (64 bits). longfloat represents large precision floats. Its size is platform dependent.
Complex	`complex64, complex128, complex, longcomplex`	The real and imaginary elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	`str, unicode`	For example, dtype='S4' would be used for an array of 4-character strings.
Object	`object`	Represent items in array as Python objects.
Records	`void`	Used for arbitrary data structures.

ENTHOUGHT

Type Casting

ASARRAY

```
>>> a = array([1.5, -3],
...           dtype=float32)
>>> a
array([ 1.5, -3.], dtype=float32)

# upcast
>>> asarray(a, dtype=float64)
array([ 1.5, -3. ])

# downcast
>>> asarray(a, dtype=uint8)
array([ 1, 253], dtype=uint8)

# asarray is efficient.
# It does not make a copy if the
# type is the same.
>>> b = asarray(a, dtype=float32)
>>> b[0] = 2.0
>>> a
array([ 2., -3.],dtype=float32)
```

ASTYPE

```
>>> a = array([1.5, -3],
...           dtype=float64)
>>> a.astype(float32)
array([ 1.5, -3.], dtype=float32)

>>> a.astype(uint8)
array([  1, 253],dtype=uint8)

# astype is safe.
# It always returns a copy of
# the array.
>>> b = a.astype(float64)
>>> b[0] = 2.0
>>> a
array([1.5, -3.])
```

ENTHOUGHT

Array Calculation Methods

SUM FUNCTION

```
>>> a = array([[1,2,3],
               [4,5,6]])

# sum() defaults to adding up
# all the values in an array.
>>> sum(a)
21

# supply the keyword axis to
# sum along the 0th axis
>>> sum(a, axis=0)
array([5, 7, 9])

# supply the keyword axis to
# sum along the last axis
>>> sum(a, axis=-1)
array([ 6, 15])
```

SUM ARRAY METHOD

```
# a.sum() defaults to adding
# up all values in an array.
>>> a.sum()
21

# supply an axis argument to
# sum along a specific axis
>>> a.sum(axis=0)
array([5, 7, 9])
```

PRODUCT

```
# product along columns
>>> a.prod(axis=0)
array([ 4, 10, 18])

# functional form
>>> prod(a, axis=0)
array([ 4, 10, 18])
```


ENTHOUGHT

Min/Max

MIN

```
>>> a = array([2.,3.,0.,1.])
>>> a.min(axis=0)
0.0
# Use NumPy's amin() instead
# of Python's built-in min()
# for speedy operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.0
```

MAX

```
>>> a = array([2.,3.,0.,1.])
>>> a.max(axis=0)
3.0

# functional form
>>> amax(a, axis=0)
3.0
```

ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# functional form
>>> argmin(a, axis=0)
2
```

ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# functional form
>>> argmax(a, axis=0)
1
```

ENTHOUGHT

Statistics Array Methods

MEAN

```
>>> a = array([[1,2,3],
               [4,5,6]])

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
>>> average(a, axis=0)
array([ 2.5,  3.5,  4.5])

# average can also calculate
# a weighted average
>>> average(a, weights=[1,2],
...         axis=0)
array([ 3., 4., 5.])
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

# variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])
>>> var(a, axis=0)
array([2.25, 2.25, 2.25])
```


ENTHOUGHT

Other Array Methods

CLIP

```
# Limit values to a range.

>>> a = array([[1,2,3],
               [4,5,6]])

# Set values < 3 equal to 3.
# Set values > 5 equal to 5.
>>> a.clip(3, 5)
array([[3, 3, 3],
       [4, 5, 5]])
```

PEAK TO PEAK

```
# Calculate max - min for
# array along columns
>>> a.ptp(axis=0)
array([3, 3, 3])
# max - min for entire array.
>>> a.ptp(axis=None)
5
```

ROUND

```
# Round values in an array.
# NumPy rounds to even, so
# 1.5 and 2.5 both round to 2.
>>> a = array([1.35, 2.5, 1.5])
>>> a.round()
array([ 1.,  2.,  2.])

# Round to first decimal place.
>>> a.round(decimals=1)
array([ 1.4,  2.5,  1.5])
```

Summary of (most) array attributes/methods (1/4)

ENTHOUGHT

	BASIC ATTRIBUTES
`a.dtype`	Numerical type of array elements: float 32, uint8, etc.
`a.shape`	Shape of array (m, n, o, …)
`a.size`	Number of elements in entire array
`a.itemsize`	Number of bytes used by a single element in the array
`a.nbytes`	Number of bytes used by entire array (data only)
`a.ndim`	Number of dimensions in the array
	SHAPE OPERATIONS
`a.flat`	An iterator to step through array as if it were 1D
`a.flatten()`	Returns a 1D copy of a multi-dimensional array
`a.ravel()`	Same as flatten(), but returns a "view" if possible
`a.resize(new_size)`	Changes the size/shape of an array in place
`a.swapaxes(axis1, axis2)`	Swaps the order of two axes in an array
`a.transpose(*axes)`	Swaps the order of any number of array axes
`a.T`	Shorthand for a.transpose()
`a.squeeze()`	Removes any length==1 dimensions from an array

Summary of (most) array attributes/methods (2/4)

ENTHOUGHT

	FILL AND COPY
`a.copy()`	Returns a copy of the array
`a.fill(value)`	Fills an array with a scalar value
	CONVERSION/COERCION
`a.tolist()`	Converts array into nested lists of values
`a.tostring()`	Raw copy of array memory into a Python string
`a.astype(dtype)`	Returns array coerced to the given type
`a.byteswap(False)`	Converts byte order (big <->little endian)
`a.view(type_or_dtype)`	Creates a new ndarray that sees the same memory but interprets it as a new datatype (or subclass of ndarray)
	COMPLEX NUMBERS
`a.real`	Returns the real part of the array
`a.imag`	Returns the imaginary part of the array
`a.conjugate()`	Returns the complex conjugate of the array
`a.conj()`	Returns the complex conjugate of the array (same as conjugate)

Summary of (most) array attributes/methods (3/4)

ENTHOUGHT

	SAVING
`a.dump(file)`	Stores binary array data to *file*
`a.dumps()`	Returns a binary pickle of the data as a string
`a.tofile(fid, sep="", format="%s")`	Formatted ASCII output to a file
	SEARCH/SORT
`a.nonzero()`	Returns indices for all non-zero elements in the array
`a.sort(axis=-1)`	Sort the array elements in place, along *axis*
`a.argsort(axis=-1)`	Finds indices for sorted elements, along *axis*
`a.searchsorted(b)`	Finds indices where elements of *b* would be inserted in *a* to maintain order
	ELEMENT MATH OPERATIONS
`a.clip(low, high)`	Limits values in the array to the specified range
`a.round(decimals=0)`	Rounds to the specified number of digits
`a.cumsum(axis=None)`	Cumulative sum of elements along *axis*
`a.cumprod(axis=None)`	Cumulative product of elements along *axis*

Summary of (most) array attributes/methods (4/4)

ENTHOUGHT

REDUCTION METHODS

All the following methods "reduce" the size of the array by 1 dimension by carrying out an operation along the specified axis. If axis is None, the operation is carried out across the entire array.

`a.sum(axis=None)`	Sums values along axis
`a.prod(axis=None)`	Finds the product of all values along axis
`a.min(axis=None)`	Finds the minimum value along axis
`a.max(axis=None)`	Finds the maximum value along axis
`a.argmin(axis=None)`	Finds the index of the minimum value along axis
`a.argmax(axis=None)`	Finds the index of the maximum value along axis
`a.ptp(axis=None)`	Calculates a.max(axis) – a.min(axis)
`a.mean(axis=None)`	Finds the mean (average) value along axis
`a.std(axis=None)`	Finds the standard deviation along axis
`a.var(axis=None)`	Finds the variance along axis
`a.any(axis=None)`	True if any value along axis is non-zero (logical OR)
`a.all(axis=None)`	True if all values along axis are non-zero (logical AND)

ENTHOUGHT

Array Creation Functions

ARANGE

arange(start,stop=None,step=1, dtype=None)

Nearly identical to Python's **range()**. Creates an array of values in the range [start,stop) with the specified step value. Allows non-integer values for start, stop, and step. Default **dtype** is derived from the start, stop, and step values.

```
>>> arange(4)
array([0, 1, 2, 3])
>>> arange(0, 2*pi, pi/4)
array([ 0.000, 0.785, 1.571,
2.356, 3.142, 3.927, 4.712,
5.497])

# Be careful…
>>> arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

ONES, ZEROS

ones(shape, dtype=float64)
zeros(shape, dtype=float64)

shape is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to **float64.**

```
>>> ones((2,3),dtype=float32)
array([[ 1.,  1.,  1.],
    [ 1.,  1.,  1.]],
       dtype=float32)
>>> zeros(3)
array([ 0., 0., 0.])
```


ENTHOUGHT

Array Creation Functions (cont.)

IDENTITY

```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> identity(4, dtype=int)
array([[ 1, 0, 0, 0],
       [ 0, 1, 0, 0],
       [ 0, 0, 1, 0],
       [ 0, 0, 0, 1]])
```

EMPTY AND FILL

```
# empty(shape, dtype=float64,
#       order='C')
>>> a = empty(2)
>>> a
array([1.78021120e-306,
 6.95357225e-308])

# fill array with 5.0
>>> a.fill(5.0)
array([5., 5.])

# alternative approach
# (slightly slower)
>>> a[:] = 4.0
array([4., 4.])
```

Array Creation Functions (cont.)

LINSPACE

```
# Generate N evenly spaced
# elements between (and
# including) start and
# stop values.
>>> linspace(0,1,5)
array([0.,0.25.,0.5,0.75, 1.0])
```

LOGSPACE

```
# Generate N evenly spaced
# elements on a log scale
# between base**start and
# base**stop (default base=10).
>>> logspace(0,1,5)
array([ 1., 1.77, 3.16,  5.62,
        10.])
```

ROW SHORTCUT

```
# r_ and c_ are "handy" tools
# (cough hacks…) for creating
# row and column arrays.

# used like arange
# -- real stride value
>>> r_[0:1:.25]
array([ 0., 0.25., 0.5, 0.75])

# used like linspace
# -- complex stride value
>>> r_[0:1:5j]
array([0.,0.25.,0.5,0.75,1.0])

# concatenate elements
>>> r_[(1,2,3),0,0,(4,5)]
array([1, 2, 3, 0, 0, 4, 5])
```


Array Creation Functions (cont.)

MGRID

```
# Get equally spaced points
# in N output arrays for an
# N-dimensional (mesh) grid.

>>> x,y = mgrid[0:5,0:5]
>>> x
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
>>> y
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

OGRID

```
# Construct an "open" grid
# of points (not filled in
# but correctly shaped for
# math operations to be
# broadcast correctly).

>>> x,y = ogrid[0:3,0:3]
>>> x
array([[0],
       [1],
       [2]])
>>> y
array([[0, 1, 2]])

>>> print x+y
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

ENTHOUGHT

Matrix Objects

MATRIX CREATION

```
# Matlab-like creation from string
>>> A = mat('1,2,4;2,5,3;7,8,9')
>>> print A
Matrix([[1, 2, 4],
        [2, 5, 3],
        [7, 8, 9]])

# matrix exponents
>>> print A**4
Matrix([[ 6497,  9580,  9836],
        [ 7138, 10561, 10818],
        [18434, 27220, 27945]])

# matrix multiplication
>>> print A*A.I
Matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

BLOCK MATRICES

```
# create a matrix from
# sub-matrices
>>> a = array([[1,2],
               [3,4]])
>>> b = array([[10,20],
               [30,40]])

>>> bmat('a,b;b,a')
matrix([[ 1,  2, 10, 20],
        [ 3,  4, 30, 40],
        [10, 20,  1,  2],
        [30, 40,  3,  4]])
```


ENTHOUGHT

Trig and Other Functions

TRIGONOMETRIC

`sin(x)`	`sinh(x)`
`cos(x)`	`cosh(x)`
`arccos(x)`	`arccosh(x)`
`arctan(x)`	`arctanh(x)`
`arcsin(x)`	`arcsinh(x)`
`arctan2(x,y)`	

OTHERS

`exp(x)`	`log(x)`
`log10(x)`	`sqrt(x)`
`absolute(x)`	`conjugate(x)`
`negative(x)`	`ceil(x)`
`floor(x)`	`fabs(x)`
`hypot(x,y)`	`fmod(x,y)`
`maximum(x,y)`	`minimum(x,y)`

VECTOR OPERATIONS

`dot(x,y)`	`vdot(x,y)`
`inner(x,y)`	`outer(x,y)`
`cross(x,y)`	`kron(x,y)`
`tensordot(x,y[,axis])`	

`hypot(x,y)`

Element by element distance calculation using $\sqrt{x^2 + y^2}$

ENTHOUGHT

More Basic Functions

TYPE HANDLING

iscomplexobj	real_if_close	isnan
iscomplex	isscalar	nan_to_num
isrealobj	isneginf	common_type
isreal	isposinf	typename
imag	isinf	
real	isfinite	

SHAPE MANIPULATION

atleast_1d	hstack	hsplit
atleast_2d	vstack	vsplit
atleast_3d	dstack	dsplit
expand_dims	column_stack	split
apply_over_axes		squeeze
apply_along_axis		

OTHER USEFUL FUNCTIONS

fix	unwrap	roots
mod	sort_complex	poly
amax	trim_zeros	any
amin	fliplr	all
ptp	flipud	disp
sum	rot90	unique
cumsum	eye	nansum
prod	diag	nanmax
cumprod	select	nanargmax
diff	extract	nanargmin
angle	insert	nanmin

ENTHOUGHT

Vectorizing Functions

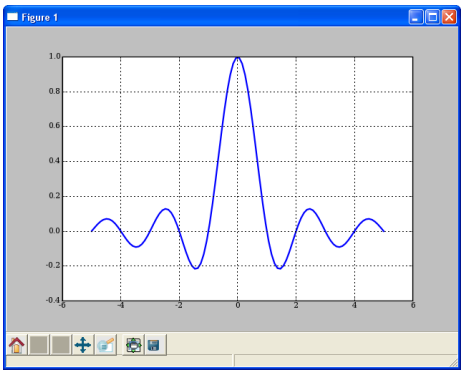
SCALAR SINC FUNCTION

```
# special.sinc already available
# This is just for show.
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```
# attempt
>>> x = array((1.3, 1.5))
>>> sinc(x)
ValueError: The truth value of
an array with more than one
element is ambiguous. Use
a.any() or a.all()
```

SOLUTION

```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc(x)
array([-0.1981, -0.2122])

>>> x2 = linspace(-5, 5, 101)
>>> plot(x2, vsinc(x2))
```



ENTHOUGHT

Mathematical Binary Operators

```
a + b → add(a,b)
a - b → subtract(a,b)
a % b → remainder(a,b)
```

```
a * b  → multiply(a,b)
a / b  → divide(a,b)
a ** b → power(a,b)
```

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN-PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead.
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```


ENTHOUGHT

Comparison and Logical Operators

```
equal         (==)
greater_equal (>=)
logical_and
logical_not
```

```
not_equal   (!=)
less        (<)
logical_or
```

```
greater     (>)
less_equal  (<=)
logical_xor
```

2-D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])

# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```

Be careful with if statements involving numpy arrays. To test for equality of arrays, don't do:

```
if a == b:
```

Rather, do:

```
if all(a==b):
```

For floating point,

```
if allclose(a,b):
```

is even better.

ENTHOUGHT

Bitwise Operators

bitwise_and (&)	invert (~)	right_shift (>>)
bitwise_or (|)	bitwise_xor (^)	left_shift (<<)

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_or(a,b)
array([ 17,  34,  68, 136])

# bit inversion
>>> a = array((1,2,3,4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)

# left shift operation
>>> left_shift(a,3)
array([ 8, 16, 24, 32], dtype=uint8)
```

When possible, operation made bitwise are another way to **speed up** computations.

ENTHOUGHT

Bitwise and Comparison Together

PRECEDENCE ISSUES

```
# When combining comparisons with bitwise operations,
# precedence requires parentheses around the comparisons.
>>> a = array([1,2,4,8])
>>> b = array([16,32,64,128])
>>> (a > 3) & (b < 100)
array([ False, False, True, False])
```

LOGICAL AND ISSUES

```
# Note that logical AND isn't supported for arrays without
# calling the logical_and function.
>>> a>3 and b<100
Traceback (most recent call last):
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()

# Also, you cannot currently use the "short version" of
# comparison with NumPy arrays.
>>> 2<a<4
Traceback (most recent call last):
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

ENTHOUGHT

Universal Function Methods

The mathematical, comparative, logical, and bitwise operators *op* that take two arguments (binary operators) have special methods that operate on arrays:

```
op.reduce(a,axis=0)
op.accumulate(a,axis=0)
op.outer(a,b)
op.reduceat(a,indices)
```


ENTHOUGHT

op.reduce()

`op.reduce(a)` applies `op` to all the elements in a 1-D array `a` reducing it to a single value.

For example:

$$y = \texttt{add.reduce(a)} = \sum_{n=0}^{N-1} a[n] = a[0] + a[1] + \ldots + a[N-1]$$

ADD EXAMPLE

```
>>> a = array([1,2,3,4])
>>> add.reduce(a)
10
```

STRING LIST EXAMPLE

```
>>> a = array(['ab','cd','ef'],
...           dtype=object)
>>> add.reduce(a)
'abcdef'
```

LOGICAL OP EXAMPLES

```
>>> a = array([1,1,0,1])
>>> logical_and.reduce(a)
False
>>> logical_or.reduce(a)
True
```

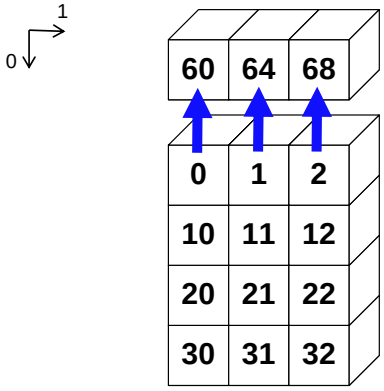
ENTHOUGHT

op.reduce()

For multidimensional arrays, **op.reduce(a,axis)** applies **op** to the elements of **a** along the specified **axis**. The resulting array has dimensionality one less than a. The default value for **axis** is 0.

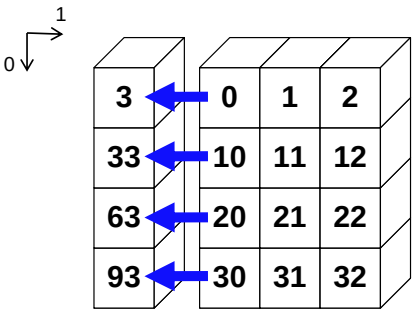
SUM COLUMNS BY DEFAULT

```
>>> add.reduce(a)
array([60, 64, 68])
```



SUMMING UP EACH ROW

```
>>> add.reduce(a,1)
array([ 3, 33, 63, 93])
```



ENTHOUGHT

op.accumulate()

op.accumulate(a) creates a new array containing the intermediate results of the **reduce** operation at each element in **a**.

For example:

$$y = \text{add.accumulate(a)} = \left[\sum_{n=0}^{0} a[n], \sum_{n=0}^{1} a[n], \ldots, \sum_{n=0}^{N-1} a[n]\right]$$

ADD EXAMPLE

```
>>> a = array([1,2,3,4])
>>> add.accumulate(a)
array([ 1,  3,  6, 10])
```

STRING LIST EXAMPLE

```
>>> a = array(['ab','cd','ef'],
...           dtype=object)
>>> add.accumulate(a)
array([ab,abcd,abcdef],
      dtype=object)
```

LOGICAL OP EXAMPLES

```
>>> a = array([1,1,0])
>>> logical_and.accumulate(a)
array([True, True, False])
>>> logical_or.accumulate(a)
array([True, True, True])
```

op.reduceat()

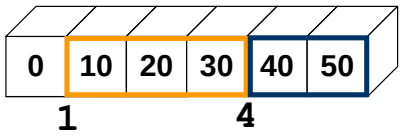
op.reduceat(a,indices)
applies `op` to ranges in the 1-D array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

For example:
`y = add.reduceat(a, indices)`

$$y[i] = \sum_{n=indices[i]}^{indices[i+1]} a[n]$$

EXAMPLE

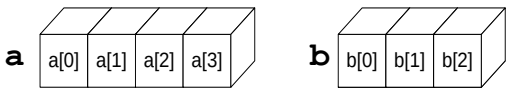
```
>>> a = array([0,10,20,30,
...            40,50])
>>> indices = array([1,4])
>>> add.reduceat(a,indices)
array([60, 90])
```



For multidimensional arrays, `reduceat()` is always applied along the *last* axis (sum of rows for 2-D arrays). This is different from the default for `reduce()` and `accumulate()`.

op.outer()

op.outer(a,b) forms all possible combinations of elements between `a` and `b` using `op`. The shape of the resulting array results from concatenating the shapes of `a` and `b`. (Order matters.)




```
>>> add.outer(a,b)
```

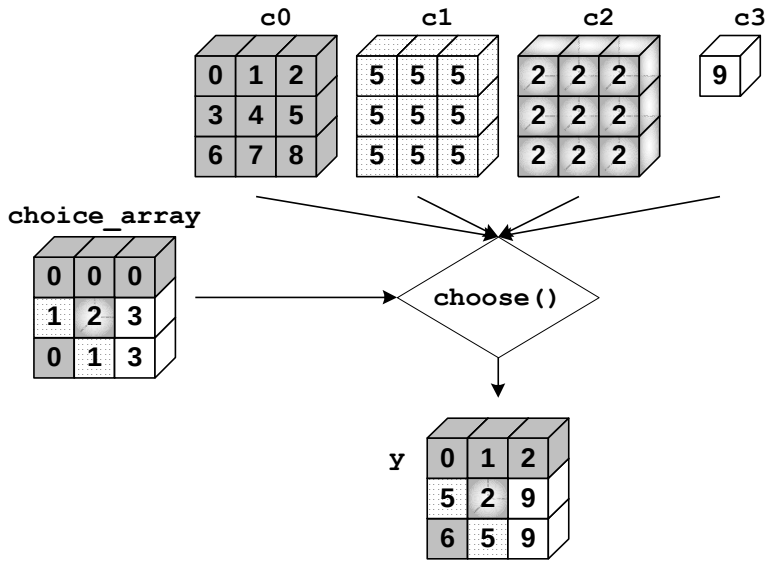
a[0]+b[0]	a[0]+b[1]	a[0]+b[2]
a[1]+b[0]	a[1]+b[1]	a[1]+b[2]
a[2]+b[0]	a[2]+b[1]	a[2]+b[2]
a[3]+b[0]	a[3]+b[1]	a[3]+b[2]

```
>>> add.outer(b,a)
```

b[0]+a[0]	b[0]+a[1]	b[0]+a[2]	b[0]+a[3]
b[1]+a[0]	b[1]+a[1]	b[1]+a[2]	b[1]+a[3]
b[2]+a[0]	b[2]+a[1]	b[2]+a[2]	b[2]+a[3]

Array Functions – `choose()`

```
>>> y = choose(choice_array,(c0,c1,c2,c3))
```



Example - `choose()`

CLIP LOWER VALUES TO 10

```
>>> a
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])

>>> a < 10
array([[True, True, True],
       [False, False, False],
       [False, False, False],
       dtype=bool)

>>> choose(a<10,(a,10))
array([[10, 10, 10],
       [10, 11, 12],
       [20, 21, 22]])
```

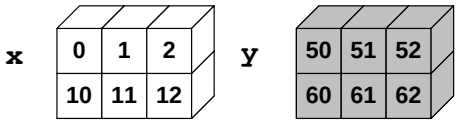
CLIP LOWER AND UPPER VALUES

```
>>> lt = a < 10
>>> gt = a > 15
>>> choice = lt + 2 * gt
>>> choice
array([[1, 1, 1],
       [0, 0, 0],
       [2, 2, 2]])
>>> choose(choice,(a,10,15))
array([[10, 10, 10],
       [10, 11, 12],
       [15, 15, 15]])
```

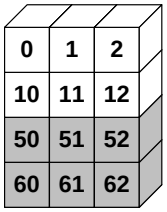
ENTHOUGHT

Array Functions – `concatenate()`

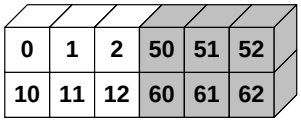
`concatenate((a0,a1,…,aN),axis=0)`
The input arrays **`(a0,a1,…,aN)`** are concatenated along the given **`axis`**. They must have the same shape along every axis *except* the one given.



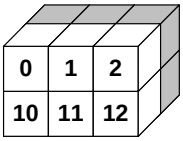

```
>>> concatenate((x,y))
```




```
>>> concatenate((x,y),1)
```




```
>>> array((x,y))
```



See also vstack(), hstack() and dstack() respectively.

ENTHOUGHT

Array Broadcasting

NumPy arrays of different dimensionality can be combined in the same expression. Arrays with smaller dimension are ***broadcasted*** to match the larger arrays, *without copying data*. Broadcasting has **two rules**.

RULE 1: PREPEND ONES TO SMALLER ARRAYS' SHAPE

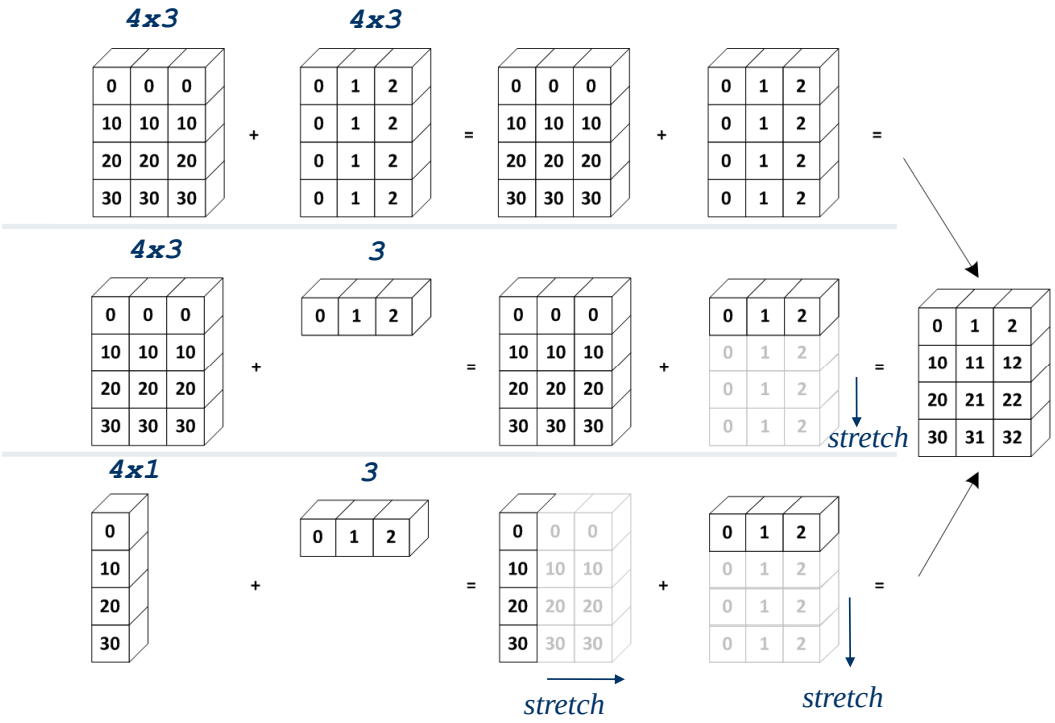
```
In [3]: a = ones((3, 5)) # a.shape == (3, 5)
In [4]: b = ones((5,)) # b.shape == (5,)
In [5]: b.reshape(1, 5) # result is a (1,5)-shaped array.
In [6]: b[newaxis, :] # equivalent, more concise.
```

RULE 2: DIMENSIONS OF SIZE 1 ARE REPEATED WITHOUT COPYING

```
In [7]: c = a + b # c.shape == (3, 5)
   is logically equivalent to...
In [8]: tmp_b = b.reshape(1, 5)
In [9]: tmp_b_repeat = tmp_b.repeat(3, axis=0)
In [10]: c = a + tmp_b_repeat
# But broadcasting makes no copies of "b"s data!
```

ENTHOUGHT

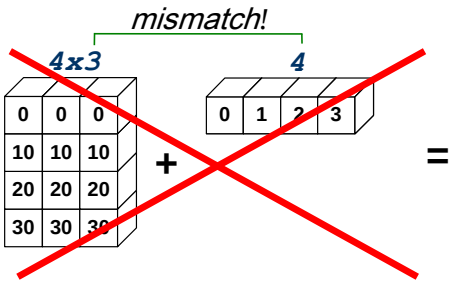
Array Broadcasting



ENTHOUGHT

Broadcasting Rules

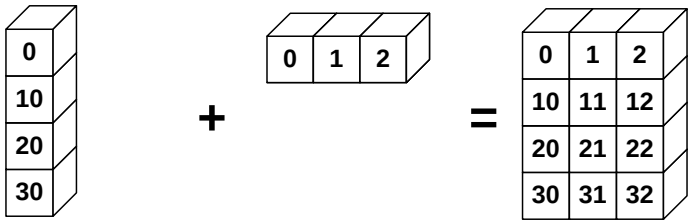
The *trailing* axes of either arrays must be 1 or both must have the same size for broadcasting to occur. Otherwise, a `"ValueError: shape mismatch: objects cannot be broadcast to a single shape"` exception is thrown.



ENTHOUGHT

Broadcasting in Action

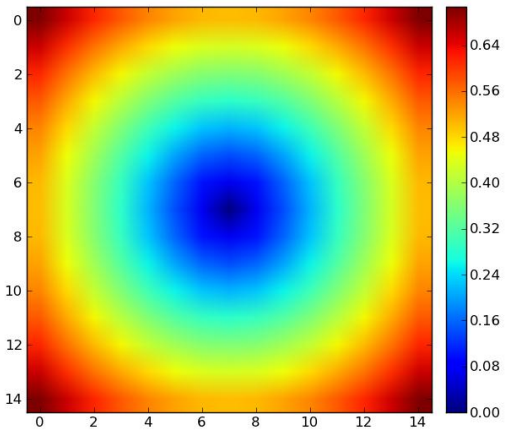
```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, newaxis] + b
```



ENTHOUGHT

Application: distance from center

```
In [1]: a = linspace(0, 1, 15) - 0.5
In [2]: b = a[:, newaxis] # b.shape == (15, 1)
In [3]: dist2 = a**2 + b**2 # broadcasting sum.
In [4]: dist = sqrt(dist2)
In [5]: imshow(dist); colorbar()
```



ENTHOUGHT

Broadcasting's usefulness

Broadcasting can often be used to replace needless data replication inside a NumPy array expression.

`np.meshgrid()` – use **`newaxis`** appropriately in broadcasting expressions.

`np.repeat()` – broadcasting makes repeating an array along a dimension of size 1 unnecessary.

MESHGRID: COPIES DATA

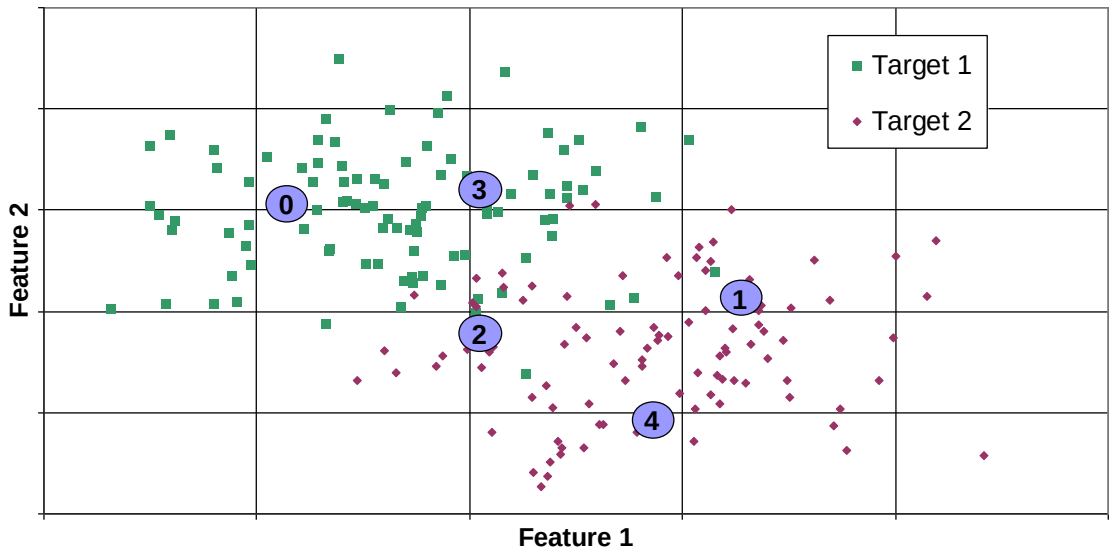
```
In [3]: x, y = \
   meshgrid([1,2],[3,4,5])
In [4]: z = x + y
```

BROADCASTING: NO COPIES

```
In [5]: x = array([1,2])
In [6]: y = array([3,4,5])
In [7]: z = \
   x[newaxis,:] + y[:,newaxis]
```

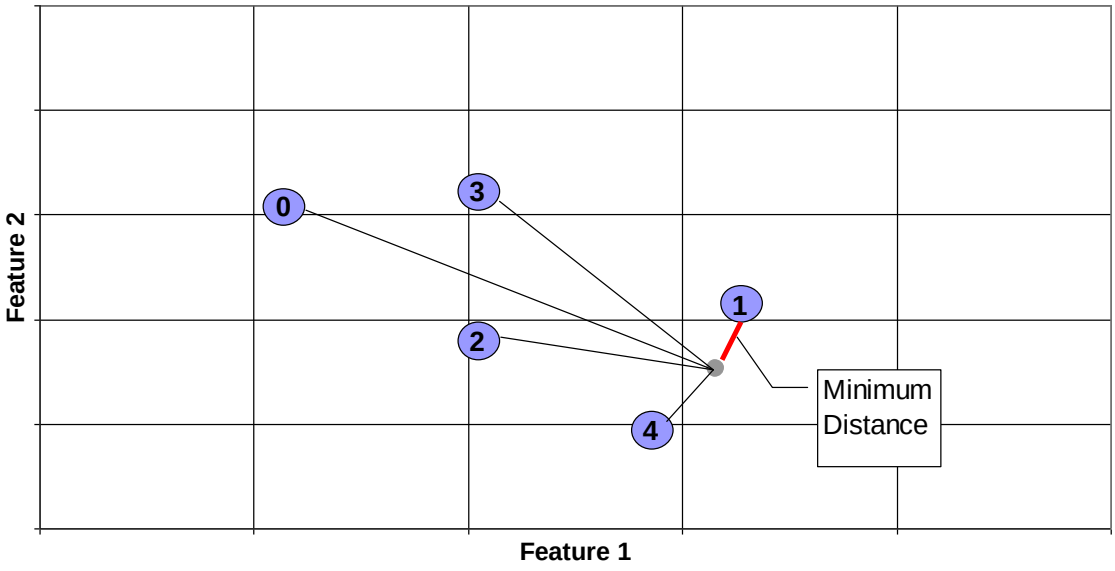

ENTHOUGHT

Vector Quantization Example



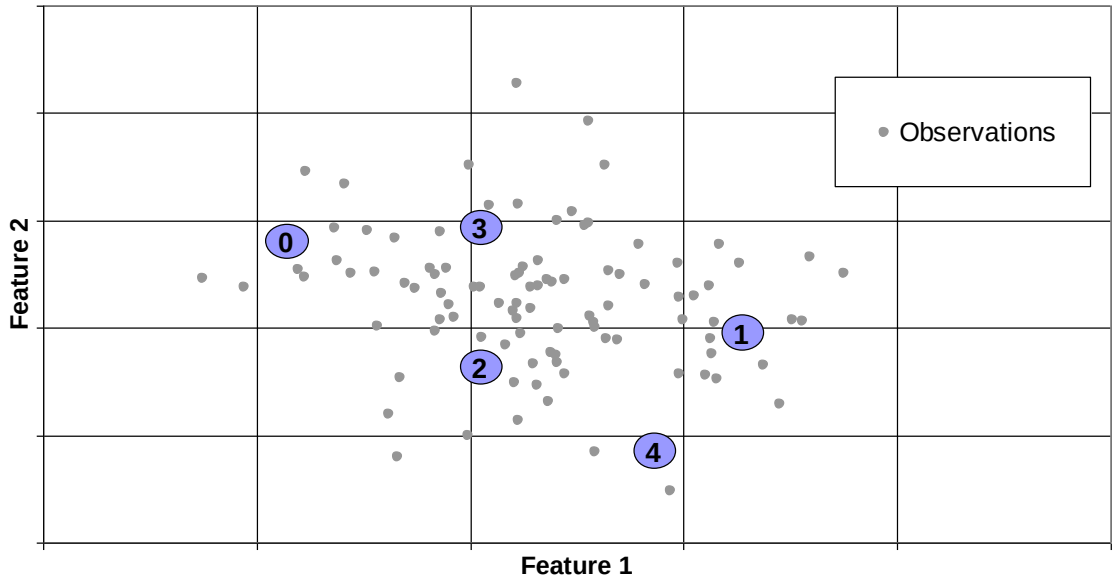
ENTHOUGHT

Vector Quantization Example



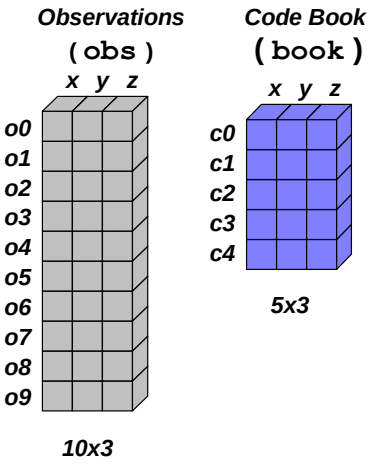
ENTHOUGHT

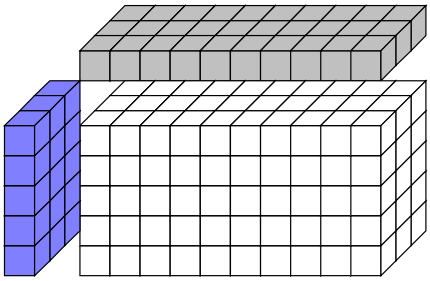
Vector Quantization Example



ENTHOUGHT

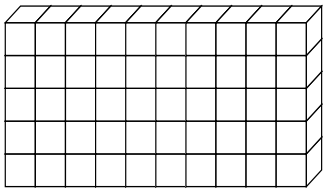
Vector Quantization Example



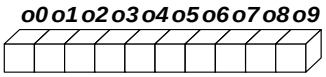


```
diff = obs[newaxis,:,:] - book[:,newaxis,:]
```

5x10x3 1x10x3 5x1x3



```
distance = sqrt(sum(diff**2,axis=-1))
```




```
code_index = argmin(distance,axis=0))
```


ENTHOUGHT

VQ Speed Comparisons

Method	Run Time (sec)	Speed Up
Matlab 5.3	1.611	-
Python VQ1, double	2.245	0.71
Python VQ1, float	1.138	1.42
Python VQ2, double	1.637	0.98
Python VQ2, float	0.954	1.69
C, double	0.066	24.40
C, float	0.064	24.40

- 4000 observations with 16 features categorized into 40 codes on Pentium III 500 MHz.
- VQ1 uses the technique described on the previous slide verbatim.
- VQ2 applies broadcasting on an observation by observation basis. This turned out to be much more efficient because it is less memory intensive.

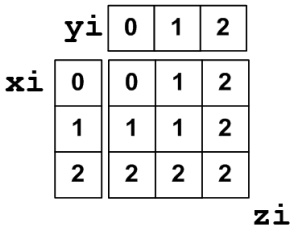
ENTHOUGHT

Broadcasting Indices

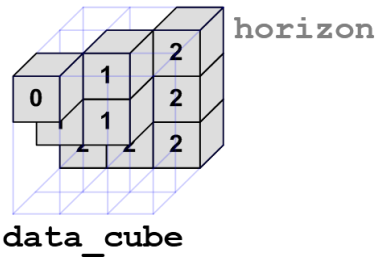
Broadcasting can also be used to slice elements from different "depths" in a 3-D (or any other shape) array. This is a *very* powerful feature of indexing.

```
>>> xi,yi = ogrid[:3,:3]
>>> zi = array([[0, 1, 2],
                [1, 1, 2],
                [2, 2, 2]])
>>> horizon = data_cube[xi,yi,zi]
```

Indices



Selected Data



ENTHOUGHT

"Structured" Arrays

```
# "Data structure" (dtype) that describes the fields and
# type of the items in each array element.
>>> particle_dtype = dtype([('mass','float32'), ('velocity', 'float32')])
# This must be a list of tuples.
>>> particles = array([(1,1), (1,2), (2,1), (1,3)],
                      dtype=particle_dtype)
>>> print particles
[(1.0, 1.0) (1.0, 2.0) (2.0, 1.0) (1.0, 3.0)]
# Retrieve the mass for all particles through indexing.
>>> print particles['mass']
[ 1.  1.  2.  1.]
# Retrieve particle 0 through indexing.
>>> particles[0]
(1.0, 1.0)
# Sort particles in place, with velocity as the primary field and
# mass as the secondary field.
>>> particles.sort(order=('velocity','mass'))
>>> print particles
[(1.0, 1.0) (2.0, 1.0) (1.0, 2.0) (1.0, 3.0)]

# See demo/multitype_array/particle.py.
```

ENTHOUGHT

“Structured” Arrays

Elements of an array can be any fixed-size data structure!

```
name char[10]
age  int
weight double
```

Brad	Jane	John	Fred
33	25	47	54
135.0	105.0	225.0	140.0
Henry	George	Brian	Amy
29	61	32	27
154.0	202.0	137.0	187.0
Ron	Susan	Jennifer	Jill
19	33	18	54
188.0	135.0	88.0	145.0

EXAMPLE

```
>>> from numpy import dtype, empty
# structured data format
>>> fmt = dtype([('name', 'S10'),
                 ('age', int),
                 ('weight', float)
                ])
>>> a = empty((3,4), dtype=fmt)
>>> a.itemsize
22
>>> a['name'] = [['Brad', … ,'Jill']]
>>> a['age'] = [[33, … , 54]]
>>> a['weight'] = [[135, … , 145]]
>>> print a
[[('Brad', 33, 135.0)
  …
  ('Jill', 54, 145.0)]]
```


ENTHOUGHT

Nested Datatype

nested.dat

Time	Size	Position				Gain	Samples (2048) …			
		Az	El	Type	ID					
1172581077060	4108	0.715594	−0.148407	1	4	40	561	1467	997	−30
1172581077091	4108	0.706876	−0.148407	1	4	40	7	591	423	
1172581077123	4108	0.698157	−0.148407	1	4	40	49	−367	−565	−35
1172581077153	4108	0.689423	−0.148407	1	4	40	−55	−953	−1151	−30
1172581077184	4108	0.680683	−0.148407	1	4	40	−719	−1149	−491	38
1172581077215	4108	0.671956	−0.148407	1	4	40	−1503	−683	661	149
1172581077245	4108	0.663232	−0.148407	1	4	40	−2731	−281	2327	291
1172581077276	4108	0.654511	−0.148407	1	4	40	−3493	−159	3277	380
1172581077306	4108	0.645787	−0.148407	1	4	40	−3255	−247	3145	385
1172581077339	4108	0.637058	−0.148407	1	4	40	−2303	−101	2079	247
1172581077370	4108	0.628321	−0.148407	1	4	40	−1495	−553	571	107
1172581077402	4108	0.619599	−0.148407	1	4	40	−955	−1491	−1207	−25
1172581077432	4108	0.61087	−0.148407	1	4	40	−875	−3009	−2987	−93
1172581077463	4108	0.602148	−0.148407	1	4	40	−491	−3681	−4193	−175
1172581077497	4108	0.593438	−0.148407	1	4	40	167	−3501	−4573	−250
1172581077547	4108	0.584696	−0.148407	1	4	40	1007	−2613	−4463	−303
1172581077599	4108	0.575972	−0.148407	1	4	40	1261	−2155	−4299	−339
1172581077650	4108	0.567244	−0.148407	1	4	40	1537	−2633	−4945	−367

ENTHOUGHT

Nested Datatype (cont'd)

The data file can be extracted with the following code:

```
>>> dt = dtype([('time', uint64),
...             ('size', uint32),
...             ('position', [('az', float32),
...                           ('el', float32),
...                           ('region_type', uint8),
...                           ('region_ID', uint16)]),
...             ('gain', uint8),
...             ('samples', int16, 2048)])

>>> data = loadtxt('nested.dat', dtype=dt, skiprows = 2)
>>> data['position']['az']
array([ 0.71559399,  0.70687598,  0.69815701,  0.68942302,
        0.68068302, ...], dtype=float32)
```


ENTHOUGHT

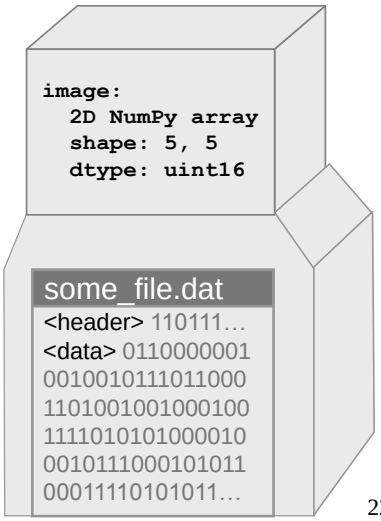
Memory Mapped Arrays

- Methods for Creating:
  - **memmap**: subclass of ndarray that manages the memory mapping details.
  - **frombuffer**: Create an array from a memory mapped buffer object.
  - **ndarray constructor**: Use the `buffer` keyword to pass in a memory mapped buffer.
- Limitations:
  - Files must be < 2GB on Python 2.4 and before.
  - Files must be < 2GB on 32-bit machines.
  - Python 2.5 and higher on 64 bit machines is theoretically "limited" to 17.2 *billion* GB (17 Exabytes).

ENTHOUGHT

Memory Mapped Example

```
# Create a "memory mapped" array where
# the array data is stored in a file on
# disk instead of in main memory.
>>> from numpy import memmap
>>> image = memmap('some_file.dat',
                   dtype=uint16,
                   mode='r+',
                   shape=(5,5),
                   offset=header_size)

# Standard array methods work.
>>> mean_value = image.mean()

# Standard math operations work.
# The resulting scaled_image *is*
# stored in main memory.  It is a
# standard numpy array.
>>> scaled_image = image * .5
```



ENTHOUGHT

memmap

The memmap subclass of array handles opening and closing files as well as synchronizing memory with the underlying file system.

```
memmap(filename, dtype=uint8, mode='r+',
       offset=0, shape=None, order=0)
```

filename Name of the underlying file. For all modes, except for 'w+', the file must already exist and contain at least the number of bytes used by the array.

dtype The numpy data type used for the array. This can be a "structured" dtype as well as the standard simple data types.

offset Byte offset within the file to the memory used as data within the array.

mode <see next slide>

shape Tuple specifying the dimensions and size of each dimension in the array. shape=(5,10) would create a 2D array with 5 rows and 10 columns.

order 'C' for row major memory ordering (standard in the C programming language) and 'F' for column major memory ordering (standard in Fortran).

ENTHOUGHT

memmap -- mode

The mode setting for memmap arrays is used to set the `access` flag when opening the specified file using the standard `mmap` module.

```
memmap(filename, dtype=uint8, mode='r+',
       offset=0, shape=None, order=0)
```

mode A string indicating how the underlying file should be opened.

'**r**' or '**readonly**': Open an existing file as an array for reading.

'**c**' or '**copyonwrite**': "Copy on write" arrays are "writable" as Python arrays, but they *never* modify the underlying file.

'**r+**' or '**readwrite**': Create a read/write array from an existing file. The file will have "write through" behavior where changes to the array are written to the underlying file. Use the `flush()` method to ensure the array is synchronized with the file.

'**w+**' or '**write**': Create the file or overwrite if it exists. The array is filled with zeros and has "write through" behavior similar to 'r+'.

ENTHOUGHT

memmap -- write through behavior

```
# Create a memory mapped "write through" file, overwriting it if it exists.
In [66]: q=memmap('new_file.dat',mode='w+',shape=(2,5))
In [67]: q
memmap([[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]], dtype=uint8)
# Print out the contents of the underlying file.  Note: It
# doesn't print because 0 isn't a printable ascii character.
In [68]: !cat new_file.dat

# Now write the ascii value for 'A' (65) into our array.
In [69]: q[:] = ord('A')
In [70]: q
memmap([[65, 65, 65, 65, 65],
        [65, 65, 65, 65, 65]], dtype=uint8)
# Ensure the OS has written the data to the file, and examine
# the underlying file. It is full of 'A's as we hope.
In [71]: q.flush()
In [72]: !cat new_file.dat
AAAAAAAAAA
```

ENTHOUGHT

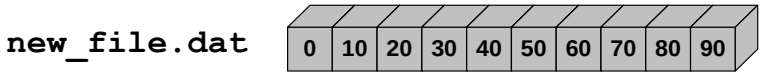
memmap -- copy on write behavior

```
# Create a copy-on-write memory map where the underlying file is never
# modified.  The file must already exist.
# This is a memory efficient way of working with data on disk as arrays but
# ensuring you never modify it.
In [73]: q=memmap('new_file.dat',mode='c',shape=(2,5))
In [74]: q
memmap([[65, 65, 65, 65, 65],
        [65, 65, 65, 65, 65]], dtype=uint8)

# Set values in array to something new.
In [75]: q[1] = ord('B')
In [76]: q
memmap([[65, 65, 65, 65, 65],
        [66, 66, 66, 66, 66]], dtype=uint8)

# Even after calling flush(), the underlying file is not updated.
In [77]: q.flush()
In [78]: !cat new_file.dat
AAAAAAAAAA
```

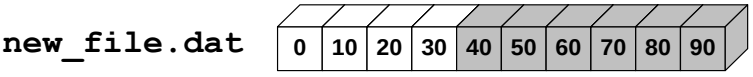

ENTHOUGHT

Using Offsets

```
# Create a memory mapped array with 10 elements.
In [1]: q=memmap('new_file.dat',mode='w+', dtype=uint8, shape=(10,))
In [2]: q[:] = arange(0,100,10)
memmap([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=uint8)
```




```
# Now, create a new memory mapped array (read only) with an offset into
# the previously created file.
In [3]: q=memmap('new_file.dat',mode='r', dtype=uint8, shape=6, offset=4)
In [4]: q
memmap([40, 50, 60, 70, 80, 90], dtype=uint8)
```

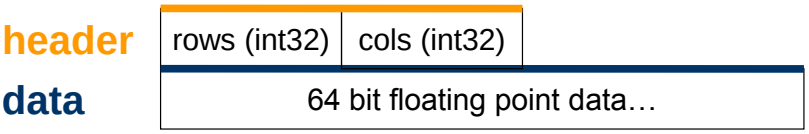



```
# The number of bytes required by the array must be equal or less than
# the number of bytes available in the file.
In [3]: q=memmap('new_file.dat',mode='r', dtype=uint8, shape=7, offset=4)
ValueError: mmap length is greater than file size
```

ENTHOUGHT

Working with file headers

File Format:

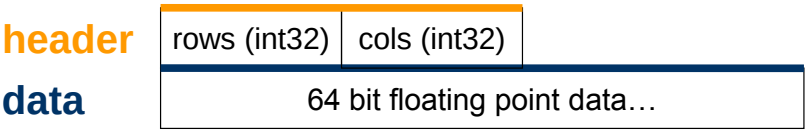



```
# Create a dtype to represent the header.
header_dtype = dtype([('rows', int32), ('cols', int32)])

# Create a memory mapped array using this dtype.  Note the shape is empty.
header = memmap(file_name, mode='r', dtype=header_dtype, shape=())

# Read the row and column sizes from using this structured array.
rows = header['rows']
cols = header['cols']

# Create a memory map to the data segment, using rows, cols for shape
# information and the header size to determine the correct offset.
data = memmap(file_name, mode='r+', dtype=float64,
              shape=(rows, cols), offset=header_dtype.itemsize)
```


ENTHOUGHT

memory maps with ndarray

File Format:

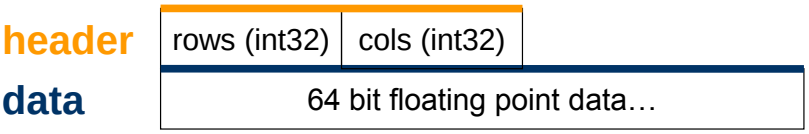



```
# mmap is a standard Python module for working with memory maps.
import mmap
import numpy

# Create a dtype to represent the header.
header_dtype = numpy.dtype([('rows', int32), ('cols', int32)])

# Open a file for read/write access in binary mode.
file = open(file_name,'r+b')

# Create a read-only memory map from the opened file with the
# correct size to read the header of the file.
mm = mmap.mmap(file.fileno(), header_dtype.itemsize,
               access=mmap.ACCESS_READ)
< continued >
```

ENTHOUGHT

memory maps with ndarray

File Format:




```
# Create a new array using the ndarray constructor.
# The first argument is the shape, and we pass in the data type and the
# memory buffer to use (mm) as keyword arguments.
header = numpy.ndarray((), dtype=header_dtype, buffer=mm)
rows = header['rows']
cols = header['cols']

# Create a writable memory map to use for the data array.  The size of the
# memory map in bytes is the size of a float64 (8) * rows * columns.
mm = mmap.mmap(file.fileno(), 8*rows*cols, access=mmap.ACCESS_WRITE)

# Create our data array using this new memory map.  Start the arrays
# data at the memory location directly after the header using offset.
data = numpy.ndarray((rows, cols), dtype=float64, buffer=mm,
                     offset=header_dtype.itemsize)
```


ENTHOUGHT

Structured Arrays

`char[12]  int64  float32`

Name	Time	Value
MSFT_profit	10	6.20
GOOG_profit	12	-1.08
MSFT_profit	18	8.40
INTC_profit	25	-0.20
⋮	⋮	⋮
GOOG_profit	1000325	3.20
GOOG_profit	1000350	4.50
INTC_profit	1000385	-1.05
MSFT_profit	1000390	5.60

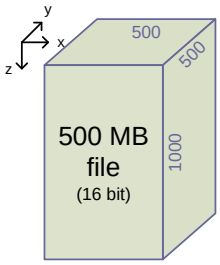
Elements of array can be any fixed-size data structure!

Example

```
>>> import numpy as np
>>> fmt = np.dtype([('name', 'S12'),
                    ('time', np.int64),
                    ('value', np.float32)])
>>> vals = [('MSFT_profit', 10, 6.20),
            ('GOOG_profit', 12, -1.08),

            ('INTC_profit', 1000385, -1.05)
            ('MSFT_profit', 1000390, 5.60)]

>>> arr = np.array(vals, dtype=fmt)
# or
>>> arr = np.fromfile('db.dat', dtype=fmt)
# or
>>> arr = np.memmap('db.dat', dtype=fmt,
                    mode='c')
```

ENTHOUGHT

Memmap Timings (3D arrays)



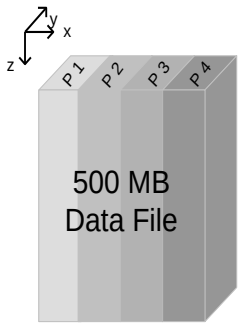
Operations (500x500x1000)	Linux		OS X	
	In Memory	Memory Mapped	In Memory	Memory Mapped
read	2103 ms	11.0 ms	3505.00	27.00
x slice	1.8 ms	4.8 ms	1.80	8.30
y slice	2.8 ms	4.6 ms	4.40	7.40
z slice	9.2 ms	13.8 ms	10.40	18.70
downsample 4x4	0.02 ms	125 ms	0.02	198.70

All times in millesconds (ms).

Linux: Ubuntu 4.10, Dell Precision 690, Dual Quad Core Zeon X5355 2.6 GHz, 8 GB Memory
OS X: OS X 10.5, MacBook Pro Laptop, 2.6 GHz Core Duo, 4 GB Memory

ENTHOUGHT

Parallel FFT On Memory Mapped File



Processors	Time (seconds)	Speed Up
1	11.75	1.0
2	6.06	1.9
4	3.36	3.5
8	2.50	4.7

```
1  from numpy import ceil
2  from ipython1.kernel import client
3  from geoio import vtio
4
5  # Execute an fft on a sub-section of a seismic cube.
6  code = \
7  """
8  from numpy import fft
9  from geoio import vtio
10
11 seismic, params = vtio.read(file_name,rescale=False)
12 start, end = id*size, (id+1) * size
13 local_seismic = vtio.unclip(seismic[start:end,:,:])
14 spectrum = fft.fft(local_seismic,axis=-1)
15 """
16
17 def equal_size_split(ary, cluster):
18     # Return the number of rows each worker should work
19     return int(ceil(float(len(ary))/len(cluster)))
20
21 # Run parallel code  on each of the remote processors
22 file_name = "500_500_1000.vt"
23 cluster = client.MultiEngineClient(('127.0.0.1',10105))
24 seismic, params = vtio.read(file_name,rescale=False)
25 cluster['size'] = equal_size_split(seismic, cluster)
26 cluster['file_name'] = file_name
27 cluster.execute(code)
```

ENTHOUGHT

Controlling Output Format

```
set_printoptions(precision=None, threshold=None,
                 edgeitems=None, linewidth=None,
                 suppress=None)
```

precision The number of digits of precision to use for floating point output. The default is 8.

threshold Array length where NumPy starts truncating the output and prints only the beginning and end of the array. The default is 1000.

edgeitems Number of array elements to print at beginning and end of array when threshold is exceeded. The default is 3.

linewidth Characters to print per line of output. The default is 75.

suppress Indicates whether NumPy suppresses printing small floating point values in scientific notation. The default is **False**.

ENTHOUGHT

Controlling Output Formats

PRECISION

```
>>> a = arange(1e6)
>>> a
array([  0.00000000e+00, 1.00000000e+00, 2.00000000e+00, ...,
         9.99997000e+05, 9.99998000e+05, 9.99999000e+05])
>>> set_printoptions(precision=3)
>>> a
array([  0.000e+00,   1.000e+00,   2.000e+00, ...,
         1.000e+06,   1.000e+06,   1.000e+06])
```

SUPPRESSING SMALL NUMBERS

```
>>> set_printoptions(precision=8)
>>> a = array((1, 2, 3, 1e-15))
>>> a
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00,
         1.00000000e-15])
>>> set_printoptions(suppress=True)
>>> a
array([ 1.,  2.,  3.,  0.])
```

ENTHOUGHT

Controlling Error Handling

```
seterr(all=None, divide=None, over=None,
       under=None, invalid=None)
```

Set the error handling flags in ufunc operations on a per thread basis. Each of the keyword arguments can be set to ‘ignore’, ‘warn’, ‘print’, ‘log’, ‘raise’, or ‘call’.

all	All error types to the specified value
divide	Divide-by-zero errors
over	Overflow errors
under	Underflow errors
invalid	Invalid floating point errors

ENTHOUGHT

Controlling Error Handling

```
>>> a = array((1,2,3))
>>> a/0.
Warning: divide by zero encountered in divide
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])

# Ignore division-by-zero.  Also, save old values so that
# we can restore them.
>>> old_err = seterr(divide='ignore')
>>> a/0.
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])

# Restore original error handling mode.
>>> old_err
{'divide': 'print', 'invalid': 'print', 'over': 'print',
'under': 'ignore'}
>>> seterr(**old_err)
>>> a/0.
Warning: divide by zero encountered in divide
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])
```

ENTHOUGHT

Scientific Analysis with SciPy

ENTHOUGHT

Overview

- Available at www.scipy.org
- Open source BSD style license
- 38 contributors to the project in 2011

CURRENT PACKAGES

- Special Functions (scipy.special)
- Signal Processing (scipy.signal)
- Image Processing (scipy.ndimage)
- Fourier Transforms (scipy.fftpack)
- Optimization (scipy.optimize)
- Numerical Integration (scipy.integrate)
- Linear Algebra (scipy.linalg)
- Input/Output (scipy.io)
- Statistics (scipy.stats)
- Fast Execution (scipy.weave)
- Clustering Algorithms (scipy.cluster)
- Sparse Matrices (scipy.sparse)
- Interpolation (scipy.interpolate)
- ...and more.

Note: “import scipy” does NOT import all these packages. Must be imported individually

Polynomials

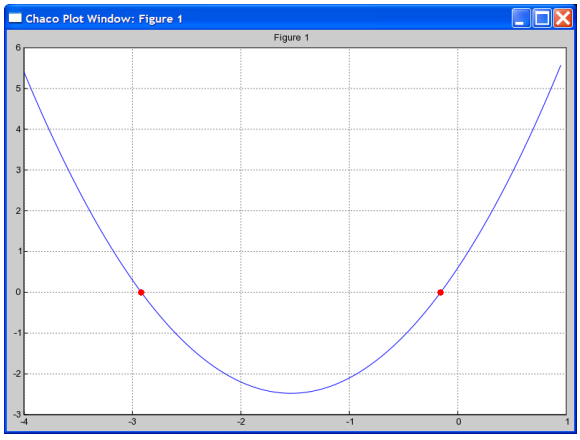
- p = poly1d(<coefficient array>)
- p.roots (p.r) are the roots
- p.coefficients (p.c) are the coefficients
- p.order is the order
- p[n] is the coefficient of x^n
- p(val) evaluates the polynomial at val
- p.integ() integrates the polynomial
- p.deriv() differentiates the polynomial
- Basic numeric operations (+,-,/,*) work
- Acts like p.c when used as an array
- Fancy printing

```
from numpy import poly1d

>>> p = poly1d([1,-2,4])
>>> print p
   2
1 x - 2 x + 4

>>> g = p**3 + p*(3-2*p)
>>> print g
   6     5      4      3      2
1 x - 6 x + 22 x - 48 x + 75 x - 70 x + 44

>>> print g.deriv(m=2)
    4       3       2
30 x - 120 x + 264 x - 288 x + 150

>>> print p.integ(m=2,k=[2,1])
         4          3     2
0.08333 x - 0.3333 x + 2 x + 2 x + 1

>>> print p.roots
[ 1.+1.73205081j  1.-1.73205081j]

>>> print p.coeffs
[ 1 -2  4]
```


Polynomials

FINDING THE ROOTS OF A POLYNOMIAL

```
>>> p = poly1d([1.3, 4.0, 0.6])
>>> print p
     2
1.3 x + 4 x + 0.6
>>> x = linspace(-4, 1.0, 101)
>>> y = p(x)
>>> plot(x,y,'-')
>>> hold(True)
>>> r = p.roots
>>> s = p(r)
>>> r
array([-0.15812627, -2.9187968 ])
>>> plot(r.real,s.real,'ro')
```



ENTHOUGHT

Optimization

scipy.optimize — Minimization and Root Finding

Univariate Function Minimization

`minimize_scalar` - minimize a scalar-valued function of a single variable: available methods include brent, bounded, golden

Multivariate Function Minimization (constrained and unconstrained)

`minimize` – minimize scalar-valued function of one or more variables: available methods include BFGS, Nelder-Mead simplex, Newton conjugate gradient, COBYLA, SLSQP, anneal, brute

Least Squares Fitting

`leastsq` - wrapper to the Fortran leastsq function from MINPACK

Curve Fitting

`curve_fit` – powerful, general, tool to fit functional parameters.

Root Finding

`brentq, brenth, ridder, bisect, newton` – find root of a scalar function

`fixed_point` – find the fixed point of a function

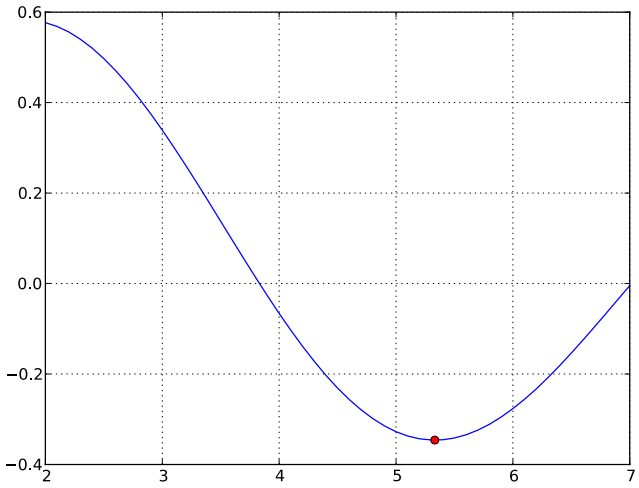
`root` – multidimensional root-finding: methods include hybr and lm for small problems and krylov, brodyen2, and anderson for large problems.

ENTHOUGHT

Optimization: 1-D Minimization

EXAMPLE: MINIMIZE BESSEL FUNCTION

```
# minimize 1st order Bessel
# function between 4 and 7
>>> from scipy.special import j1
>>> from scipy.optimize import \
...   minimize_scalar

>>> x = np.linspace(2, 7, 200)
>>> j1x = j1(x)
>>> plot(x, j1x)
>>> result = minimize_scalar(j1,
...   method="bounded",
...   bounds=[4, 7])
>>> j1_min = j1(result.x)
>>> plot(result.x, j1_min,'ro')
```



ENTHOUGHT

Result Object

Output of **`minimize`**, **`minimize_scalar`**, and **`root`** is a **`Result`** object with attributes:

`x`	:	solution to the optimization
`success`	:	True/False – did the optimization succeed?
`status`	:	integer termination status
`message`	:	termination status message
`fun, jac, hess`	:	values of the function, Jacobian, Hessian, (if available) at the optimized value
`nfev, njev, nhev`	:	number of evaluations of the function, Jacobian, or Hessian (if available)
`...`	:	other attributes are added by specific methods

Note: older legacy functions (fmin, fmin_bgfs, fminbound, fsolve, etc.) do not have a unified API and are replaced by the above.

ENTHOUGHT

Optimization: Using Derivatives

minimize: WITHOUT DERIVATIVE

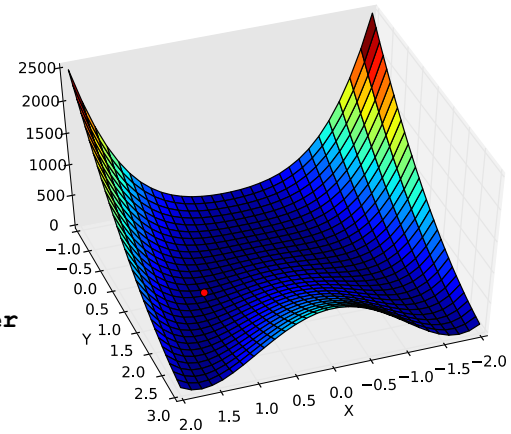
```
>>> from scipy.optimize import rosen
>>> x0 = [1.3, 1.6, -0.5, -1.8, 0.8]
>>> result = minimize(rosen, x0)
>>> print result.x
[ 1.  1.  1.  1.  1.]
>>> print result.nfev
462
```

minimize: USING DERIVATIVE

```
>>> from scipy.optimize import rosen_der
>>> result = minimize(rosen, x0,
...      jac=rosen_der)
>>> print result.x
[ 1.  1.  1.  1.  1.]
>>> print result.nfev, result.njev
66 66
```

Rosenbrock function

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_i - x_{i-1}^2\right)^2 + (1 - x_{i-1})^2 .$$



ENTHOUGHT

Optimization: Solving Nonlinear Equations

ROOT

```
>>> def equations(x,a,b,c):
...     x0, x1, x2 = x
...     eqs = \
...     [3 * x0 - cos(x1*x2) + a,
...     x0**2 - 81*(x1+0.1)**2 + sin(x2) + b,
...     exp(-x0*x1) + 20*x2 + c]
...     return eqs

>>> from scipy.optimize import root
# coefficients
>>> a = -0.5 ; b = 1.06
>>> c = (10 * pi - 3.0) / 3
# Optimization start location.
>>> initial_guess = [0.1, 0.1, -0.1]
# Solve the system of non-linear equations.
>>> result = root(equations, initial_guess, args=(a, b, c))
>>> print "root:", result.x
root: [ 0.5    0.    -0.52]
>>> print "solution at root:", result.fun
solution at root: [ 0. -0.  0.]
```

SYSTEM OF EQUATIONS

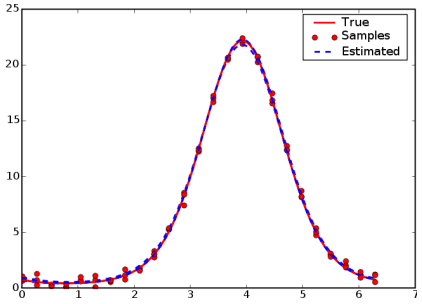
$$3x_0 - \cos(x_1 x_2) + a = 0$$
$$x_0^2 - 81(x_1 + 0.1)^2 + \sin(x_2) + b = 0$$
$$e^{-x_0 x_1} + 20x_2 + c = 0$$

ENTHOUGHT

Optimization: Data Fitting

NONLINEAR LEAST SQUARES CURVE FITTING

```
>>> from scipy.optimize import curve_fit
>>> from scipy.stats import norm
# Define the function to fit.
>>> def function(x, a, b, f, phi):
...     result = a * exp(-b * sin(f * x + phi))
...     return result

# Create a noisy data set.
>>> actual_params = [3, 2, 1, pi/4]
>>> x = linspace(0,2*pi,25)
>>> exact = function(x, *actual_params)
>>> noisy = exact + 0.3*norm.rvs(size=len(x))
# Use curve_fit to estimate the function parameters from the noisy data.
>>> initial_guess = [1,1,1,1]
>>> estimated_params, err_est = curve_fit(function, x, noisy, p0=initial_guess)
>>> estimated_params
array([3.1705, 1.9501, 1.0206, 0.7034])
# err_est is an estimate of the covariance matrix of the estimates
#   (i.e. how good of a fit is it)
>>> err_est.diagonal()
array([ 0.0572,  0.006362,  0.0005834,  0.008979])
```

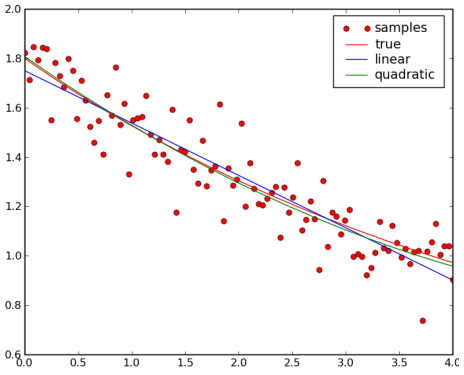


ENTHOUGHT

Fitting Polynomials (NumPy)

POLYFIT(X, Y, DEGREE)

```
>>> from numpy import polyfit, poly1d
>>> from scipy.stats import norm
# Create clean data.
>>> x = linspace(0, 4.0, 100)
>>> y = 1.5 * exp(-0.2 * x) + 0.3
# Add a bit of noise.
>>> noise = 0.1 * norm.rvs(size=100)
>>> noisy_y = y + noise

# Fit noisy data with a linear model.
>>> linear_coef = polyfit(x, noisy_y, 1)
>>> linear_poly = poly1d(linear_coef)
>>> linear_y = linear_poly(x)

# Fit noisy data with a quadratic model.
>>> quad_coef = polyfit(x, noisy_y, 2)
>>> quad_poly = poly1d(quad_coef)
>>> quad_y = quad_poly(x)
```



ENTHOUGHT

Linear Algebra

scipy.linalg — FAST LINEAR ALGEBRA

- Uses optimized BLAS and MKL if available — very fast
- Low-level access to BLAS and LAPACK routines in modules **`linalg.blas`**, and **`linalg.lapack`** (FORTRAN order)
- High level matrix routines
  - Linear Algebra Basics: **`inv, pinv, solve, det, norm, lstsq, ...`**
  - Decompositions: **`eig, lu, svd, orth, cholesky, qr, schur, ...`**
  - Matrix Functions: **`expm, logm, sqrtm, cosm, coshm, funm`** (general matrix functions)

ENTHOUGHT

Solving A*X = B

SOLVE Ax=b

```
>>> from scipy import linalg
>>> a = array([[1.0, 3.0, 2.0],
...            [4.0, 0.0, 1.0],
...            [2.0, 2.0, 2.0]])
>>> b = array([2.0, 1.0, 0.0])

>>> x = linalg.solve(a, b)

>>> x
array([ 1.5,  3.5, -5. ])

# Check:
>>> dot(a, x)
array([ 2.,  1.,  0.])
```

LEAST SQUARES Ax=b

```
# Over-specified problem:
>>> a = array([[1.0, 4.0],
...            [3.0, 0.0],
...            [2.0, 1.0]])
>>> b = array([2.0, 1.0, 0.0])
>>> x, res, rnk, svals = \
...     linalg.lstsq(a, b)

# Least squares solution:
>>> x
array([ 0.1832,  0.4059])
# Sum of squared residuals:
>>> res
0.83663366336633671
# Effective rank of `a`:
>>> rnk
2
# Singular values of `a`:
>>> svals
array([ 4.6567,  3.0521])
```


ENTHOUGHT

LU FACTORIZATION

To solve many equations AX=B for different matrices B, factor A =L*U , L is lower triangular (L) and U is upper-triangular

```
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# time consuming factorization
>>> lu, piv = linalg.lu_factor(a)

# fast solve for 1 or more
# right hand sides.
>>> b = array([10,8,3])
>>> linalg.lu_solve((lu, piv), b)
array([-7.82608696,  4.56521739,
        0.82608696])
# check
>>> dot(a, _)
array([ 10.,   8.,   3.])
```

QR FACTORIZATION

To solve Ax=b, factor A = QR, where Q is orthogonal and R is upper triangular.

```
>>> a = array([[12, -51,   4],
...            [ 6, 167, -68],
...            [-4,  24, -41]])
>>> q, r = linalg.qr(a)
>>> q
array([[-0.8571,  0.3943,  0.3314],
       [-0.4286, -0.9029, -0.0343],
       [ 0.2857, -0.1714,
0.9429]])
>>> r
array([[ -14.,  -21.,   14.],
       [   0., -175.,   70.],
       [  -0.,    0.,  -35.]])
```

See the `qr` docstring for more advanced options (e.g. pivoting for rank-revealing QR decomposition).

ENTHOUGHT

Eigenvalues, eigenvectors

EIGENVALUES AND VECTORS

```
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# compute eigenvalues/vectors
>>> vals, vecs = linalg.eig(a)
# print eigenvalues
>>> vals
array([ 9.39895873+0.j,
       -0.73379338+0.j,
        3.33483465+0.j])
# eigenvectors are in columns
# print first eigenvector
>>> vecs[:,0]
array([-0.57028326,
       -0.41979215,
       -0.70608183])
# norm of vector should be 1.0
>>> linalg.norm(vecs[:,0])
1.0
```

SCHUR DECOMPOSITION

Factor A = ZTZ^{-1}, where Z is unitary and T is upper triangular.

```
>>> t, z = linalg.schur(a)
>>> t
array(
 [[ 156.137,   64.737,   85.651],
  [   0.   ,   16.06 ,   15.814],
  [   0.   ,    0.   ,  -34.197]])
>>> z
array([[ 0.328, -0.94 ,  0.098],
       [-0.937, -0.337, -0.093],
       [-0.121,  0.061,  0.991]])

# Diagonals in t are eigenvalues:
>>> linalg.eigvals(a)
array([ 156.137+0.j,   16.060+0.j,
-34.197+0.j])
```


ENTHOUGHT

Linear Algebra

SINGULAR VALUE DECOMPOSITION

Factor A = U S V*, where U and V are unitary, and S is diagonal. This generalizes diagonalization.

```
>>> a = array([[2, 1],
...            [2, 0]])
>>> u, s, vt = linalg.svd(a)
>>> u
array([[-0.75 , -0.662],
       [-0.662,  0.75 ]])
# s holds the singular values.
>>> s
array([ 2.921,  0.685])
>>> vt
array([[-0.966, -0.257],
       [ 0.257, -0.966]])
# Use to inverse A
>>> inv_a = dot(vt.T,
           dot(diag(1/s), u.T))
>>> dot(a, inv_a)
array([[  1.0,  -4.44089210e-16],
       [ -2.22044605e-16,   1.0]])
```

```
# Nonsquare example:
underspecified
# set of equations
>>> a = array([[2, 1, -4],
               [2, 3,  1]])
>>> u, s, vt = linalg.svd(a)
>>> u
array([[-0.938, -0.347],
       [-0.347,  0.938]])
>>> s
array([ 4.702,  3.59 ])
>>> vt
array([[-0.546, -0.421,  0.724],
       [ 0.329,  0.687,  0.648],
       [ 0.77 , -0.592,  0.237]])
```

ENTHOUGHT

Inverse, and determinants

MATRIX INVERSION

```
# Square matrix
>>> from scipy import linalg
>>> a = array([[2, 1],
...            [2, 0]])
>>> a_inv = linalg.inv(a)
>>> a_inv
array([[ 0. ,  0.5],
       [ 1. , -1. ]])
# Check
>>> dot(a, a_inv)
array([[ 1.,  0.],
       [ 0.,  1.]])

# Alternatively (slower)
>>> linalg.solve(a, indentity(2))
array([[ 0. ,  0.5],
       [ 1. , -1. ]])
```

```
# Non-square mat: Pseudo-inverse
# with least-square method
>>> b = array([[2, 1, -4],
               [2, 3,  1]])
>>> linalg.pinv(b)
array([[ 0.07719298,
0.12631579],
       [ 0.01754386,
0.21052632],
       [-0.20701754,
0.11578947]])
```

MATRIX DETERMINANT

```
>>> linalg.det(a)
-2.0

#If LU decomposition available:
>>> lu, _ = linalg.lu_factor(a)
>>> np.prod(lu.diagonal())
-2.0
```


ENTHOUGHT

Matrix Objects

STRING CONSTRUCTION

```
>>> from numpy import mat
>>> a = mat('[1,3,5;2,5,1;2,3,6]')
>>> a
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 6]])
```

TRANSPOSE ATTRIBUTE

```
>>> a.T
matrix([[1, 2, 2],
        [3, 5, 3],
        [5, 1, 6]])
```

INVERTED ATTRIBUTE

```
>>> a.I
matrix([[-1.1739,  0.1304,  0.956],
        [ 0.4347,  0.1739, -0.391],
        [ 0.1739, -0.130,  0.0434]
      ])
```

DIAGONAL

```
>>> a.diagonal()
matrix([[1, 5, 6]])
>>> a.diagonal(-1)
matrix([[2, 3]])
```

SOLVE

```
>>> b = mat('10;8;3')
>>> a.I*b
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])

>>> from scipy import linalg
>>> linalg.solve(a,b)
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])
```

ENTHOUGHT

Interpolation

scipy.interpolate — General purpose Interpolation

- **1D Interpolating Class**
  - Constructs callable function from data points and desired spline interpolation order.
  - Function takes vector of inputs and returns interpolated value using the spline.
- **Radial basis functions**
  - Simple but effective N-dimensional interpolation
  - Works with scattered data

ENTHOUGHT

1D Spline Interpolation

```
>>> from scipy.interpolate import interp1d
```

interp1d(x, y, kind='linear', axis=-1, copy=True, bounds_error=True, fill_value=numpy.nan)

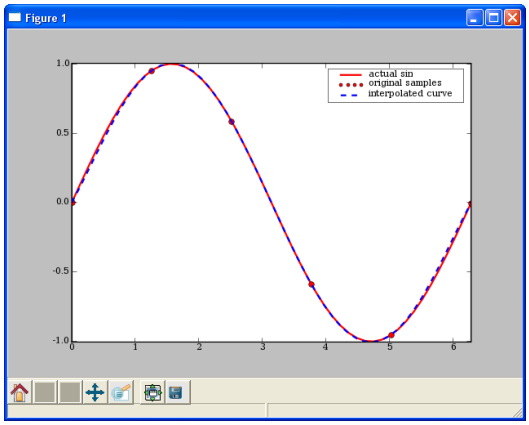
Returns a function that uses interpolation to find the value of new points.

- **x** – 1d array of increasing real values which cannot contain duplicates
- **y** – Nd array of real values whose length along the interpolation axis must be len(x)
- **kind** – kind of interpolation (*e.g.* 'linear', 'nearest', 'quadratic', 'cubic'). Can also be an integer n>1 which returns interpolating spline (with minimum sum-of-squares discontinuity in nth derivative).
- **axis** – axis of y along which to interpolate
- **copy** – make internal copies of x and y
- **bounds_error** – raise error for out-of-bounds
- **fill_value** – if bounds_error is False, then use this value to fill in out-of-bounds.

ENTHOUGHT

1D Spline Interpolation

```
# demo/interpolate/spline.py
from scipy.interpolate import interp1d
from pylab import plot, axis, legend
from numpy import linspace

# sample values
x = linspace(0,2*pi,6)
y = sin(x)

# Create a spline class for interpolation.
# kind='nearest' -> zeroth older hold.
# kind='linear' -> linear interpolation
# kind=n -> use an nth order spline
spline_fit = interp1d(x,y,kind=5)
xx = linspace(0,2*pi, 50)
yy = spline_fit(xx)

# display the results.
plot(xx, sin(xx), 'r-', x, y, 'ro', xx, yy, 'b--', linewidth=2)
axis('tight')
legend(['actual sin', 'original samples', 'interpolated curve'])
```



ENTHOUGHT

Radial Basis Functions

```
>>> from scipy.interpolate import Rbf
```

```
func = Rbf(x, y, …, function='multiquadric',
epsilon=None, smooth=0, norm=euclidean_norm)
```

Returns a function that uses interpolation to find the value of new points.

- **x** – array of real values
- **y** – array of real values the same shape as x
- … - additional array arguments can be provided for N-d interpolation
- **function** – basis function used to interpolate; one of 'multiquadric', 'inverse multiquadric', 'gaussian', 'linear', 'cubic', 'quintic', 'thin-plate'
- **epsilon** – adjustable constant for 'gaussian' or 'multiquadrics' functions. Defaults to approximate average distance between nodes.
- **smooth** – Greater than zero increases smoothness of approximation. Default of 0 is interpolation.
- **norm** – function (vectorized) that returns the distance between two points

Radial Basis Functions

The function is approximated by a sum of radial functions with certain weights n_j:

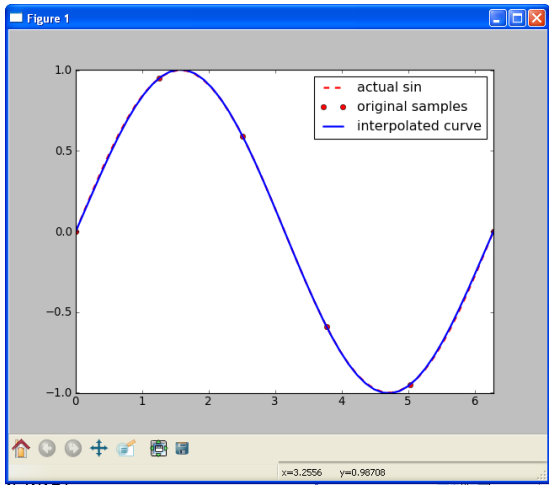
$$f(\mathbf{x}) = \sum_j h(||\mathbf{x} - \mathbf{x}_j||)n_j \qquad f(\mathbf{x}_i) = \sum_j h(||\mathbf{x}_i - \mathbf{x}_j||)n_j$$

Typical radial functions:

$$\begin{aligned}
h_{\text{multiquadric}}(r) &= \sqrt{\frac{r^2}{\epsilon^2} + 1} \\
h_{\text{inverse multiquadric}}(r) &= \frac{1}{\sqrt{\frac{r^2}{\epsilon^2} + 1}} \\
h_{\text{gaussian}}(r) &= \exp\left(-\frac{r^2}{\epsilon^2}\right) \\
h_{\text{linear}}(r) &= r \\
h_{\text{cubic}}(r) &= r^3 \\
h_{\text{quintic}}(r) &= r^5 \\
h_{\text{thin-plate}}(r) &= r^2 + \log(r)
\end{aligned}$$

1D RBF Interpolation

```
# demo/interpolate/spline.py
from scipy.interpolate import Rbf
from pylab import plot, axis, legend
from numpy import linspace

# sample values
x = linspace(0,2*pi,6)
y = sin(x)

# Create a new function
fit = Rbf(x,y, function='gaussian')
xx = linspace(0,2*pi, 50)
yy = fit(xx)

# display the results.
plot(xx, sin(xx), 'r--', x, y, 'ro', xx, yy, 'b-', linewidth=2)
axis('tight')
legend(['actual sin', 'original samples', 'interpolated curve'])
```

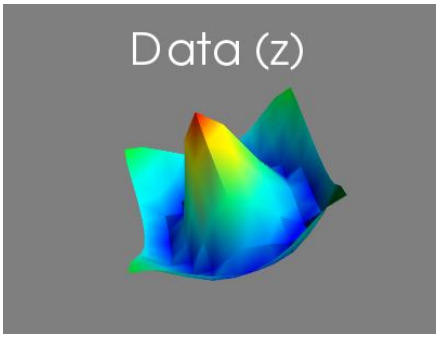


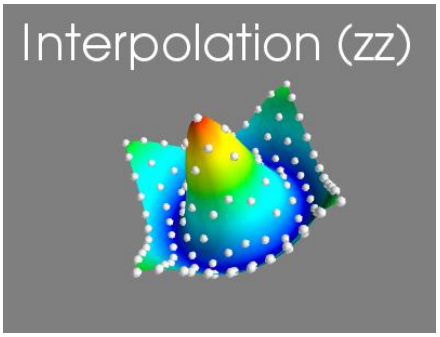
ENTHOUGHT

2D RBF Interpolation

EXAMPLE

```
>>> from scipy.interpolate import \
...       Rbf
>>> from numpy import hypot, mgrid
>>> from scipy.special import j0
>>> x, y = mgrid[-5:6,-5:6]
>>> z = j0(hypot(x,y))
>>> newfunc = Rbf(x, y, z)
>>> xx, yy = mgrid[-5:5:100j,
                   -5:5:100j]
# xx and yy are both 2-d
# result is evaluated
#  element-by-element
>>> zz = newfunc(xx, yy)
>>> from mayavi import mlab
>>> mlab.surf(x, y, z*5)
>>> mlab.figure()
>>> mlab.surf(xx, yy, zz*5)
>>> mlab.points3d(x,y,z*5,
                  scale_factor=0.5)
```





ENTHOUGHT

Integration

scipy.integrate — General purpose Integration

- **Ordinary Differential Equations (ODE)**

`integrate.odeint, integrate.ode`

- **Samples of a 1-D function**

`integrate.trapz` (trapezoidal Method), `integrate.simps` (Simpson Method), `integrate.romb` (Romberg Method)

- **Arbitrary callable function**

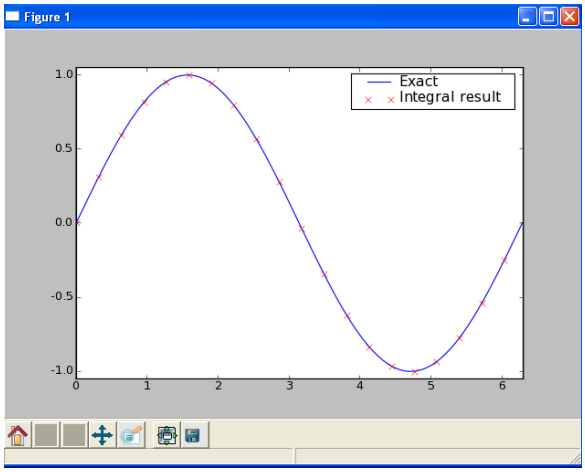
`integrate.quad` (general purpose), `integrate.dblquad` (double integration), `integrate.tplquad` (triple integration), `integrate.fixed_quad` (fixed order Gaussian integration), `integrate.quadrature` (Gaussian quadrature to tolerance), `integrate.romberg` (Romberg)

ENTHOUGHT

Integration

EXAMPLE (INTEGRATE A CALLABLE)

```
# Compare sin to integral(cos)
>>> def func(x):
...   # Quad returns (result, error).
...   # Only return the result.
...   return integrate.quad(cos,0,x)[0]
>>> vecfunc = vectorize(func)

>>> x = linspace(0, 2*pi, 1010)
>>> x2 = x[::5]
>>> y = sin(x)
>>> y2 = vecfunc(x2)
>>> plot(x,y,x2,y2,'rx')
>>> legend(['Exact',
...          'Integral Result'])
```

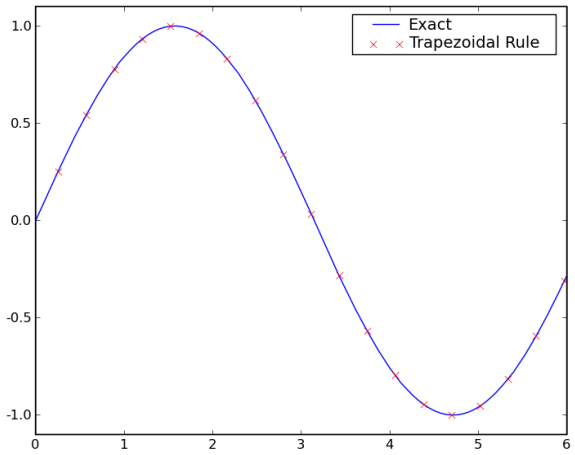


ENTHOUGHT

Integration

EXAMPLE (INTEGRATE FROM SAMPLES)

```
# Compare sin to integral(cos)
>>> x = r_[0:2*pi:100j]
>>> y = sin(x)
>>> fx = cos(x)
>>> N = len(x)+1
>>> y2 = [trapz(fx[:i],x[:i])
...       for i in range(5,N,5)]
>>> x2 = x[4::5]
>>> plot(x,y,x2,y2,'rx')
>>> legend(['Exact',
...          'Trapezoidal Rule'])
```



ENTHOUGHT

Ordinary Differential Equations

```
>>> from scipy.integrate import odeint
```

Find $y(t)$ given $y(t_0)$ and

$$\frac{dy}{dt} = f(y,t)$$

```
odeint(func, y0, t, args=(),**kwargs)
```

```
func   - func(y,tn,...) calculates dy/dt at tn, y can
         be a vector; func should then return the
         same length vector
y0     - initial value of y (can be a vector)
t      - a sequence of time points at which to solve for
         y, the initial value, t0, should be the first
         element of the sequence
args   - any extra arguments needed by func
kwargs - additional ODE solver arguments
```


ENTHOUGHT

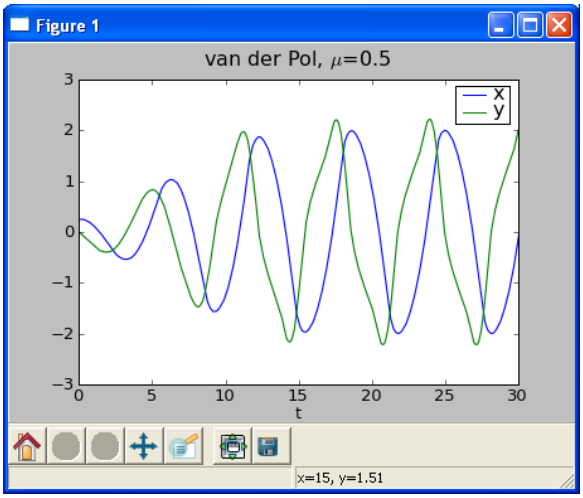
ODEINT Example

```
from scipy.integrate import odeint
from numpy import linspace
from matplotlib.pyplot import *

def vanderpolsys(w, t, mu):
    """ Van der Pol system vector
    field. """
    x, y = w
    f = [ y, -mu*(x**2-1)*y - x]
    return f

t = linspace(0, 30.0, 301)
mu = 0.5
init = [0.25,0.0]
sol = odeint(vanderpolsys, init, t,
    args=(mu,), atol=1.0e-8, rtol=1.0e-
    10)

plot(t, sol[:,0], label='x')
plot(t, sol[:,1], label='y')
title('van der Pol, $\mu$=%g' % mu)
```

van der Pol system:

$x' = y$

$y' = -\mu(x^2-1)y - x$



ENTHOUGHT

Clustering

scipy.cluster — Basic Clustering Algorithms

- **Observation whitening** `cluster.vq.whiten`
- **Vector quantization** `cluster.vq.vq`
- **K-means algorithm** `cluster.vq.kmeans`

ENTHOUGHT

`scipy.io` - MATLAB FILES

SAVE FILE

```
>>> from scipy.io import savemat
>>> a = array([[1.0, 2.0, 3.0],
               [1.5, 2.5, 3.5]])
>>> savemat('tst.mat',
...         dict(a=a, x=1))
```

LOAD MATLAB 7.1 OR BEFORE

```
>>> from scipy.io import loadmat
>>> mf = loadmat('tst.mat')
>>> print mf['a']
[[ 1.   2.   3. ]
 [ 1.5  2.5  3.5]]
>>> print mf['x']
[[1]]
```

LOAD - MATLAB 7.3 OR AFTER

```
# Matlab 7.3 uses HDF files
# by default.  The 'tables'
# library handles HDF files
# well.
# Suppose the array 'y' was
# save in 'foo.mat'.
>>> import tables
>>> f = tables.openFile('foo.mat')
>>> y = f.root.y.read()
>>> print y
[[ 1.5  2.5  1. ]
 [ 2.5  3.   2.5]
 [ 4.5  2.5  1. ]
 [ 1.   2.5  2. ]]
```

ENTHOUGHT

scipy.io - Additional Formats

IDL

```
>>> from scipy.io import readsav
```

`readsav(file_name, ...)`
Read an IDL ".sav" file.

ARFF

```
>>> from scipy.io.arff import \
        loadarff
```

`loadarff(f)`
Read an ARFF file.

HARWELL-BOEING SPARSE

```
>>> from scipy.io import \
        hb_read, hb_write
```

`hb_read(file)`
Read HB-format file.
`hb_write(file, m, hb_info=None)`
Write HB-format file.

NETCDF

```
>>> from scipy.io.netcdf import \
      netcdf_file, netcdf_variable
```

`netcdf_file(filename, ...)`
A file object for NetCDF data.
`netcdf_variable(data,typecode,...)`
A data object for the netcdf module.

(See also the third-party **netCDF** library.)

WAV

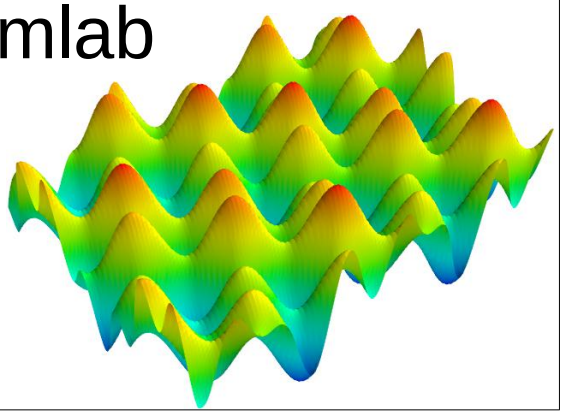
```
>>> from scipy.io.wavfile import \
        read, write
```

`read(file)`
Read sample rate and data.
`write(filename, rate, data)`
Write a numpy array as a WAV file.

(See also the standard library **wave**.)

ENTHOUGHT

Interactive 3D Visualization with Mayavi-mlab

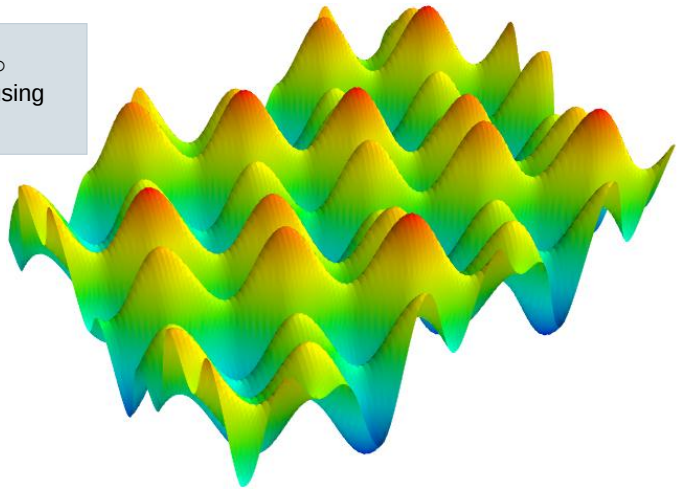


ENTHOUGHT

3D Visualization with Mayavi

```
C:\ ipython --pylab

>>> from mayavi import mlab
```

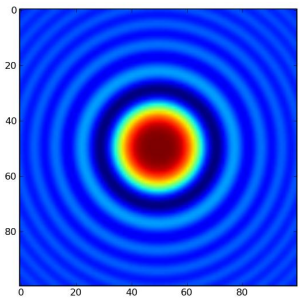

`matplotlib` also has an `mlab` namespace. Be sure you are using the one from **`mayavi`**



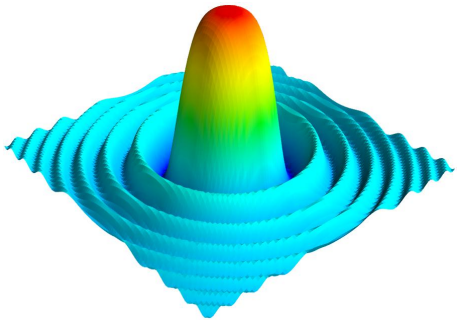
ENTHOUGHT

Interactive Example

```
# create arrays
>>> x, y = mgrid[-5:5:100j,-5:5:100j]
>>> r = x**2 + y**2
>>> z = sin(r)/r

# plot with pylab
>>> imshow(z)
```

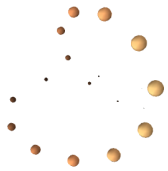



```
# plot with mayavi
>>> from mayavi import mlab
>>> mlab.surf(x,y,5*z)
```



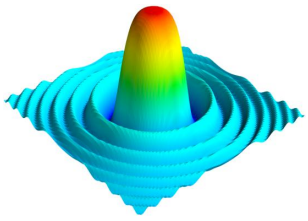
ENTHOUGHT

Some Plotting Commands



0D data

```
mlab.points3d(x, y, z)
```

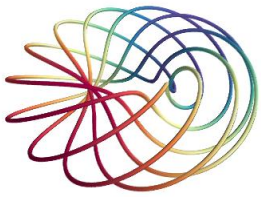


2D data

```
mlab.surf(x, y, z)
```

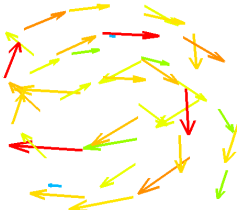
3D data

```
mlab.contour3d(x, y, z)
```



1D data

```
mlab.plot3d(x, y, z)
```



Vector field

```
mlab.quiver(x, y, z, u, v, w)
```


ENTHOUGHT

Points in 3D

scaling applied

```
mlab.points3d(x, y, z, color=(1.0,0.0,1.0), mode='sphere', scale_factor=0.1)
```

x.shape == y.shape == z.shape

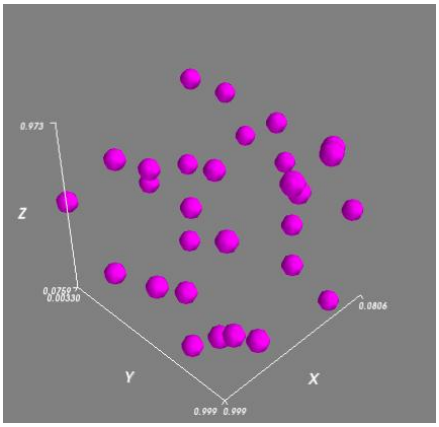
color = (R, G, B)

0.0 <= R, G, B <= 1.0

default is (1.0, 1.0, 1.0)

mode = 'sphere', 'cone', 'cube', 'arrow', 'cylinder', 'point', '2darrow', '2dcircle', '2dcross', '2ddash', '2ddiamond', '2dhooked_arrow', '2dsquare', '2dthick_arrow', '2dthick_cross', '2dtriangle', '2dvertex'

```
from numpy.random import rand
x,y,z = rand(30),rand(30),rand(30)
mlab.axes()
```



ENTHOUGHT

Lines in 3D

mlab.plot3d(x, y, z, color=(0.0,1.0,1.0), tube_radius=0.03)

1-d arrays giving end-points of connected line-segments

len(x) == len(y) == len(z)

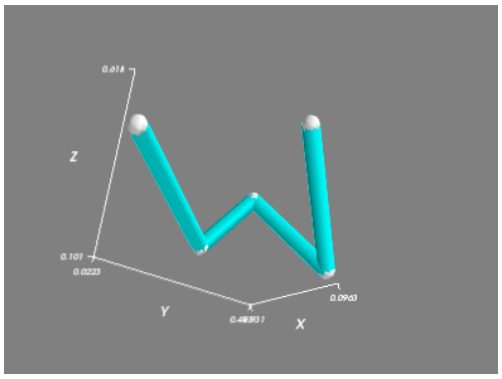
color = (R, G, B)

0.0 <= R, G, B <= 1.0

default is (1.0, 1.0, 1.0)

A float giving the radius of the tubes representing the lines.
default is 0.025

```
from numpy.random import rand
x,y,z = rand(5), rand(5), rand(5)
mlab.points3d(x,y,z,scale_factor=0.06)
mlab.axes()
```



ENTHOUGHT

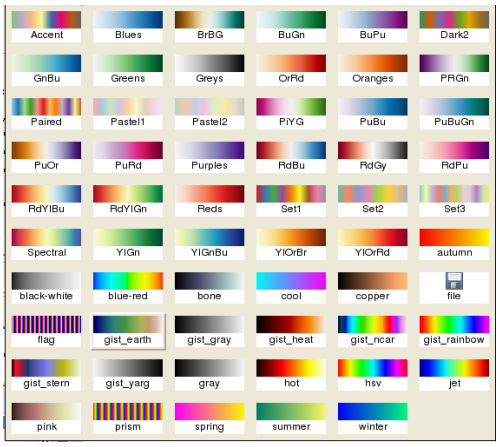
Surfaces in 3D

mlab.surf(z, colormap='RdBu') or mlab.surf(x, y, z, colormap='RdBu')

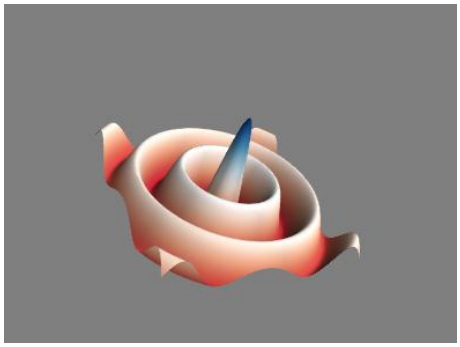
2d array of "heights"

2d array of "heights"

1d (or 2d) arrays producing a (possibly non-uniform) orthogonal grid (such as returned from ogrid or mgrid)




```
from numpy import ogrid, hypot
from scipy.special import j0
x,y = ogrid[-15:15:100j, -15:15:100j]
z = j0(hypot(x,y))
```



ENTHOUGHT

Volumes in 3D

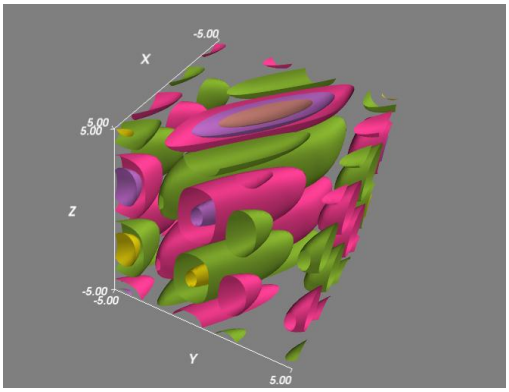
mlab.contour3d(data, contours=6, colormap='Dark2')

3d array of volume data

Integer number of contours or

List of specific contours


```
from numpy import ogrid, tanh, cos
from scipy.special import j0
x,y,z = ogrid[-5:5:64j, -5:5:64j,
              -5:5:64j]
data = cos(0.5*y)*j0(x+y*1.3)*sin(0.8*z)

mlab.axes(ranges=[-5,5]*3)
```



ENTHOUGHT

Vector Fields in 3D – quiver3d

mlab.quiver3d(u, v, w) or mlab.quiver3d(x, y, z, u, v, w)

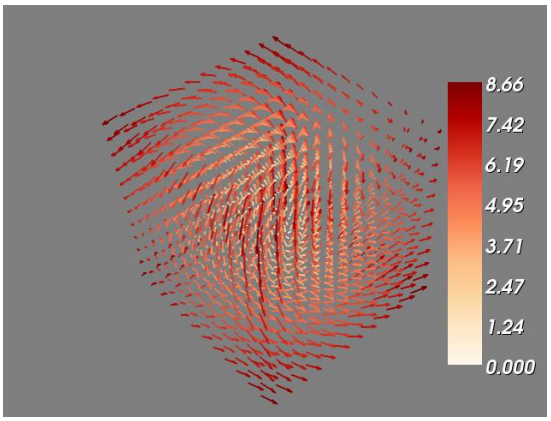
3-d arrays each representing a component of a 3-d vector field. All must be the same shape.

1, 2, or 3-d arrays of the same shape as u, v, and w providing real locations of the vectors.

1, 2, or 3-d arrays of the same shape as x, y, and z representing components of the vectors.

any colormap argument will color glyphs according to magnitude of vector.

```
from numpy import mgrid
x,y,z = mgrid[-5:5:12j, -5:5:12j, -5:5:12j]
u,v,w = -z, y, x
mlab.quiver3d(u,v,w, colormap='OrRd')
```



ENTHOUGHT

Vector Fields in 3D – flow

mlab.flow(u, v, w, linetype='tube', seedtype='sphere', colormap='PuRd')

3-d arrays representing vector field. Can also be called with `x,y,z,u,v,w` where x, y, and z are all 3-d arrays.

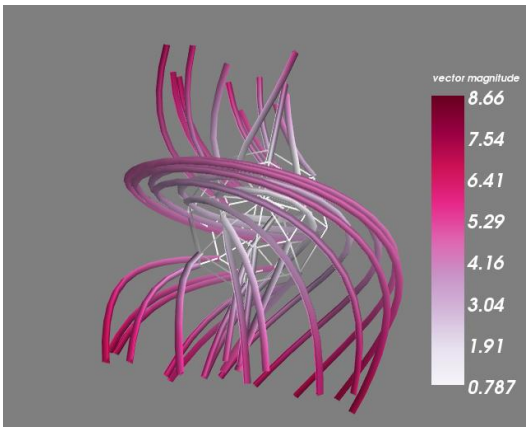
linetype= 'line', 'ribbon', or 'tube'

seedtype = 'sphere', 'plane', 'line', or 'point'

streamlines are calculated from seedpoints spaced throughout the specified seed widget. The seed widget is shown and can be interacted with to compute new streamlines.

any colormap argument will color according to magnitude of vector.

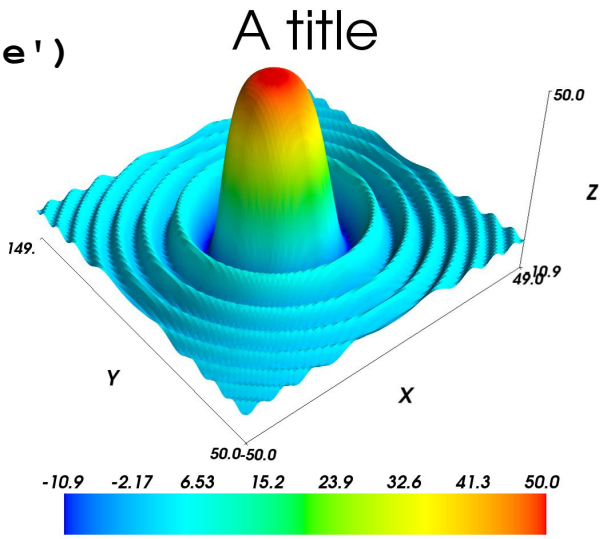
```
from numpy import mgrid
x,y,z = mgrid[-5:5:12j, -5:5:12j, -5:5:12j]
u,v,w = -z, y, x
```



ENTHOUGHT

Figure Decorations

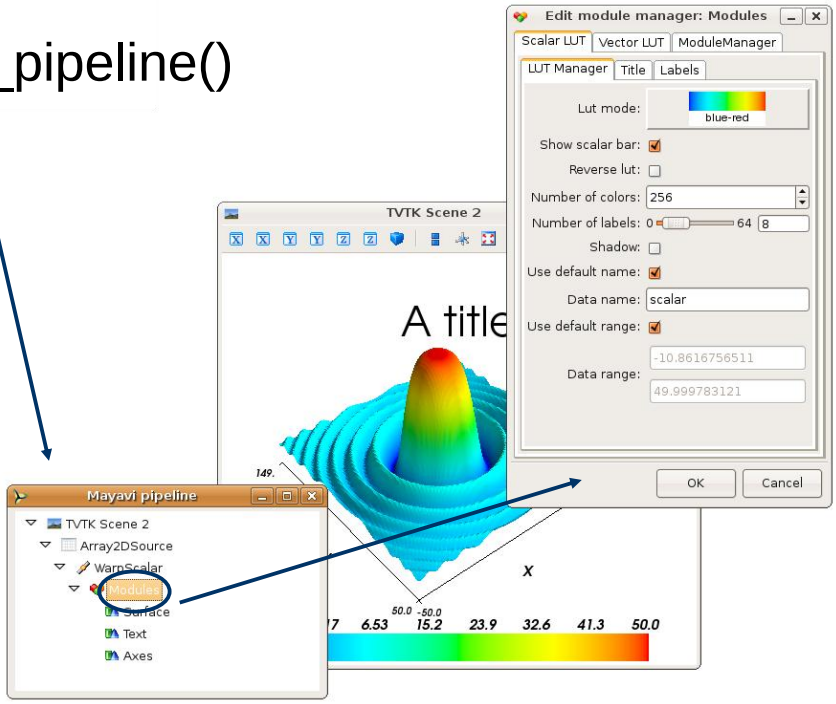
```
>>> mlab.title('A title')
>>> mlab.axes()
>>> mlab.colorbar()
>>> zlabel('Depth')

>>> mlab.clf()
>>> mlab.figure()
>>> mlab.gcf()
```



ENTHOUGHT

Graphical User Interface

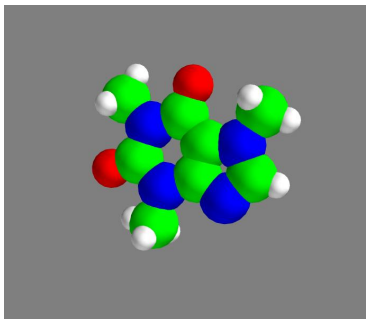
mlab.show_pipeline()



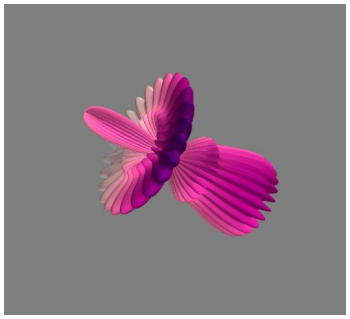
ENTHOUGHT

Examples and Demos

>>> mlab.test_<TAB>
mlab.test_points3d()
mlab.test_plot3d()
mlab.test_surf()
mlab.test_contour3d()
mlab.test_quiver3d()



mlab.test_molecule()
mlab.test_flow()
mlab.test_mesh()

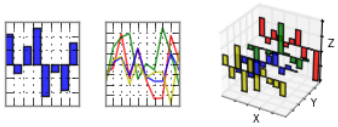


Use ?? in IPython to look at the source code of these examples.

Time-series management with

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas

Pandas (version 0.12.0, Jul 2013) wraps 1,2,3D Numpy arrays for timeseries functionalities.

Author: Wes McKinney, http://pandas.pydata.org/

License: BSD

GOALS

- New array-based data-structures with axis labeling + nice representation in ipython,
- Date/time management built-in, including timezones,
- Data alignment, data merging, data aggregation (group by), missing data management,
- Statistical tools (describe, moving stats, covariance, correlation, panel regression),
- Easy visualization (line plot, bar chart, boxplot, scatter plot matrix, ...) with Matplotlib.

RELATED PROJECTS

- Built on top of: `Numpy`, `pytables`, `matplotlib`.
- Replaces `scikits.timeseries`. Is now an optional dependency of `statsmodels`.
- Offers similar features to `larry`, `datarray` (labeled arrays).

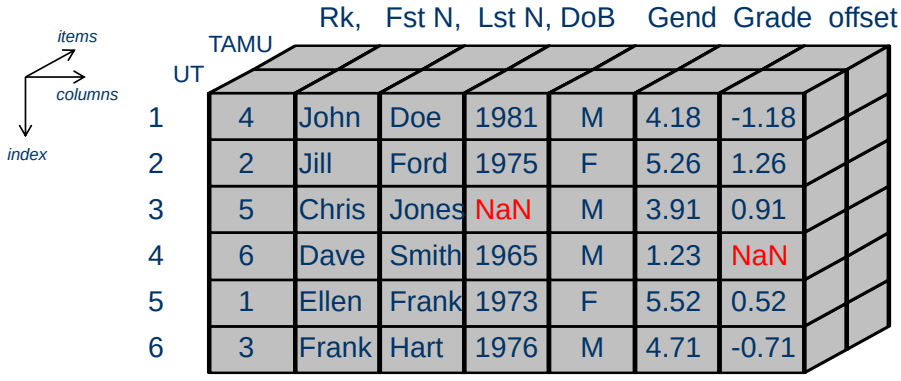
ENTHOUGHT

New Data-structures

PANDAs = PANel Data = multi-dimensional data in stats & econometrics.

Data-structures:

- A Series = 1D data-structure. Derives from np.ndarray & generic panda object (index).
- A DataFrame = 2D data-structure and can be viewed as a dict of series. It is NOT a subclass of an numpy array.
- A Panel is a is a 3D data-structure and can be viewed as a dict of DataFrames.



ENTHOUGHT

Creating (Time)Series

FROM LIST AND DICT

```
# Data and corresponding indices can
# be stored in lists.
>>> index = ['a', 'b', 'c', 'd']
>>> Series(range(4), index=index,
           name = 'first series')
a    0
b    1
c    2
d    3
Name: first series
# data + indices in a dict
>>> d = {'a':0,'b':1,'c':2,'d':1}
>>> s=Series(d, name='first series')
>>> print s.name
'first series'
>>> s.name = ''
>>> print s.index
Index([a, b, c, d], dtype=object)
>>> print s.values, type(s.values)
array([0, 1, 2, 1], dtype=int64)
numpy.ndarray
>>> s.dtype
dtype('int64')
```

FROM A NUMPY ARRAY

```
>>> from numpy.random import randn
>>> Series(randn(4), index=index)
a   -1.062984
b   -0.961625
c   -0.720323
d    1.100681
```

ACCESS ELEMENTS

```
# Access elements like an array
>>> s[2]
2
>>> s[:2]
a    0
b    1
# Access elements like a dictionary
>>> print s['c'], 'c' in s
2 True
# Slicing works on indices!
>>> s['a':'c']
a    0
b    1
c    2
```

ENTHOUGHT

Creating DataFrames

FROM A DICT OF SERIES

```
# DF from a dict of series: keys are
# column names.
>>> s2=Series(randn(3), name='B',
              index=index[1:])
>>> d = {'A':s, 'B':s2}
>>> df = DataFrame(d)
   A         B
a  0       NaN
b  1 -0.961625
c  2 -0.720323
d  1  1.100681
>>> df.index, df.columns
Index([a, b, c, d], dtype=object)
Index([A, B], dtype=object)
>>> df.shape, df.dtypes
(4,2)
A          int64
B        float64
>>> df.values
array([[ 0.        ,         nan],
       [ 1.        , -0.96162549],
       [ 2.        , -0.72032263],
       [ 1.        , 1.100680533]])
```

FROM A NUMPY ARRAY

```
>>> DataFrame(randn(4,4), index=index,
    columns=['A','B','C','D'])
          A        B        C        D
a  0.28164 -0.36826  0.04011  1.25030
b -0.71049 -1.23956 -0.08504 -0.08336
c -1.29446  0.70709  1.39642  0.49035
d  0.74632 -0.03512 -0.69237  0.81488
# List(series) interpreted as an array
>>> df2 = concat([s,s2], axis=1,
    keys=['A', 'B'])
```

ACCESS OR ADD ELEMENTS

```
>>> col1 = df['A']
>>> col2 = df.B
>>> df['flag'] = df['B'] > 0
>>> df.ix['c']
A              2
B      -0.7203226
flag        False
Name: c
>>> df.ix['c','B']
-0.7203226322119064
```


ENTHOUGHT

Creating Panels

FROM A DICT OF DATAFRAMES

```
# Panel from a dict of dataframes:
# keys used for third 'items' axis.
>>> df2=df.ix[::2,:]
>>> data = {'item1': df,'item2': df2}
>>> wp = Panel(data)

>>> wp.items, wp.major_axis,
wp.minor_axis
Index([item1, item2], dtype=object)
Index([a, b, c, d], dtype=object)
Index([A, B, flag], dtype=object)
>>> wp.shape
(2, 4, 3)
>>> wp.values
array([[[0, nan, False],
        [1, -0.96162549, False],
        [2, -0.72032263, False],
        [1, 1.100680533, True]],
       [[0.0, nan, False],
        [nan, nan, nan],
        [2.0, -0.72032263, False],
        [nan, nan, nan]]],
dtype=object)
```

FROM A NUMPY ARRAY

```
>>> items = ['foo','bar','baz','biz']
>>> Panel(randn(4,4,4), items=items)
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 4 (major) x 4
(minor)
Items: foo to biz
Major axis: 0 to 3
Minor axis: 0 to 3
```

ACCESS OR ADD ELEMENTS

```
>>> df1 = wp['item1']
   A          B  flag
a  0        NaN  False
b  1 -0.9616255  False
c  2 -0.7203226  False
d  1   1.100681   True
>>> df2 = wp.item2
>>> del wp['item1']
>>> wp.ix['item2','c','A']
2.0
```

ENTHOUGHT

Pandas IO

READING FUNCTIONS

Format	Method, Function, Class
txt, csv	read_table, read_csv
clipboard	read_clipboard
pickle	read_pickle
HDF5	read_hdf, HDFStore
Excel	read_excel, ExcelFile
R	pandas.rpy.common.load_data

WRITING FUNCTIONS

Format	Method, Function, Class
txt, csv	to_string, to_csv
html	to_html
pickle	to_pickle
HDF5	to_hdf, HDFStore
Excel	to_excel, ExcelWriter

EXAMPLES

```
# Pickle is supported
>>> df.to_pickle('foo.pickle')
>>> read_pickle('foo.pickle')


# To load a table from an ascii file
df = read_table('foo.txt', sep=',',
header = 1, skiprows=2, index_col=0,
names=['Paris', 'NY', 'Tokyo',
'London'], parse_dates=True,
na_values = [-9999])


# Long term storage: HDF5
>>> st = HDFStore('foo.h5')
>>> st['data1'] = df
>>> st['ser1'] = s
>>> s2 = st['ser1']
>>> st.close()
```


ENTHOUGHT

Indexes and data alignment

RE-INDEXING

```
# reindex shuffles the existing index
>>> s.reindex(['c', 'b', 'a', 'e'])
c     2
b     1
a     0
e   NaN
# The index attr can be overwritten
>>> s.index = ['p', 'q', 'r', 's']
# 'rename' modifies index label
>>> s.rename(lambda a: a.upper())
P    0
Q    1
R    2
S    3
# Sort by values
>>> s.values[:] = [5, 3, 0, 2]
>>> s.order()
r   0
s   2
q   3
p   5
# Explicit alignment of BOTH series
>>> s,s2 = s.align(s2)
```

ALIGNMENT FROM OPERATIONS

```
# Operations automatically align on
# the index (different from ndarray)
>>> s[1:] + s[:-2]
a    NaN
b      1
c      2
d    NaN
e    NaN

# Sort a DF by a (list of) column(s)
>>> df.sort('A')
   A         B  flags
a  0       NaN  False
b  1 -0.961625  False
d  1  1.100681   True
c  2 -0.720323  False
```

ENTHOUGHT

More on indexing

INDEX TO/FROM A COLUMN

```
# Any dataframe column can become the
# index
>>> df
   A         B
a  0       NaN
b  1 -0.961625
c  2 -0.720323
d  3  1.100681
>>> df2 = df.set_index('A')
          B
A
0       NaN
1 -0.961625
2 -0.720323
3  1.100681
# The opposite operation converts the
# index to a column
>>> df2.reset_index()
   A         B
0  0       NaN
1  1 -0.961625
2  2 -0.720323
3  3  1.100681
```

REDUNDANT INDEXES

```
# Although dangerous, it is possible
# to have redundant indices
>>> index = ['a', 'b', 'c', 'c']
>>> s=Series(range(4), index=index)
a    0
b    1
c    2
c    3
>>> s['c']
c    2
c    3
>>> type(s['c'])
pandas.core.series.Series
```


ENTHOUGHT

Multi-indexed pandas

MULTI-INDEXING SERIES

```
>>> ind1=['bar', 'bar', 'baz', 'baz']
>>> ind2=['one', 'two', 'one', 'two']
>>> ind = [ind1,ind2]
>>> s = Series(randn(4), index=ind)
bar  one    0.03
     two    0.80
baz  one    0.56
     two   -2.08
>>> s.index
MultiIndex
[('bar', 'one') ('bar', 'two') ('baz',
'one') ('baz', 'two')]
>>> s['bar']
one    0.03
two    0.80
>>> s.reindex([('baz', 'two'),
               ('bar', 'one')])
baz  two   -2.08
bar  one    0.03
```

MULTI-INDEXING DATAFRAMES

```
>>> df = DataFrame(randn(4,2),
                   index=ind)
             0     1
bar one   0.24  0.81
    two  -0.96  0.49
baz one   1.23  0.29
    two  -0.97  1.18
>>> df.ix['bar']
         0     1
one   0.24  0.81
two  -0.96  0.49
>>> df.ix['bar', 'one']
0    0.24
1    0.81
Name: one
# Partial slicing works!
>>> df.ix[('bar','two'):('baz','two')]
             0     1
bar two  -0.96  0.49
baz one   1.23  0.29
    two  -0.97  1.18
```

ENTHOUGHT

Multi-indexed pandas II

MANIPULATING INDEXES

```
# By default each index is identified
# by a level 0 (outer), 1 (inner)
>>> s.swaplevel(0,1)
one     bar      0.03
two     bar      0.80
one     baz      0.56
two     baz     -2.08
# reorder_levels generalizes that
>>> s.reorder_levels([1,0])
(...)
>>> s.index.names=['first','second']
>>> s
first  second
bar    one       0.03
       two       0.80
baz    one       0.56
       two      -2.08
```

SLICING

```
>>> df.index.names=['first','second']
>>> df.xs("one", level="second")
           0     1
first
bar     0.24  0.81
baz     1.23  0.29
```

SORTING

```
>>> s.sortlevel(level='second')
bar    one       0.03
baz    one       0.56
bar    two       0.80
baz    two      -2.08
```


ENTHOUGHT

Computation with pandas

SERIES

```
# Series are subclasses of ndarrays:
# computations are element by element
>>> i = ['a', 'b', 'c', 'd']
>>> s=Series(range(4), index=i)
>>> s+1
a    1
b    2
c    3
d    4
# All Numpy operators can be applied
>>> s2 = np.exp(s)
>>> s = s.astype(np.float32)
# including fancy indexing with masks
>>> s[s > 1.5] = 1.5
```

DATAFRAMES

```
>>> df
   A      B  flags
a  0    NaN  False
b  1   -0.9  False
c  2   -0.7  False
d  1    1.1   True
```

```
# Dataframe computations are applied
# columns by columns
>>> df2 = df+1
# Adding a series or re-scaling
>>> row = df.ix[1]
>>> df-row
    A    B   flag
a  -1  NaN  False
b   0    0  False
c   1  0.2  False
d   0  2.0   True
# This would have been equivalent
>>> rescaled = df.sub(row, axis=0)
# Adding another dataframe can be done
# with a filler.
>>> df1.add(df2, fill_value = 0)
# 'apply' a custom function to columns
>>> f = lambda x: x.max()-x.min()
>>> df.apply(f, axis = 0)
A     2
B     2
C  True
```

ENTHOUGHT

Statistical Analysis

DESCRIPTIVE STATS

```
# Descriptive stats available:
# count, sum, mean, median, min, max,
# abs, prod, std, var, skew, kurt,
# quantile, cumsum, cumprod, cummax
# Stats on DF are column per column
>>> print df.mean(), df.mean(axis=1)
A       1.000000      a   -0.354328
B      -0.227542      b    0.500000
flag    0.250000      c    0.426559
                      d    1.033560
# Stats func accept a level as kw arg
# applicable for multi-index pandas
>>> df.describe()
              A         B
count  4.000000  3.000000
mean   1.000000 -0.227542
std    0.816497  1.162964
min    0.000000 -1.062984
25%    0.750000 -0.891653
50%    1.000000 -0.720323
75%    1.250000  0.190179
max    2.000000  1.100681
>>> pandas.rolling_<TAB>
```

CORRELATIONS & REGRESSION

```
>>> cov = df.cov()
>>> ts.corr(ts2, method = 'kendall')
0.066666666666666693
>>> ols(y = ts2, x = ts)
Formula: Y ~ <x> + <intercept>
Number of Observations:         20
Number of Degrees of Freedom:   2
R-squared:         0.9936
Adj R-squared:     0.9932
Rmse:              0.0582
F-stat(1,18): 2772.7496, p-value: 0.0
Degrees of Freedom: model 1, resid 18
(...)
```

For more stats, see `statsmodels`.

ENTHOUGHT

Dealing with missing data

FIND & REMOVE/REPLACE NaN

```
# A boolean mask with 'null' values:
# None, np.NaN, np.inf, -np.inf
>>> np.all(isnull(s) | s.notnull())
True
# Replace missing values manually
>>> s[isnull(s)] = 0.
# Automatic version with forward fill
>>> s.fillna(method = 'ffill')
# Inverse operation
>>> s[s == -9999] = np.nan
# Remove all entries w/ missing values
>>> df.dropna(how='all')
# Interpolation also removes NaN
>>> s.interpolate()
```

SPARSE DATASETS

```
# If mostly missing data, convert to
# SparseSeries, SparseDataFrame, ...
# Only stores non-null values & loc
>>> print len(s),s.count(),s.nbytes
1000, 10, 8000
>>> sp_s = s.to_sparse()
>>> print sp_s.nbytes
80
```

MERGING DATASETS

```
# Combining overlapping datasets with
# a priority
>>> df1 = DataFrame({
    'A' : [1.,nan,3.,5.,nan],
    'B' : [nan,2.,3.,nan,6.]})
>>> df2 = DataFrame({
    'A' : [5.,2.,4.,nan,3.,7.],
    'B' : [nan,nan,3.,4.,6.,8.]})
>>> df1.combine_first(df2)
   A   B
0  1 NaN
1  2   2
2  3   3
3  5   4
4  3   6
5  7   8
# More custom merging to be
# implemented using combine and a
# custom function
```

ENTHOUGHT

Pivot tables and reshaping

PIVOTING

```
# Repeating columns can be viewed as
# an additional axis
>>> df
```

```
         date variable     value
0  2000-01-03        A  0.469112
1  2000-01-04        A -0.282863
2  2000-01-05        A -1.509059
3  2000-01-03        B -1.135632
4  2000-01-04        B  1.212112
5  2000-01-05        B -0.173215
```

```
>>> df.pivot(index='date',
  columns='variable', values='value')
```

```
variable           A         B
date
2000-01-03  0.469112 -1.135632
2000-01-04 -0.282863  1.212112
2000-01-05 -1.509059 -0.173215
```

(UN-)STACKING

```
# Stacking a DF creates a
# series with multiple indexes made
# from the column names.
>>> s
```

```
     a  b
one  1  2
two  3  4
```

```
>>> stacked = s.stack()
```

```
one a    1
    b    2
two a    3
    b    4
```

```
# Opposite operation unstacks the
# last level by default. If more cplx
# DF, can specify a level to unstack
>>> stacked.unstack()
```

```
     a  b
one  1  2
two  3  4
```

```
>>> stacked.unstack(level=0)
```

```
  one   two
a   1     3
b   2     4
```


ENTHOUGHT

Pivot tables and reshaping II

PIVOT_TABLE

```
# Another way to reshape a DF and
# aggregate at the same time
>>> df
```

```
   A   B   C      D
0  foo one small  1
1  foo one large  2
2  foo one large  2
3  foo two small  3
4  foo two small  3
5  bar one large  4
6  bar one small  5
7  bar two small  6
8  bar two large  7
```

```
>>> table=pivot_table(df, values='D',
          rows=['A', 'B'], cols=['C'],
          aggfunc=np.sum)
>>> table
```

```
          small  large
foo  one  1      4
     two  6      NaN
bar  one  5      4
     two  6      7
```

ENTHOUGHT

Data aggregation

Aggregation in a dataset is done in 3 steps: Split > Apply > Combine

SPLIT WITH groupby

```
# Group data by one column's value
>>> gps = df.groupby('flags')
# gps = iterator of tuples with
# group name and sub part of df
>>> gps.groups
{False: ['a', 'b', 'c'], True:
['d']}
# groupby can also take a custom
# function that acts on index labels
>>> df = df.reset_index()
>>> myfunc = lambda x: x%2 ==0
>>> gps2 = df.groupby(myfunc)
# Series can also be grouped by
>>> df['A'].groupby(myfunc)
# which is identical to
>>> gps2['A']
```

APPLY WITH aggregate()

```
# Values in groups can be aggregated
>>> gps.sum()
       A    B
flags
False  3 -1.6
True   1  1.1
# Version more flexible but slower
>>> gps.aggregate(np.sum)
# agg is even more flexible
>>> gps.agg([np.mean, np.std])
>>> gps.agg({'A':'sum','B':'std'})
       A         B
flags
False  3  0.141421
True   1       NaN
```


ENTHOUGHT

Data aggregation II

APPLY WITH transform()

```
# Values in groups can be
# transformed (length conserved)
>>> f=lambda x: x - x.mean()
>>> gps.transform(f)
       A     B
a     -1   NaN
b      0  -0.1
c      1   0.1
d      0   0.0
```

APPLY WITH apply()

```
# Computations from values in
# groups can be turned into a DF
# of calcs
>>> desc = lambda x: x.describe()
>>> gps['A'].apply(desc).unstack()
       count mean std  min  25% 50% 75% max
flags
False      3    1   1    0  0.5   1 1.5   2
True       1    1 NaN    1  1.0   1 1.0   1

>>> def f(group):
    return DataFrame({'original':group,
          'demeaned': group - group.mean()})
>>> gps['A'].apply(f)
       demeaned  original
index
a            -1         0
b             0         1
c             1         2
d             0         1
```

ENTHOUGHT

Dealing with date & time

CREATING DATE/TIME INDEXES

```
# The index can be a list of
# dates+times locations that can be
# automatically generated
>>> date_range('1/1/2000',periods=4)
<class
'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-04
00:00:00]
Length: 4, Freq: D, Timezone: None
# Specify frequency: us,ms,S,T,H,D,B,
# W,M,3min, 2h20min, 2W,...
>>> r=date_range('1/1/2000',periods=72,
          freq='H')
>>> i=date_range('1/1/2000',periods=4,
        freq=datetools.YearEnd())
>>> i=date_range('1/1/2000',periods=4,
             freq='3min')
>>> ts=Series(range(4), index = i)
2000-01-01 00:00:00    0
2000-01-01 00:03:00    1
2000-01-01 00:06:00    2
2000-01-01 00:09:00    3
Freq: 3T
```

UP-/DOWN-SAMPLING

```
>>> ts.resample('T')
2000-01-01 00:00:00     0
2000-01-01 00:01:00   NaN
2000-01-01 00:02:00   NaN
2000-01-01 00:03:00     1
2000-01-01 00:04:00   NaN
2000-01-01 00:05:00   NaN
2000-01-01 00:06:00     2
2000-01-01 00:07:00   NaN
2000-01-01 00:08:00   NaN
2000-01-01 00:09:00     3
Freq: T
# Group hourly data into daily
>>> ts2 = Series(randn(72),index=r)
>>> ts2.resample('D', how='mean',
         closed='left', label='left')
01-Jan-2000    0.397501
02-Jan-2000    0.186568
03-Jan-2000    0.327240
Freq: D
>>> my_avg = lambda x: x[1:-1].mean()
>>> ts3=ts2.resample('D', how=my_avg)
```


ENTHOUGHT

Dealing with date & time II

RANGES OF PERIODS

```
# The index can be a list of
# dates/times intervals
>>> PeriodIndex(['2011-1', '2011-2',
'2011-3', '2011-4'], freq='M')
<class 'tseries.period.PeriodIndex'>
freq: M
[Jan-2011, ..., Apr-2011]
length: 4
# They can be automatically generated
>>> pr = period_range('1/1/2000',
             periods=4, freq = 'M')
freq: M
[Jan-2000, ..., Apr-2000]
length: 4
>>> ps=Series(range(4), index = pr)
Jan-2000    0
Feb-2000    1
Mar-2000    2
Apr-2000    3
Freq: M
```

CONVERT TIMESTAMP<->PERIOD

```
>>> ts = ps.to_timestamp()
2000-01-31    0
2000-02-29    1
2000-03-31    2
2000-04-30    3
Freq: M
>>> all(ts.to_period() == ps)
True
```

TIMEZONES

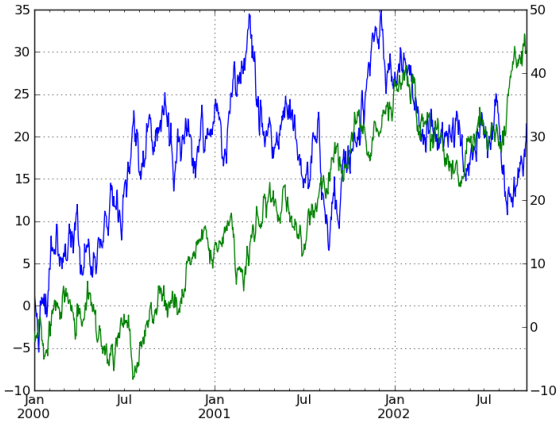
```
# Timestamps/pandas can be tz aware
>>> ts_utc = ts.tz_localize('UTC')
>>>
ts_us=ts_utc.tz_convert('US/Eastern')
2012-01-30 19:00:00-05:00   -0.425792
2012-02-28 19:00:00-05:00    0.788903
2012-03-30 20:00:00-04:00    0.009502
2012-04-29 20:00:00-04:00    1.775263
2012-05-30 20:00:00-04:00   -1.656727
Freq: M
>>> ts.index == ts_us.index
True
```

ENTHOUGHT

Visualizing (Time)series

```
>>> ts.plot()
```


```
>>> ts2.plot(secondary_y=True,
style='g')
```



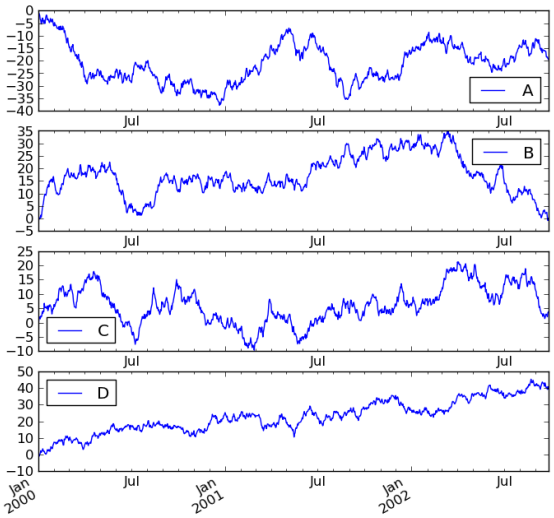
See also

tools.plotting.lag_plot,
tools.plotting.autocorrelation_plot

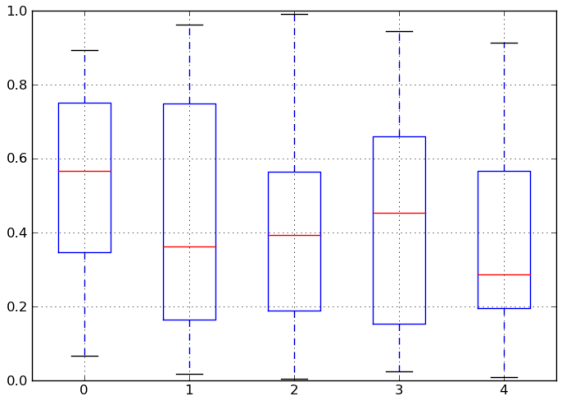
ENTHOUGHT

Visualizing DataFrames

```
>>> plt.figure()
>>> df.plot(subplots=True)
>>> plt.legend(loc='best')
```




```
>>> df.boxplot()
```



See also:

plot with kind='bar','barh', 'kde' keywd

hist method

tools.plotting.scatter_matrix,
andrews_curves, lag_plot

ENTHOUGHT

Extension Modules

Interfacing with other languages

- C/C++ Integration
  - SWIG www.swig.org
  - ctypes Python standard library
  - Cython www.cython.org
  - boost www.boost.org/libs/python/doc/index.html
  - weave www.scipy.org/site_content/weave
- FORTRAN Integration
  - f2py www.scipy.org/F2py

ENTHOUGHT

Cython

ENTHOUGHT

What is Cython?

Cython is a Python-like language for writing extension modules. It allows one to mix Python with C or C++ and significantly lowers the barrier to speeding up Python code.

The **`cython`** command generates a C or C++ source file from a Cython source file; the C/C++ source is then compiled into a heavily optimized extension module.

Cython has built-in support for working with NumPy arrays and Python buffers, making numerical programming nearly as fast as C and (nearly) as easy as Python.

`http://www.cython.org/`

ENTHOUGHT

A really simple example

PI.PYX

```
# Define a function. Include type information for the argument.
def multiply_by_pi(int num):
   return num * 3.14159265359
```

SETUP_PI.PY

```
# Cython has its own "extension builder" module that knows how
# to build cython files into python modules.
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext = Extension("pi", sources=["pi.pyx"])

setup(ext_modules=[ext],
      cmdclass={'build_ext': build_ext})
```

See demo/cython for this example.
Build it using build.bat.

ENTHOUGHT

A simple Cython example

CALLING MULTIPLY_BY_PI FROM PYTHON

```
$ python setup_pi.py build_ext --inplace -c mingw32

>>> import pi
>>> pi.multiply_by_pi()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> pi.multiply_by_pi("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
>>> pi.multiply_by_pi(3)
9.4247779607700011
```


ENTHOUGHT

(some of) the generated code

C CODE GENERATED BY CYTHON

```
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds); /*proto*/
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds) {
  int __pyx_v_num;
  PyObject *__pyx_r;
  PyObject *__pyx_1 = 0;
  static char *__pyx_argnames[] = {"num",0};
  if (!PyArg_ParseTupleAndKeywords(__pyx_args, __pyx_kwds, "i",
__pyx_argnames,
&__pyx_v_num)) return 0;

  /* "C:\pi.pyx":2 */
  __pyx_1 = PyFloat_FromDouble((__pyx_v_num * 3.14159265359));
if (!__pyx_1) {__
pyx_filename = __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}
  __pyx_r = __pyx_1;
  __pyx_1 = 0;
```

ENTHOUGHT

Def vs. CDef

DEF — PYTHON FUNCTIONS

```
# Python callable function.
def inc(int num, int offset):
    return num + offset

# Call inc for values in sequence.
def inc_seq(seq, offset):
    result = []
    for val in seq:
        res = inc(val, offset)
        result.append(res)
    return result
```

INC FROM PYTHON

```
# inc is callable from Python.
>>> inc.inc(1,3)
4
>>> a = range(4)
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

CDEF — C FUNCTIONS

```
# cdef becomes a C function call.
cdef int fast_inc(int num,
                  int offset):
    return num + offset
# fast_inc for a sequence
def fast_inc_seq(seq, offset):
    result = []
    for val in seq:
        res = fast_inc(val, offset)
        result.append(res)
    return result
```

FAST_INC FROM PYTHON

```
# fast_inc not callable in Python
>>> inc.fast_inc(1,3)
Traceback: ... no 'fast_inc'
# But fast_inc_seq is 2x faster
# for large arrays.
>>> inc.fast_inc_seq(a, 3)
[3,4,5,6]
```


ENTHOUGHT

CPdef: combines def + cdef

CPDEF — C AND PYTHON FUNCTIONS

```
# cdef becomes a C function call.
cpdef fast_inc(int num, int offset):
    return num + offset

# Calls compiled version inside Cython file
def inc_seq(seq, offset):
    result = []
    for val in seq:
        res = fast_inc(val, offset)
        result.append(res)
    return result
```

FAST_INC FROM PYTHON

```
# fast_inc is now callable in Python via Python wrapper
>>> inc.fast_inc(1,3)
4
# No speed degradation here
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

ENTHOUGHT

Functions from C Libraries

EXTERNAL C FUNCTIONS

```
# len_extern.pyx
# First, "include" the header file you need.
cdef extern from "string.h":
    # Describe the interface for the functions used.
    cdef extern int strlen(char *c)

def get_len(char *message):
    # strlen can now be used from Cython code (but not Python)…
    return strlen(message)
```

CALL FROM PYTHON

```
>>> import len_extern
>>> len_extern.strlen
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'strlen'
>>> len_extern.get_len("woohoo!")
```


ENTHOUGHT

Structures from C Libraries

TIME_EXTERN.PYX

```
cdef extern from "time.h":
    # Describe the structure you are using.
    struct tm:
        ...
        int tm_mday # Day of the month: 1-31
        int tm_mon  # Months *since* january: 0-11
        int tm_year # Years since 1900
        ...
    ctypedef long time_t
    tm* localtime(time_t *timer)
    time_t time(time_t *tloc)

def get_date():
    """ Return a tuple with the current day, month, and year."""
    cdef time_t t
    cdef tm* ts
    t = time(NULL)
    ts = localtime(&t)
    return ts.tm_mday, ts.tm_mon + 1, ts.tm_year
```

CALLING FROM PYTHON

```
>>> extern_time.get_date()
(8, 4, 2011)
```

ENTHOUGHT

Classes

SHRUBBERY.PYX

```
cdef class Shrubbery:
  # Class level variables
  cdef int width, height

  def __init__(self, w, h):
    self.width = w
    self.height = h
  def describe(self):
    print "This shrubbery is",self.width,"by ",self.height," cubits."
```

CALLING FROM PYTHON

```
>>> import shrubbery
>>> x = shrubbery.Shrubbery(1, 2)
>>> x.describe()
This shrubbery is 1 by 2 cubits.
>>> print x.width
AttributeError: 'shrubbery.Shrubbery' object has no attribute 'width'
```


ENTHOUGHT

Classes from C++ libraries

rectangle_extern.h

```
class Rectangle {
   public:
       int x0, y0, x1, y1;
       Rectangle(int x0, int y0, int x1, int y1);
       ~Rectangle();
       int getLength();
       int getHeight();
       int getArea();
       void move(int dx, int dy);};
```

The implementation of the class and methods is done inside rectangle_extern.cpp. See demo/cython for this example.

ENTHOUGHT

Classes from C++ libraries

rectangle.pyx

```
cdef extern from "rectangle_extern.h":
    cdef cppclass Rectangle:
        Rectangle(int, int, int, int)
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int, int)

cdef class PyRectangle:
    cdef Rectangle *thisptr   # hold a C++ instance which we're wrapping
    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.thisptr = new Rectangle(x0, y0, x1, y1)
    def __dealloc__(self):
        del self.thisptr
    def getLength(self):
        return self.thisptr.getLength()
```


ENTHOUGHT

Classes from C++ libraries

SETUP.PY

```
from distutils.core import setup
from Cython.Distutils import build_ext
from distutils.extension import Extension
setup(
  ext_modules=[Extension("rectangle", sources = ["rectangle.pyx",
                                                 "rectangle_extern.cpp"],
                language = "c++")],
  cmdclass = {'build_ext': build_ext})
```

CALLING FROM PYTHON

```
In [1]: import rectangle
In [2]: r = rectangle.PyRectangle(1,1,2,2)  # calls __cinit__
In [3]: r.getLength()
Out[3]: 1
In [4]: r.getHeight()
AttributeError:rectangle.PyRectangle object has no attribute getHeight
In [5]: del r  # calls __dealloc__
```

ENTHOUGHT

Using NumPy with Cython

```
#cython: boundscheck=False
# Import the numpy cython module shipped with Cython.
cimport numpy as np
ctypedef np.float64_t DOUBLE

def sum(np.ndarray[DOUBLE] ary):
    # How long is the array in the first dimension?
    cdef int n = ary.shape[0]
    # Define local variables used in calculations.
    cdef unsigned int i
    cdef double sum
    # Sum algorithm implementation.
    sum = 0.0
    for i in range(0, n):
        sum = sum + ary[i]
    return sum
```

```
C:\demo\cython>test_sum.py
elements: 1,000,000
python sum(approx sec, result): 7.030
numpy sum(sec, result): 0.047
cython sum(sec, result): 0.047
```


ENTHOUGHT

Problem: Make This Fast!

```
def mandelbrot_escape(x, y, n):
    z_x = x
    z_y = y
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i

def generate_mandelbrot(xs, ys, n):
    d = empty(shape=(len(ys), len(xs)))
    for j in range(len(ys)):
        for i in range(len(xs)):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

ENTHOUGHT

Step 1: Add Type Information

Type information can be added to function signatures:

```
def mandelbrot_escape(double x, double y, int n):
    ...
def generate_mandelbrot(xs, ys, int n):
    ...
```

Variables can be declared to have a type using 'cdef':

```
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    ...
```


ENTHOUGHT

Step 2: Use Cython C Functions

In Cython you can declare functions to be C functions using 'cdef' instead of 'def':

```
cdef int mandelbrot_escape(float x, float y, int n):
    ...
```

This makes the functions:

- Generate actual C functions, so they are much faster.
- Not visible to Python, but freely usable in your Cython module.

Arbitrary Python objects can still be passed in and out of C functions using the 'object' type.

ENTHOUGHT

Solution 2: This is Fast!

```
cdef int mandelbrot_escape(double x, double y, int n):
    cdef double z_x = x
    cdef double z_y = y
    cdef int i
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```


ENTHOUGHT

Step 3: Use NumPy in Cython

In our example, we are still using Python-level numpy calls to do our array indexing:

```
d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
```

If we use the Cython interface to NumPy, we can declare our arrays to be C-level numpy extension types, and gain even more speed.

NumPy arrays are declared using a special buffer notation:

```
cimport numpy as np
...
cdef np.ndarray[int, ndim=2] my_array
```

You must declare both the type of the array, and the number of dimensions. All the standard numpy types are declared in the numpy cython declarations.

ENTHOUGHT

Solution 3: This is *Really* Fast!

mandel.pyx

```
cimport numpy as np
...
def generate_mandelbrot(np.ndarray[double, ndim=1] xs,
        np.ndarray[double, ndim=1] ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    cdef np.ndarray[int, ndim=2] d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

setup.py

```
...
import numpy
...
ext = Extension("mandel", ["mandel.pyx"],
    include_dirs = [numpy.get_include()])
```


ENTHOUGHT

Step 4: Parallelization using OpenMP

Cython supports native parallelism. To use this kind of parallelism, the Global Interpreter Lock (GIL) must be released. It currently supports OpenMP (more backends might be supported in the future).

1) Release the GIL before a block of code:

```
with nogil:
    # This block of code is executed after releasing the GIL
```

2) Declare that a cdef function can be called safely without the GIL:

```
cdef int mandelbrot_escape(double x, double y, int n) nogil:
    ...
```

3) Parallelize for-loops with prange:

```
from cython.parallel import prange
...
for j in prange(M):
    for i in prange(N):
        d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
```

ENTHOUGHT

Solution 4: Even faster!

mandel.pyx

```
from cython.parallel import prange
...
cdef int mandelbrot_escape(double x, double y, int n) nogil:
...
def generate_mandelbrot(np.ndarray[double, ndim=1] xs,
                        np.ndarray[double, ndim=1] ys, int n):
    """ Generate a mandelbrot set """
    cdef ...
    with nogil:
        for j in prange(M):
            for i in prange(N):
                d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

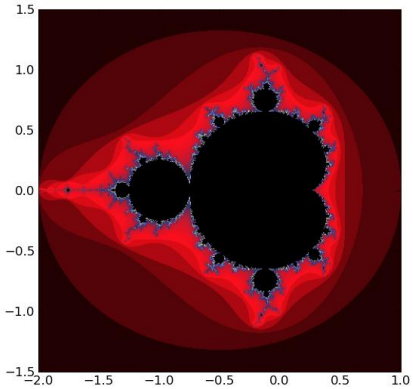
setup.py

```
ext = Extension("mandel", ["mandelbrot.pyx"],
                include_dirs = [numpy.get_include()],
                extra_compile_args=['-fopenmp'],
                extra_link_args=['-fopenmp'])
```


ENTHOUGHT

Conclusion

Solution	Time	Speed-up
Pure Python	630.72 s	x 1
Cython (Step 1)	2.7776 s	x 227
Cython (Step 2)	1.9608 s	x 322
Cython+Numpy (Step 3)	0.4012 s	x 1572
Cython+Numpy+prange (Step 4)	0.2449 s	x 2575



Timing performed on a 2.3 GHz Intel Core i7 MacBook Pro with 8GB RAM using a 2000x2000 array and an escape time of n=100.
[July 20, 2012]

ENTHOUGHT

Statistics with Python

ENTHOUGHT

Overview: scipy.stats

- *Probability distribution objects*
- *Random number generation*
- *Summary statistics*
- *Statistical tests*
- *Gaussian kernel density estimator (KDE)*

ENTHOUGHT

Continuous distributions

scipy.stats — CONTINUOUS DISTRIBUTIONS

over 80 continuous distributions!

METHODS

pdf entropy

cdf nnlf

rvs moment

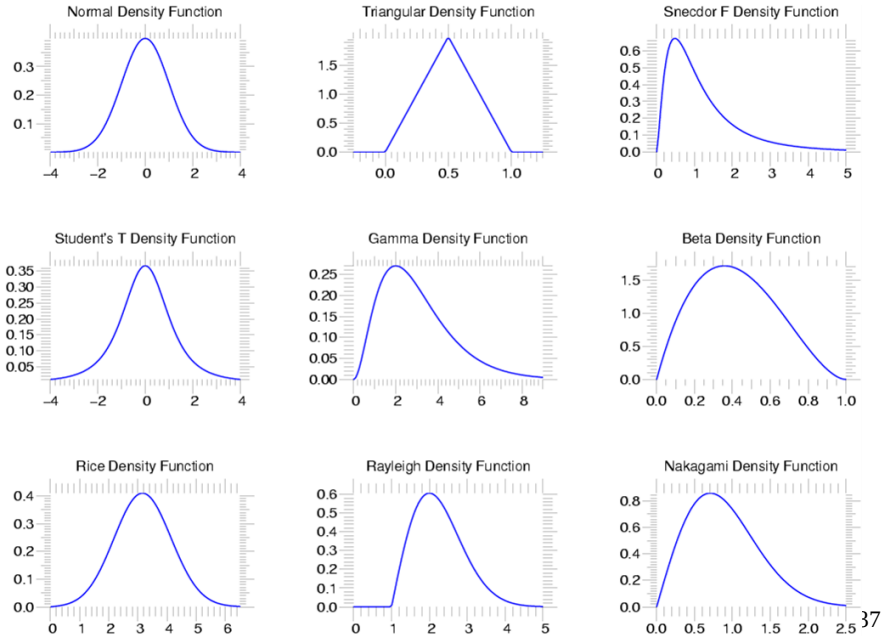
ppf freeze

stats

fit

sf

isf

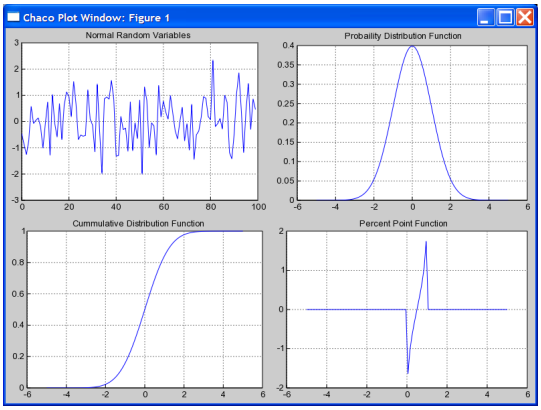


ENTHOUGHT

Using stats objects

DISTRIBUTIONS

```
>>> from scipy.stats import norm
# Sample normal dist. 100 times.
>>> samp = norm.rvs(size=100)

>>> x = linspace(-5, 5, 100)
# Calculate probability dist.
>>> pdf = norm.pdf(x)
# Calculate cumulative dist.
>>> cdf = norm.cdf(x)
# Calculate Percent Point Function
>>> ppf = norm.ppf(x)
```

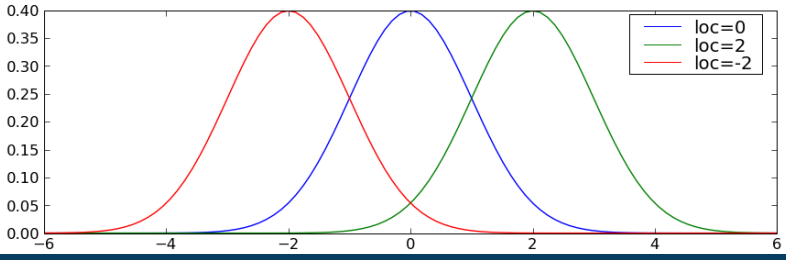


ENTHOUGHT

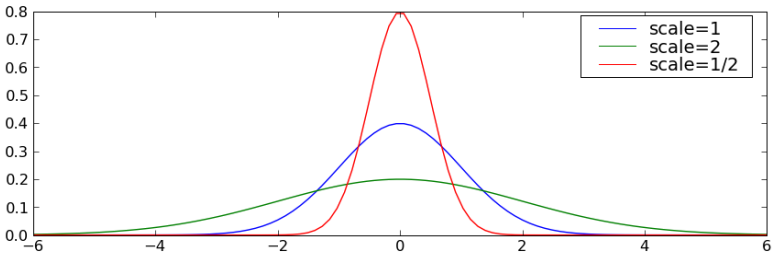
Distribution objects

Every distribution can be modified by `loc` and `scale` keywords
(many distributions also have required *shape* arguments to select from a family)

LOCATION (`loc`) --- shift left (<0) or right (>0) the distribution



SCALE (`scale`) --- stretch (>1) or compress (<1) the distribution

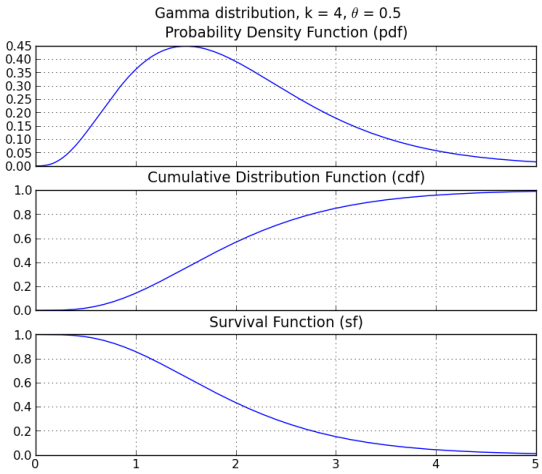


ENTHOUGHT

Distribution objects

Be careful! `loc` only makes sense for distributions defined on the entire real line. A gamma or lognormal distribution, for example, should not be shifted.

EXAMPLE - GAMMA



location	offset from zero (do not use)
scale	***θ***
shape	**k**

mean = kθ, variance = kθ^2

```
>>> from scipy.stats import gamma
>>> gm4 = gamma(4, scale=0.5)
>>> x = linspace(0, 5, 200)
>>> pdf = gm4.pdf(x)
>>> cdf = gm4.cdf(x)
>>> sf = gm4.sf(x)
```

ENTHOUGHT

Example distributions

NORM (norm) – N(μ,σ)

Only location and scale arguments:

location	mean	μ
scale	standard deviation	σ

LOG NORMAL (lognorm)

log(S) is N(μ, σ)

S is lognormal

one shape parameter!

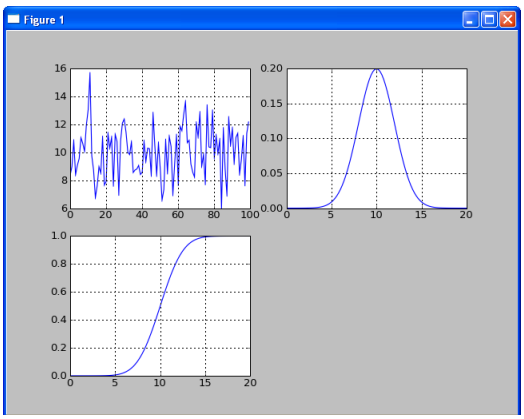
location	offset from zero (rarely used)
scale	e^{μ}
shape	σ

ENTHOUGHT

Setting location and scale

NORMAL DISTRIBUTION

```
>>> from scipy.stats import norm
# Normal dist with mean=10 and std=2
>>> dist = norm(loc=10, scale=2)

>>> x = linspace(-5, 15, 100)
# Calculate probability dist.
>>> pdf = dist.pdf(x)
# Calculate cumulative dist.
>>> cdf = dist.cdf(x)

# Get 100 random samples from dist.
>>> samp = dist.rvs(size=100)

# Estimate parameters from data
>>> mu, sigma = norm.fit(samp)
>>> print "%4.2f, %4.2f" % (mu, sigma)
10.07, 1.95
```



.fit returns best `shape + (loc, scale)` that explains the data

ENTHOUGHT

Continuous Distributions

norm	alpha	anglit	arcsine	beta	betaprime
bradford	burr	fisk	cauchy	chi	chi2
cosine	dgamma	dweibull	erlang	expon	exponweib
exponpow	fatiguelife	foldcauchy	f	foldnorm	frechet_r
frechet_l	genlogistic	genpareto	genexpon	genextreme	gausshyper
gamma	gengamma	genhalflogistic	gompertz	gumbel_r	gumbel_l
halfcauchy	halflogistic	halfnorm	hypsecant	invgamma	invnorm
invweibull	johnsonsb	johnsonsu	laplace	logistic	loggamma
loglaplace	lognorm	gilbrat	lomax	maxwell	mielke
nakagami	ncx2	ncf	t	nct	pareto
powerlaw	powerlognorm	powernorm	rdist	reciprocal	rayleigh
rice	recipinvgauss	semicircular	triang	truncexpon	truncnorm
tukeylambda	uniform	von_mises	wald	weibull_min	weibull_max
wrapcauchy	ksone	kstwobign			

ENTHOUGHT

Discrete Distributions

scipy.stats — Discrete Distributions

10 standard discrete distributions (plus any finite RV)

METHODS

pmf moment

cdf entropy

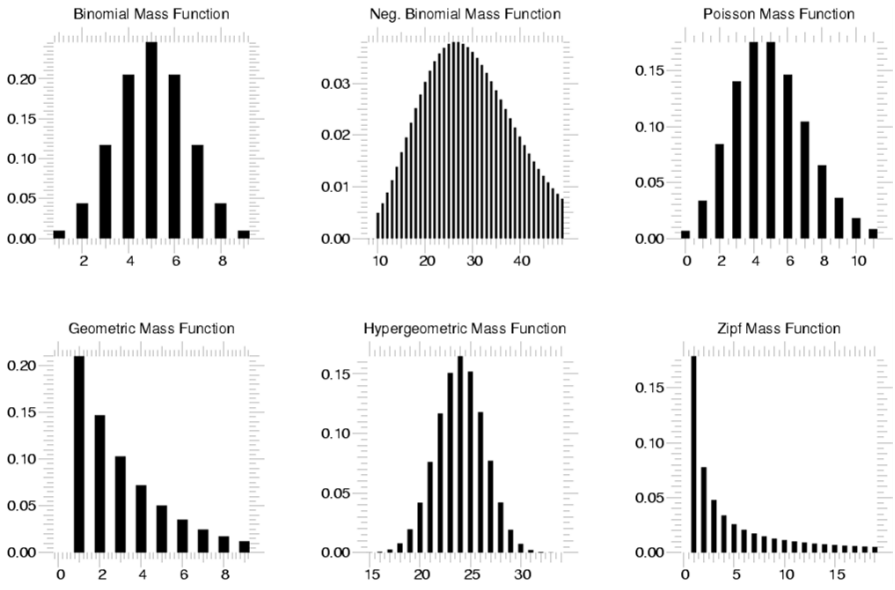
rvs freeze

ppf

stats

sf

isf



ENTHOUGHT

Discrete distributions

The discrete distributions:

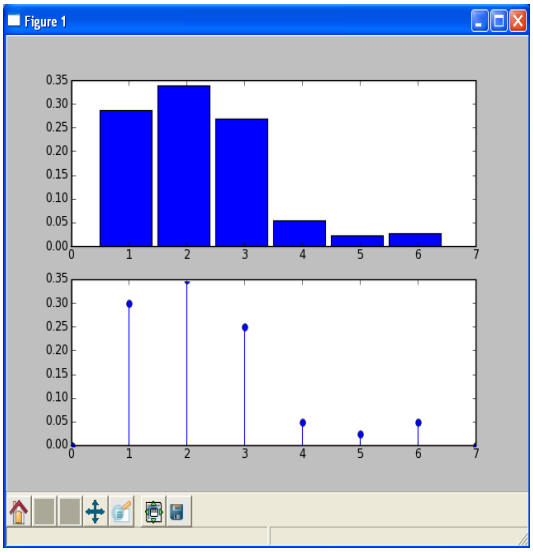
binom	bernoulli	nbinom	geom	hypergeom	logser
poisson	planck	boltzmann	randint	zipf	dlaplace

ENTHOUGHT

Using stats objects

CREATING NEW DISCRETE DISTRIBUTIONS

```
# Create loaded dice.
>>> from scipy.stats import rv_discrete
>>> xk = [1,2,3,4,5,6]
>>> pk = [0.3,0.35,0.25,0.05,
     0.025,0.025]
>>> new = rv_discrete(name='loaded',
              values=(xk,pk))

# Calculate histogram
>>> samples = new.rvs(size=1000)
>>> bins=linspace(0.5,5.5,6)
>>> subplot(211)
>>> hist(samples,bins=bins,normed=True)

# Calculate pmf
>>> x = range(0,8)
>>> subplot(212)
>>> stem(x,new.pmf(x))
```



ENTHOUGHT

Summary Statistics

NUMPY AND SCIPY.STATS

```
>>> import numpy as np
>>> from scipy import stats
# Make some data to demo.
>>> x = stats.gamma.rvs(4.0, 0.5, size=1000)
# Demo assorted numpy and scipy.stats
# summary statistics functions.
# Mean:
>>> x.mean(), np.mean(x)
(4.4569, 4.4569)
# Standard deviation:
>>> x.std(), np.std(x)
(2.0035, 2.0035)
# Variance:
>>> x.var(), np.var(x)
(4.0141, 4.0141)
>>> x.var(ddof=1)
4.0181
# Median:
>>> np.median(x)
4.1610
# Trimmed mean:
>>> stats.tmean(x, (0.0, 10.0)
4.3626
# Skew:
>>> stats.skew(x)
1.0238
# Kurtosis:
>>> stats.kurtosis(x)
1.6433
# Score at the given percentile:
>>> stats.scoreatpercentile(x, per=90)
7.2195
# Percent below the given score:
>>> stats.percentileofscore(x, 3.0)
24.8999
# Rank the data points.
>>> stats.rankdata([1., 2., 2., 3., 4.])
array([ 1. ,  2.5,  2.5,  4. ,  5. ])
# Mode; return value(s) and count(s).
>>> d = [1, 2, 3, 2, 4, 2, 1, 4, 3]
>>> stats.mode(d)
(array([ 2.]), array([ 3.]))
```


ENTHOUGHT

Summary Statistics

COVARIANCE AND CORRELATION COEFFICIENTS

How correlated are the variations of 2 variables?

```
>>> x = np.random.random(size=(2,100))
# x is 2x100; each row is a variable.
>>> np.cov(x)
array([[ 0.08365514,  0.01120372],
       [ 0.01120372,  0.08369897]])
# Variables given as separate args:
>>> np.cov(x[0], x[1])
array([[ 0.08365514,  0.01120372],
       [ 0.01120372,  0.08369897]])

# Three variables with correlation
>>> x = np.random.random(size=(3,500))
>>> w = x + 0.005 * x.cumsum(axis=1)
>>> np.cov(w)
array([[0.208823, 0.124881, 0.1314642],
       [0.124881, 0.226147, 0.1394896],
       [0.131464, 0.139489, 0.2111896]])

# Correlation coefficients:
>>> np.corrcoef(w)
array([[ 1.      , 0.574661, 0.62600],
       [ 0.574661, 1.      , 0.63827],
       [ 0.626009, 0.638277, 1.     ]])
```

ENTHOUGHT

Cookbook

A Few Useful Stats Recipes

ENTHOUGHT

Random Numbers: numpy.random

BASIC UTILITIES

`numpy.random` implements a Mersene-twister PRNG. `scipy.stats` relies on it.

`from numpy.random import \`

`random` : Uniformly distributed random values from [0, 1).

`bytes` : Uniformly distributed bytes.

`random_integers` : Uniformly distributed integers.

`permutation` : Randomly permute a sequence.

`shuffle` : Randomly permute a sequence in place.

`seed` : Seed the random number generator.

EXAMPLES

```
>>> random()
0.19230919154236259
>>> random(size=(2,3))
array([[ 0.253003, 0.883954, 0.541616],
       [ 0.465056, 0.790178, 0.806152]])
>>> random_integers(1, 10, size=5)
array([ 8,  7,  2, 10,  6])
>>> permutation([1,2,3,4,5])
array([4, 1, 5, 3, 2])
>>> x = array([10, 20, 30, 40])
>>> shuffle(x)
>>> x
array([20, 10, 40, 30])
>>> bytes(4)
'4\x1f\xe70'
```

ENTHOUGHT

Random Numbers: numpy.random

Functions generating random numbers following different distributions:

UNIVARIATE DISTRIBUTIONS

Continuous

beta, chisquare, exponential, f, gamma, gumbel, laplace, logistic, lognormal, noncentral_chisquare, noncentral_f, normal, pareto, power, rayleigh, triangular, uniform, vonmises, wald, weibull

Discrete

binomial, geometric, hypergeometric, logseries, negative_binomial, poisson, zipf

MULTIVARIATE DISTRIBUTIONS

Continuous

dirichlet, multivariate_normal

Discrete

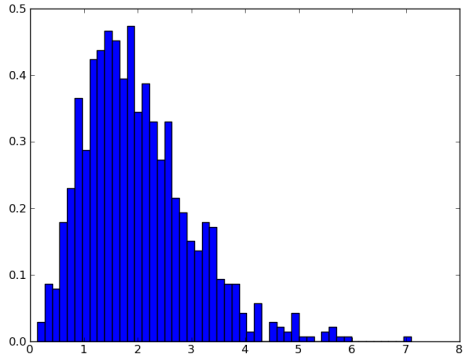
multinomial

ENTHOUGHT

Random Numbers: numpy.random

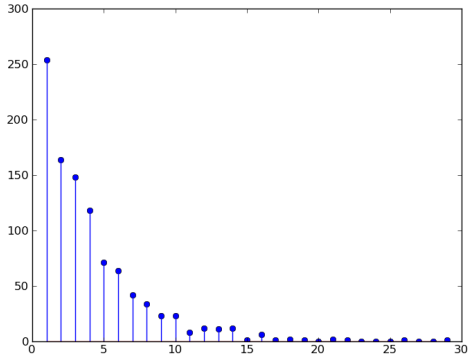
EXAMPLE - GAMMA

```
>>> k = 4.0
>>> theta = 0.5
>>> x = gamma(k, theta, size=1000)
>>> x.mean(), k*theta
(2.0015056922094678, 2.0)
>>> hist(x, 50, normed=True)
```



EXAMPLE - GEOMETRIC

```
>>> p = 0.25
>>> x = geometric(p, size=1000)
>>> x.mean(), 1/p
(4.085, 4.0)
>>> h = histogram(x, bins=x.ptp()+1,
                  range=(x.min()-0.5, x.max()+0.5))
>>> stem(h[1][:-1]+0.5, h[0])
```

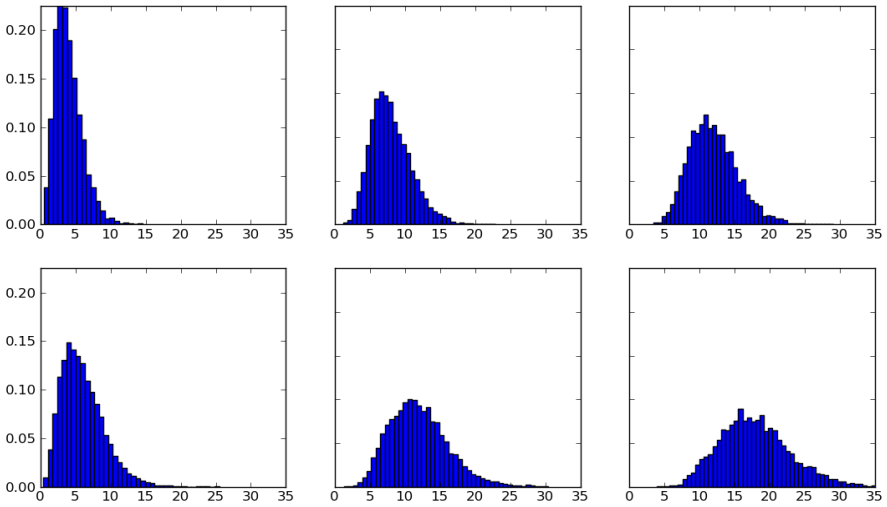


ENTHOUGHT

Random Numbers: numpy.random

BROADCASTING

```
# Three shape parameters, two scales.
>>> x = gamma([4.0, 8.0, 12.0], [[1.0],[1.5]], size=(5000,2,3))
```

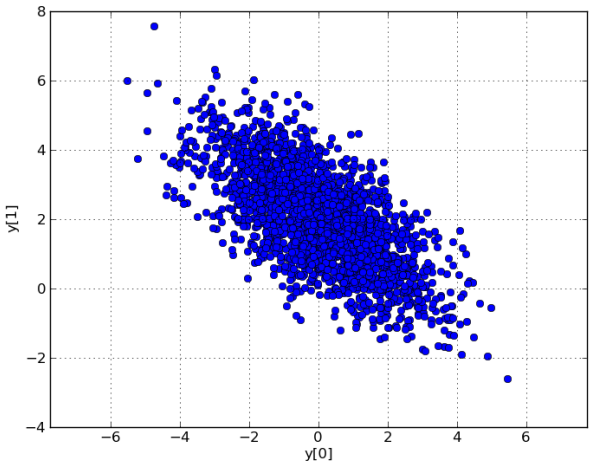


ENTHOUGHT

Random Numbers: numpy.random

MULTIVARIATE NORMAL

```
>>> from numpy.random import \
                        multivariate_normal
# mu is the vector of means.
>>> mu = array([0.0, 2.0])
# S is the desired covariance matrix.
>>> S = array([[2.7, -1.5], [-1.5, 1.8]])
>>> n = 2500
>>> y = multivariate_normal(mu, S, size=n).T
>>> cov(y)
array[[ 2.53192146, -1.4509589 ]
      [-1.4509589,   1.82226571]]
```



ENTHOUGHT

Linear Regression

scipy.stats.linregress

Compute the linear regression for the given data. 2 signatures:

linregress(x,y)

x, y both 1D with the same length

linregress(x)

x has shape (n, 2) or (2, n)

Returns:

(slope, intercept, rvalue, pvalue, stderr)

slope, intercept: linear coefficients

rvalue: Pearson product-moment correlation coefficient [-1, 1]

pvalue: for hypothesis “slope is 0”

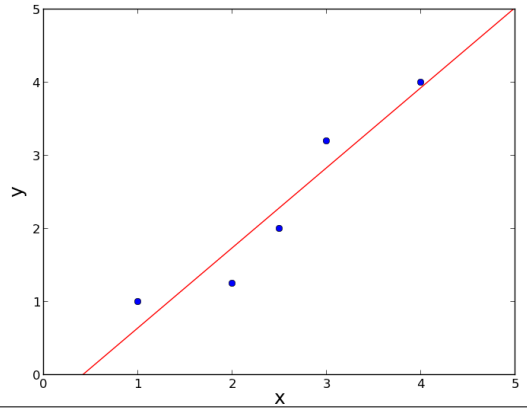
stderr: estimate of standard deviation of the data

ENTHOUGHT

Linear Regression

EXAMPLES

```
>>> from scipy.stats import \
                      linregress
>>> x = array([1.,2.,2.5,3.,4.])
>>> y = array([1,1.25,2.,3.2,4])
>>> sl, intercept, r, p, se = \
        linregress(x, y)
>>> sl, intercept
(1.095, -0.447)
```



DIAGNOSTICS

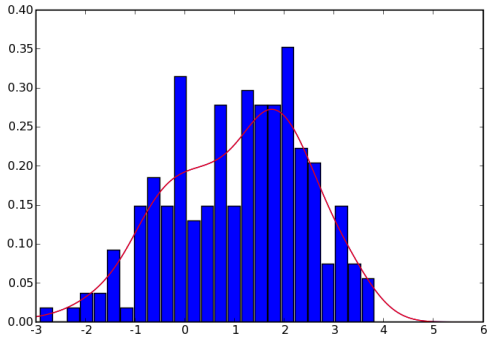
```
# Evaluation of std on sl and intercept
>>> mx = x.mean()
>>> sx2 = ((x-mx)**2).sum()
>>> sd_intercept = se *
   sqrt(1./len(x) + mx*mx/sx2)
>>> sd_slope = se *
   sqrt(1./sx2)
# “goodness of fit”
>>> r
0.962
# Probability that x and y are not
# correlated
>>> p
0.0116
```

ENTHOUGHT

Gaussian Kernel Density Estimation

CONTINUOUS DISTRIBUTION ESTIMATION USING GAUSSIAN KERNELS

```
# Sample two normal distributions
# and create a bi-modal distribution
>>> rv1 = stats.norm()
>>> rv2 = stats.norm(2.0,0.8)
>>> samples = hstack([rv1.rvs(size=100),
                      rv2.rvs(size=100)])


# Use a Gaussian kernel density to
# estimate the PDF for the samples.
>>> from scipy.stats.kde import gaussian_kde
>>> approximate_pdf = gaussian_kde(samples)

# Compare the histogram of the samples to
# the PDF approximation.
>>> hist(samples, bins=25, normed=True)
>>> x = linspace(-3,6,200)
>>> plot(x, approximate_pdf(x),'r')
```

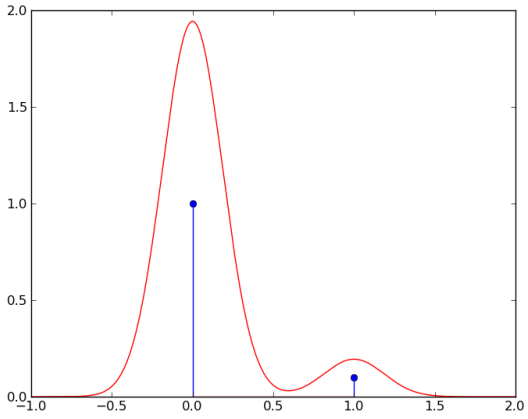


ENTHOUGHT

Gaussian Kernel Density Estimation

CONTINUOUS DISTRIBUTION ESTIMATION USING GAUSSIAN KERNELS

```
# 10 zeros, one one:
>>> samples2 = [0.0]*10 + [1.0]
# Use a Gaussian kernel density to
# estimate the PDF for the samples.
>>> pdf2 = gaussian_kde(samples2)

>>> x2 = linspace(-1.0, 2.0, 500)
>>> plot(x2, approximate_pdf2(x2),'r'
>>> stem([0, 1], [1, 0.1])
```

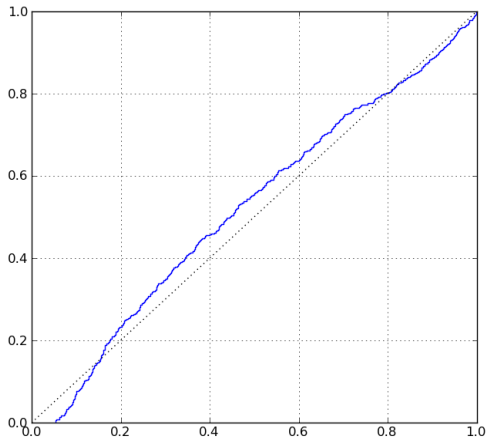


ENTHOUGHT

Q-Q Plot

COMPARING DISTRIBUTIONS WITH A Q-Q PLOT

```
>>> from scipy import stats
>>> def ecdf(x):
....     "Empirical CDF."
....     nx = x.size
....     u, i = np.unique(x,
....                      return_inverse=True)
....     if u.size == nx:
....         e = np.linspace(1./nx, 1, nx)
....     else:
....         n = np.bincount(i)
....         e = n.cumsum() / float(nx)
....     return u, e
# Make some data to demo.
>>> x = stats.gamma.rvs(4.0, 0.5, size=500)
>>> x.sort()
>>> u, e = ecdf(x)
# Make the Q-Q plot against a normal dist.
>>> ncdf = stats.norm.cdf(x, loc=x.mean(),
                          scale=x.std())
>>> step(ncdf, e)
# Plot y=x.
>>> plot([0,1], [0,1], 'k:')
>>> grid(True)
```



Deviation of the blue curve from y=x shows the non-normality of the data.

ENTHOUGHT

Z Functions

Z FUNCTIONS

The *standard score* or *z-score* is the number of std dev a value is above the mean.

$$z = \frac{x - \mu}{\sigma}$$

zmap(x, a) - Convert *x* to *z*, using *a*'s mean and std. dev.

zscore(a) - Convert array *a* to an array *z*, using *a*'s mean and std. dev.

zprob(z1) - Prob(z < z1) for z-distrib.

EXAMPLES

```
>>> from scipy.stats import \
        zscore, zmap, zprob

>>> a = array([1.3,1.5,1.3,1.8])
>>> zscore(a)
array([-0.85518611, 0.12216944,
  -0.85518611, 1.58820278])
>>> (a - a.mean())/a.std()
array([-0.85518611, 0.12216944,
  -0.85518611, 1.58820278])
# zscore for a give value
>>> zmap(1.4, a)
-0.36650833306891545
# probability to be in [-1, 1]
>>> zprob(1) - zprob(-1)
0.68268949213708585
```

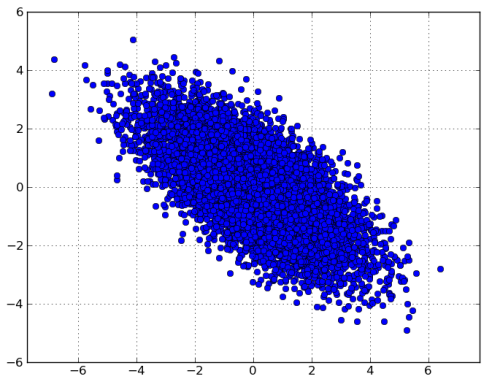
Correlated Random Samples

*Use **numpy.random.multivariate_normal,** or:*

Let Σ be the desired covariance matrix.

1. Generate uncorrelated samples.
2. Multiply by the matrix C, where
   $CC^T = \Sigma$
   The *Cholesky factorization* of Σ gives such a matrix.

```
>>> from numpy.random import normal
>>> from scipy.linalg import cholesky
>>> # S is the desired covariance matrix.
>>> S = array([[2.7, -1.5], [-1.5, 1.8])
>>> c = cholesky(S, lower=True)
>>> x = normal(size=(2, 10000))
>>> y = dot(c, x)
>>> cov(y)
array([[ 2.71840618, -1.52092278]
       [-1.52092278,  1.83228834]])
```



A spectral decomposition can also be used:

```
>>> from scipy.linalg import eigh
>>> evals, evecs = eigh(S)
>>> c = dot(evecs, diag(sqrt(evals)))
```

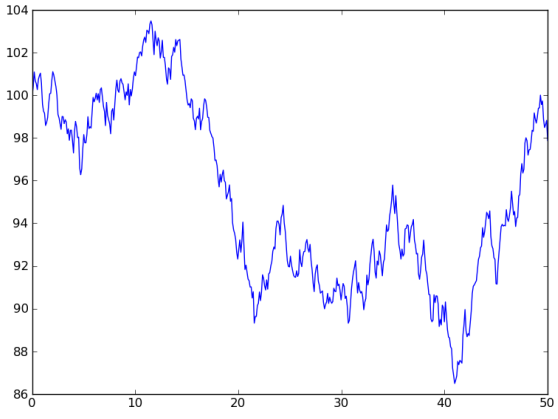

Brownian Motion

Brownian motion (Wiener process):

$X(t+dt) = X(t) + N(0,\sigma^2 dt; t, t+dt)$

where $N(a,b; t_1, t_2)$ is normal with mean a and variance b, and independent on disjoint time intervals.

```
>>> from scipy.stats import norm
>>> x0 = 100.0
>>> dt = 0.5
>>> sigma = 1.5
>>> n = 100
>>> steps = norm.rvs(size=n,
                     scale=sigma*sqrt(dt))
>>> steps[0] = x0  # Make i.c. work
>>> x = steps.cumsum()
>>> t = linspace(0, (n-1)*dt, n)
>>> plot(t, x)
```

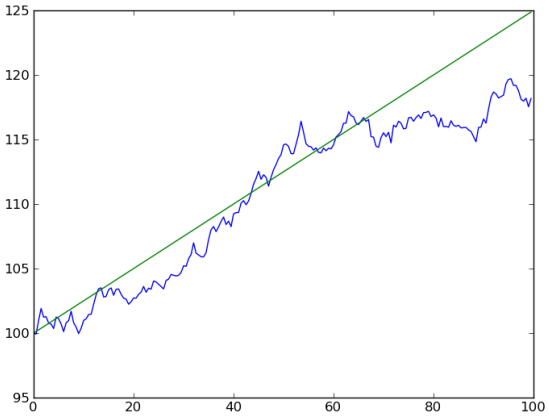


Brownian Motion

With ***drift*** (generalized Wiener process):

$X(t+dt) = X(t) + \mu dt + N(0,\sigma^2 dt; t, t+dt)$

```
>>> x0 = 100.0
>>> dt = 0.5
>>> sigma = 0.65
>>> mu = 0.25
>>> n = 200
>>> steps = mu*dt + norm.rvs(size=n,
              scale=sigma*sqrt(dt))
>>> steps[0] = x0
>>> x = steps.cumsum()
>>> t = linspace(0, (n-1)*dt, n)
>>> plot(t, x)
>>> plot(t, x0 + v*t)
```



Geometric Brownian Motion

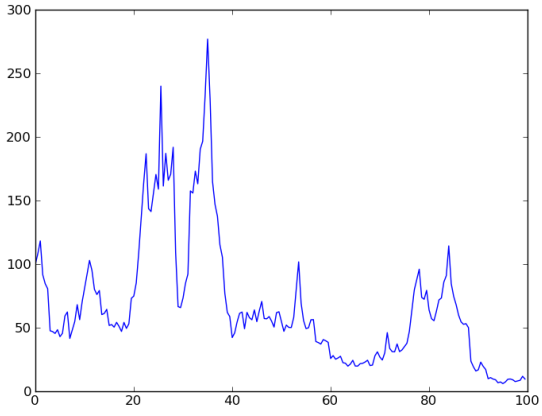
Geometric Brownian Motion (GBM):

$\Delta X / X = \mu\, dt + N(0,\sigma^2 dt; t, t+dt)$

or

$X(t+dt) = X(t)\,(1 + \mu\, dt + N(0,\sigma^2 dt; t, t+dt)$

```
>>> x0 = 100.0
>>> dt = 0.5
>>> sigma = 0.25
>>> mu = 0.05
>>> n = 200
>>> steps = 1 + mu*dt + \
norm.rvs(size=n, scale=sigma*sqrt(dt))
>>> steps[0] = x0
>>> x = steps.cumprod()
>>> t = linspace(0, (n-1)*dt, n)
>>> plot(t, x)
```

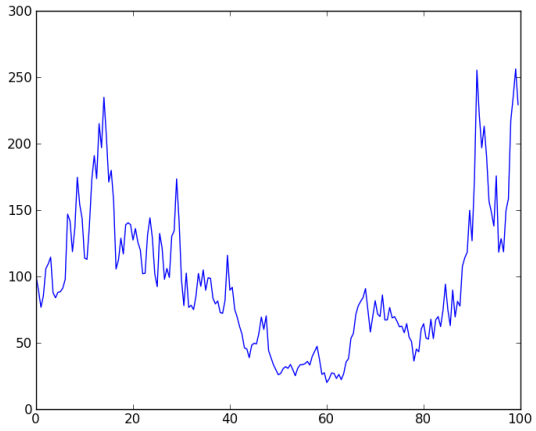


Geometric Brownian Motion

Or, even better, use the explicit formula for the solution to GBM:

$X(t) = X(0)\exp((\mu - \sigma^2/2)t + N(0,\sigma^2 t))$

```
>>> x0 = 100.0
>>> dt = 0.5
>>> sigma = 0.25
>>> mu = 0.05
>>> n = 200
>>> steps = exp((mu-sigma**2/2)*dt + \
norm.rvs(size=n, scale=sigma*sqrt(dt)))
>>> steps[0] = x0
>>> x = steps.cumprod()
>>> t = linspace(0, (n-1)*dt, n)
>>> plot(t, x)
```



Correlated GBM

Two GBM processes, with correlated random variables:

$X_1(t) = X_1(0)\exp((\mu_1 - \sigma_1^2/2)t + t^{1/2}V_1)$

$X_2(t) = X_2(0)\exp((\mu_2 - \sigma_2^2/2)t + t^{1/2}V_2)$

where ***V*** *is a random vector of two normal random variables, with covariance*

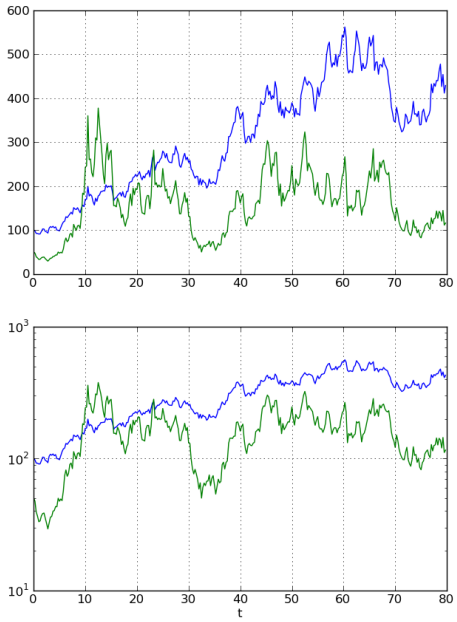
$$\Sigma = \begin{vmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{vmatrix}$$

```
>>> from math import sqrt
>>> from scipy.stats import norm
>>> from numpy.random import \
                    multivariate_normal
>>> dt = 0.25
>>> n = 320
>>> # Desired variances:
>>> var1 = 0.1**2
>>> var2 = 0.25**2
>>> # The desired correlation:
>>> corr = 0.75
>>> # Create the covariance matrix:
>>> covar12 = corr * sqrt(var1*var2)
>>> Sigma = array([[var1, covar12],
                   [covar12, var2]])
```

ENTHOUGHT

Correlated GBM

```
>>> # Create the correlated samples:
>>> v = multivariate_normal([0,0],
                            Sigma, size=n)
>>> # Generate the GBM realizations:
>>> mu = array([0.02, 0.025])
>>> var = Sigma.diagonal()
>>> steps = exp((mu - 0.5*var)*dt +
                sqrt(dt) * v).T
>>> # Initial values:
>>> steps[:,0] = array([100.0, 50.0])
>>> # The correlated GBM realizations:
>>> x = steps.cumprod(axis=1)
```

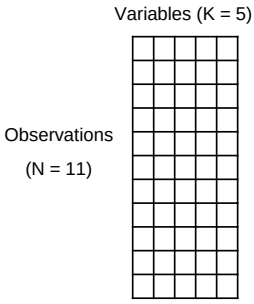


ENTHOUGHT

Principal Component Analysis

PCA is a mathematical procedure to decompose a data set into the directions of maximum variance.

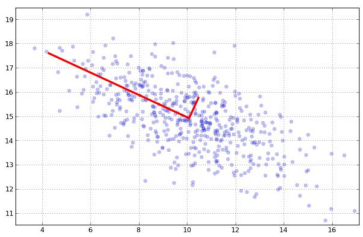
PCA is applied to a collection of N *observations* or *samples*. Each observation has measurements of K *variables* or *features*.



The data set is interpreted as a collection of points in K-dimensional space.

- First principal component accounts for as much of the variability in the data as possible.
- Succeeding components have the highest variance possible under the constraint that they are orthogonal to (uncorrelated with) the preceding components.
- Can be computed using an eigenvalue decomposition of the covariance matrix, or with a singular value decomposition.

- The orthogonal vectors are often called the *loadings*.
- The projections of the observations onto the loading vectors give the *scores*.



Example with K = 2; see demo/pca/plot_pca_2d.py

ENTHOUGHT

Principal Component Analysis

SVD IMPLEMENTATION

```
# We'll use the wind data from the "wind statistics" numpy exercise.
>>> x = loadtxt('wind.data')[:, 3:]   # Drop the first 3 cols (dates)
# Subtract the mean.
>>> xz = x - x.mean(axis=0)
>>> u, s, vt = np.linalg.svd(xz, full_matrices=False)
# The rows of vt are the loadings.
# Get the scores by multiplying xz by vt.T
>>> score = np.dot(xz, vt.T)
```


ENTHOUGHT

Principal Component Analysis

VARIANCES

```
# Find the variances for each component.
>>> num_samples = xz.shape[0]
>>> var = (s ** 2) / num_samples
>>> var
array([ 229.48 ,   23.101,   14.807,    8.066,    7.819,    3.735,
          3.276,    2.431,    1.985,    1.838,    1.355,    1.166])
# Relative variance components
>>> var / var.sum()
array([ 0.767,  0.077,  0.05 ,  0.027,  0.026,  0.012,  0.011,  0.008,
        0.007,  0.006,  0.005,  0.004])
>>> var.cumsum() / var.sum()
array([ 0.767,  0.845,  0.894,  0.921,  0.947,  0.96 ,  0.971,  0.979,
        0.985,  0.992,  0.996,  1.   ])
```

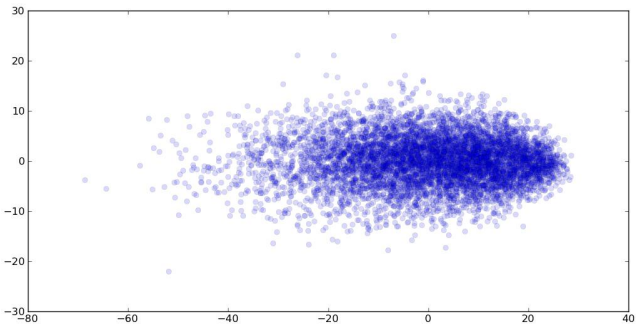
First component accounts for 77% of the variance.

First five components account for 95% of the variance.

ENTHOUGHT

Principal Component Analysis

```
# Project the data onto the first two components (reduce dimension)
>>> y = dot(xz, vt[:2].T)
# Make a scatter plot of the projected data.
>>> plot(y[:,0], y[:,1], 'bo', alpha=0.2)
>>> axis('equal')
```



See demo/pca/pca_wind_data.py

ENTHOUGHT

Principal Component Analysis

COMPARISON TO MATLAB'S PRINCOMP

For MATLAB users...

The following use of the MATLAB function **princomp**

```
>> [coeff, score, latent] = princomp(b)    % MATLAB code
```

can be implemented with the SVD function as follow

```
>>> u, s, vt = np.linalg.svd(b, full_matrices=False)
>>> coeff = vt.T
>>> score = np.dot(b, coeff)
>>> latent = (s ** 2) / (b.shape[0] - 1)
# (MATLAB uses an unbiased variance calculation, hence the -1 above.)
```

ENTHOUGHT

Principal Component Analysis

SCIKITS-LEARN PCA

The scikits-learn library has an implementation of PCA.

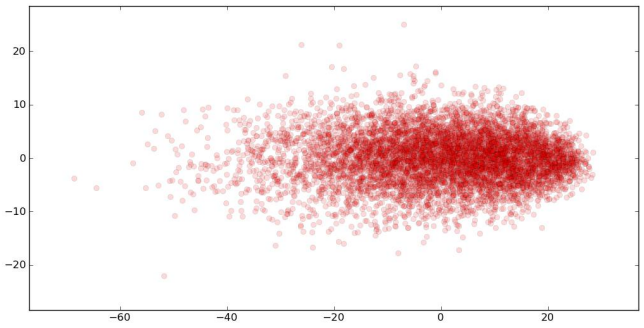
EXAMPLE (CONTINUED)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA().fit(x)
>>> print pca.explained_variance_
array([ 229.48 ,   23.101,   14.807,    8.066,    7.819,    3.735,
          3.276,    2.431,    1.985,    1.838,    1.355,    1.166])
>>> print pca.explained_variance_ratio_
array([ 0.767,  0.077,  0.05 ,  0.027,  0.026,  0.012,  0.011,  0.008,
        0.007,  0.006,  0.005,  0.004])
# Project onto the first two components.
>>> pca2 = PCA(n_components=2).fit(x)
>>> x2 = pca2.fit_transform(x)
```


ENTHOUGHT

Principal Component Analysis

```
# Make a scatter plot of the projected data.
>>> plot(x2[:,0], x2[:,1], 'ro', alpha=0.15)
>>> axis('equal')
```



See demo/pca/pca_wind_data2.py

ENTHOUGHT

Hypothesis Testing

ENTHOUGHT

Chi-squared test

Are 2 samples likely to have been drawn from the same distributions? For binned observation, chi^2 test is standard.

GOODNESS OF FIT

```
from scipy.stats import chisquare

# Some observation counts events in 3 different bins
>>> observed = array([73, 10, 17])

# Expected distribution
>>> distr = array([0.6, 0.2, 0.2])
>>> expected = distr *observed.sum()

>>> chi2, p = chisquare(observed, expected)
>>> print chi2, p
8.26667, 0.0160

# The probability is too small: it is unlikely that the
# observations are explained by this model.
```

Contingency Table

ENTHOUGHT

CONTINGENCY TABLE

```
>>> from scipy.stats import \
        chi2_contingency
>>> obs = array([[75, 25],
                 [65, 35]])
>>> chi2, p, dof, ex = \
        chi2_contingency(obs)

# Expected obs
>>> ex
[[ 70.  30.]
 [ 70.  30.]]
# chi-square value
>>> chi2
1.9285714285714288
# Probability value
>>> p
0.16491482255330137
```

FISHER'S EXACT TEST

```
>>> from scipy.stats import \
        fisher_exact
>>> oddsr, p = fisher_exact(obs)
# Probability value
>>> p
0.16463667384528796
```

NOTES

- **`fisher_exact`** is for 2x2 arrays only.
- **`chi2_contingency`** can handle n-dimensional contingency tables of arbitrary size but the chi-square test is not valid if any entry is too small, typically less than 5.

t-Test

ENTHOUGHT

T-TEST FUNCTIONS

ttest_1samp(a, propmean) – t-test for the mean of one group of scores.

ttest_ind(a, b, axis=0) – t-test for the means of two independent samples of scores.

ttest_rel(a, b, axis=0) – t-test for the means of two related samples of scores.

All return (t, prob). prob is the probability of the tested hypothesis. Below 0.05, the convention is to reject the hypothesis.

EXAMPLE – ONE SAMPLE

```
>>> from scipy.stats import \
                    ttest_1samp
>>> a = array([1.45, 1.44, 1.51,
          1.49, 1.46, 1.46, 1.50])

# Was this sample drawn from a
# population with mean 1.5?
>>> ttest_1samp(a, 1.5)
(-2.6692695630078465, 0.0370)

>>> d = norm(loc = 1.5, scale = 3)
>>> x = d.rvs(100)
>>> ttest_1samp(a, 1.5)
(-1.0391329947533454, 0.3012)
>>> x = d.rvs(1000)
>>> ttest_1samp(x, 1.5)
(0.4832862997311918, 0.6289)
```

ENTHOUGHT

t-Test

EXAMPLE – TWO SAMPLES

```
# Test if 2 independent samples
# are drawn from distributions
# with the same mean

>>> from scipy.stats import norm,\
 ttest_ind, ttest_rel, gamma

# Create data
>>> x = norm.rvs(loc=50, scale=10,
      size=30)
# y is from a gamma distribution
# with shape=25 and scale=2
# (mean=50, std=10)
>>> y = gamma.rvs(25, scale=2,
      size=50)
>>> ttest_ind(x, y)
(-1.2898780346752314, 0.20090)
```

p > 0.05; can not reject H0.

EXAMPLE – TWO SAMPLES (REL)

```
# Test the hypothesis that 2
# datasets are from the same
# distribution.
# Reaction times before treatment
>>> x = [783, 599, 1005, 882, 934]
# Reaction times after treatment
>>> y = [890,1032, 1178, 900, 918]
>>> ttest_rel(x, y)
(-1.7930981620792279, 0.14741)
```

p > 0.05; can not reject H0. Get more data!

```
>>> x += [1010, 850, 942]
>>> y += [1021, 1138, 1013]
>>> ttest_rel(x, y)
(-2.4586636970210622, 0.04355)
```

At 0.05 sig. level, reject H0.

ENTHOUGHT

ANOVA

ANOVA – Analysis of Variance

Evaluate the probability that *independent*, *normally distributed* datasets a, b, ... have the same mean.

`f_oneway(a, b, …)` – one-way ANOVA.

Returns (t, prob).

EXAMPLE

```
>>> from scipy.stats import \
                    f_oneway
```

Data from three geographic regions

```
>>> a = array([200, 240, 230, 198,
              221, 204, 285, 199])
>>> b = array([215, 245, 233, 212,
              230, 204])
>>> c = array([231, 218, 205, 244,
    261, 288, 233, 241, 257, 236])
>>> a.mean(), b.mean(), c.mean()
(222.125, 223.166, 240.40)
```

Are the differences significant?

```
>>> f_oneway(a, b, c)
(1.5489673412104141, 0.23578)
```

We can not conclude that the differences are significant.

ENTHOUGHT

Tests for Distributions

ANDERSON-DARLING

```
>>> from scipy.stats import \
          anderson, gamma
# Make some data to test.
>>> x = gamma.rvs(2, size=500)
# anderson returns
# (stat,critvals,siglevels)
>>> anderson(x, 'norm')
(9.9589399398296905,
 array([ 0.571,  0.651,  0.781,
0.911,  1.083]),
 array([ 15. ,  10. ,   5. ,
2.5,   1. ]))
>>> x = gamma.rvs(32, size=500)
>>> a2, c, s = anderson(x, 'norm')
>>> a2
0.64929834112859908
```

KOLMOGOROV-SMIRNOV

This is the standard test to compare an observed dataset following a continuous distribution (not binned) to a known distribution.

```
>>> from scipy.stats import \
          zscore, kstest
>>> x = gamma.rvs(2, size=500)

# kstest returns (stat, p)
>>> kstest(zscore(x), 'norm')
(0.12257, 5.265e-07)

>>> x = gamma.rvs(32, size=500)
>>> kstest(zscore(x), 'norm')
(0.035188, 0.5664)
>>> x = gamma.rvs(32, size=5000)
>>> kstest(zscore(x), 'norm')
(0.023561, 0.0077655)
```


ENTHOUGHT

More Hypothesis Tests

pearsonr	Pearson correlation coefficient and p-value
spearmanr	Spearman rank-order coefficient and p-value
pointbiserialr	Point biserial coefficient and p-value
kendalltau	Kendall's tau, a correlation for ordinal data
mannwhitneyu	Mann-Whitney rank test
ranksums	Wilcoxon rank-sum statistic for two samples
wilcoxon	Wilcoxon signed-rank test
kruskal	Kruskal-Wallis H-test for independent samples
friedmanchisquare	Friedman test for repeated measurements

ENTHOUGHT

More Hypothesis Tests

ansari	Ansari-Bradley test for equal scale parameters
bartlett	Bartlett's test for equal variances
levene	Levene test for equal variances
shapiro	Shapiro-Wilk test for normality
binom_test	Test that the probability of success is p
fligner	Fligner's test for equal variances
mood	Mood's test for equal scale parameters
oneway	Test for equal means in samples from the normal distribution

ENTHOUGHT

scikits.statsmodels

`scikits.statsmodels` is a library for statistical and econometric analysis in Python.

Provides tools for:
- Regression
- Generalized Linear Models
- Robust Linear Models
- Time Series Analysis
- and more...

Web page: **`http://statsmodels.sourceforge.net/`**

ENTHOUGHT

scikits.statsmodels

MODEL

Sketch of the base class for the models:

```
class Model(object):
    def __init__(self, endog, exog=None):
        self.endog = endog
        if exog is not None:
            self.exog = exog
        ...

    def fit(self):
        ...

    def predict(self):
        ...
```


ENTHOUGHT

scikits.statsmodels

REGRESSION

Linear models:

$$Y = X\beta + \varepsilon$$

X is the design matrix, β is the vector of linear coefficients, and ε is the noise or error.

For example, for a simple linear fit (slope and intercept) of an observed Y, X has two columns: one holds the independent variable associated with Y, and the other is all 1's.

ENTHOUGHT

scikits.statsmodels

OLS - ORDINARY LEAST SQUARES REGRESSION

Least squares solution: $\beta = (X^T X)^{-1} X^T Y$

```
>>> import numpy as np
>>> import scikits.statsmodels.api as sm

# Make some noisy sample data.
>>> n = 41
>>> x = linspace(0, 10, n)
>>> y = 2.5 + 0.5*x + 0.08*np.random.randn(n)

# Create the design matrix for a linear fit.
# add_constant(x) appends a column of 1s (for the intercept).
>>> X = sm.add_constant(x)

# Create the OLS instance with this data, and call fit()
>>> results = sm.OLS(y, X).fit()
>>> print results.summary()
```


ENTHOUGHT

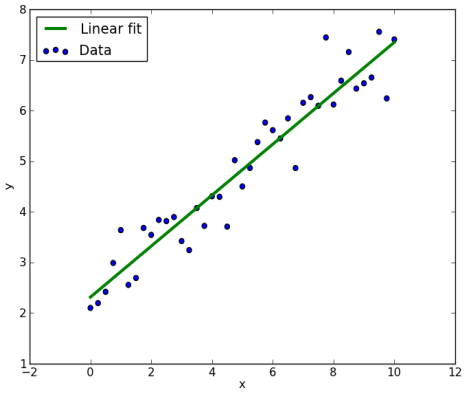
scikits.statsmodels

```
 Summary of Regression Results
=======================================
| Dependent Variable:            ['y']|
| Model:                           OLS|
| Method:                Least Squares|
| Date:               Fri, 09 Dec 2011|
| Time:                       00:45:22|
| # obs:                          41.0|
| Df residuals:                   39.0|
| Df model:                        1.0|
===============================================================================
|                   coefficient     std. error    t-statistic          prob. |
-------------------------------------------------------------------------------
| x1                     0.5034        0.02451        20.5396         0.0000 |
| const                   2.315         0.1424        16.2563         0.0000 |
===============================================================================
|                         Models stats                         Residual stats  |
-------------------------------------------------------------------------------
| R-squared:                  0.9154   Durbin-Watson:                2.155  |
| Adjusted R-squared:         0.9132   Omnibus:                     0.6192  |
| F-statistic:                 421.9   Prob(Omnibus):               0.7337  |
| Prob (F-statistic):      1.613e-22   JB:                         0.01779  |
| Log likelihood:             -25.69   Prob(JB):                    0.9911  |
| AIC criterion:               55.37   Skew:                       0.05445  |
| BIC criterion:               58.80   Kurtosis:                     3.222  |
-------------------------------------------------------------------------------
```

ENTHOUGHT

scikits.statsmodels

```
# Plot the data.
>>> scatter(x, y, label='Data')
# Compute the fitted straight line.
>>> Y = np.dot(X, results.params)
# Plot the fit.
>>> plot(x, Y, 'g-', linewidth=3,
...      label='Linear fit')
>>> legend(loc='best')
>>> xlabel('x')
>>> ylabel('y')
>>> show()
```



Now see: **demo/statsmodels/ols_quadratic.py**

ENTHOUGHT

scikits.statsmodels

WLS - WEIGHTED LEAST SQUARES REGRESSION

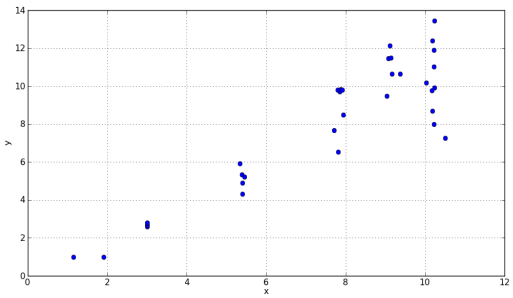
For when the variances of the error are not the same for all the observations.

Weighted least squares solution:

$$\beta = (X^T W X)^{-1} X^T W Y$$

where W is diagonal and

$$W_{ii} = 1 / \sigma_i^2$$



How to estimate W depends on the data.

Detailed example in
demo/statsmodels/wls.py

ENTHOUGHT

Enthought Tool Suite

ENTHOUGHT

Enthought Tool Suite

TRAITS

Initialization, Validation, Observation, and Visualization of Python class attributes

KIVA

2D primitives supporting path based rendering, affine transforms, alpha blending and more.

ENABLE

Object based 2D drawing canvas

CHACO

Plotting toolkit for building complex, interactive 2D plots

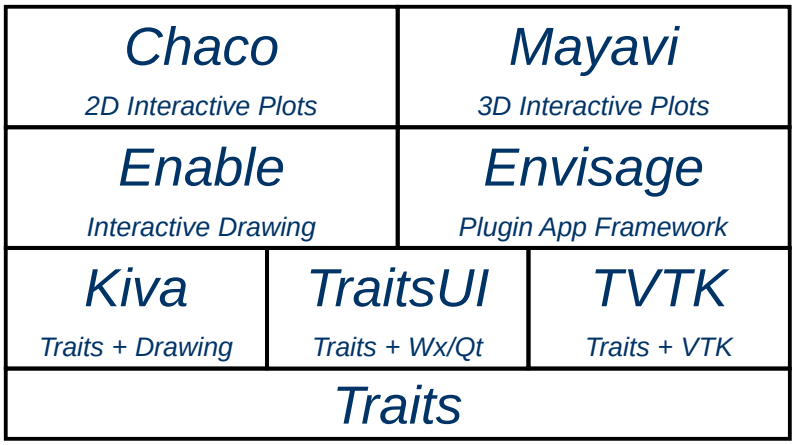
MAYAVI

3D Visualization of Scientific Data based on VTK

ENVISAGE

Application plugin framework for building scriptable and extensible applications

Enthought Tool Suite



- BSD-style license
- Actively used and supported by Enthought and community
- More information at http://code.enthought.com

Traits

ENTHOUGHT

What are traits?

Traits provide additional characteristics for Python object attributes:

- Standardization
  - Initialization
  - Validation
  - Delegation
- Visualization (TraitsUI)
- Notification
- Documentation

ENTHOUGHT

A Note About Examples

- The code in the scripts do not exactly mimic the slides. It has usually been simplified for brevity.
- Most non-ui examples are simple scripts that can be run from the command line or IPython:

```
In[1] run rect_1.py
```

- Some examples have a `main()` method that must be run to get their output. This is typically done for examples that throw tracebacks as part of the demo.
- Most UI examples must be run from IPython started using an option such as '--pylab' or '--gui=wx'.

ENTHOUGHT

Defining Simple Traits — rect_1.py

```
from traits.api import HasTraits, Float

class Rectangle(HasTraits): # <----- Derive from HasTraits
    """ Simple rectangle class with two traits.
    """
    # Width of the rectangle
    width = Float    # <----- Declare Traits

    # Height of the rectangle
    height = Float() # <----- Declare Traits
```

Note: Run `main()` for this example to execute similar commands to those below.

```
In[1]: run rect_1.py
In[2]: main()
```

```
# Demo Code
>>> rect = Rectangle()
>>> rect.width
0.0

# Set rect width to 1.0
>>> rect.width = 1.0
>>> rect.width
1.0
```

```
# Float traits convert integers
>>> rect.width = 2
>>> rect.width
2.0

# THIS WILL RAISE EXCEPTION
>>> rect.width = "1.0"
TraitError: The 'width' trait of a
Rectangle instance must be a float,
but a value of '1.0' <type 'str'>
was specified.
```


ENTHOUGHT

Default Values — rect_2.py

```
from traits.api import HasTraits, Float

class Rectangle(HasTraits):
    """ Simple rectangle class with two traits.
    """

    # Width of the rectangle
    width = Float(1.0)  # <----- Set default to 1.0

    # Height of the rectangle
    height = Float(2.0) # <----- Set default to 2.0
```

```
# Demo Code
>>> rect = Rectangle()
>>> rect.width
1.0
>>> rect.height
2.0
```

```
# Initialization via
# keyword arguments
>>> rect = Rectangle(width=2.0,
                     height=3.0)
>>> rect.width
2.0
>>> rect.height
3.0
```

ENTHOUGHT

Coercion and Casting — rect_3.py

```
from traits.api import HasTraits, Float, CFloat

class Rectangle(HasTraits):
    """ Simple rectangle class with two traits.
    """

    # Basic traits allow "widening" coercion (Int->Float).
    width = Float

    # CFloat traits apply float() to any assigned variable.
    height = CFloat  # <----- CFloat is the casting version
                     #        of the basic Float trait
```

```
# Demo Code
>>> rect = Rectangle()
>>> rect.height = "2.0" # <----- This Works!
>>> rect.width = "2.0"
TraitError: The 'width' trait of a Rectangle instance must be a float,
but a value of '2.0' <type 'str'> was specified.
```


ENTHOUGHT

Traits for Basic Python Types

Coercing Trait	Casting Trait	Python Type	Default Value
Bool	CBool	bool	False
Complex	CComplex	complex	0+0j
Float	CFloat	float	0.0
Int	CInt	int	0
Long	CLong	long	0L
Str	CStr	str or unicode (whichever assigned)	''
Unicode	CUnicode	unicode	u''

ENTHOUGHT

Traits Speed

Attribute Access Method	Get Attribute		Set Attribute	
	Time (µs)	Speed-up	Time (µs)	Speed-up
Global Module Variable	0.017	**2.35**	0.029	**2.10**
Old Style Instance Attribute	0.031	**1.32**	0.045	**1.34**
New Style Instance Attribute	0.041	-	0.061	-
Standard Python Property	0.248	0.17	0.345	0.18
“Any” Traits Attribute	0.023	**1.78**	0.061	0.98
“Int” Traits Attribute	0.024	**1.73**	0.067	0.90
“Range” Traits Attribute	0.023	**1.76**	0.068	0.89
Statically Observed Trait	0.024	**1.75**	1.297	0.05
Dynamically Observed Trait	0.023	**1.78**	2.126	0.03
Delegated Trait	0.090	0.46	2.943	0.02
Delegated Trait (2 levels)	0.153	0.27	5.789	0.01
Delegated Trait (3 levels)	0.224	0.18	8.772	0.007

Comparison of setting various types of traits to setting standard Python class attributes and properties (from traits/tests/check_timing.py on 2.3GHz Mac Book Pro as of 8 May 2013)

ENTHOUGHT

Properties — rect_4.py

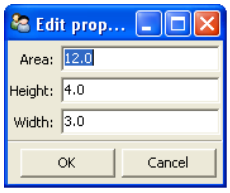
```
from traits.api import HasTraits, \
    Float, Property

class Rectangle(HasTraits):
    """ Rectangle class with
        read-only area property.
    """
    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # The area of the rectangle
    # Defined as a property.
    area = Property

    # specially named method
    # automatically associated
    # with area.
    def _get_area(self):
        return self.width * self.height
```

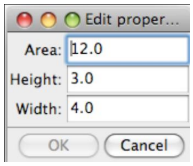
```
# Demo Code

>>> rect = Rectangle(width=2.0,
                     height=3.0)
>>> rect.area
6.0
>>> rect.width = 4.0
>>> rect.area
12.0
>>> rect.area = 16.0
TraitError: The 'area' trait of a
Rectangle instance is 'read
only'.
```

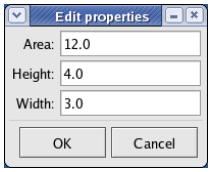
ENTHOUGHT

Traits UI – Default Views

```
>>> rect = Rectangle(width=3.0, height = 4.0)
# Create a UI to edit the traits of the object.
>>> rect.edit_traits() # or rect.configure_traits()
```





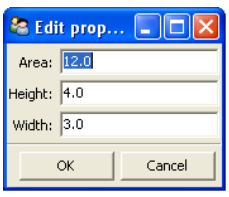


ENTHOUGHT

Properties Dependencies — rect_5.py

```
from traits.api import HasTraits, Float, Property

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)

    # Specify dependencies with 'depends_on' meta-data.
    # This updates the area whenever width or height change.
    area = Property(depends_on=['width','height'])

    def _get_area(self):
        return self.width * self.height
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits()
```

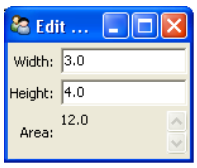


ENTHOUGHT

Default UI Views — rect_6.py

```
from traits.api import HasTraits, Float, Property
from traitsui.api import View, Item

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)
    area = Property(depends_on=['width','height'])

    # Define a default view with the area as a readonly editor.
    traits_view = View('width', 'height',
                       Item('area', style='readonly'))

    def _get_area(self):
        return self.width * self.height
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits()
```

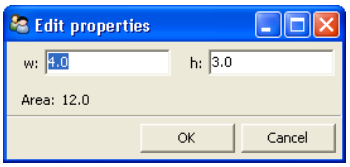


ENTHOUGHT

Simple UI Layout — rect_6.py

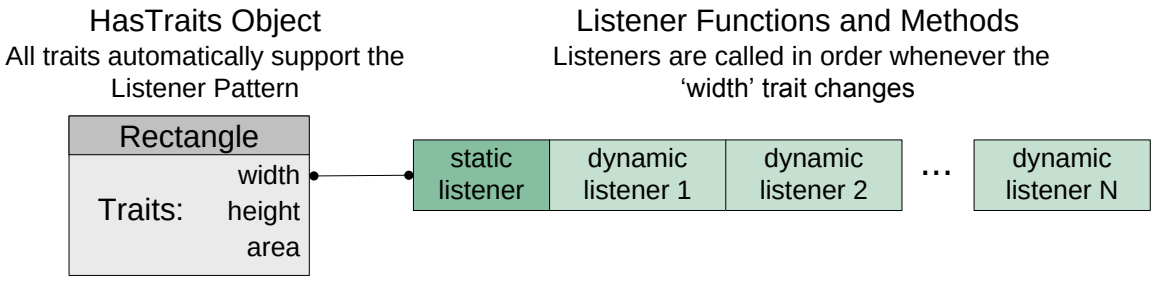
```
from traitsui.api import View, HGroup, VGroup, Item
from traitsui.menu import OKCancelButtons

view1 = View(
             VGroup( # Create a vertical layout group.
                    HGroup(# And a horizontal group within that.
                           # Change the labels on each item.
                           Item('width',label='w'),
                           Item('height', label='h')
                           ),
                    Item('area', style='readonly'),
                   ),
             # Add OK, Cancel buttons to the UI
             buttons=OKCancelButtons
            )
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits(view=view1)
```



ENTHOUGHT

Trait Listeners



ENTHOUGHT

Static Trait Notification—amplifier_1.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, value=5.0)

    # Static observer method called whenever volume is set.
    def _volume_changed(self, old, new):
       if new == 11.0:
          print "This one goes to eleven"
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
This one goes to eleven
>>> spinal_tap.volume = 11.0 # nothing is printed because
                             # the value didn't change.
```

ENTHOUGHT

Valid Static Trait Notification Signatures

```
def _volume_changed(self):
# no arguments...

def _volume_changed(self, new):
# new – the just-set value of volume

def _volume_changed(self, old, new):
# old – the previous value of volume
# new – the just-set value of volume

def _volume_changed(self, name, old, new):
# name – the name of the trait ('volume')
# old – the previous value of volume
# new – the just-set value of volume

# This signature is usually used for the
# _anytrait_changed() observer that is called
# whenever any trait is changed on the object.
```


ENTHOUGHT

Dynamic Trait Notification – amplifier_2.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, value=5.0)


def printer(value):
   print "new value:", value
```

```
# Demo Code
>>> spinal_tap = Amplifier()
# In the following, name can also be a list of trait names
>>> spinal_tap.on_trait_change(printer, name='volume')
>>> spinal_tap.volume = 11.0
new value: 11.0
```

ENTHOUGHT

Valid Dynamic Trait Notification Signatures

```
def observer():
      # no arguments...

def observer(new):
      # new – the new value of the changed trait

def observer(name, new):
      # name – the name of the changed trait
      # new – the new value of the changed trait

def observer(object, name, new):
      # object – the object containing the changed trait
      # name – the name of the changed trait
      # new – the new value of the changed trait

def observer(object, name, old, new):
      # object – the object containing the changed trait
      # name – the name of the changed trait
      # old – the previous value of the changed trait
      # new – the new value of the changed trait
```


ENTHOUGHT

Dynamic Trait Notification — amplifier_3.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """
    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, value=5.0)

class Listener(object):
    """ Class that will listen to the Amplifier volume
    """
    def printer(self, value):
        print "new value:", value
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> listener = Listener()
>>> spinal_tap.on_trait_change(listener.printer, name='volume')
>>> spinal_tap.volume = 11.0
new value: 11.0

# on_trait_change has a weak reference to the class method.
# When the class goes away, the method no longer fires.
>>> del listener
>>> spinal_tap.volume = 10.0
```

ENTHOUGHT

@on_trait_change decorator — amplifier_4.py

```
from traits.api import HasTraits, Range, on_trait_change

class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    volume = Range(0.0, 11.0, value=5.0)
    reverb = Range(0.0, 10.0, value=5.0)

    # The on_trait_change decorator can listen to multiple traits
    # Note the "list" of traits is specified as a string.
    @on_trait_change('reverb, volume')
    def update(self, name, value):
        print 'trait %s updated to %s' % (name, value)
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
trait volume updated to 11.0
>>> spinal_tap.reverb = 2.0
trait reverb updated to 2.0
```


ENTHOUGHT

Listeners on a different thread — amplifier_5.py

```
class Amplifier(HasTraits):
    volume = Range(0.0, 11.0, value=5.0)

    def __init__(self, *args, **kw):
        super(Amplifier, self).__init__(*args, **kw)

        # Use the "dispatch" keyword to run the listener on a
        # different thread.
        self.on_trait_change(self.update, 'volume', dispatch='new')

    def update(self, name, value):
        print 'thread %s sleeping for 1 second' % thread.get_ident()
        sleep(1.0)
        print 'trait %s updated to %s' % (name, value)
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
main thread: -1601355872
thread identity -1340948480 sleeping for 1 second
trait volume updated to 11.0
```

ENTHOUGHT

Dynamic defaults — default.py

```
from datetime import date, timedelta
from traits.api import HasTraits, Str, Date
class Task(HasTraits):
    """A task to be performed."""
    # Description of the task.
    description = Str
    # Due date of the task.
    due_date = Date
    def _due_date_default(self):
        """Default due date is one week from today."""
        return date.today() + timedelta(days=7)
```

The first time the trait 'due_date' referenced, this function is called to get the initial value.

```
# Demo Code
>>> task = Task(description="File taxes")
>>> task.due_date
datetime.date(2012, 1, 16)
>>> task.due_date = date(2012, 4, 15)
>>> task.due_date
datetime.date(2012, 4, 15)
```


ENTHOUGHT

More Advanced Trait Types

In this section, we'll take a look at a few more Trait types:

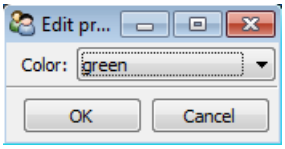
- Enum
- List
- Dict
- Array
- Instance
- Event
- Button
- File

ENTHOUGHT

Enum Trait – traffic_light_1.py

```
from traits.api import HasTraits, Enum

class TrafficLight(HasTraits):

    color = Enum("green", "yellow", "red")
```

```
# Demo Code
>>> light = TrafficLight()
>>> light.color
"green"
>>> light.edit_traits()
```




```
>>> light.color = "blue"    # This will raise an exception.
TraitError: The 'color' trait of a TrafficLight instance must be
'green' or 'yellow' or 'red', but a value of 'blue' <type 'str'> was
specified.
```


ENTHOUGHT

List Traits

```
class MultipleLists(HasTraits):
    # same as List(Any)
    a = List
    # List of Strings
    b = List(Str)
    # List of Person objects
    c = List(Instance(Person))
    # List of Ints with 3-5 elements.
    # The default value is [1,2,3]
    d = List([1,2,3], Int, minlen=3, maxlen=5)
```

ENTHOUGHT

List Trait – school_class_1.py

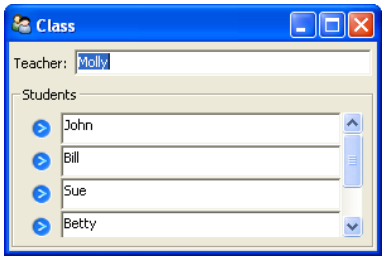
```
class SchoolClass(HasTraits):

    # List of the students in the class
    students = List(Str)                        # <-- List of strings

    def _students_changed(self, old, new):      # <-- called when list replaced
        print "The entire class has changed:", new

    def _students_items_changed(self, event): # <-- called when list items changed
        """ event.added -- A list of the items added to students
            event.removed -- A list of the items removed from students
            event.index -- Start index of the items that were added/removed
        """
        if event.added: print "added (index,name):", event.index, event.added
        else:  print "removed (index,name):", event.index, event.removed
```

```
# Demo Code
>>> school_class = SchoolClass()
>>> school_class.students = ["John", "Jane", "Jill"] # initial set of students.
The entire class has changed: ['John', 'Jane', 'Jill']
>>> school_class.students.append("Bill")              # add a student
students added (index,name): 3 ['Bill']
>>> del school_class.students[1:3]                    # remove some students
students removed (index,name): 1 ['Jane', 'Jill']
```


ENTHOUGHT

List Trait UI

```
# Create a customized view of the list.
>>> view = View('teacher',
                Group(
                      Item('students',
                           style='custom',
                           editor=ListEditor(rows=5),
                           show_label=False
                          ),
                      show_border=False,
                      label='Students'
                      ),
                title = 'Class',
                width=300,
                height=200,
                resizable=True
                )

>>> school_class.edit_traits(view=view)
```



ENTHOUGHT

Dict Traits

```
class MultipleDicts(HasTraits):

  # Signature: Dict(key_type, value_type)

  # Basic dictionary with unchecked key/value types
  # Same as Dict(Any, Any)
  a = Dict

  # Dictionary with checked Str key type
  # Same as Dict(Str, Any)
  b = Dict(Str)

  # Dictionary with string for keys and floats for values
  c = Dict(Str, Float)

  # Default value specified for the dictionary
  d = Dict(Str, Float, value={"hello": 1.0})
```


ENTHOUGHT

Array Traits

```
from numpy import float32, int32
from traits.api import Array, HasTraits

class TriangleMesh(HasTraits):

    # An Nx3 floating point array of points (vertices) within the mesh.
    points = Array(dtype=float32, shape = (None,3))

    # An Mx3 integer array of indices into the points array.
    # Each row defines a triangle in the mesh.
    triangles = Array(dtype=int32, shape=(None,3))

# Demo Code
points = numpy.array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], dtype=float32)
triangles = numpy.array([[0,1,3], [0,3,2], [1,2,3], [0,2,1]], dtype=int32)

# Demo Code
>>> tetra = TriangleMesh()
# Set the data points and connectivity
>>> tetra.points = points
>>> tetra.triangles = triangles
# THIS WILL RAISE AN EXCEPTION
>>> tetra.points = points[:,:2]
TraitError: The 'points' trait of a TriangleMesh instance must be an array of float32
    values with shape ('*', 3), but a value of array([[ 0.,  0.],
```

ENTHOUGHT

Instance Objects – instance_1.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self): return 'Person("%s %s")' % (self.first_name, self.last_name)

class Family(HasTraits):
    # Instantiate the default Person
    dad = Instance(Person,args=())

    # Instantiate a Person object with a different first name
    mom = Instance(Person, args=(), kw={'first_name':'Jane'})

    # Son is a Person object, but it defaults to 'None'
    son = Instance(Person)

    # In case you need "forward" declarations, you can use
    # the name as a string.  Default is None
    daughter = Instance('Person')
```

```
# Demo Code
>>> family = Family()
>>> family.dad
Person("John Doe")
>>> family.mom
Person("Jane Doe")
>>> family.son
None
```

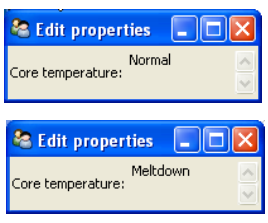
```
>>> family.daughter
None
>>> family.son = Person(first_name="Bubba")
>>> family.daughter = Person(first_name='Sissy')
>>> family.son
Person("Bubba Doe")
>>> family.daughter
Person("Sissy Doe")
```


ENTHOUGHT

Model View Pattern – model_view_1.py

```
class Reactor(HasTraits):
   core_temperature = Range(-273.0, 100000.0)

class ReactorModelView(HasTraits):

    model = Instance(Reactor)

    # The "dummy" view of the reactor should be a warning string.
    core_temperature = Property(depends_on='model.core_temperature')

    def _get_core_temperature(self):
        temp = self.model.core_temperature
        if temp <= 500.0:
           return 'Normal'
        if temp < 2000.0:
           return 'Warning'
        return 'Meltdown'

my_view = View(Item('core_temperature', style = 'readonly'))
```

```
# Demo Code
>>> reactor = Reactor( core_temperature = 200.0 )
>>> view = ReactorModelView(model=reactor)
>>> view.edit_traits(view=my_view)

# Now change the temperature
>>> reactor.core_temperature = 5000.0
```



ENTHOUGHT

Delegation – instance_delegate.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self):
        return 'Person("%s %s")' % (self.first_name, self.last_name)

class Child(Person):
    parent = Instance(Person, args=())

    # Define last_name to "delegate" to the parent's last name
    last_name = DelegatesTo('parent', 'last_name')

# Demo Code
>>> dad = Person(first_name="Sam", last_name="Barns")
>>> child = Child(first_name="Jane", parent=dad)
>>> dad
Person("Sam Barns")
>>> child
Person("Jane Barns")
# Changing the child's last name changes the parent's as well
>>> child.last_name = "Smith"
>>> print dad, child
Person("Sam Smith") Person("Jane Smith")
```


ENTHOUGHT

Prototyping – instance_prototype.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self):
        return 'Person("%s %s")' % (self.first_name, self.last_name)

class Child(Person):
    parent = Instance(Person, args=())

    # Define last_name to "prototype" using the parent's last name
    last_name = PrototypedFrom('parent', 'last_name')

# Demo Code
>>> dad = Person(first_name="Sam", last_name="Barns")
>>> child = Child(first_name="Jane", parent=dad)
>>> dad
Person("Sam Barns")
>>> child
Person("Jane Barns")
# Copy on write, but linked before that point
>>> child.last_name = "Smith"
>>> print dad, child
Person("Sam Barns") Person("Jane Smith")
```

ENTHOUGHT

Instance List — instance_observer_1.py

```
class Person(HasTraits):
    name = Str
    age = Int

class SchoolClass(HasTraits):
    teacher = Instance(Person)
    students = List(Person)

    def _age_changed_for_teacher(self, object, name, old, new):
        print 'The teacher is now ', new, ' years old.'s

    def _age_changed_for_students(self, object, name, old, new):
        print object.name, ' is now ', new, ' years old.'
```

```
# Demo Code
>>> the_class = SchoolClass()
>>> teacher_ben = Person(name="Ben",
                         age=35)
>>> the_class.teacher = teacher_ben
>>> bob = Person(name="Bob", age=10)
>>> the_class.students.append(bob)
>>> jane = Person(name="Jane", age=11)
>>> the_class.students.append(jane)
```

```
# This calls the SchoolClass observer
>>> teacher_ben.age = 36
The teacher is now 36 years old.
# This calls the SchoolClass observer
>>> bob.age = 11
Bob is now 11 years old.
# Remove Bob from the Class and the
# observer is no longer called.
>>> the_class.students.remove(bob)
>>> bob.age = 12
```


ENTHOUGHT

Event Traits – event_1.py

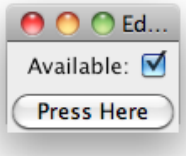
```
class Rectangle(HasTraits):

    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # Set to notify others that you have made changes to a Rectangle
    # Listen to this if you want to react to any changes to a Rectangle
    updated = Event

def rect_printer(rect, name, value):
    print 'rectangle (width, height):', rect.width, rect.height
```

```
# Demo Code
>>> rect = Rectangle()
# Hook up a dynamic listener to respond whenever rect is updated.
>>> rect.on_trait_change(rect_printer, name='updated')
# update multiple items
>>> rect.width = 10
>>> rect.height = 20
# now explicitly tell the item that it is updated
>>> rect.updated = True
rectangle (width, height): 10.0 20.0
```

ENTHOUGHT

Button Traits – button_1.py

```
class ButtonDemo(HasTraits):

    # Enables the button.
    available = Bool(True)

    # A "button" trait.
    click = Button("Press Here")

    traits_view = View(Group('available'),
                       UItem('click', enabled_when='available'))

    def _click_fired(self):
        print "You clicked the button."
```



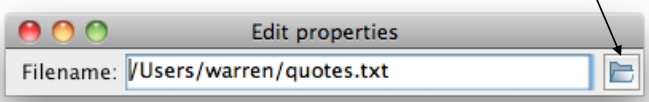
This code also demonstrates a simple use of the '**enabled_when**' option of the Item and UItem objects. Its value can be a string containing a python expression using the object's traits. If the expression evaluates to False, the GUI item is disabled.

ENTHOUGHT

File Traits – file_trait.py

```
from traits.api import HasTraits, File
from traitsui.api import View

class CharacterCount(HasTraits):
    filename = File()
    traits_view = View('filename', width=400)

    def _filename_changed(self):
        fname = self.filename
        try:
            with open(fname, 'rb') as f:
                n = len(f.read())
            print '"{:s}" has {:d} chars.'.format(fname, n)
        except IOError:
            # Ignore file errors.
            pass

if __name__ == "__main__":
    cc = CharacterCount()
    cc.edit_traits()
```

Click to open a file dialog.



ENTHOUGHT

Editors

- An ***editor*** is the GUI element that allows a user to view and modify a trait.
- Editors are defined in the **`traitsui`** package. Import them from **`traitsui.api`**.
- The **`Item`** object allows you to override the default editor of a trait.
- The **`style`** keyword argument of **`Item`** can be used to select different styles of the editor.
- Traits has *many* editors. We'll look at three: **`RangeEditor`**, **`CheckListEditor`**, and **`ArrayEditor`**.

RangeEditor

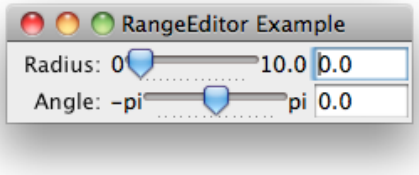
- The RangeEditor is the default editor for a Range trait.
- It can also be used as the editor for any numerical trait.

ENTHOUGHT

Editors

EXAMPLE -- range_editor_1.py

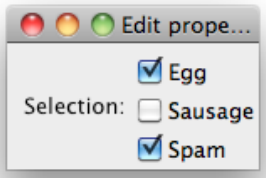
```
from math import pi
from traits.api import HasTraits, Float
from traitsui.api import View, Item, RangeEditor

class Example(HasTraits):
    radius = Float
    angle = Float
    traits_view = View(
        Item('radius', editor=RangeEditor(low=0, high=10)),
        Item('angle', editor=RangeEditor(low=-pi, high=pi,
                                         low_label='-pi', high_label='pi')),
        title="RangeEditor Example")

if __name__ == "__main__":
    example = Example()
    example.edit_traits()
```



Editors

CheckListEditor

- A **`CheckListEditor`** provides an editor for a list.
- The 'simple' style lets the user select one element from a drop-down list.
- The 'custom' style shows all the options with check boxes.



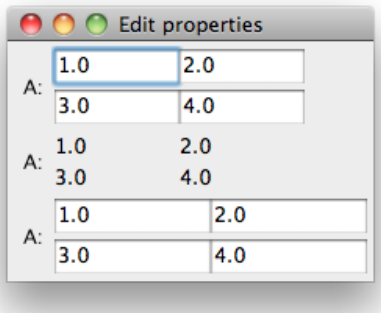
EXAMPLE -- checklist_editor_1.py

```
from traits.api import HasTraits, List, Str
from traitsui.api import View, Item, CheckListEditor

class Example(HasTraits):
    selection = List(Str)
    traits_view = View(
        Item('selection', style='custom',
             editor=CheckListEditor(
                 values=['egg','sausage','spam'])))
    def _selection_changed(self):
        print "selection is", self.selection

if __name__ == "__main__":
    example = Example()
    example.edit_traits()
```


Editors

ArrayEditor

- An **`ArrayEditor`** provides an editor for a 1D or 2D Array trait.
- The 'width' keyword alters the width of each field.



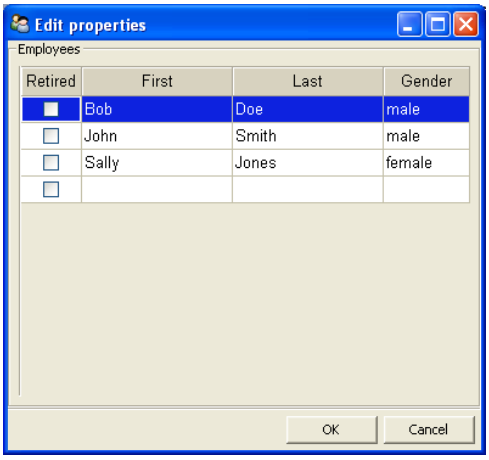
EXAMPLE -- array_editor.py

```
import numpy as np
from traits.api import HasTraits, Array
from traitsui.api import View, Item, ArrayEditor

class Data(HasTraits):
    a = Array()
    def _a_default(self):
        a = np.array([[1.0, 2.0], [3.0, 4.0]])
        return a
    trauts_view = \
        View(
            Item('a', editor=ArrayEditor()),
            Item('a', editor=ArrayEditor(),
                                  style='readonly'),
            Item('a', editor=ArrayEditor(width=48))
        )

if __name__ == "__main__":
    data = Data()
    data.configure_traits()
```

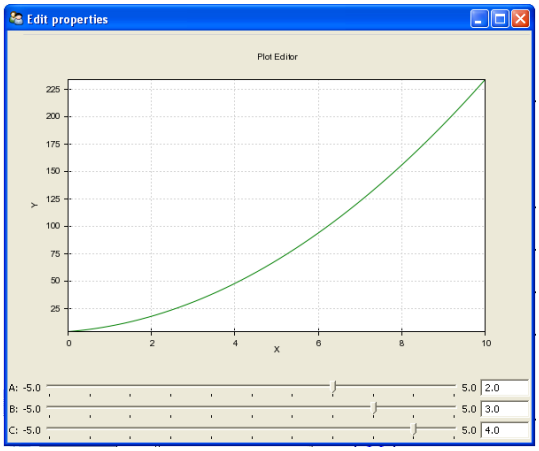
ENTHOUGHT

UI Demos

Table Demo



Polynomial Demo



ENTHOUGHT

More Traits UI

ENTHOUGHT

edit_traits()

The edit_traits() and configure_traits() methods take the following arguments.

ARGUMENTS

view: the view to use (if not the default)

kind: the kind of the view

handler: the handler for the view

parent: the parent ui to embed a panel into.

context: the context for the ui.

UI OBJECTS

The edit_traits() and configure_traits() methods return a UI object. For modal dialogs the UI object has a result trait that is True if the dialog was not cancelled.

The UI also has traits keeping track of the main moving parts of a user interaction: the context, view, handler and UIInfo, as well as the toolkit's control.

KINDS

modal: the view is in a modal dialog, object updated when dialog closes (or apply button pressed)

livemodal: the view is a modal dialog, object is updated live

nonmodal: the view is a nonmodal dialog, object updated on close or apply.

live: nonmodal, live updates.

panel: a view embedded in another window, has command buttons.

subpanel: a view embedded in another window, no command buttons

wizard: a wizard dialog. Wizards are live and modal, but with their own standard button set. Each group in the view is a page in the wizard.

ENTHOUGHT

View

There are a quite a number of traits available to control the appearance and behaviour of the View.

TRAITS

height, width: the requested height and width as pixels or proportion of screen

x, y: the requested x and y coordinates for the window (positive for top/left, negative for bottom/right, either pixels or proportions)

resizable: can the view be resized?

scrollable: should scroll bars be added if the size of the window is too small

title: the title to display on the window

icon: the icon to display on the window

dock: how to arrange subgroups: 'fixed', 'horizontal', 'vertical' or 'tabbed'.

image: the image to display in tabs

close_result: what to return if the window is closed via the window's close button

handler: a handler for the view (see later)

menubar: the menubar for the view

toolbar: a toolbar for the view

statusbar: a statusbar for the view

buttons: the buttons in the view

key_bindings: key bindings for the view

style: the default style for editors

id: a globally unique id for preferences

object: the object being edited

ENTHOUGHT

Group

Items in view can be linked together in Groups, and layed out in various ways.

TRAITS

height, width: the requested height and width as pixels or proportion of screen

padding: the amount of padding about the group

show_border: should a border be shown or not

label: the label to display on the group

layout: the layout style of the group, one of 'normal', 'flow', 'split' or 'tabbed'

orientation: the orientation of the group

columns: the number of columns in the group.

dock: dock style of sub-groups

show_labels: show labels of items?

show_left: show labels on the left or the right

selected: in a tabbed layout, should this be the visible tab?

image: image to show on tabs

springy: use extra space in the parent layout?

defined_when: expression that determines inclusion of group in parent

visible_when: expression that determines visibility of group

enabled_when: expression that determines whether of group can be edited

ENTHOUGHT

Item

Items are the basic units of layout, and usually contain trait editors.

TRAITS

label: the label to display on the item

name: the name of the trait being edited

editor: the editor to use

style: the editor style

height, width: the requested height and width as pixels or proportion of screen

padding: the amount of padding about the group

resizable: can the item be resized to use extra space

springy: use extra space in the parent layout?

full_size: expand to use all available space in non-layout direction?

has_focus: should this item get initial focus?

emphasized: should label text be emphasized?

tooltip: tooltip to display on mouse-over

image: image to show on tabs

format_str: % format string for text

format_func: format function for text

invalid: trait name for holding editor state

defined_when: expression that determines inclusion of group in parent

visible_when: expression that determines visibility of group

enabled_when: expression that determines whether of group can be edited

ENTHOUGHT

Subclasses

Several subclasses of Group and Item are defined for convenience

GROUP SUBCLASSES

HGroup: a horizontally arranged group

HFlow: a horizontally arranged group that flows vertically once horizontal space is used up

HSplit: a horizontally arranged group with a splitter bar to separate it from other groups

Tabbed: a group displayed as a notebook tab

VGroup: a vertically arranged group

VFlow: as HFlow, but vertical

VFold: a vertically arranged group where items can be collabes by clicking their titles

VGrid: a grid, default 2 columns

ITEM SUBCLASSES

Label: an item which is a label

Heading: an item which is a heading

Spring: an item that injects springy space into a layout

Custom: an item with a custom editor style

Readonly: an item with readonly editor style

UItem, UCustom, UReadonly: as Item, Custom or Readonly, but with no label.

ENTHOUGHT

A Word On Using Traits Themes

Don't!

ENTHOUGHT

Helpful resources

- Docs
  http://docs.enthought.com/traits
- Mailing List Help
  enthought-dev@enthought.com
- GitHub
  https://github.com/enthought/traits

ENTHOUGHT

Interactive Plotting with Chaco

ENTHOUGHT

About this tutorial

Recommended prerequisites:

- Know a little NumPy
- Know a little bit about GUI programming
- Know a little bit about Traits and Traits UI (from previous tutorial)

ENTHOUGHT

Goals

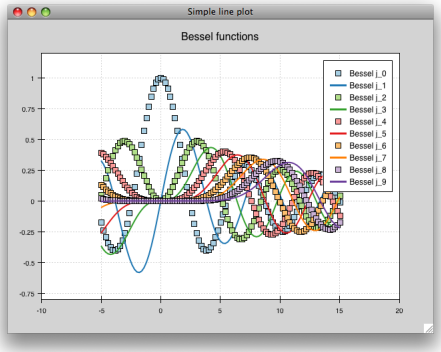
By the end of this tutorial, you will have learned how to:

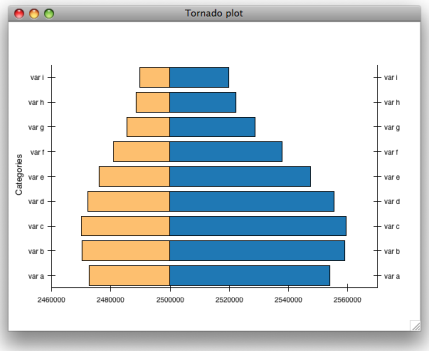
- Create Chaco plots of various types
- Arrange plots of data items in various layouts
- Configure and interact with your plots using Traits UI
- Create a custom plot renderer
- Create a custom tool that interacts with the mouse

ENTHOUGHT

Introduction

- Chaco is a *plotting application toolkit*
- You can build simple, static plots

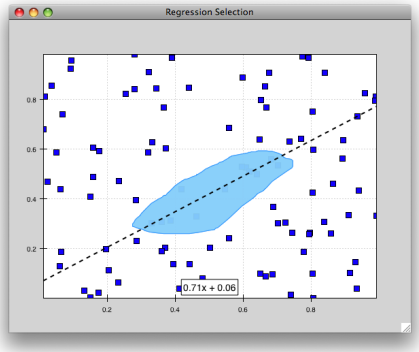


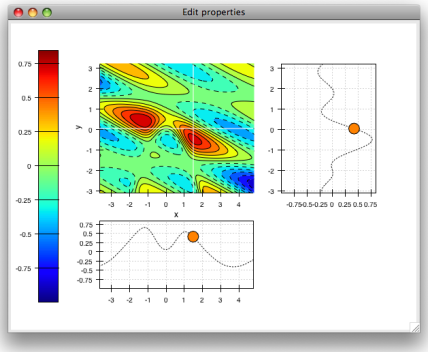


ENTHOUGHT

Introduction

- Chaco is a *plotting application toolkit*
- You can build simple, static plots
- You can also build rich, interactive visualizations:

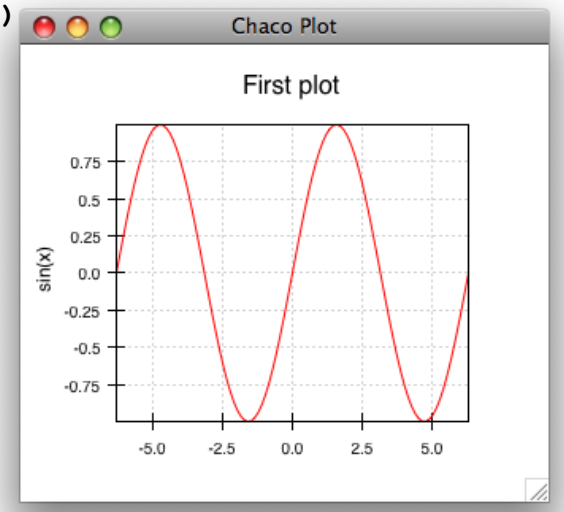




ENTHOUGHT

“Script-oriented” Plotting

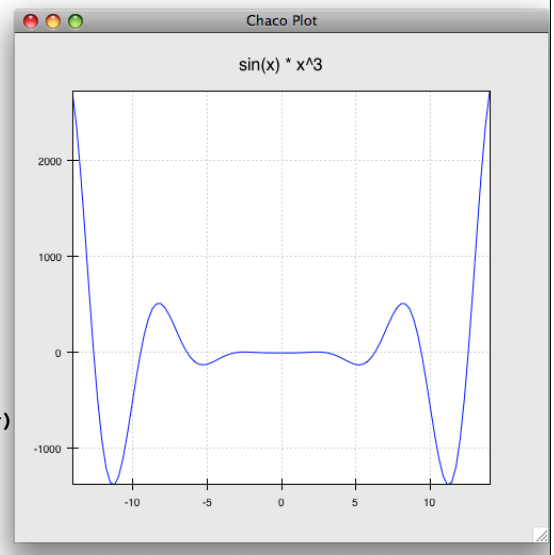
```
>>> from numpy import *
>>> from chaco.shell import *

>>> x = linspace(-2*pi, 2*pi, 100)
>>> y = sin(x)

>>> plot(x, y, 'r-')
>>> title('First plot')
>>> ytitle('sin(x)')
```



ENTHOUGHT

“Application-oriented” Plotting

```
from chaco.api import Plot, ArrayPlotData
from enable.api import ComponentEditor
from traits.api import HasTraits, Instance
from traitsui.api import View, Item
import numpy as np

class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default (self):
        x = np.linspace(-14, 14, 100)
        y = np.sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line",
                  color="blue")
        plot.title = "sin(x) * x^3"
        return plot

if __name__ == "__main__":
    LinePlot().configure_traits()
```



first_plot.py

ENTHOUGHT

Details 1/2: the Plot instance

```
from chaco.api import Plot
<snip>
class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")
<snip>
```

- The easiest way to embed a Chaco plot in a window is using Traits. Here `plot` is an instance of Chaco's `Plot` (a kind of `Trait`) is contained in the view (another kind of `Trait`).
- The view will use the standard editor for a generic Enable component.

ENTHOUGHT

Details 2/2: Initializing the Plot

```
from chaco.api import Plot, ArrayPlotData
<snip>

    def _plot_default (self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        plot.title = "sin(x) * x^3"
        return plot
```

- The initialization of the plot requires adding numpy arrays to an ArrayPlotData, and creating the `Plot` instance from it. It is a dict-like HasTraits object that stores numpy arrays with keys 'x' and 'y'. Modifying the underlying data then triggers a redraw of the plot using Traits listeners architecture.
- Drawing a line representing 'y' as a function of 'x' requires the creation of a renderer using the `plot` method of the `Plot` instance.

Conclusion: the central object is a Chaco plot is an instance of a `chaco.plot.Plot`. It links the datasets to the renderers. It is both a factory and a container for all renderers.

ENTHOUGHT

Conclusion: architecture of Chaco

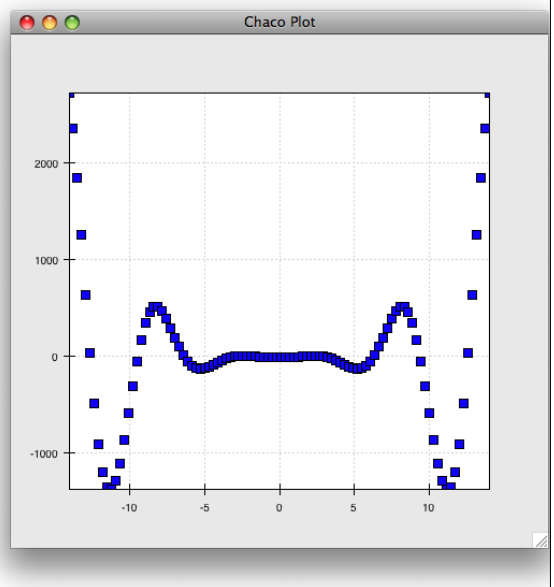
Creating a Chaco plot requires 3 central components:

1. An `ArrayPlotData` stores `np` arrays.
2. A Chaco instance of a `Plot` holds pointers to this data and controls the plotting structure
3. Renderers such as `lineplot`, `scatter` are added to the `Plot` instance to represent data.

ENTHOUGHT

Scatter Plot

```
class ScatterPlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter",
                  color="blue")
        return plot

if __name__ == "__main__":
    ScatterPlot().configure_traits()
```



scatter.py

ENTHOUGHT

Image Plot

```
class ImagePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default(self):
        x = linspace(0, 10, 51)
        y = linspace(0, 5, 51)
        X, Y = meshgrid(x, y)
        z = exp(-(X**2 + Y**2) / 100)
        plotdata = ArrayPlotData(zdata=z[:-1,:-1])
        plot = Plot(plotdata)
        plot.img_plot("zdata", xbounds=x,
                      ybounds=y, colormap=jet)
        return plot

if __name__ == "__main__":
    ImagePlot().configure_traits()
```

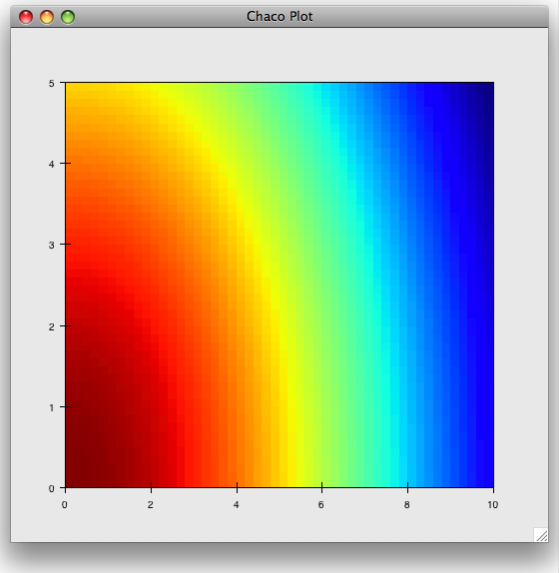
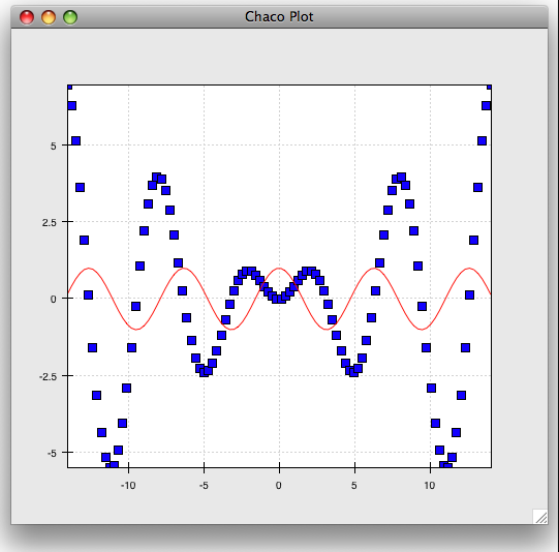


image.py

ENTHOUGHT

Multiple Line Plots

```
class OverlappingPlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = x/2 * sin(x)
        y2 = cos(x)
        plotdata = ArrayPlotData(x=x,y=y,y2=y2)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter",
            color="blue")
        plot.plot(("x", "y2"), type="line",
                  color="red")
        return plot

if __name__ == "__main__":
    OverlappingPlot().configure_traits()
```

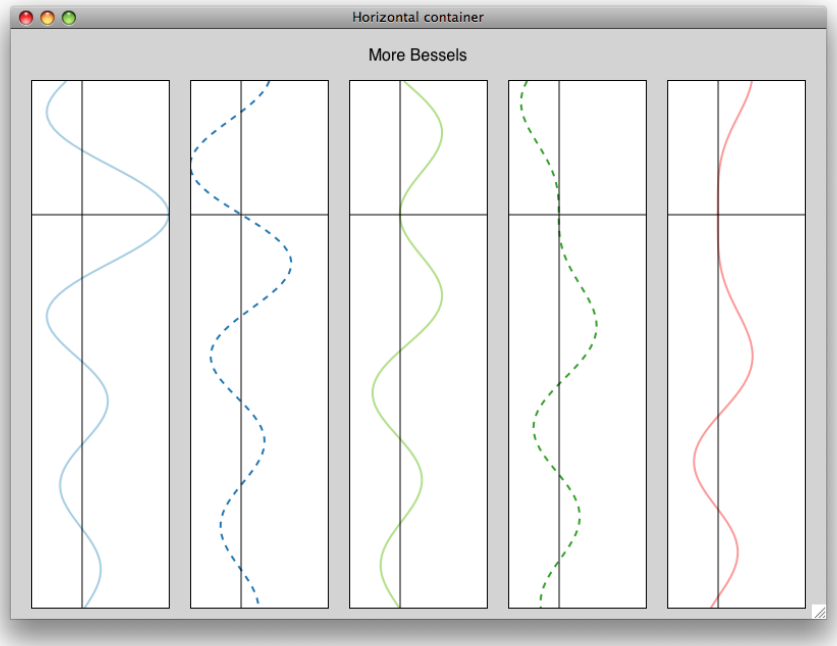


overlapping.py

ENTHOUGHT

HPlotContainer

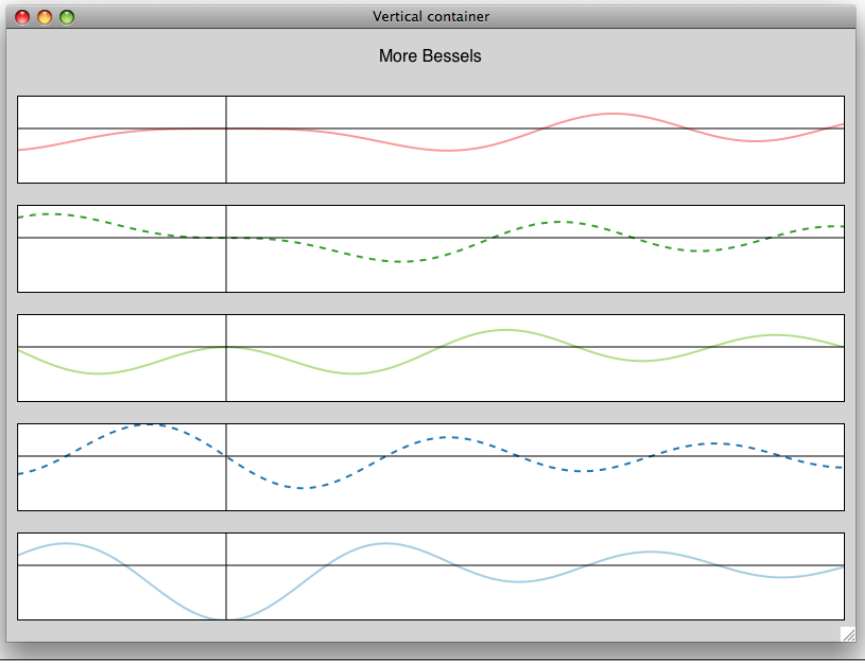
There are multiple types of containers to organize the Plots horizontally, …



ENTHOUGHT

VPlotContainer

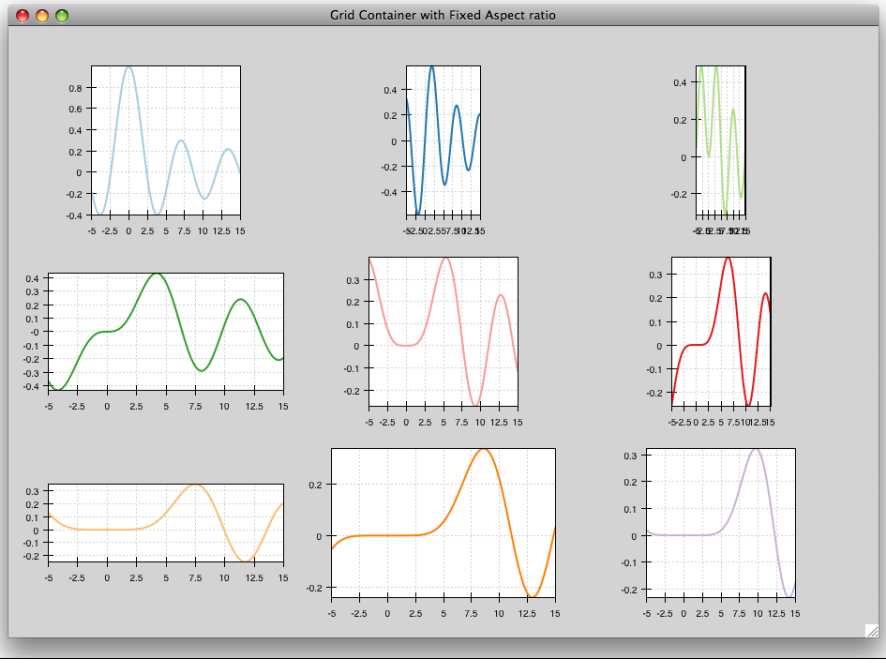
… vertically, …



ENTHOUGHT

GridContainer

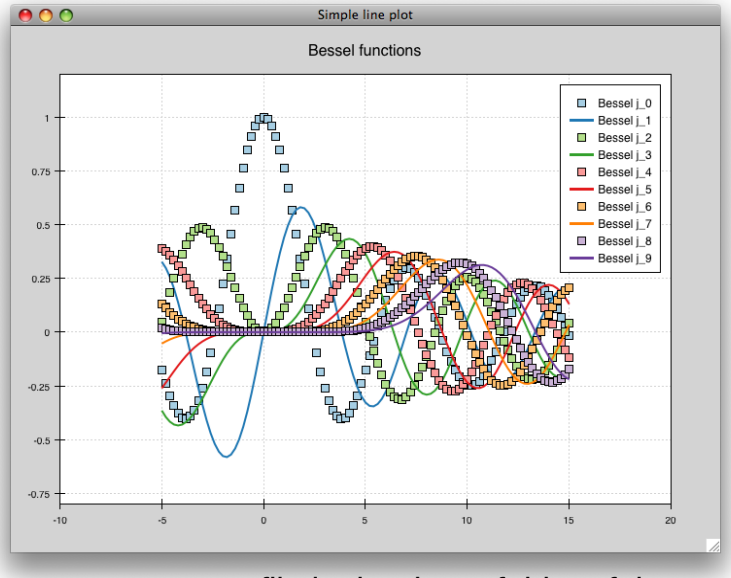
… or into a grid.



ENTHOUGHT

OverlayPlotContainer

In an overlayPlotContainer, plots are superimposed on top of each other. They can share both axes…

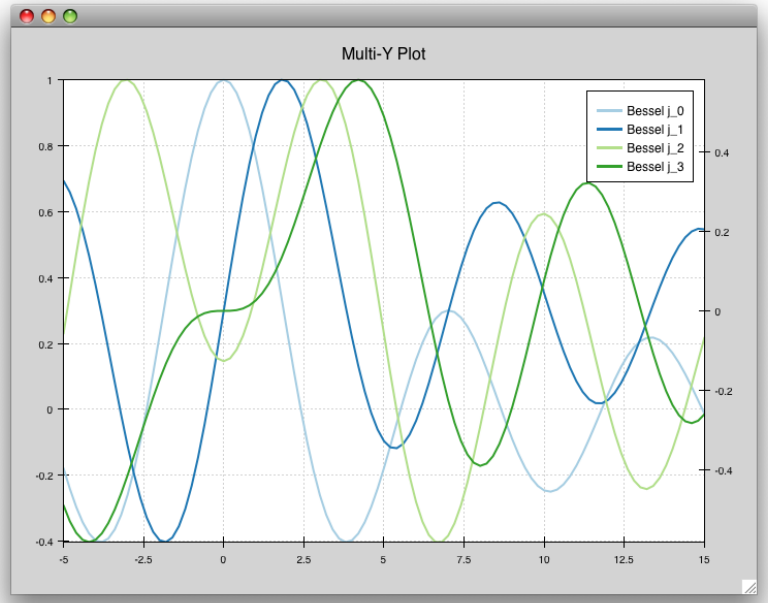


See the `simple_line.py` file in the demo folder of the source code.

ENTHOUGHT

OverlayPlotContainer

… or only one of the axes. In this case they all only share the x axis:



See the `multiaxis.py` file in the demo folder of the source code.

ENTHOUGHT

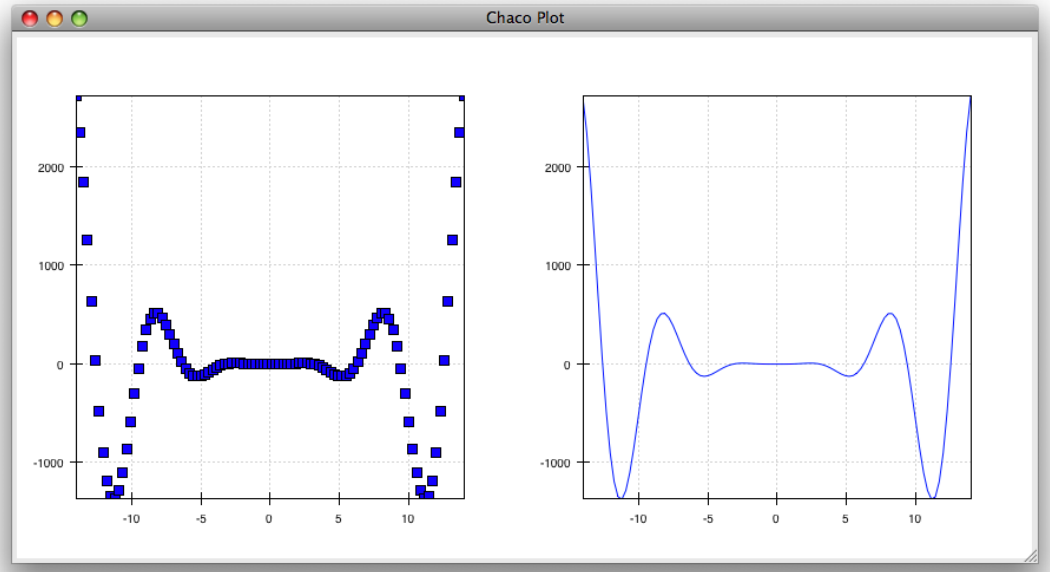
Using a Container

The Plot is replaced by a container and several instances of Plot are passed into the container:

```
from chaco.api import HPlotContainer, Plot
<snip>

class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)
    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")

        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        container = HPlotContainer(scatter, line)
        return container
```

ENTHOUGHT

Using a Container



container.py

ENTHOUGHT

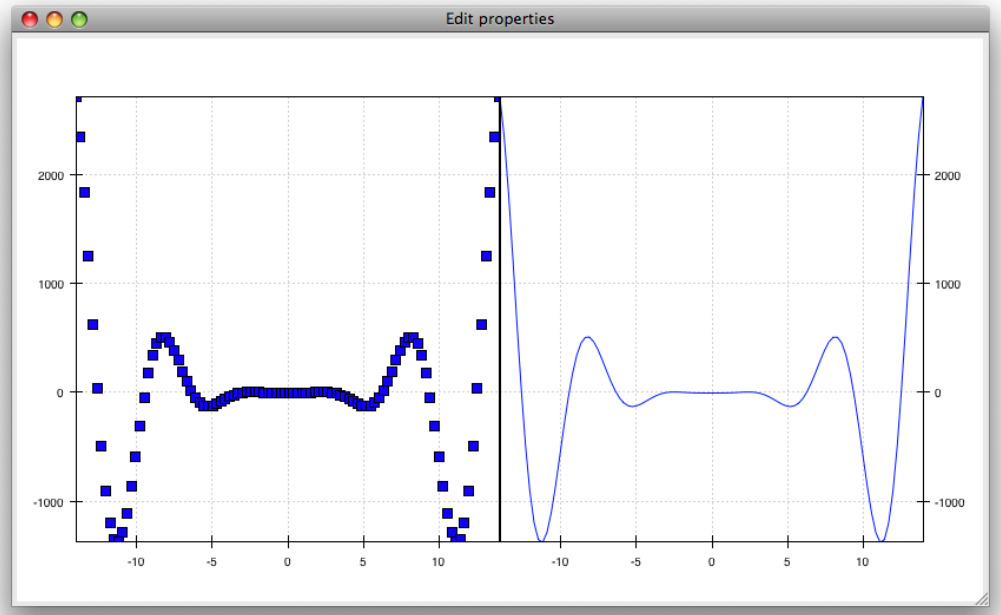
Configuring the Container

Let's modify the previous example to have the two plots touch each other:

```
class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)
    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")

        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        container = HPlotContainer(scatter, line)
        container.spacing = 0
        scatter.padding_right = 0
        line.padding_left = 0
        line.y_axis.orientation = "right"
        return container
```

ENTHOUGHT

Configuring the Container



`container_nospace.py`

ENTHOUGHT

Controlling the plot attributes

Let's build a traitsUI application that controls some of the scatter plot's attributes:

```
from traits.api import Int, Property
from enable.api import ColorTrait
from chaco.api import marker_trait

class ScatterPlotTraits(HasTraits):
    plot = Instance(Plot)
    color = ColorTrait("blue")
    marker = marker_trait
    marker_size = Int(4)
    renderer = Property(depends_on=['plot'])
    traits_view = View(
        VGroup(
            Item('color', label="Color", style="custom"),
            Item('marker', label="Marker"),
            Item('marker_size', label="Size"),
            Item('plot', editor=ComponentEditor(),
                 show_label=False)),
        width=500, height=500,
        resizable=True, title="Chaco Plot")
```

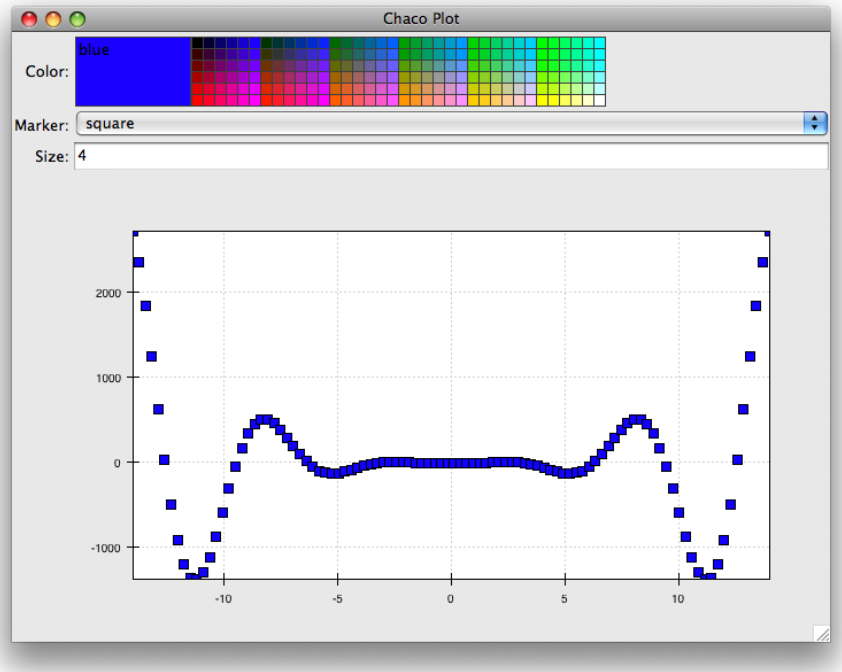
ENTHOUGHT

Controlling the plot attributes

```
def _plot_default(self):

    x = linspace(-14, 14, 100)
    y = sin(x) * x**3
    plotdata = ArrayPlotData(x = x, y = y)

    plot = Plot(plotdata)

    plot.plot(("x", "y"), type="scatter", color="blue", name="XY")
    return plot

def _get_renderer(self):
    return self.plot.plots['XY'][0]

def _color_changed(self):
    self.renderer.color = self.color

def _marker_changed(self):
    self.renderer.marker = self.marker

def _marker_size_changed(self):
    self.renderer.marker_size = self.marker_size
```


ENTHOUGHT

Controlling the plot attributes



traits_demo.py

ENTHOUGHT

Data Chooser

Let's create another traits application that offers an interactive way to change the underlying data (in this case the function) to be plotted.

First we will create like in the previous example some addition traits to offer these controls to the user.

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    data = Dict
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(),
          show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")
<snip>
```


ENTHOUGHT

Data Chooser

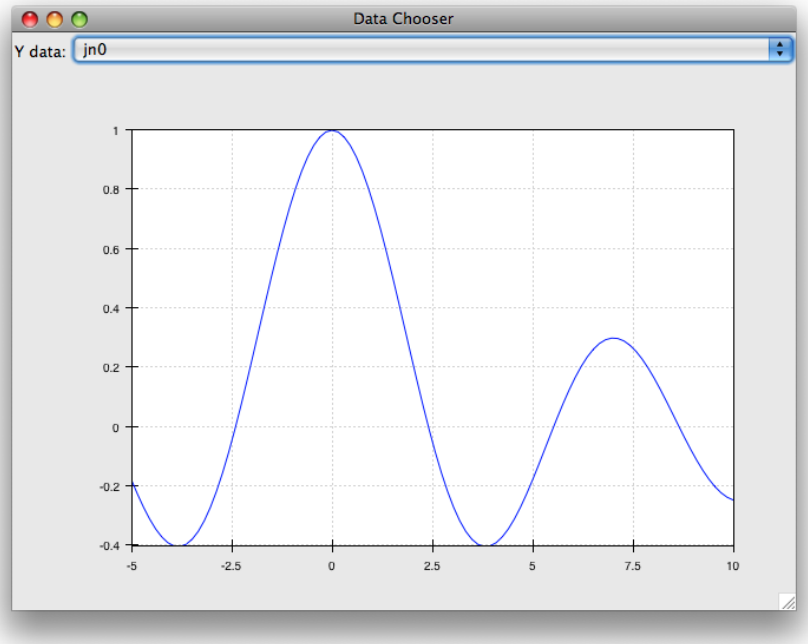
First, to speed things up and simplify code, we will pre-compute all possible functions in the plot initialization and store it in the `data` dict.

But the most important step is to add a listener to changes to the `data_name` trait (changed in the UI), which needs to trigger a change of the y entry in the `ArrayPlotData` instance. The latter is accessible through the `data` attribute of the `Plot` object.

```
<snip>
   def _plot_default(self):
        x = linspace(-5, 10, 100)
        self.data = dict(jn0=jn(0,x),jn1=jn(1,x),jn2=jn(2,x))
        plotdata = ArrayPlotData(x=x,y=self.data[self.data_name])
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        return plot

    def _data_name_changed(self):
        self.plot.data.set_data("y", self.data[self.data_name])
```

ENTHOUGHT

Data Chooser



data_chooser.py

ENTHOUGHT

Chaco tools

- Tools in Chaco are usually attached to a `Plot` object, which has hooks to add them through its `tools` (and optionally `overlays`) attribute(s).
- Standard tools in Chaco can be found in `chaco.tools.api`.
- Tools keep a reference to the `Plot` (or Enable component) they are acting on.
- Chaco offers a framework to define custom tools (see later).

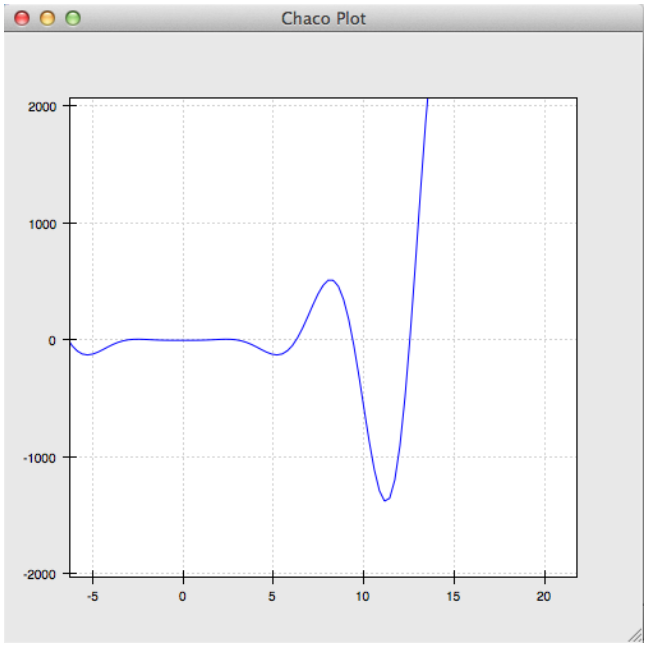
ENTHOUGHT

Example: tools to the line plot

```
from chaco.tools.api import PanTool, ZoomTool, DragZoom

class ToolsExample(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")

        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        plot.tools.append(DragZoom(component=plot,
                                   drag_button="right"))
        return plot
```


ENTHOUGHT

Example: tools to the line plot



tools.py

ENTHOUGHT

Quick Recap

- Plot creation
  - Using ArrayPlotData to link data to plots
  - Using Plot to create different types of plots and multiple overlapping plots
- Plot layout using containers
- Plot configuration
  - Using Traits UI to modify plot attributes
  - Using Traits UI to modify plot data
- Tools

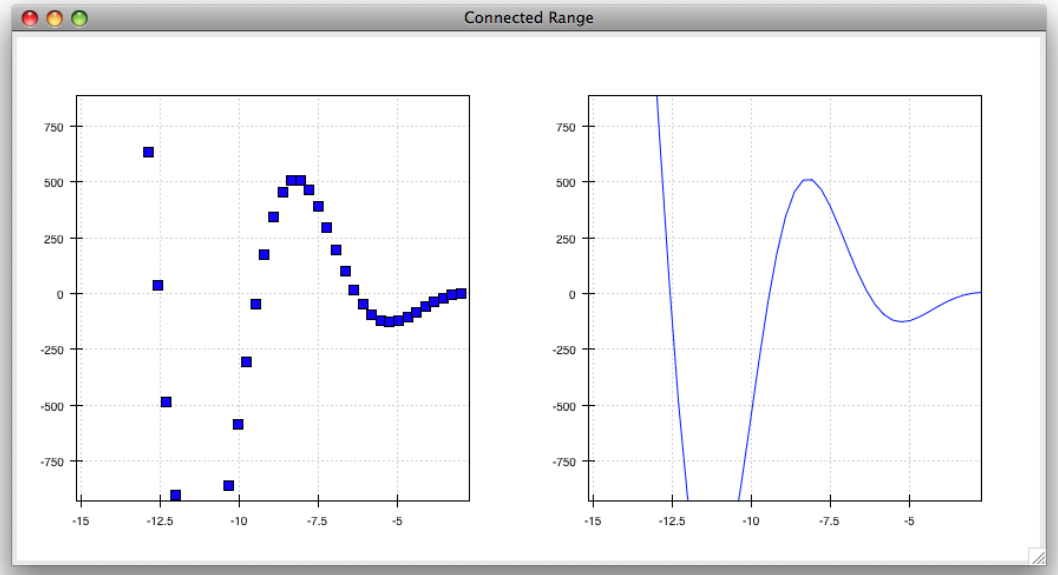
ENTHOUGHT

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def _container_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))
        # connect the ranges so that zooming in one zooms in the other
        scatter.range2d = line.range2d

        return HPlotContainer(scatter, line)
```

ENTHOUGHT

Connected Plots



connected_range.py

ENTHOUGHT

Partially Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def _container_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))
        # Connect only the range of x axis
        scatter.index_range = line.index_range

        return HPlotContainer(scatter, line)
```

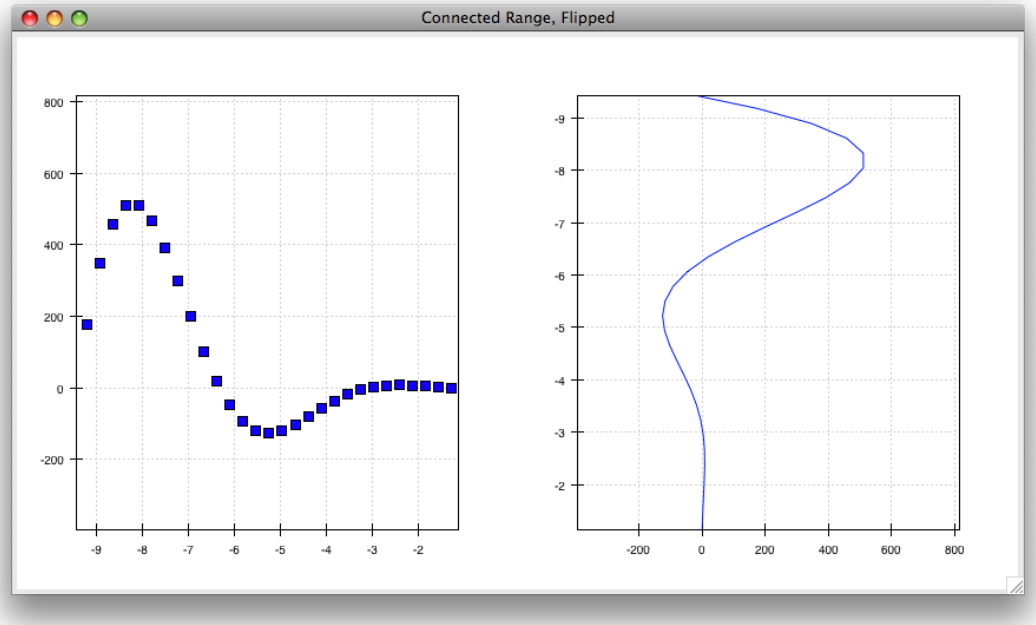
ENTHOUGHT

Flipped Connected Plots

```
class FlippedExample(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range, Flipped")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        # Make the x axis vertical
        line = Plot(plotdata, orientation="v", default_origin="top left")
        line.plot(("x", "y"), type="line", color="blue")

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))
        scatter.range2d = line.range2d

        return HPlotContainer(scatter, line)
```


ENTHOUGHT

Flipped Connected Plots



connected_orientation.py

ENTHOUGHT

Connected figures

Let's illustrate connecting plots that are in distinct figures. We will offer the possibility for each plot to control its type and orientation.

```
class PlotEditor(HasTraits):
    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                        width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        return plot
```


ENTHOUGHT

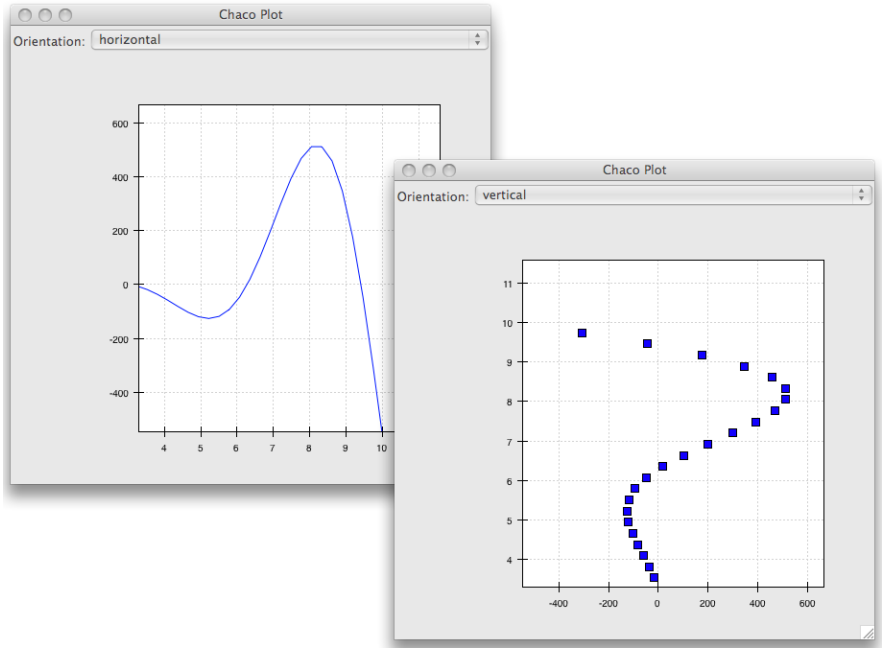
Two Windows

We define a listener for changes to the orientation attribute. We also connect the ranges between the two Plot attributes of our instances, despite the fact that they are their own distinct traitsUI windows.

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"

if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```

ENTHOUGHT

Two Windows



connected_widgets.py

ENTHOUGHT

Interacting with IPython

Interestingly part of the interactivity of the previous example was in the __main__ (that is interactive) part of the script.

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"

if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```

ENTHOUGHT

Interacting with IPython

```
$ ipython --gui 'wx'
Python 2.7.2 |EPD 7.3-1 (32-bit)| (default, Sep  7 2011, 09:16:50)
Type "copyright", "credits" or "license" for more information.

IPython 0.14.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: from ploteditor import PlotEditor

In [2]: p1 = PlotEditor()

In [3]: p2 = PlotEditor()

In [4]: p3 = PlotEditor()

In [5]: p1.edit_traits(); p2.edit_traits(); p3.edit_traits()
Out[5]: <enthought.traits.ui.ui.UI object at 0x195ce210>
Out[5]: <enthought.traits.ui.ui.UI object at 0x199d2660>
Out[5]: <enthought.traits.ui.ui.UI object at 0x199fb0c0>
```


ENTHOUGHT

Writing a Custom Tool

custom_tool_screen.py

```
from enable.api import BaseTool

class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

class ScatterPlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=800, height=600, resizable=True,
                       title="Custom Tool")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter", color="blue")
        plot.tools.append(CustomTool(plot))
        return plot
```

ENTHOUGHT

Listening to clicks

Other event names beyond “mouse_move” are “mouse_enter”, “mouse_leave”, “mouse_wheel”, “left_down”, “left_up”, “right_down”, “right_up”, “key_pressed”. So, we can modify the tool to do things on mouse clicks as well.

```
class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

    def normal_left_down(self, event):
        print "Mouse went down at", event.x, event.y

    def normal_left_up(self, event):
        print "Mouse went up at:", event.x, event.y
```

custom_tool_click.py

ENTHOUGHT

A multi-state custom tool

Increasing complexity can be obtained by modifying the state of the tool: so far it was always in the default (“normal”) state. Here we define a “mousedown” state as well:

```
class CustomTool(BaseTool):
        event_state = Enum("normal", "mousedown")

        def normal_mouse_move(self, event):
            print "Screen:", event.x, event.y

        def normal_left_down(self, event):
            self.event_state = "mousedown"
            event.handled = True

        def mousedown_left_up(self, event):
            self.event_state = "normal"
            event.handled = True
```

ENTHOUGHT

Accessing coord in data space

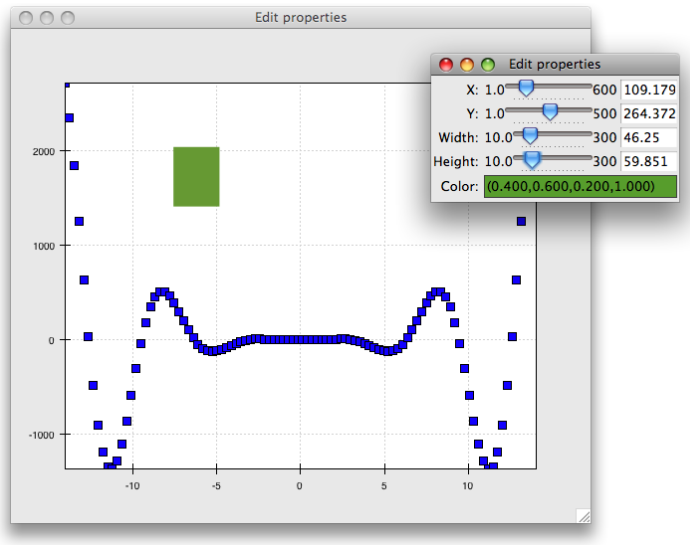
The event only knows about the screen coordinates. The map_data method on the Plot object can convert them to data coordinates:

```
class CustomTool(BaseTool):

        event_state = Enum("normal", "mousedown")
        def normal_mouse_move(self, event):
            print "Screen:", event.x, event.y

        def normal_left_down(self, event):
            self.event_state = "mousedown"
            event.handled = True

        def mousedown_mouse_move(self, event):
            print "Data:", self.component.map_data((event.x,
                                                    event.y))
        def mousedown_left_up(self, event):
            self.event_state = "normal"
            event.handled = True
```


ENTHOUGHT

Additional Tutorial Examples

In addition to tools, Chaco offers the possibility to create custom overlays:

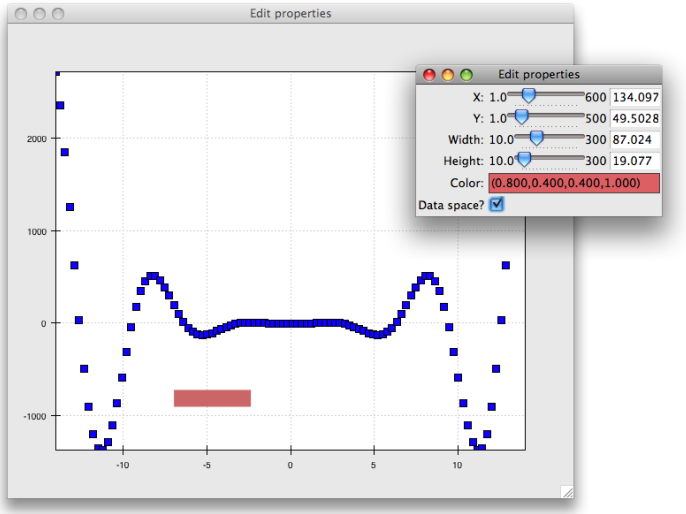


custom_overlay.py

ENTHOUGHT

Additional Tutorial Examples

These examples demonstrate custom overlays using dataspace coordinates:



`custom_overlay_dataspace.py`,
`custom_overlay_movetool.py`

ENTHOUGHT

More information

- Source on GitHub:
  https://github.com/enthought/chaco
- Documentation:
  http://docs.enthought.com/chaco
- Annotated examples:
  http://docs.enthought.com/chaco/user_manual/annotated_examples.html
- Mailing list:
  enthought-dev@enthought.com