SciPyTutorial Documentation

Release 0.0.1

VG

January 31, 2011

CONTENTS

CHAPTER

ONE

BASICS

Contents:

1.1 A Dialog on Python for Scientific Computing

Why should I learn Python?

```
If you consider the possibility that you will work in a different domain than
pure mathematics, then there is a good chance that you will need it at a given
moment of your career.
```

It is used in the industry?

```
Yes, it is used in many companies and by research institutions that use
computers as an instrument.
```

Why? I thought there is Java and C++ and so on...

```
With Python you can develop and prototype your products much more quickly. If
needed, parts of your production code can later be migrated to
C/C++/Fortran/Java or whatever.
```

Whatever?

```
Yes. One of the qualities that qualified Python to large industrial
application is its affinity to many programming languages. This makes it easy
to embed applications in both directions.
```

Aha, so you glue together tools.

```
Yes, Python is a perfect scripting language, but here I meant that you can
really mix programming languages in an easy and consistent way.
```

What is scripting?

```
Scripting is a way to let very different applications play together to
achieve your aim. This is another quality that makes Python attractive for the
industry.
```

What about research and scientific work?

```
As a researcher, you would like to know quite soon if your idea makes
sense. You would like to run simulations as soon as possible, so you want
quickly a correct prototype and not a prototype that runs quickly...
```

But that was one of the attractions of matlab, so what's new?

```
If the idea works and the code gives you correct results, but you want it more
efficiently or you want to scope it to other codes or professional visualization
tools, then it is much simpler to migrate parts of python code than to re-write
the code from scratch, as usually happens with matlab projects.
```

I see. So with python I can glue applications together and can migrate critical numerical parts to C/Fortran. But if I do not need to? Isn't matlab enough?

```
Enough? This is a good word!
It depends. Earlier I used to say: if you are satisfied with matlab and you do
not feel you need something more, then stay with matlab.
```

And now?

```
Now I would be more careful. I noticed that people prefer to use what they
learned first than to learn something new. This means that when people get a
difficult new problem, they make great efforts to solve it with the tools they
know better. This gives complicated solutions after great efforts, where
simple rush solutions were possible with new knowledge. You have to hit the
wall, and the hit must really hurt, in order to be really willing to learn
something new. I am sure that people are more efficient and more likely to
obtain good and rush solutions to hard problems if they learn python (and
scientific tools) first.
```

You mean that you can solve more difficult problems with python than with matlab?

```
I mean that with python et al you have better chances, especially for large projects.
```

Why should we learn or use python for teaching numerical methods? Isn't it much too complex? Isn't it simpler to use matlab for this aim?

```
When you start from zero, you need the same effort to learn matlab and Python.

matlab is a very good software. It has also its price, which is very high,
if you leave the university.  Most of our students will not remain in a
university, so working in a small company would be a clear handicap for
our alumni.

matlab is a good software because it was developed over several years under
very strict guidelines. Hence, it is more stable than the on-going development
versions of python et al.

For the very same reasons that make matlab very stable, matlab is very rigid.
Having to cope with the main ideas of 1970-80 makes improvements very hard
to realize. Object-orientation is the best example in this sense. It had to
be introduced in order to make matlab survive, but apart from its marketing
effect, it is useless, since it rather hampers than facilitates development
and code-reuse. Respecting old guidelines makes matlab strong, but rigid and
rather past-oriented. On the contrary, our students will live in the future,
let us give them a chance to make a better world!
```

1.2 Introduction

The aim of these pages is to provide enough information in order to help students to start using python's tools for computational science, numerical methods and implementation of numerical algorithms.

Python is not a Matlab clone, but a very flexible and very powerful (be aware!) high-level programming language. Still, it is so simple to use, that we'll learn it by doing our own job... We focus on scientific computing now, but you might use it for very different tasks, from controlling your experimental machines, by making high-performance visualizations, to sorting your mp3's on your mobile phone.

While coming with 'batteries included' python is only a small kernel that should be completed by modules specialized for your aims.

We'll use the following python modules:

- numpy
- scipy
- matplotlib
- ipython
- mayavi2

Please, consult *Installation tips* before retrieving these modules.

Work flow Personally, I use my preferred editor (emacs) and the improved interactive environment IPython for the design and test of the scripts; the actual tutorial assumes this working style. You might consider also one of the following alternatives to (editor+ipython):

- spyder (similar to the Matlab-environment)
- eric (similar to the Eclipse-environment)

The editor scite comes with the pythonxy package for Windows and is well suited for writing python files.

1.3 Installation tips

1.3.1 All at once if you are allowed

The main difficulty in the bottom-up installation is to discover the correct dependences of the modules. You can avoid this difficulty by using packages:

- Linux package manager: look for enthought distribution (e.g. present on Mandriva Free 2010.0); if this is not present then look for mayavi2 and scipy, then for ipython, matplotlib and nose (for having testing enabled)
- On Windows 32 bits you can use pyhtonxy that includes apart the main scientific modules and spyder and scite, too.
- The Enthought Python Distribution (EPD) is available for several operation systems (including Windows and MacOS) free of charge for academic use (it requires registration).

In case of difficulties or special wishes, consult the web pages of scipy.

A basic installation (without spyder) is available on the computers for the students slab.

This tutorial was tested on fedora core 13 machines of D-MATH and slab which have Python 2.6.4. as well as on a WinXP-machine with Python(x,y)-2.6.5.3.

1.3.2 Installation of the Spyder IDE as non-root on linux

A short description on how to install the Spyder IDE as non-root on Linux, useful for instance on one of the computers slab*.ethz.ch.

The Spyder IDE is available for download at http://code.google.com/p/spyderlib/.

Prerequisites

The program needs the following dependencies which are grouped into required and extension packages.

- Dependencies (required):
  - Python 2.x (x>=5)
  - PyQt4 4.x (x>=3 ; recommended x>=4)
  - QScintilla2 2.x (x>=1) (PyQt4 extension)
- Dependencies (optional):
  - pylint (code analysis)
  - numpy (N-dimensional arrays)
  - scipy (signal/image processing)
  - matplotlib (2D plotting)

Of course the last three optional dependencies are important and not really optional for us. However, these packages are already installed on these computers.

Installation process

Now let's go through the installation process step by step. We will install the program to the destination:

```
~/opt/spyder/
```

Adapt the shell variable *pythonpath* by executing the following bash command:

```
export PYTHONPATH="~/opt/spyder/lib/python:$PYTHONPATH"
```

(Note that if we don't set the python path *before* running the installer, it will refuse to work.) Now it's time to fetch the source code (you should check for a more recent versions than 2.0 beta 3):

```
wget http://code.google.com/p/spyderlib/downloads/detail?name=spyder-2.0.0beta3.tar.gz
```

unpack the source and go to the source directory:

```
tar -xzvf spyder-2.0.0beta3.tar.gz
cd spyder-2.0.0beta3
```

and finally run the installer script:

```
python setup.py install --home=~/opt/spyder
```

Usage

You can now run the Spyder program with this command:

```
~/opt/spyder/bin/spyder
```

1.3.3 Installation of the Spyder IDE on Debian GNU/Linux

A short description on how to install the Spyder IDE on the Debian GNU/Linux operating system.

The spyder IDE

The Spyder python IDE is contained in the official package repositories of the Debian GNU/Linux distribution. Thus installing the software is rather trivial. You just have to tell the package manager to install the package and he will automatically pull and install all required dependent packages too. (This includes other python packages but also more low level stuff like the BLAS routines if necessary.)

The shell command doing all the magic is as simple as:

```
apt-get install spyder
```

(You can use any other package manager frontend like `aptitude` or `synaptic` instead of `apt-get`.) This will work on Ubuntu Linux too, the most recent version `Maverick Meerkat` ships with the spyder package. (But maybe it's not installed by default.)

Other python packages on Debian

In the following we list some more Debian packages related to python and the tools we are going to use in this tutorial. Many of these will get installed as dependencies of the spyder package already, others are optional. You may want to try some of them nevertheless.

- `ipython`: enhanced interactive Python shell
- `mayavi2`: A scientific visualization package for 2-D and 3-D data
- `winpdb`: Platform independent Python debugger
- `spe`: Stani's Python Editor (Another python IDE)
- `pylint`: python code static checker and UML diagram generator
- `pyflakes`: passive checker of Python programs
- `python-numpy`: Numerical Python adds a fast array facility to the Python language
- `python-numpy-doc`: NumPy documentation
- `python-scipy`: scientific tools for Python
- `python-symeig`: Symmetrical eigenvalue routines for NumPy
- `python-matplotlib`: Python based plotting system in a style similar to Matlab
- `python-sympy`: Computer Algebra System (CAS) in Python
- `python-imaging`: Python Imaging Library
- `python-mpmath`: library for arbitrary-precision floating-point arithmetic
- `python-enthoughtbase`: Core packages for the Enthought Tool Suite
- `python-epydoc`: tool for documenting Python modules
- `python-sphinx`: tool for producing documentation for Python projects

1.4 First steps in Python

1.4.1 Fast-track Python

Start an IPython session

```
[gradinar@localhost Learn]$ ipython
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
In [1]:
```

In Python everything is an object and the types are dynamic

```
In [1]: a = 1.5 # set a to a floating point number

In [2]: type(a)
Out[2]: <type 'float'>

In [3]: a = 1 # redefine a as an integer

In [4]: type(a)
Out[4]: <type 'int'>

In [5]: a = 1e-10 # redefine a as a float with scientific notation

In [6]: type(a)
Out[6]: <type 'float'>

In [7]:  a = 1+5j # redefine a as complex

In [8]: type(a)
Out[8]: <type 'complex'>

In [9]: print 'a=', a
('a=', (1+5j))

In [10]: print a
(1+5j)

In [11]: a
Out[11]: (1+5j)

In [12]: print 2**6
64

In [13]: a**8
Out[13]: (-3824-456960j)

In [14]: 2**6
Out[14]: 64
```

str, unicode - string types:

```
In [15]: s1 = 'hello'

In [16]: s2 = u'hi'  # prepend u, gives unicode string

In [17]: s1[0]
Out[17]: 'h'
```

```
In [18]: s1[0], s1[1]
Out[18]: ('h', 'e')

In [19]: type(s1)
Out[19]: <type 'str'>

In [20]: type(s2)
Out[20]: <type 'unicode'>
```

list - mutable sequence:

```
In [21]: ell = [1,2,'three'] # make list

In [22]: type ell[2]
Out[22]: <type 'str'>

In [23]: ell[2] = 3

In [24]: ell.append(4)

In [25]: ell
Out[25]: [1, 2, 3, 4]
```

tuple - immutable sequence:

```
In [26]: t = (1,2,'four')

In [27]: t[2]
Out[27]: 'four'

In [28]: t[2] = 4

TypeError                                 Traceback (most recent call last)

/home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

TypeError: 'tuple' object does not support item assignment
```

python - loop:

```
In [29]: for element in xrange(1,21,2):
....::     print element
....::
....::
1
3
5
7
9
11
13
15
17
19

In [30]: for element in xrange(1,21,2):
....::   print element,
    ------------------------------------------------------------
    IndentationError: expected an indented block (<ipython console>, line 2)
```

hence indentation is the delimiter for blocks (Here IPython takes care for you. Pressing Enter is enough.):

```
In [31]: for element in xrange(1,21,2):
   ....:     print element,
   ....:
   ....:
   1 3 5 7 9 11 13 15 17 19
```

pythonic way of adding first n numbers

```
def sf(n):
    k = 0; result = 0
    while True:
        print 'current number to add =', k
        result += k
        k += 1
        if k > n: break

    return result
```

In the above example note that the indentation delimits the blocks and that no semi-colons are needed to separate commands written on different lines.

Python files have the extention .py.

Copy the above definition of the function in a file, let us call it sumy.py; then start ipython in the same directory, and type:

```
from sumy import sf
sf(5)
```

Generators:

```
def squares(lastterm):
    for n in range(lastterm):
        yield n**2

for i in squares(4): print i,
```

List comprehensions:

```
[sf(k) for k in squares(5)]
```

is then very slow, because our way of adding first n numbers is very inefficient; look into *Basic Tutorial* to see how to do it a bit better. For now, try a simpler function:

```
[k**2 for k in squares(5)]
```

Anonymous function (lambda function) is a way to inline code:

```
In [32]: lambda x,y : x + y - 2
Out[32]: <function <lambda> at 0x7f9d9741d9b0>

In [33]: f = lambda x,y : x + y - 2

In [34]: f(1,1)
Out[34]: 0
```

Warning: Arguments are always passed by assignement.
Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the C language's argument-passing model in practice:

- Immutable arguments are effectively passed "by value." Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in-place anyhow, the effect is much like making a copy.
- Mutable arguments are effectively passed "by pointer." Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers – mutable objects can be changed in-place in the function, much like C arrays. Of course, if you've never used C, Python's argument-passing mode will seem simpler still – it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.

```
In [2]: def f(a): # a is assigned to (references) passed object
...:       a = 99    # changes local variable a only: here simply resets a to a completely different
...:
...:

In [3]: b = 88

In [4]: f(b)     # a and b both reference same 88 initially

In [5]: print(b) # b not changed
88
```

Assignment to an argument name inside a function (e.g., a=99) does not magically change a variable like b in the scope of the function call. Argument names may share passed objects initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends.

That is the case for assignment to argument names themselves. When arguments are passed mutable objects like lists and dictionaries, we also need to be aware that inplace changes to such objects may live on after a function exits, and hence impact callers. Here's an example that demonstrates this behavior:

```
In [7]: def changer(a,b):  # Arguments assigned references to objects
...:         a = 2         # Changes local name's value only
...:         b[0] = 'spam' # Changes shared object in-place
...:

In [8]: X = 1

In [9]: L = [1, 2]          # Caller

In [10]: changer(X,L)       # Pass immutable and mutable objects

In [11]: X, L               # X is unchanged, L is different!
Out[11]: (1, ['spam', 2])
```

If we don't want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects. For function arguments, we can always copy the list at the point of call:

```
L = [1, 2]
changer(X, L[:]) # Pass a copy, so our 'L' does not change
```

We can also copy within the function itself, if we never want to change passed-in objects, regardless of how the function is called:

```
def changer(a, b):
     b = b[:] # Copy input list so we don't impact caller
     a = 2
     b[0] = 'spam' # Changes our list copy only
```

Both of these copying schemes don't stop the function from changing the object – they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to immutable objects to force the issue. Tuples, for example, throw an exception when changes are attempted:

```
L = [1, 2]
changer(X, tuple(L)) # Pass a tuple, so changes are errors
```

Elements of Python **style** are on Google Python Style Guide or PythonStyle.

Finally, the Zen of Python (hat you might find useful outside of Pyhton, too):

```
In [35]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.4.2 More on the language

You can find more on Python in on-line tutorials and books. I enjoyed *Learning Python* by M. Lutz and *Python in a Nutshell* by A. Martelli.

For your convenience, you might consult Raoul Bourquin's syntheses of the python tutorial at python.org:

An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
SPAM = 1                 # and this is the second comment
                         # ... and now a third!
STRING = "# This is not a comment."
```

Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

Numbers The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign (`'='`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Variables must be "defined" (assigned a value) before they can be used, or an error will occur:

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of `j` or `J`. Complex numbers with a nonzero real component are written as `(real+imagj)`, or can be created with the `complex(real, imag)` function.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number *z*, use `z.real` and `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (`float()`, `int()` and `long()`) don't work for complex numbers — there is no one correct way to convert a complex number to a real number. Use `abs(z)` to get its magnitude (as a float) or `z.real` to get its real part.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

```
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

Strings Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print hello
```

Note that newlines still need to be embedded in the string using `\n` – the newline following the trailing backslash is discarded. This example would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Or, strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

produces the following output:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

If we make the string literal a "raw" string, `\n` sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."
```

```
print hello
```

would print:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

The interpreter prints the result of string operations in the same way as they are typed for input: inside quotes, and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. (The `print` statement, described later, can be used to write strings without quotes or escapes.)

Strings can be concatenated (glued together) with the + operator, and repeated with `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Two string literals next to each other are automatically concatenated; the first line above could also have been written `word = 'Help' 'A'`; this only works with two literals, not with arbitrary string expressions:

```
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'   #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'     #  <-  This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the *slice notation*: two indices separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # Everything except the first two characters
'lpA'
```

Unlike a C string, Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

However, creating a new string with the combined content is easy and efficient:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Here's a useful invariant of slice operations: `s[:i] + s[i:]` equals `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Degenerate slice indices are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indices may be negative numbers, to start counting from the right. For example:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

Out-of-range negative slice indices are truncated, but don't try this for single-element (non-slice) indices:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Lists Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list *a*:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are *immutable*, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
```

```
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function `len()` also applies to lists:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Note that in the last example, `p[1]` and `q` really refer to the same object! We'll come back to *object semantics* later.

First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

  ```
  >>> i = 256*256
  >>> print 'The value of i is', i
  The value of i is 65536
  ```

  A trailing comma avoids the newline after the output:

  ```
  >>> a, b = 0, 1
  >>> while b < 1000:
  ...     print b,
  ...     a, b = b, a+b
  ...
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
  ```

  Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**`(`*x*`)`
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**`(`*L*`)`
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.`**`insert`**`(`*i*, *x*`)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**`(`*x*`)`
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`sort`**`()`
Sort the items of the list, in place.

`list.`**`reverse`**`()`
Reverse the elements of the list, in place.

An example that uses most of the list methods:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Functional Programming Tools There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If *sequence* is a `string` or `tuple`, the result will be of the same type; otherwise, it is always a `list`. For example, to compute some primes:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this. New in version 2.3.

List Comprehensions List comprehensions provide a concise way to create lists without resorting to use of `map()`, `filter()` and/or `lambda`. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

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

List comprehensions are much more flexible than `map()` and can be applied to complex expressions and nested functions:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see *typesseq*). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

Dictionaries

Another useful data type built into Python is the *dictionary* (see *typesmapping*). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sort()` method to the list of keys). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

The `dict()` constructor builds dictionaries directly from lists of key-value pairs stored as tuples. When the pairs form a pattern, list comprehensions can compactly specify the key-value list.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Later in the tutorial, we will learn about Generator Expressions which are even better suited for the task of supplying key-values pairs to the `dict()` constructor.

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

`if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword '`elif`' is short for 'else if', and is useful to avoid excessive indentation. An `if` ... `elif` ... `elif` ... sequence is a substitute for the `switch` or `case` statements found in other languages.

`for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists). If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

```
>>> for x in a[:]: # make a slice copy of the entire list
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see *Looping Techniques*.

break and continue Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the smallest enclosing for or while loop.

The continue statement, also borrowed from C, continues with the next iteration of the loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section *tut-docstrings*.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

```
>>> fib(0)
>>> print fib(0)
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
```

[1] Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

```
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see *Classes*) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

Default Argument Values The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword Arguments Functions can also be called using keyword arguments of the form `keyword = value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

could be called in any of the following ways:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

but the following calls would all be invalid:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument following keyword
parrot(110, voltage=220)     # duplicate value for argument
parrot(actor='John Cleese')  # unknown keyword
```

In general, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It's not important whether a formal parameter has a default value or not. No argument may receive a value more than once — formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
...
```

```
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see *typesmapping*) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print "-" * 40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ":", keywords[kw]
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the `sort()` method of the list of keyword argument names is called before printing the contents of the `keywords` dictionary; if this is not done, the order in which the arguments are printed is undefined.

Lambda Forms By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Classes

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to "break into the definition." The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of data.

In C++ terminology, all class members (including the data members) are *public*, and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

Class Definition Syntax The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

Class Objects Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Instance Objects Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.) Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

Method Objects Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print xf()
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

Python Packages

Required, optional or otherwise noteworthy python packages are presented in the following short table. We do not pretend that this table is in any way exhaustive. For a much more comprehensive list take a look at the Python package index at: http://pypi.python.org/pypi.

- numpy

NumPy is the fundamental package needed for scientific computing with Python. It contains among other things a powerful N-dimensional array object

  – URL: http://www.numpy.org/

- scipy

  The SciPy library is built to work with NumPy arrays, and provides many user-friendly and efficient numerical routines such as routines for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

  – URL: http://www.scipy.org/

- matplotlib

  Matplotlib is a python 2D plotting library for the Python programming language which produces publication quality figures in a variety of formats. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code.

  – URL: http://matplotlib.sourceforge.net/

- sympy

  SymPy is a Python library for (limited) symbolic computations inside python programms.

  – URL: http://sympy.org/

- PIL

  Python Imaging Library is a library for the Python programming language that provides support for opening, manipulating, and saving many different image file formats.

  – URL: http://www.pythonware.com/products/pil/

- mpmath

  Mpmath is a Python library for multiprecision floating-point arithmetic. It provides an extensive set of transcendental functions, unlimited exponent sizes, complex numbers, interval arithmetic, numerical integration and differentiation, root-finding, linear algebra, and much more

  – URL: http://code.google.com/p/mpmath/

- MeshPy

  MeshPy offers quality triangular and tetrahedral mesh generation for Python. Meshes of this type are chiefly used in finite-element simulation codes, but also have many other applications ranging from computer graphics to robotics.

  – URL: http://mathema.tician.de/software/meshpy

- PyX

  PyX is a Python package for the creation of PostScript and PDF files. It combines an abstraction of the PostScript drawing model with a TeX/LaTeX interface. This package can be used for two dimensional (geometric) drawings.

  – URL: http://pyx.sourceforge.net/

- h5py

  The HDF5 library is a versatile, mature library designed for the storage of numerical data. The h5py package provides a simple, Pythonic interface to HDF5. A straightforward high-level interface allows the manipulation of HDF5 files, groups and datasets using established Python and NumPy metaphors.

  – URL: http://h5py.alfven.org/

1.5 Basic Tutorial

1.5.1 NumPy

- N-dimensional homogeneous arrays
- **Universal functions (ufunc)**
  - built-in linear algebra, FFT, random number generators, etc...
- Tools for integration with C/C++/Fortran
- **Heavy lifting done by optimized C/Fortran libraries**
  - ATLAS, UMFPACK, FFTW, etc...

Let us initialize an array with first n natural numbers:

```
import numpy as np # import the module and use a shorter namespace
x = np.arange(4)
print x
```

and try the following commands:

```
print np.sum(x)
print x**2
f = lambda x: np.sum(x)
print f(x**2)
print [f(t) for t in x**2]
print [f(np.arange(t)) for t in x**2]
```

Need help?

```
help np.sum
help np.arange
help np
```

Under ipython you could also try:

```
In [1]: import numpy as np

In [2]: np.linalg.<TAB>
```

where <TAB> refers to the TAB-key. You should see:

```
In [2]: np.linalg.
np.linalg.LinAlgError        np.linalg.__path__           np.linalg.eigvalsh
np.linalg.Tester             np.linalg.__reduce__         np.linalg.info
np.linalg.__builtins__       np.linalg.__reduce_ex__      np.linalg.inv
np.linalg.__class__          np.linalg.__repr__           np.linalg.lapack_lite
np.linalg.__delattr__        np.linalg.__setattr__        np.linalg.linalg
np.linalg.__dict__           np.linalg.__sizeof__         np.linalg.lstsq
np.linalg.__doc__            np.linalg.__str__            np.linalg.matrix_power
np.linalg.__file__           np.linalg.__subclasshook__   np.linalg.norm
np.linalg.__format__         np.linalg.bench              np.linalg.pinv
np.linalg.__getattribute__   np.linalg.cholesky           np.linalg.qr
np.linalg.__hash__           np.linalg.cond               np.linalg.solve
np.linalg.__init__           np.linalg.det                np.linalg.svd
np.linalg.__name__           np.linalg.eig                np.linalg.tensorinv
np.linalg.__new__            np.linalg.eigh               np.linalg.tensorsolve
np.linalg.__package__        np.linalg.eigvals            np.linalg.test
```

IPython examined the `np.linalg` module, and returned all the possible completions. Once you see an interesting function, you'd like to know how to use it:

```
In[3]: np.transpose?<ENTER>
```

In order to view the actual source code, use two question marks instead of one.

Slicing Arrays:

```
In [1]: import numpy as np

In [2]: a = np.array([[1,2,3,4],[9,8,7,6],[1,6,5,4]])

In [3]: a
Out[3]:
array([[1, 2, 3, 4],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

In [4]: arow = a[0,:] # get slice referencing row zero

In [5]: arow
Out[5]: array([1, 2, 3, 4])

In [6]: cols = a[:,[0,2]] # get slice referencing columns 0 and 2

In [7]: cols
Out[7]:
array([[1, 3],
       [9, 7],
       [1, 5]])
```

NOTE: slices (as `arow` & `cols`) are **NOT copies**, they point to the original data:

```
In [8]: arow[:] = 0
In [9]: arow
Out[9]: array([0, 0, 0, 0])

In [10]: a   # arow is a reference to elements of a
Out[10]:
array([[0, 0, 0, 0],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

In [11]: copyrow = arow.copy()    # copying need to be invoked explicitly
In [11]: copyrow = a[0,:].copy()  # same result without using arow
```

`NumPy` gives the possibility to apply operations to many elements with a single call (broadcasting):

```
In [14]: a = np.arange(10)

In [15]: a
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [16]: a.reshape((2,5))
Out[16]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

In [18]: b = a.reshape((2,5))
In [19]: b+1
Out[19]:
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])

In [25]: b + np.array(([1],[10])) # add 1 to 1st row, 10 to 2nd row
Out[25]:
array([[ 1,  2,  3,  4,  5],
       [15, 16, 17, 18, 19]])

In [26]: b +  ([1],[10]) # same as above
Out[26]:
array([[ 1,  2,  3,  4,  5],
   [15, 16, 17, 18, 19]])

In [27]: b +  [[1],[10]] # same as above
Out[27]:
array([[ 1,  2,  3,  4,  5],
       [15, 16, 17, 18, 19]])

In [28]: a**([2],[3])

Out[28]:
array([[  0,   1,   4,   9,  16,  25,  36,  49,  64,  81],
       [  0,   1,   8,  27,  64, 125, 216, 343, 512, 729]])

In [29]: b**([2],[3]) # raise 1st row to power 2, 2nd to 3
Out[29]:
array([[  0,   1,   4,   9,  16],
       [125, 216, 343, 512, 729]])
```

NOTE: be careful with types (especially by inplace operations as +=, *=, etc.):

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3, 4])

In [3]: a[0] = a[0]+1

In [4]: a
Out[4]: array([2, 2, 3, 4])

In [5]: a[0] = a[0]+1.1

In [6]: a                      # the result is 3 instead of 3.1
Out[6]: array([3, 2, 3, 4])

In [7]: a += 1 # add 1 to variable a (inplace/register level)

In [8]: a
Out[8]: array([4, 3, 4, 5])

In [9]: a += 1.1            # again the .1 is truncated

In [10]: a
Out[10]: array([5, 4, 5, 6])

In [11]: type(a[0])         # the entries of a are integers
```

```
Out[11]: <type 'numpy.int32'>

In [12]: a = a + 1.1         # this redefines also the type of a

In [13]: a
Out[13]: array([ 6.1,  5.1,  6.1,  7.1])

In [14]: type(a[0])          # the entries of a are now floats
Out[14]: <type 'numpy.float64'>

In [15]: f = np.array([1.,2.,3.,4.])   # this creates an array of floats

In [16]: type(f[0])
Out[16]: <type 'numpy.float64'>

In [17]: f = np.zeros(4)               # this creates an array of floats filled with zeros

In [18]: f
Out[18]: array([ 0.,  0.,  0.,  0.])

In [19]: type(f[0])
Out[19]: <type 'numpy.float64'>
```

Experiment this:

```
In [1]: import math

In [2]: import numpy as np

In [3]: x = np.linspace(0.,np.pi,100) # 100 evenly spaced points in [0.,pi]

In [4]: math.sin(x)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/home/gradinar/Documents/Bibl/Programming/<ipython console> in <module>()

TypeError: only length-1 arrays can be converted to Python scalars

In [5]: y = np.sin(x)

In [6]: import pylab as plb

In [7]: plb.plot(x,y)
/usr/lib64/python2.6/site-packages/matplotlib/backends/backend_gtk.py:621: DeprecationWarning: Use th
self.tooltips = gtk.Tooltips()
Out[7]: [<matplotlib.lines.Line2D object at 0x7f159cf90890>]

In [8]: plb.show()
```

NOTE: mathematical functions from module math are very fast, but cannot be applied to vectors; for those, we use the module `numpy`.

NOTE: if you need complex numbers, you have to allocate space for them!:

```
In [9]: x += 1j # remember, x was created with np.linspace(0.,np.pi,100) at In[3]

In [10]: x[0]
Out[10]: 0.0
```

```
In [11]: x[1]
Out[11]: 0.031733259127169629

In [12]: 1j
Out[12]: 1j

In [13]: type(1j)
Out[13]: <type 'complex'>

In [14]: type(x[0])
Out[14]: <type 'numpy.float64'>
```

for instance via multiplication trick:

```
In [15]: x = x+1j

In [16]: x
Out[16]:
array([ 0.00000000+1.j,  0.03173326+1.j,  0.06346652+1.j,  0.09519978+1.j,
        0.12693304+1.j,  0.15866630+1.j,  0.19039955+1.j,  0.22213281+1.j,
        0.25386607+1.j,  0.28559933+1.j,  0.31733259+1.j,  0.34906585+1.j,
        0.38079911+1.j,  0.41253237+1.j,  0.44426563+1.j,  0.47599889+1.j,
        0.50773215+1.j,  0.53946541+1.j,  0.57119866+1.j,  0.60293192+1.j,
        0.63466518+1.j,  0.66639844+1.j,  0.69813170+1.j,  0.72986496+1.j,
        0.76159822+1.j,  0.79333148+1.j,  0.82506474+1.j,  0.85679800+1.j,
        0.88853126+1.j,  0.92026451+1.j,  0.95199777+1.j,  0.98373103+1.j,
        1.01546429+1.j,  1.04719755+1.j,  1.07893081+1.j,  1.11066407+1.j,
        1.14239733+1.j,  1.17413059+1.j,  1.20586385+1.j,  1.23759711+1.j,
        1.26933037+1.j,  1.30106362+1.j,  1.33279688+1.j,  1.36453014+1.j,
        1.39626340+1.j,  1.42799666+1.j,  1.45972992+1.j,  1.49146318+1.j,
        1.52319644+1.j,  1.55492970+1.j,  1.58666296+1.j,  1.61839622+1.j,
        1.65012947+1.j,  1.68186273+1.j,  1.71359599+1.j,  1.74532925+1.j,
        1.77706251+1.j,  1.80879577+1.j,  1.84052903+1.j,  1.87226229+1.j,
        1.90399555+1.j,  1.93572881+1.j,  1.96746207+1.j,  1.99919533+1.j,
        2.03092858+1.j,  2.06266184+1.j,  2.09439510+1.j,  2.12612836+1.j,
        2.15786162+1.j,  2.18959488+1.j,  2.22132814+1.j,  2.25306140+1.j,
        2.28479466+1.j,  2.31652792+1.j,  2.34826118+1.j,  2.37999443+1.j,
        2.41172769+1.j,  2.44346095+1.j,  2.47519421+1.j,  2.50692747+1.j,
        2.53866073+1.j,  2.57039399+1.j,  2.60212725+1.j,  2.63386051+1.j,
        2.66559377+1.j,  2.69732703+1.j,  2.72906028+1.j,  2.76079354+1.j,
        2.79252680+1.j,  2.82426006+1.j,  2.85599332+1.j,  2.88772658+1.j,
        2.91945984+1.j,  2.95119310+1.j,  2.98292636+1.j,  3.01465962+1.j,
        3.04639288+1.j,  3.07812614+1.j,  3.10985939+1.j,  3.14159265+1.j])

In [17]: type(x[0])
Out[17]: <type 'numpy.complex128'>
```

or via constructor giving the data type:

```
In [17]: z = np.zeros(100, dtype=complex)
```

Run this example in order to be aware about possible errors in using python's pure range command. Avoid `np.arange`, that is numerically unstable:

```
from numpy import arange, linspace#, arrayrange
#from Numeric import arange#, arrayrange

nerrors = 0
for n in range(1,101):
    x1 = arange(0, 1, 1./n)[-1] # should be less than 1
    print n, x1
    if abs(x1 - 1.0) < 1e-6: nerrors += 1

print 'arange leading to ', nerrors, ' unexpected cases'

nerrors = 0
for n in range(1,101):
    x1 = linspace(0, 1, num=n, endpoint=False )[-1] # should be less than 1
    #print n, x1
    if abs(x1 - 1.0) < 1e-6: nerrors += 1

print 'linspace leading to ', nerrors, ' unexpected cases'
```

Use `np.linspace` to generate equidistant sequences or `np.r_`, for example:

```
In [1]: import numpy as np

In [2]: np.r_[:9:10j]
Out[2]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [3]: x = np.r_[:9:10j]

In [4]: x[0]
Out[4]: 0.0

In [5]: x[9]
Out[5]: 9.0

In [6]: x[10]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

IndexError: index out of bounds

In [4]: np.r_[0:9:3j]
Out[4]: array([ 0. ,  4.5,  9. ])

In [5]: np.r_[0:9:3]
Out[5]: array([0, 3, 6])
```

More in indicies of an array:

```
In [3]: a = np.linspace(1,8,8)

In [4]: a
Out[4]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])

In [5]: a[[1,6,7]] = 10

In [6]: a
Out[6]: array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])

In [7]: a[range(2,8,3)] = -2

In [8]: a
Out[8]: array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
```

```
In [9]: a[a < 0] # pick out the negative elements of a
Out[9]: array([-2., -2.])

In [10]: a[a < 0] = a.max()

In [11]: a
Out[11]: array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
```

Other usefull array operations:

```
# a is an array
a.clip(min=3, max=12) # clip elements
a.mean(); mean(a)     # mean value
a.var(); var(a)       # variance
a.std(); std(a)       # standard deviation
median(a)
cov(x,y)              # covariance
trapz(a)              # Trapezoidal integration
diff(a)               # finite differences (da/dx)
```

and more Matlab-like functions:: corrcoeff, cumprod, diag, eig, eye, fliplr, flipud, max, min, prod, ptp, rot90, squeeze, sum, svd, tri, tril, triu

Solving a linear system:

```
In [1]: import numpy as np

In [2]: A = np.array([
   ...: [3,2,-1],
   ...: [2,-1,4],
   ...: [-1,0.5,-1]
   ...: ])

In [3]: b = np.array([1,-2,0])

In [4]: x = np.solve(A,b)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

/home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

AttributeError: 'module' object has no attribute 'solve'

In [5]: x = np.linalg.solve(A,b)

In [6]: x
Out[6]: array([ 0.57142857, -0.85714286, -1.        ])

In [7]: dot(A,x)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

/home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

NameError: name 'dot' is not defined

In [8]: np.dot(A,x)
Out[8]: array([  1.00000000e+00,  -2.00000000e+00,  -1.11022302e-16])
```

```
In [9]: Ax = np.dot(A,x)

In [10]: np.allclose(Ax,b)
Out[10]: True
```

Vectorization:

```
In [1]: import numpy as np

In [2]: def myfunc(x):
...:     if x >= 0: return x**2
...:     else: return -x
...:
...:

In [3]: myfunc(2)
Out[3]: 4

In [5]: myfunc(np.array([-2,2]))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/home/gradinar/LaTeX_doc/Numcourses/Numcourses/SciPyTools/Start/<ipython console> in <module>()

/home/gradinar/LaTeX_doc/Numcourses/Numcourses/SciPyTools/Start/<ipython console> in myfunc(x)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.al

In [6]: vecfunc = np.vectorize(myfunc, otypes=[float]) # it is important to declare the return type

In [8]: vecfunc(np.array([-2,2]))
Out[8]: array([ 2.,  4.])

In [9]: def myvecfunc(x):
...:     return np.where(x<0, -x, x**2)
...:

In [10]: myvecfunc(np.array([-2,2]))
Out[10]: array([2, 4])
```

NOTE that the hand written vectorization (last version) will run quicker than the automatically vectorized function.

- Simple Input/Output
- Professional I/O with HDF5

1.5.2 SciPy - Scientific tools for Python

SciPy is an Open Source library of scientific tools for Python. It depends on the NumPy library, and it gathers a variety of high level science and engineering modules together as a single package. SciPy provides modules for

- statistics
- optimization
- numerical integration
- linear algebra
- Fourier transforms

- signal processing
- image processing
- ODE solvers
- special functions

and more.

Root finding:

```
In [1]: coeffs = [1, -5, 6]

In [2]: import scipy as sp

In [3]: sp.roots(coeffs)
Out[3]: array([ 3.,  2.])
```

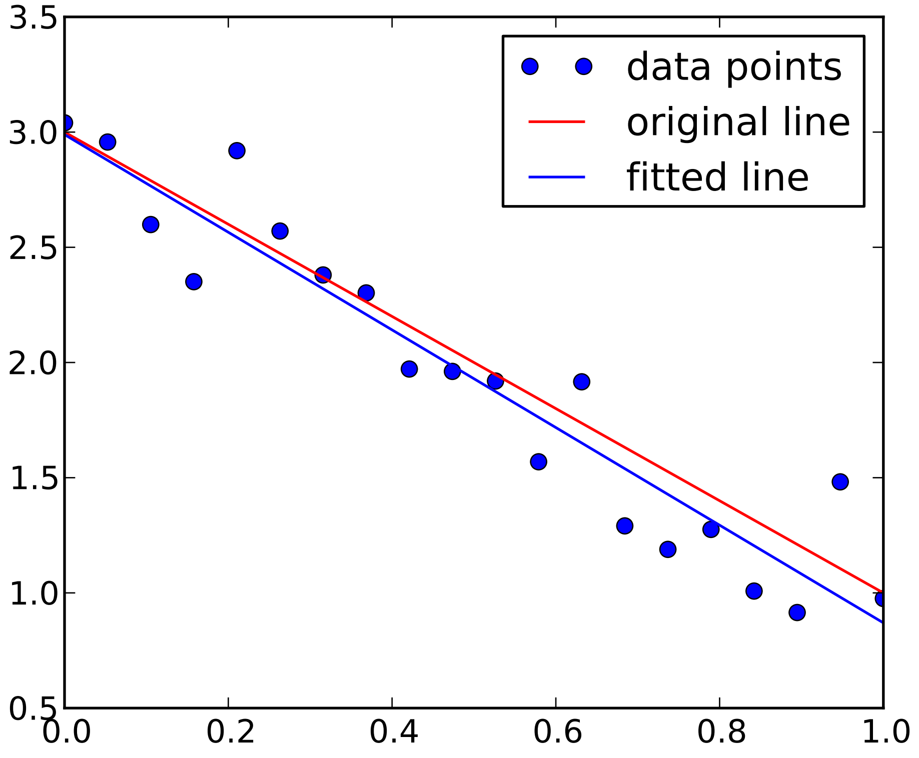
and now a more complicated function:

```
In [4]: from scipy.optimize import fsolve

In [5]: fsolve(sin(z)+cos(z)*cos(z),0)
        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)

        /home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

        NameError: name 'sin' is not defined

In [6]: fsolve(sp.sin(z)+sp.cos(z)*sp.cos(z),0)
        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)

        /home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

        NameError: name 'z' is not defined

In [7]: z = sp.linspace(-sp.i,sp.pi)
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        /home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

        TypeError: bad operand type for unary -: 'str'

In [8]: z = sp.linspace(-sp.pi,sp.pi)

In [9]: fsolve(sp.sin(z)+sp.cos(z)*sp.cos(z),0)
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        /home/gradinar/Programming/Python/Learn/<ipython console> in <module>()

        /usr/lib64/python2.6/site-packages/scipy/optimize/minpack.pyc in fsolve(func, x0, args, 
            111     n = len(x0)
            112     if type(args) != type(()): args = (args,)
        --> 113     check_func(func,x0,args,n,(n,))
            114     Dfun = fprime
            115     if Dfun is None:
```

```
            /usr/lib64/python2.6/site-packages/scipy/optimize/minpack.pyc in check_func(thefunc, x0,
                 10
                 11 def check_func(thefunc, x0, args, numinputs, output_shape=None):
            ---> 12     res = atleast_1d(thefunc(*((x0[:numinputs],)+args)))
                 13     if (output_shape is not None) and (shape(res) != output_shape):
                 14         if (output_shape[0] != 1):

            TypeError: 'numpy.ndarray' object is not callable

In [10]: g = lambda z: sp.sin(z)+sp.cos(z)*sp.cos(z)

In [11]: g(0)
Out[11]: 1.0

In [12]: fsolve(g,0)
Out[12]: -0.66623943249251527

In [13]: from matplotlib import pyplot as plt

In [14]: plt.plot(z,g(z))
Out[14]: [<matplotlib.lines.Line2D object at 0x7fd7ce10a9d0>]

In [15]: plt.show()
```

An example on least squares:

```
import scipy.linalg as splalg
import scipy as sp
from matplotlib import pyplot as plt

n = 20 # sample length
x = sp.linspace(0.,1.,n)
y = -2*x+3 # line
z = y + sp.random.normal(0.,0.25,n) # + noise
A = sp.array([x, sp.ones(n)])
A = A.transpose()
r = splalg.lstsq(A,z)
# result is a 4-tuple, the solution (a,b) is the 1st entry
a, b = r[0]
plt.plot(x,z,'o', x,y,'r', x, a*x+b, 'b')
plt.legend(('data points', 'original line', 'fitted line'))
plt.show()
```



1.5.3 Random number generator

For the interested, the pseudorandom number generator in NumPy implements the Mersenne Twister, which is generally considered a good generator for many applications, and is also used by the Python standard library's random module, as well as R, Maple, Ruby and recent versions of MATLAB.

1.5.4 What is in numpy that isn't in scipy and vice versa?

NumPy contains core numerical functionality. The main thing it provides is the ndarray object. It also provides things like a high quality pseudorandom number generator (an implementation of the Mersenne twister, with generators for several distributions), fast Fourier transforms, scalar types that correspond to underlying C data types and make their size explicit (e.g. float64, int32, complex128), a variety of ufuncs for things like trigonometric functions and transcendentals, linear algebra routines (that can use an optimized BLAS such as ATLAS or the Intel Math Kernel Library if you have them available), and lots of utility functions for working with ndarrays: creating them, merging them, splitting them, etc. It also provides f2py which is a tool for generating Python wrappers for Fortran code.

SciPy is a much broader package that is best thought of as a set of toolboxes, built on top of NumPy, and also wrapping suitably licensed packages (including many numerical libraries written in Fortran using f2py). Things like clustering, signal processing, even more linear algebra routines than NumPy provides, sparse matrix support, numerical optimization; all of this stuff falls under SciPy. You're best looking through the SciPy Reference Guide for more detailed guide to what is included.

1.5.5 Why do both numpy.linalg and scipy.linalg exist? What is the difference?

scipy.linalg is a more complete wrapping of Fortran LAPACK using f2py, but both NumPy and SciPy may be using the underlying LAPACK available on your platform if you have installed it along with the header files (-dev or -devel packages on most Linux distributions). One of the design goals of NumPy was to make it buildable without a Fortran compiler, and if you don't have LAPACK available NumPy will use its own implementation. SciPy requires a Fortran compiler to be built, and heavily depends on wrapped Fortran code.

You can check the underlying detected implementation for both of these with show_config:

```
import numpy as np
import scipy as sc
print np.show_config()
print sc.show_config()
```

The linalg modules in NumPy and SciPy have some common functions but with different help lines. scipy.linalg contains more, such as functions related to LU decomposition and the Schur decomposition, multiple ways of calculating the pseudoinverse, and matrix transcendentals like the matrix logarithm. Some functions that exist in both have augmented functionality in scipy.linalg; for example scipy.linalg.eig() can take a second matrix argument for solving generalized eigenvalue problems.

Using numpy.linalg (if you can) will likely make your code more portable, since NumPy is easier to install and necessary for using NumPy arrays in the first place. If numpy.linalg can't do what you're looking for, however, try scipy.linalg.

1.5.6 A LAPACK bug propagated into scipy

Many scipy programms are just simple wrappers around standard Fortran or C packages. As a consequence, bugs are propagated, as in the folowing example that needs the matrix file

```
import scipy.linalg
import scipy.io
L = scipy.io.loadmat('Lmat.mat')['L']
# let us compute the first (smallest) 5 eigenvectors
d,v= scipy.linalg.eigh(L,eigvals=(0,4))
# let us check if the eigenvectors are orthogonal
print scipy.multiply(v[:,0],v[:,1]).sum()
# answer is zero (to a floating point error). Good!
# now let us compute the first (smallest) 2 eigenvectors
d,v= scipy.linalg.eigh(L,eigvals=(0,1))
# let us check if the eigenvectors are orthogonal
print scipy.multiply(v[:,0],v[:,1]).sum()
# they are not orthogonal!! the second eigenvector was not calculated correctly!
```

The bug in LAPACK makes this use of eigh prohibite; independently of this, it is much more efficient to use the Arnoldi method as implemented by ARPACK:

```
import scipy.linalg
import scipy.io

nev = 6

L = scipy.io.loadmat('Lmat.mat')['L']
n = L.shape[0]

d,v= scipy.linalg.eigh(L,eigvals=(n-nev,n-1))
# let us check if the eigenvectors are orthogonal
```

```
print scipy.multiply(v[:,0],v[:,1]).sum()

from scipy.sparse.linalg.eigen.arpack import arpack
#from scipy.sparse.linalg.eigen.arpack import speigs
from scipy.sparse import csr_matrix

L = csr_matrix(L)
#L = scipy.matrix(L)
#L = L.tocsr()


#da,va = arpack.eigen(L, k=nev)
da,va = arpack.eigen_symmetric(L, k=nev)
#da,va = speigs.ARPACK_eigs(L.matvec,n,nev, which='LM')
va =  scipy.matrix(va)
print d
I = abs(da).argsort()
da.sort(); vac = va[:,I]
print da



print (vac.H*vac-scipy.eye(nev)).max()#scipy.multiply(v[:,0],v[:,1]).sum()
#print 'eigh Arpack EV:', v
v = scipy.matrix(v)
ve = v - vac
#print v.shape, va.shape
from scipy.linalg import norm
print ve.shape
z = scipy.array([norm(ve[:,k]) for k in xrange(nev)])
print z
#print z.max()

#from scipy.sparse.linalg.eigen.arpack.arpack import eigen
```

1.5.7 Partial evaluation of functions

Assume you have a function of multiple arguments and already know some of them but not all. And you want or need to bind the known parameters now. A common example is plotting of functions. Take the bessel functions, they require an argument `n` which is the order (an integer in the most simple case) additionally the argument `x`. What can we do now? Assume the plot command can not handle such functions. (In the example below we could just plug in the order in the plot command, but there are other plot commands that only take the function object expecting something like `f(x)`.)

An example:

```
# Import bessel_j
from scipy.special import jn

# Evaluate J_n with n=2 at x=1.0
jn(2, 1.0)
0.11490348493190049

# The partial evaluation tool
from functools import partial

j = partial(jn, [2])
```

```

# Evaluate J_2 at x=1.0
j(1.0)
0.11490348493190049

# Plot
x = linspace(-5, 5, 1000)
plot(x, f(x))
```

1.6 IPython tips

IPython is an interactive computing environment; not only it is an enhanced interactive Python shell, but it is also an architecture for interactive parallel computing...

In case you use the pythonxy distribution under Windows, it is a good idea to start the IPython with mlab-mode enabled; you find several modes in the menu-entry "enhanced console" of pythonxy.

Note that the magic commands discussed below didn't worked in the spyder 1.1.5 version coming with pythonxy 2.6.5.3 under Windows, but in all IPython shells of pythonxy.

We resume here the commands that we use frequently in our work.

```
In [1]: import numpy as np

In [2]: np.linalg.<TAB>
```

where <TAB> refers to the TAB-key. You should see:

```
In [2]: np.linalg.
np.linalg.LinAlgError        np.linalg.__path__          np.linalg.eigvalsh
np.linalg.Tester             np.linalg.__reduce__        np.linalg.info
np.linalg.__builtins__       np.linalg.__reduce_ex__     np.linalg.inv
np.linalg.__class__          np.linalg.__repr__          np.linalg.lapack_lite
np.linalg.__delattr__        np.linalg.__setattr__       np.linalg.linalg
np.linalg.__dict__           np.linalg.__sizeof__        np.linalg.lstsq
np.linalg.__doc__            np.linalg.__str__           np.linalg.matrix_power
np.linalg.__file__           np.linalg.__subclasshook__  np.linalg.norm
np.linalg.__format__         np.linalg.bench             np.linalg.pinv
np.linalg.__getattribute__   np.linalg.cholesky          np.linalg.qr
np.linalg.__hash__           np.linalg.cond              np.linalg.solve
np.linalg.__init__           np.linalg.det               np.linalg.svd
np.linalg.__name__           np.linalg.eig               np.linalg.tensorinv
np.linalg.__new__            np.linalg.eigh              np.linalg.tensorsolve
np.linalg.__package__        np.linalg.eigvals           np.linalg.test
```

IPython examined the np.linalg module, and returned all the possible completions. Once you see an interesting function, you'd like to know how to use it:

```
In[3]: np.transpose?<ENTER>
```

In order to view the actual source code, use two question marks instead of one.

The magic commands %pdoc, %pdef, %psource and %pfile will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found). If automagic is on (it is by default), you don't need to type the '%' explicitly:

```
np.linalg.svd?
pdoc np.linalg.svd
```

```
pdef np.linalg.svd
psource np.linalg.svd
```

Consider now the files sattelite.py that implements the simplified trajectory of a sattelite:

```python
import numpy as np
from matplotlib import pyplot as plt

op = 2*np.pi; os = 14.*np.pi
R = 4.; r = 0.25
t = np.linspace(0, 2, 2000)
xp = np.cos(op*t)
yp = np.sin(op*t)
xs = xp + r*np.cos(os*t)
ys = yp + r*np.sin(os*t)
plt.plot(xs,ys); plt.show()
```

Start IPython with the option -pylab:

```
ipython -pylab
Your PyGtk has set_interactive(), so you can use the
more stable single-threaded Gtk mode.
See https://bugs.launchpad.net/ipython/+bug/270856
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

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

In [1]:
```

Now, we can use all functions implemented by pylab without importing them explicitly, a la Matlab. However, the **prefferd style** is to use:

```python
from matplotlib import pyplot as plt
```

Exit by ^D(Ctrl-D)

The %run magic command:

```
run sattelite.py
run -t sattelite.py
run -p sattelite.py
run -d sattelite.py
```

%run also has special flags for timing the execution of your scripts (-t) and for executing them under the control of either Python's pdb debugger (-d) or profiler (-p). With all of these, %run can be used as the main tool for efficient interactive development of code which you write in your editor of choice.

All output results are automatically stored in a global dictionary named Out and variables named _1, _2, etc. alias them. For example, the result of input line 4 is available either as Out[4] or as _4.

Put a ';' at the end of a line to suppress the printing of output. The _* variables and the Out[] list do get updated with the contents of the output, even if it is not printed. You can thus still access the generated results this way for further processing.

The %history command can show you all previous input, without line numbers if desired (option -n) so you can directly copy and paste code either back in IPython or in a text editor. You can also save all your history by turning on logging via %logstart; these logs can later be either reloaded as IPython sessions or used as code for your programs.

Related to the history:

- Up and down arrows (Ctrl-p/Ctrl-n)
- Search: Ctrl-r and start typing
- Ctrl-a: go to start of line
- Ctrl-e: end of line
- Ctrl-k: kill to end of line

IPython is a line-oriented program, without full control of the terminal. Therefore, it doesn't support true multiline editing. However, it has a number of useful tools to help you in dealing effectively with more complex editing.:

```
%edit 10-20 24 28 # opens an editor with these lines pre-loaded for modification
%save <filename> # saves the lines directly to a named file on disk
```

Here comes an example showing the working flow with python.

Suppose we want to plot the tajectory of a sattelite of the earth as seen from the sun. After thinking about the mathematics of the problem, we might start by trying an implementation interactively in ipython:

```
[gradinar@localhost TutCodes]$ ipython
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: import numpy as np

In [2]: op = 2*np.pi; os = 14.*np.pi

In [3]: R = 4.; r = 2

In [4]: t = np.linspace(0, 2, 100)

In [5]: xp = np.cos(op*t)

In [6]: yp = np.sin(op*t)

In [7]: xs = xp + r*np.cos(os*t)

In [8]: ys = yp + r*np.sin(os*t)

In [9]: from matplotlib import pyplot as plt

In [10]: plt.plot(xs,ys); plt.show()
/usr/lib64/python2.6/site-packages/matplotlib/backends/backend_gtk.py:621: DeprecationWarning: Use t
self.tooltips = gtk.Tooltips()
Out[10]: [<matplotlib.lines.Line2D object at 0x7f5d8c822050>]
```

The result is far from satisfactory. Something went wrong with the radia:

```
In [11]: R = 4.; r = 0.25

In [12]: plt.plot(xs,ys); plt.show()
Out[12]: [<matplotlib.lines.Line2D object at 0x7f5d8c853510>]
```

Nothing changed? Of course, because we haven't recomputed the trajectories yet! We do not want to retype everything, so look at the history:

```
In [13]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
10: plt.plot(xs,ys); plt.show()
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
```

OK, so lines 6 to 10 has to be repeated, but I forgot how it works:

```
In [14]: rep?
  Type:             Magic function
  Base Class:       <type 'instancemethod'>
  String Form:   <bound method InteractiveShell.rep_f of <IPython.iplib.InteractiveShell object at 0x
  Namespace:        IPython internal
  File:             /usr/lib/python2.6/site-packages/IPython/history.py
  Definition:       rep(self, arg)
  Docstring:
                   Repeat a command, or get command to input line for editing

                   - %rep (no arguments):

                   Place a string version of last computation result (stored in the special '_'
                   variable) to the next input prompt. Allows you to create elaborate command
                   lines without using copy-paste::

                       $ l = ["hei", "vaan"]
                       $ "".join(l)
                       ==> heivaan
                       $ %rep
                       $ heivaan_ <== cursor blinking

                   %rep 45

                   Place history line 45 to next input prompt. Use %hist to find out the
                   number.
```

Aha!

```
In [15]: rep 6-10
lines [u'yp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nfrom matplotlib impor
Out[20]: [<matplotlib.lines.Line2D object at 0x7f5d8c629b90>]
```

Hmm, still not good, what if we take more points?

```
In [21]: t = np.linspace(0, 2, 2000)

In [22]: rep 6-10
lines [u'yp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nfrom matplotlib impor
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/home/gradinar/LaTeX_doc/Numcourses/Numcourses/SciPyTools/Start/TutCodes/<ipython console> in <module

ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Upps, one of the variables is longer. What was there?

```
In [25]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
10: plt.plot(xs,ys); plt.show()
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
14: #?rep
15: _ip.magic("rep 6-10")
16: yp = np.sin(op*t)
17: xs = xp + r*np.cos(os*t)
18: ys = yp + r*np.sin(os*t)
19: from matplotlib import pyplot as plt
20: plt.plot(xs,ys); plt.show()
21: t = np.linspace(0, 2, 2000)
22: _ip.magic("rep 6-10")
23: yp = np.sin(op*t)
24: xs = xp + r*np.cos(os*t)
25: _ip.magic("hist ")
```

Aha, I rerun the wrong lines; let’s do it again:

```
In [26]: rep 5-10
lines [u'xp = np.cos(op*t)\nyp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nf
Out[32]: [<matplotlib.lines.Line2D object at 0x7f5d8c556750>]
```

It looks well; let us save our valuable work in a file. What to save?

```
In [33]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
```

```
10: plt.plot(xs,ys); plt.show()
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
14: #?rep
15: _ip.magic("rep 6-10")
16: yp = np.sin(op*t)
17: xs = xp + r*np.cos(os*t)
18: ys = yp + r*np.sin(os*t)
19: from matplotlib import pyplot as plt
20: plt.plot(xs,ys); plt.show()
21: t = np.linspace(0, 2, 2000)
22: _ip.magic("rep 6-10")
23: yp = np.sin(op*t)
24: xs = xp + r*np.cos(os*t)
25: _ip.magic("hist ")
26: _ip.magic("rep 5-10")
27: xp = np.cos(op*t)
28: yp = np.sin(op*t)
29: xs = xp + r*np.cos(os*t)
30: ys = yp + r*np.sin(os*t)
31: from matplotlib import pyplot as plt
32: plt.plot(xs,ys); plt.show()
33: _ip.magic("hist ")
```

Aha, let us save the lines 1 31 2 11 21 5-8 10 in this order:

```
In [34]: save testsat.py 1 31 2 11 21 5-8 10
The following commands were written to file `testsat.py`:
import numpy as np
from matplotlib import pyplot as plt
op = 2*np.pi; os = 14.*np.pi
R = 4.; r = 0.25
t = np.linspace(0, 2, 2000)
xp = np.cos(op*t)
yp = np.sin(op*t)
xs = xp + r*np.cos(os*t)
ys = yp + r*np.sin(os*t)
plt.plot(xs,ys); plt.show()


In [35]:
```

%who/%whos: These functions give information about identifiers you have defined interactively (not things you loaded or defined in your configuration files). %who just prints a list of identifiers and %whos prints a table with some basic details about each identifier.

The point about % with an example on the magic comand %cd:

```
[gradinar@localhost Start]$ ipython
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
```

```
In [1]: cd ..
/home/gradinar/Documents/SciPyTutorial

In [2]: cd = 5

In [3]: cd
Out[3]: 5

In [4]: cd Start
------------------------------------------------------------
File "<ipython console>", line 1
    cd Start
          ^
SyntaxError: invalid syntax


In [5]: %cd Start
/home/gradinar/Documents/SciPyTutorial/Start

In [6]:
```

More nice documentation on IPython here.

1.7 Matplotlib tips

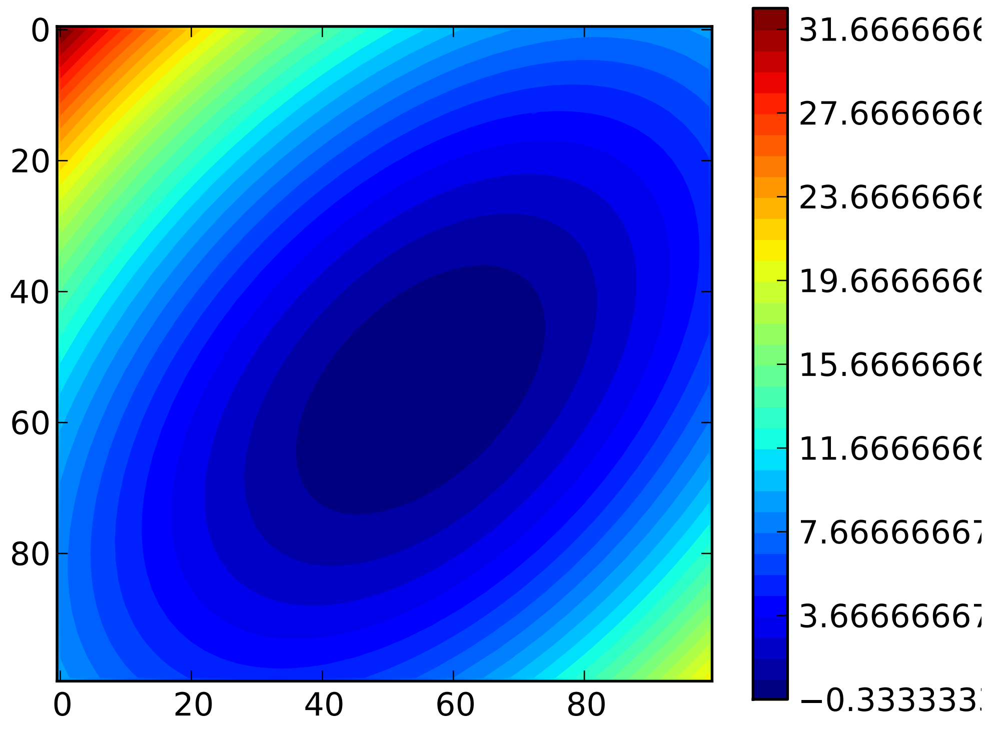
Matplotlib has a beautiful documentation.

You might start by the tutorial or by screen-shots

For special ploting wishes, look into the gallery, you might get either your ploting program already tipped or good visualization ideas.

```python
import numpy as np
from matplotlib import pyplot as plt

# Data of the quadratic form
A = np.array([
    [2., 1.],
    [1., 2.]
    ])

b = np.array([1., 1.])
# mesh
Nx = 100; Ny = 100
X,Y = np.ogrid[-3.:3:Nx*1j, -3.:3:Ny*1j]
z = np.zeros(2)
Z = np.zeros(Nx*Ny)
# compute values
k = 0
for x in X[:,0]:
    z[0] = x
    for y in Y[0,:]:
         z[1] = y
         Z[k] = 0.5*np.dot(z,np.dot(A,z)) - np.dot(z,b)
         k += 1

Z = Z.reshape(Nx,Ny)
```

```

ax = plt.subplot(111)
im = plt.imshow(Z, cmap=plt.cm.jet)
#im.set_interpolation('nearest')
#im.set_interpolation('bicubic')
im.set_interpolation('bilinear')
#ax.set_image_extent(-3, 3, -3, 3)

levels = np.arange(Z.min(), Z.max()+0.01, 1.) # Boost the upper limit to avoid truncation errors.
plt.contourf(Z, levels,
                        cmap=plt.cm.get_cmap('jet', len(levels)-1),
                        )

plt.colorbar()
plt.show()

## fig = plt.figure()
## ax = fig.add_subplot(111, projection='3d') # only newer matplotlib
## ax.plot_surface(x, y, Z, rstride=1, cstride=1, cmap=cm.jet)
## plt.show()


#from enthought.mayavi import mlab
#mlab.surf(x,y,Z)
#mlab.show()
```

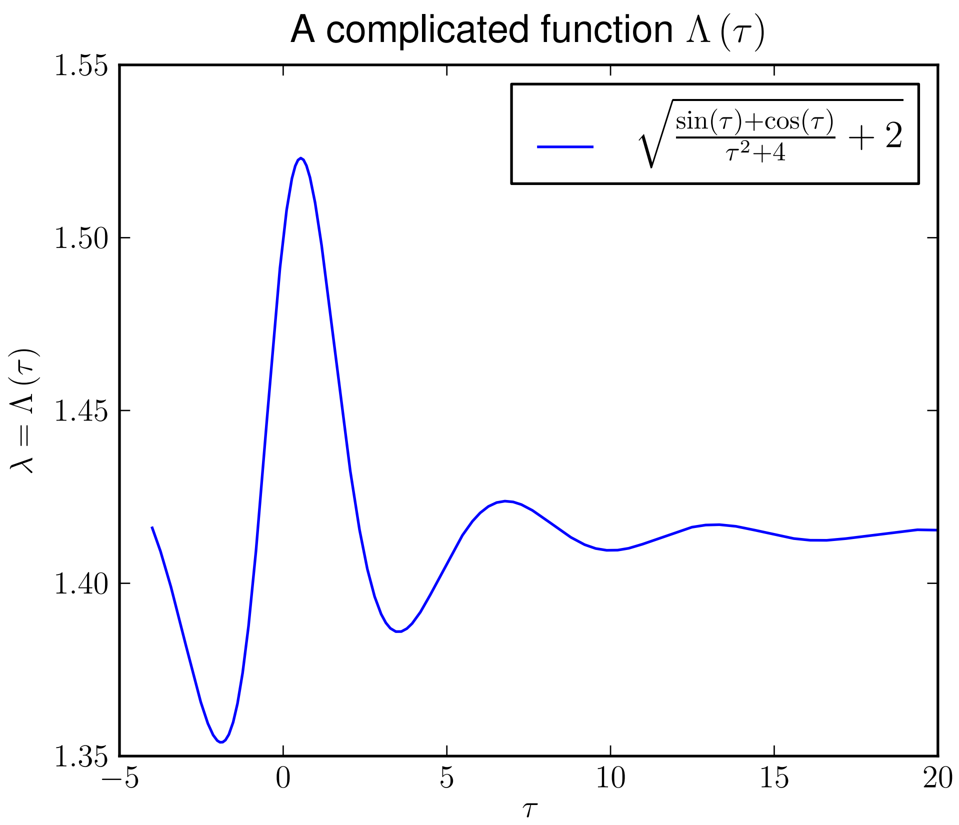


1.7.1 TeX labels

The plotting utilities of matplotlib can use the power of a TeX system if one is installed. We just have to tell matplotlib that it should generate the labels with the help of TeX. We then can use anything that is a valid TeX expression we would otherwise write between the dollar signs in a TeX file. The following code presents a small example showing TeX labels in action.

NOTE: you may have issues with the following usetex-options in case that latex is bad configured on MacOS or Windows.

```python
from numpy import *
from pylab import *
from matplotlib import rc

# Enable usage of real TeX for labels and captions
rc('text', usetex=True)

# Build a complicated function
x = linspace(-8, 8, 5000)

y = zeros(x.shape)
for n in xrange(1,11,2):
    y += 1.0 / n * sin(n * pi * x / 6.0)
y = pi / 4.0 * y

# Plot and label with full TeX support
figure()
plot(x, y, label=r"$\frac{4}{\pi} \sum \frac{1}{n} \sin\left( \frac{n \pi x}{L} \right)$")
xlabel(r"$\xi$")
ylabel(r"$\mathcal{F}\left(f\left(x\right)\right)\left(\xi\right)$")
legend(loc="lower right")
title(r"Fourier series of a square wave: $\mathcal{F}\left(f\right) = \frac{4}{\pi} \sum_{n=1,3,5,\lo
savefig("tex_labels.png")
```



Note that we have to use so called raw strings, i.e. strings with the character "r" as a prefix. Additionally we use the dollar signs inside the strings for delimiting formulas as usual.

1.8 Mayavi2 tips

The *enthought.mayavi.mlab* module, that we call mlab, provides an easy way to visualize data in a script or from an interactive prompt with one-liners as done in the matplotlib `pylab` interface but with an emphasis on 3D visualization using Mayavi2. This allows users to perform quick 3D visualization while being able to use Mayavi's powerful features.

Warning: When using IPython with mlab, as in the following examples, IPython must be invoked with the `-wthread` command line option like so:

```
$ ipython -wthread
```

If you are using the Enthought Python Distribution, or the latest Python(x,y) distribution, the Pylab menu entry will start ipython with the right switch. In older releases of Python(x,y) you need to start "Interactive Console (wxPython)".

Scripting with mayavi2

Reference manual

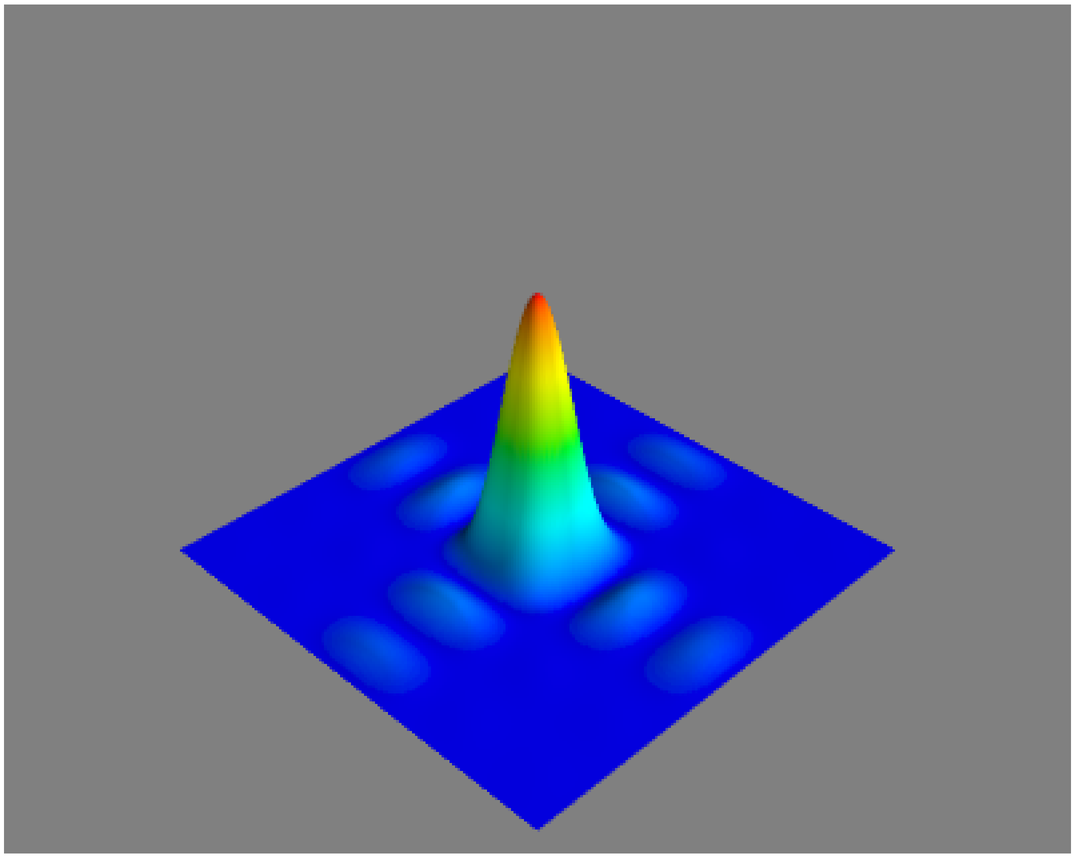
Try:

```
In [1]: from enthought.mayavi import mlab

In [2]: mlab.test_[TAB]

In [3]: mlab.test_contour3d??

In [4]: mlab.test_contour3d()
Out[4]: <enthought.mayavi.modules.iso_surface.IsoSurface object at 0x7fb8e8646b30>

In [5]: import numpy as np

In [6]: t = np.linspace(0, 2*np.pi, 50)

In [7]: u = np.cos(t)*np.pi

In [8]: x, y, z = np.sin(u), np.cos(u), np.sin(t)

In [9]: mlab.clf()

In [10]: mlab.points3d(x,y,z)

In [11]: mlab.clf()

In [12]: mlab.points3d(x,y,z,t, scale_mode='none')

In [13]: mlab.points3d?

In [23]: x, y = np.mgrid[-3:3:100j, -3:3:100j ]

In [24]: z = np.sin(x*x + y*y)

In [25]: mlab.clf()

In [26]: mlab.surf(x,y,z)

In [28]: mlab.mesh(x,y,z)
```

A simple example showing a 3D plot done with the help of mayavi2. The following code shows what it takes to produce such an image. The plotting is essentially only two lines of code!

```
from numpy import mgrid, real, conj, ones, zeros
from numpy.fft.fftpack import fft2
from numpy.fft.helper import fftshift
from enthought.mayavi import mlab

# A mesh grid
X,Y = mgrid[-100:100, -100:100]

# The initial function: a 2D unit step
Z = zeros((200,200))
Z[0:6,0:6] = 0.3*ones((6,6))

# The fourier transform: a 2D sinc(x,y)
W = fftshift(fft2(Z))
W = real(conj(W)*W)

# Display the data with mayavi
# Plot the original function
```

```
#mlab.mesh(X, Y, Z)
# Plot the fourier transformed function
mlab.mesh(X, Y, W)
mlab.savefig("mayavi_fft_plot.png")
```



An Advanced example

And now an animated example:

```
'''Progam menat to test funcyionality of scipy, mayavi2 and matplotlib
   to be run either with
   mayavi2 -x testy.py
   or
   ipython -pylab -wthread
   run testy.py
'''
from enthought.mayavi  import mlab
from scipy import sin, ogrid, array
from pylab import plot, show
# prepare data, hence test scipy elements
x , y = ogrid[-3:3:100j , -3:3:100j]
z = sin(x**2 + y**2)
# test matplotlib
plot(x, sin(x**2)); show()
#now mayavi2
#mlab.options.offscreen = True
obj=mlab.surf(x,y,z)
P = mlab.pipeline
scalar_cut_plane = P.scalar_cut_plane(obj,
                                   plane_orientation='y_axes',
                                   )
scalar_cut_plane.enable_contours = True
scalar_cut_plane.contour.filled_contours = True
scalar_cut_plane.implicit_plane.widget.origin = array([  0.00000000e+00,   1.46059210e+00,  -2.02655

scalar_cut_plane.warp_scalar.filter.normal = array([ 0.,  1.,  0.])
scalar_cut_plane.implicit_plane.widget.normal = array([ 0.,  1.,  0.])
## mlab.draw()
f = mlab.gcf()
f.scene.camera.azimuth(10)

f.scene.show_axes = True
f.scene.magnification = 4 # or 4
mlab.axes()
#engine=mlab.getengine()
## mlab.draw()


# Now animate the data.
dt = 0.01; N = 40
ms = obj.mlab_source
for k in xrange(N):
    x = x + k*dt
    scalars = sin(x**2 + y**2)
    ms.set(x=x, scalars=scalars)
```

The next example demonstrates how to use scalar cut planes in order to evidence subtle characteristics of a plot. Here, we would like to show the Gibbs oscillations in the Fourier series approximation of jumps.

The workflow for such a problem is:

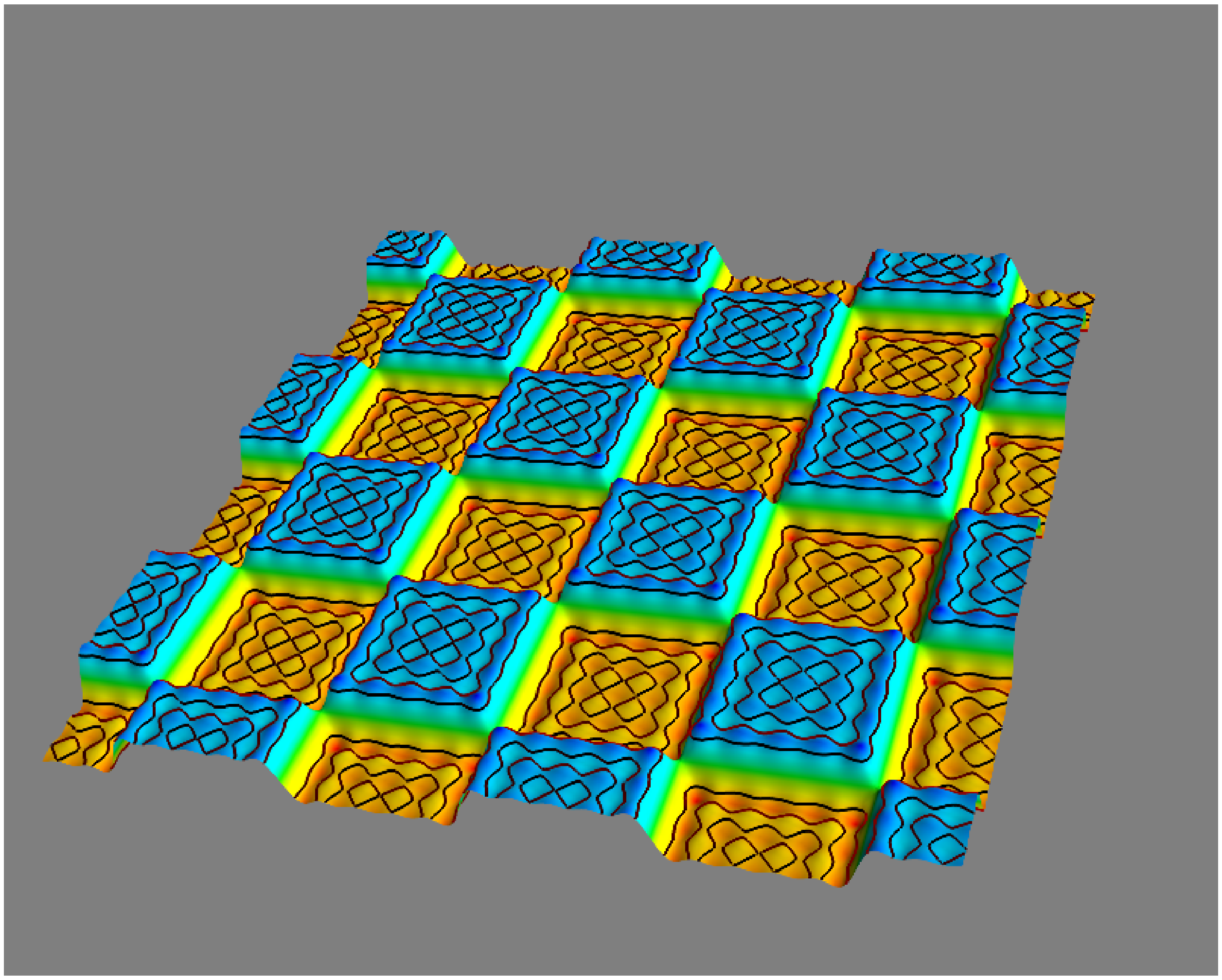
- write a simple program (let us call it pl.py) that plots your object
- run mayavi2 -x pl.py (python pl.py works too for just executing the script)
- use the 'spion' of mayavi2: click on the red point in the menu: the protocol of everything you do visually with the mouse is python-scripted in the opened window
- play with the possibilities and variables that mayavi2 offers, till the image that you see satisfies your exigences
- copy from the 'spion'-window the relevant information for your result into your pl.py file
- save the picture, if you wish; CAUTION: the pictures that mayavi2 produces became huge, if saved in vector graphics ... so eps is not recommeded here!
- re-run mayavi2 -x pl.py to see if this is really what you wanted
- iterate the above process, if wished
- improve speed and quality of the plot by avoiding rendering and increasing size

In the following code (that was developped following this receipt) you find hints for your formulations.

```
from numpy import mgrid, zeros, sin, pi, array
from enthought.mayavi import mlab

Lx = 4
Ly = 4

S = 10
```

```
dx = 0.05
dy = 0.05

# A mesh grid
X,Y = mgrid[-S:S:dx, -S:S:dy]

# The initial function:
#Z = zeros(X.shape)
#Z = sin(pi*X/Lx)*sin(pi*Y/Ly)
#Z[Z>0] = 1
#Z[Z<0] = -1

# The Fourier series:
W = zeros(X.shape)
m = 10
for i in xrange(1,m,2):
    for j in xrange(1,m,2):
        W += 1.0 / (i*j) * sin(i * pi * X / Lx) * sin(j * pi * Y / Ly)
W *= pi / 4.0


# prepare scene
scene = mlab.gcf()
# next two lines came at the very end of the design
scene.scene.off_screen_rendering = True # quicker execution because no redenring
scene.scene.magnification = 4
# plot the object !
obj = mlab.mesh(X, Y, W)
P = mlab.pipeline
# first scalar_cut_plane
scalar_cut_plane = P.scalar_cut_plane(obj,
                               plane_orientation='z_axes',
                               )
scalar_cut_plane.enable_contours = True
scalar_cut_plane.contour.filled_contours = True
# where ?
scalar_cut_plane.implicit_plane.widget.origin = array([-0.025, -0.025,  0.48])
scalar_cut_plane.warp_scalar.filter.normal = array([ 0.,  0.,  1.])
scalar_cut_plane.implicit_plane.widget.normal = array([ 0.,  0.,  1.])
scalar_cut_plane.implicit_plane.widget.enabled = False # do not show the widget
# second scalar_cut-plane
scalar_cut_plane_2 = P.scalar_cut_plane(obj,
                               plane_orientation='z_axes',
                               )
scalar_cut_plane_2.enable_contours = True
scalar_cut_plane_2.contour.filled_contours = True
# where ?
scalar_cut_plane_2.implicit_plane.widget.origin = array([-0.025, -0.025,  -0.48])
scalar_cut_plane_2.warp_scalar.filter.normal = array([ 0.,  0.,  1.])
scalar_cut_plane_2.implicit_plane.widget.normal = array([ 0.,  0.,  1.])
scalar_cut_plane_2.implicit_plane.widget.enabled = False # do not show the widget
# see it from a closer view point
scene.scene.camera.position = [-31.339891336951567, 14.405281950904936, -27.156389988308661]
scene.scene.camera.focal_point = [-0.025000095367431641, -0.025000095367431641, 0.0]
scene.scene.camera.view_angle = 30.0
scene.scene.camera.view_up = [0.63072643330371414, -0.083217283169475589, -0.77153033000256477]
scene.scene.camera.clipping_range = [4.7116394000124906, 93.313165745624019]
scene.scene.camera.compute_view_plane_normal()
```

```
#scene.scene.render() # commented out at the end, we do not want to render anything
mlab.savefig("mayavi_fourier_series_plot.png")
#mlab.show()
```



1.9 Other noteworthy plot commands

While you saw several plot command and know how to do different 2D and 3D plots, we will list here some other plot command that might be interesting.

1.9.1 Plots in the complex plane

Plots of a possibly complex function in the complex plane can be easily done with plot routines from the `mpmath` python package.

The example plots the gamma function over the complex plane:

```
import mpmath as mp

mp.cplot(mp.gamma, re=[-4,4], im=[-4,4], points=10000, file="gamma.png")
```

This is all for plotting the gamma function in the rectangle -4 < Re(z) < 4 and -4 < Im(z) < 4. The above call saves the result to the file “gamma.png” which looks like this:

1.10 Examples

Here we put a few complete examples that should be relevant for the numerics lectures.

1.10.1 First example

Define $f(x) = \frac{1}{\sqrt{1-a\sin(2\pi x-1)}}$.

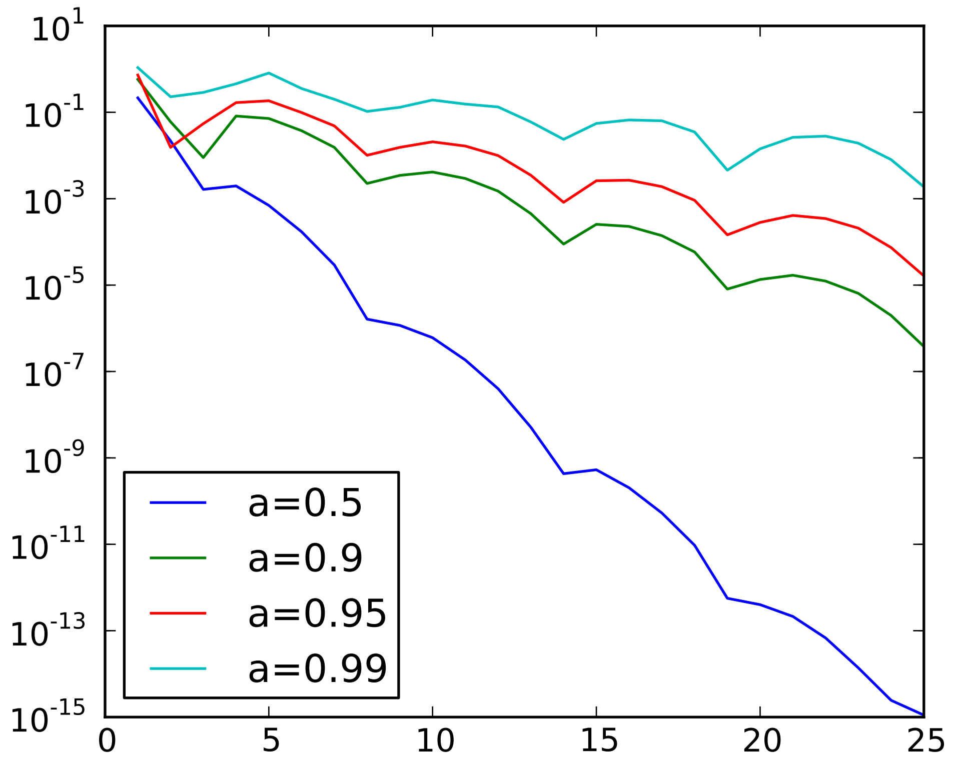
We want to plot the error made by using the trapezoidal rule when approximating $\int_0^1 f(x)dx$ and $\int_0^{0.5} f(x)dx$.

.

As exact value we take the value of the integral as produced by the function quad from the sub-package integrate of scipy. This function is just a wrapper to the Fortran library QUADPACK. Please, use IPython facility to consult the documentation of it.

The aim of the plots is to put in evidence the exponential convergence of the trapezoidal rule when the smoothness of the function extends above its periodical domain. The plots should be done for several possible parameters $0 < a < 1$.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

def trapr(func, a, b, N):
    #h = (b-a)/N; x = np.arange(a, b + h, h)
    x = np.linspace(a,b, N+1); h = x[1]-x[0]
    res = np.sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
    res *= h
    return res

N = 25 #number of quadrature points used
res = np.zeros(N) # alocate space for results
left = 0.; right = 1. # integration limits

lista = [0.5, 0.9, 0.95, 0.99]
for a in lista:
    myf = lambda x: 1./np.sqrt(1.- a*np.sin(2*np.pi*x-1))
    for n in xrange(N):
        res[n] = trapr(myf, left, right, n+1)
    exact, aerr = integrate.quad(myf, left, right)
    #exact = trapr(myf, left, right, 500)
    print 'a=',a,'quad:', exact, 'aerr=', aerr
    err = np.abs(res-exact)
    plt.semilogy(np.arange(1,N+1),err, label='a='+str(a))

plt.legend(loc=0)
plt.show()
```



At lines 1-3 we import the libraries we are using. At lines 5-10 we define our quadrature rule function. Note that there are several possibilities to prepare the array of sampling points. The integrand is evaluated once at all points (vectorial) and the function sum of numpy is used in order to add the necessary values in most efficient way possible.

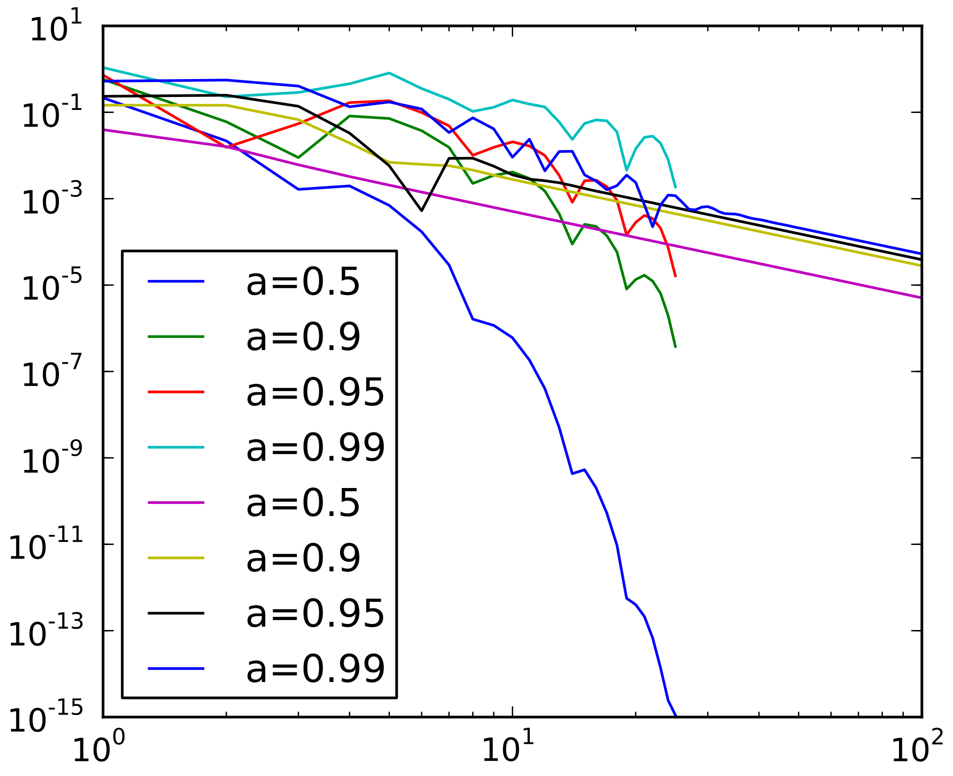
After preparing the data, we run the loop over the interesting values of the parameter a.

At line 18 we define the function to be integrated as a lambda function, i.e. it will be inlined. Note that if we define it as a function of two parameters, we'd have difficulties in the later call of the trapr-function.

We would like to have a program that computes both integrals and makes both plots at one. Here's a possibility:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

def trapr(func, a, b, N):
    """
    Compute the integral from a to b of func via trap. rule with N points
    """
    #h = (b-a)/N; x = np.arange(a, b + h, h)
    x = np.linspace(a,b, N+1); h = x[1]-x[0]
    res = np.sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
    res *= h
    return res

def ploterr(left, right, N, lista, pf):
    res = np.zeros(N) # alocate space for results
    for a in lista:
        myf = lambda x: 1./np.sqrt(1.- a*np.sin(2*np.pi*x-1))
```

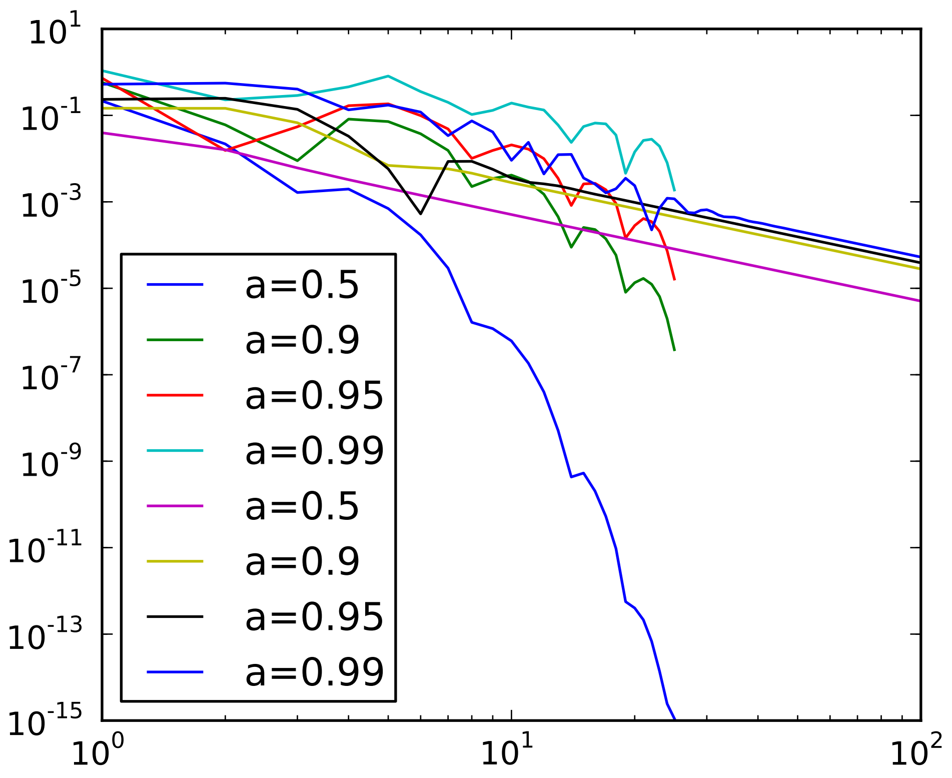
```
19          for n in xrange(N):
20              res[n] = trapr(myf, left, right, n+1)
21          exact, aerr = integrate.quad(myf, left, right)
22          #exact = trapr(myf, left, right, 500)
23          print 'a=',a,'quad:', exact, 'aerr=', aerr
24          err = np.abs(res-exact)
25          pf(np.arange(1,N+1),err, label='a='+str(a))
26
27      plt.legend(loc=0)
28      plt.savefig('err'+str(right)+'.png')
29      plt.close()
30
31  # ----------------------------------------------------------
32  if __name__ == "__main__": # not really necessary here, but you see how to do modules and test them.
33      lista = [0.5, 0.9, 0.95, 0.99]
34      ploterr(0., 1., 25, lista, pf=plt.semilogy)
35      ploterr(0., 0.5, 100, lista, pf=plt.loglog)
```



In last code, we incorporated the loop in a function that gets the integration range, the list of wished parameters and the preferred plot function (semilogy is better for the identification of the exponential convergence). Note that in the picture generated for the web-tutorial, the images appear together, while the code itself write two figures on the disc (line 27). The type of the file is decided by the extension of the name of the file. Note also the string concatenation used in the construction of the file name.

Finally, we would like to rewrite the code such that we avoid the trick based on inlining and locality of the variable a. We have now the occasion to see a very powerful trait of python: class function.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

def trapr(func, a, b, N):
    """
    Compute the integral from a to b of func via trap. rule with N points
    """
    #h = (b-a)/N; x = np.arange(a, b + h, h)
    x = np.linspace(a,b, N+1); h = x[1]-x[0]
    y = func(x)
    res = np.sum(y[1:-1]) + 0.5*(y[0]+y[-1])
    res *= h
    return res

class Myf:
    """
    models a parametric function suited for evidencing
    exponential convergence of trap. rule
    """
    def __init__(self, ina): self.a = ina
    def __str__(self): return str(self.a)+'\n'
    def __call__(self,x):
        res = 1./np.sqrt(1.- self.a*np.sin(2*np.pi*x-1))
        return res

def ploterr(f, left, right, N, lista, pf):
    res = np.zeros(N) # alocate space for results
    for a in lista:
        f.a = a; print f
        for n in xrange(N):
            res[n] = trapr(f, left, right, n+1)
        exact, aerr = integrate.quad(f, left, right)
        print 'a=',a,'quad:', exact, 'aerr=', aerr
        err = np.abs(res-exact)
        pf(np.arange(1,N+1),err, label='a='+str(a))

    plt.legend(loc=0)
    plt.savefig('err'+str(right)+'.png')
    plt.close()

# ----------------------------------------------------------
if __name__ == "__main__": # not really necessary here, but you see how to do modules and test them.
    lista = [0.5, 0.9, 0.95, 0.99]
    myf = Myf(lista[0]) # initialize the function
    ploterr(myf, 0., 1., 25, lista, pf=plt.semilogy)
    ploterr(myf, 0., 0.5, 100, lista, pf=plt.loglog)
```



The class Myf creates callable objects. On line 43 we create one such object that models our parameter function, having the parameter set by the first element of the list of interesting parameters. During the loop at line 29, we modify the parameter of the object, which we use as a function to be integrated by our quadrature routines at lines 32 and 33.

This is a very simple example, but imagine the power at your hands given by the class function when you have to optimize a very complicated function (e.g. given only implicitly as the result of complicated algorithms) depending on several parameters.

1.10.2 Second example

Serial and parallel Monte-Carlo integration via IPython. Serial code:

```python
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz by raw MC.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np
import time

t1 = time.time()

M = 100 # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
```

```python
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance        MC I0 val'
print '------       --------        ---------'
k = 5 # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
for n in N:
    for m in xrange(M):
        x = np.random.rand(n) # sample
        x = np.exp(np.cos(-np.pi*x))
        ex[m] = sum(x)/n # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    print n, vex, ev

t2 = time.time()
t = t2-t1

print "Serial calculation completed, time = %s s" % t
```

Parallel code:

```python
import sys
import time
from IPython.kernel import client
import numpy as np
from matplotlib import pyplot as plt


def mcquad():
    k = 5 # how many experiments
    N = 10**np.arange(1,k+1)
    ex = []
    for n in N:
        x = np.random.rand(n) # sample
        x = np.exp(np.cos(-np.pi*x))
        ex += [np.sum(x)/n] # quadrature
    return ex

#-----------------------------------------------------------------------------
# Setup for parallel calculation
#-----------------------------------------------------------------------------

# The MultiEngineClient is used to setup the calculation and works with all
# engine.
mec = client.MultiEngineClient()

# The TaskClient is an interface to the engines that provides dynamic load
# balancing at the expense of not knowing which engine will execute the code.
tc = client.TaskClient()

# Initialize the common code on the engines.
mec.execute('import numpy as np')


#-----------------------------------------------------------------------------
```

```python
# Perform parallel calculation
#------------------------------------------------------------------------------

print "Running parallel calculation"
M = 100 # number of times we run our MC inetgration

# Submit tasks to the TaskClient for each N as a MapTask.
t1 = time.time()
taskids = []
for m in xrange(M):
        t = client.MapTask(
            mcquad
        )
        taskids.append(tc.run(t))

print "Submitted tasks: ", len(taskids)
sys.stdout.flush()

# Block until all tasks are completed.
tc.barrier(taskids)
t2 = time.time()
t = t2-t1

print "Parallel calculation completed, time = %s s" % t
print "Collecting results..."

# Get the results using TaskClient.get_task_result.
results = [tc.get_task_result(tid) for tid in taskids]
ex = np.array(results)
ev = ex.sum(axis=0)/M
vex = (ex**2).sum(axis=0)/M
vex = vex - ev**2
print vex
```

To use the parallel code, start first an IPython controller with:

```
gradinar@cmath-2% ipcluster local -n 4
```

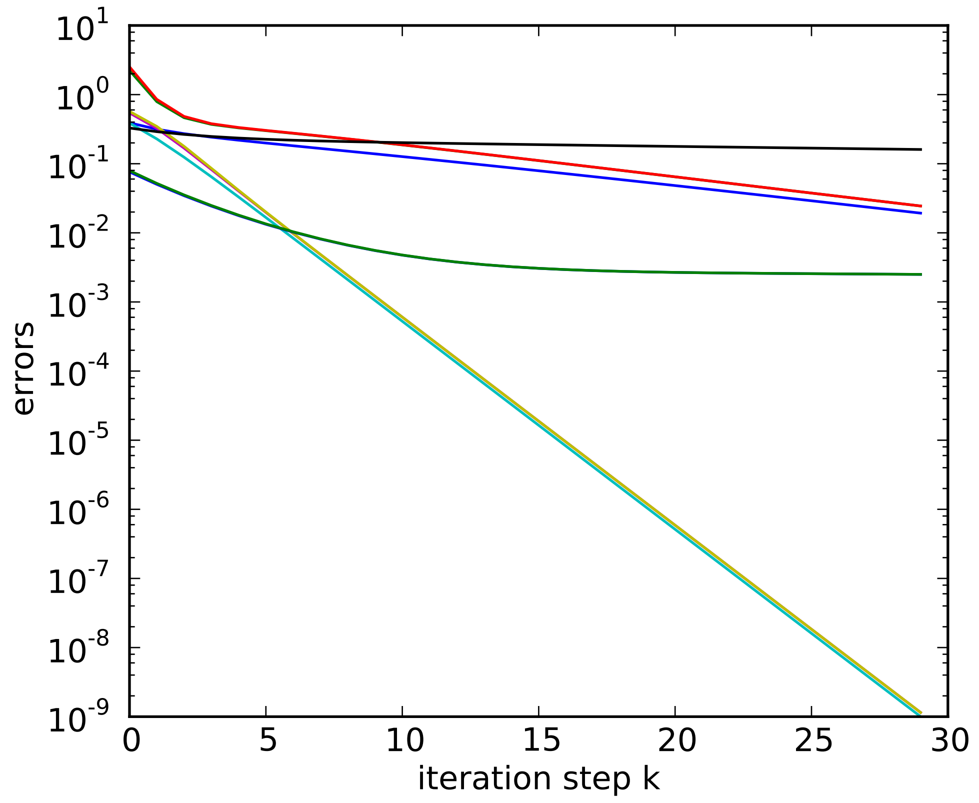
then open a new window on the same machine and start ipython:

```
gradinar@cmath-2% ipython
Python 2.6.4 (r264:75706, Jun  4 2010, 18:20:31)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: run -i par01.py
class FCTaskControllerFromTaskController(Referenceable):
Running parallel calculation
Submitted tasks:  100
Parallel calculation completed, time = 1.54661202431 s
Collecting results...
[  6.04508921e-02   6.88178837e-03   7.46274110e-04   6.50926168e-05   5.63334832e-06]
```

More on Using IPython for parallel computing

1.10.3 Third example

```python
"""
Calculates eigenvalues through direct power method

Originaly, there were the input parameters d and maxit,
but in the skript, d = [1:10]' and maxit = 30 are used.
"""

import numpy as np

def pm(A,z,sgn,ev,ew,maxit=30):
    '''
    performs maxit iterations of the power method
    for the matrix A and the initial vector z
    and computes the error for the largest eigenvalue ew
    having the sign sgn and the corresponding eigenvector ev
    '''
    res = np.zeros((maxit,5))
    s = 1;
    for k in xrange(maxit):
        w = np.dot(A,z)
        no = np.linalg.norm(w)
        rq  = np.real(np.vdot(w,z))
        z = w/no
        err_aew = abs(no-abs(ew))
        err_ew = abs(sgn*rq-ew)
        err_ev = np.linalg.norm(s*z-ev)
        res[k]  = np.array([k, no, err_ev, err_aew, err_ew])
        s = s*sgn
    return res

def runpm(d, fname):
    '''
    performs the analysis of the power iteration starting from
    the array of eigenvalues d; the picture with the results goes
    to the file indicate by fname
    '''
    n = len(d) # size of the matrix
    S = np.triu(np.diag(np.r_[n:0:-1]) + np.ones((n,n)))
    A = np.dot(S, np.dot(np.diag(d), np.linalg.inv(S)) )
    # need exact ew and ew for error calculation
    w, V = np.linalg.eig(A)
    t = np.where(w == abs(w).max())
    k = t[0];
    if len(k) > 1:
        print 'Error: no single larges EV'
        raise ValueError
    ev = V[:,k[0]];  ev /= np.linalg.norm(ev)
    ew = w[k[0]]
    ## # another way to do this is:
    ## t = (w == abs(w).max()) # array of booleans showing where w ==max(|w|)
    ## if t.sum()>1:
    ##     print 'Error: no single larges EV'
    ##     raise ValueError
    ## k = t.argmax()
    ## ev = V[:,k]; ev /= np.linalg.norm(ev)
    ## ew = w[k]
    sgn = 1
    if ew<0: sgn = -1
    print 'ew =', ew, ' sgn =', sgn, '\n ev =', ev
    z = np.random.rand(n); z /= np.linalg.norm(z)
    res = pm(A,z,sgn,ev,ew)
    from matplotlib import pyplot as plt
    plt.semilogy(res[:,0], res[:,2])
    plt.semilogy(res[:,0], res[:,3])
    plt.semilogy(res[:,0], res[:,4])
    plt.xlabel('iteration step k')
    plt.ylabel('errors')
    plt.savefig(fname)
    plt.show()


# ---------------------------------------
if __name__=='__main__':
    n = 10 # size of the matrix
    d = 1.+np.r_[0:n] # eigenvalues
    print 'd =', d, 'd[-2]/d[-1] =', d[-2]/d[-1]
    runpm(d, 'pm1.eps')
    d = np.ones(n); d[-1] = 2. # eigenvalues
    print 'd =', d, 'd[-2]/d[-1] =', d[-2]/d[-1]
    runpm(d, 'pm2.eps')
    d = 1. - 0.5**np.r_[1:5.1:0.5] # eigenvalues
    print 'd =', d , 'd[-2]/d[-1] =', d[-2]/d[-1]
    runpm(d, 'pm3.eps')
```



1.11 Useful links

1.11.1 Tutorials, Manuals and Books

- Tentative NumPy Tutorial
- NumPy for Matlab Users
- Documentations on NumPy and SciPy
- Additional documentation from SciPy
- Scipy tutorial
- Kittens tutorial
- SciPy FAQ page
- NumPy
- Short NumPy introduction
- IPython documentation.
- SymPy documentation
- h5py project website and h5py user guide
- Ask scipy page
- A very good book on the subject is H. Langtangen, "Python Scripting for Computational Science", available electronically on NEBIS, too.
- A very good French overview

1.11.2 Local copies at a given moment:

- Tentative NumPy Tutorial
- NumPy for Matlab Users
- Numpy 1.3 Reference Guide [HTML+zip]
- Scipy 0.7 Reference Guide [HTML+zip]
- French overview

1.11.3 Additional software

- The spyder IDE
- winpdb, a python debugger
- mayavi2, 3D scientific data visualization
- image magic, batch image processing
- mplayer, versatile video player and encoder
- hdfview, HDF file editor

1.12 Installing python packages

There are thousands of python packages available [2] and not all of them can be found in the package repositories of the common Linux distributions. Even if Debian for example consists of over 30‘000 software packages, there are python packages that are not part of the Debian distribution. For this reason you need to install python packages by hand sometimes. This is not a difficult task but you have to obey some constraints.

If you download a python package, it’s usually an archive file in the `.zip` or `.tar.gz` format. Within this archive there is a file called `setup.py`, this acts as an installer. (For more information about the topis, see the official documentation on the `Distutils` python module [3].)

Installing the python package locally on your Linux system is as easy as running the `setup.py` script with the argument `install`. (Uninstalling works similar, but we don’t go into details here.):

```
python setup.py install
```

This tries to install the package on a system-wide basis. If you do not have the permissions to do this or you just want to try a new package quickly, then it’s preferable to install the package only locally. Assume you want to install it in `~/my_python/`. Then you first create that directory:

```
cd ~
mkdir my_python
```

and then tell the setup script where to install the package:

```
python setup.py install --home=~/my_python/
```

that’s it. But you can not `import` the new module yet.

The installation directory lies outside of the so called `python path` where python searches for referenced modules. (This is similar to how the shell searches for binaries in the directories listed in the variable `$PATH`.) To resolve this issue you just have to prepend the new directory path to the `$PYTHONPATH` variable:

```
export PYTHONPATH="~/my_python/lib/python:$PYTHONPATH"
```

where the part `lib/python/` is the exact path to the python module. You can put this last line of shell code into your `~/.bashrc` file to avoid the need of manually executing it every time you want to use the new python module.

[2] Python Package Index http://pypi.python.org/pypi.

[3] For more information on how to distribute python software see http://docs.python.org/distutils/index.html.

CHAPTER

TWO

ADVANCED STUFF

Contents:

2.1 Optimization of Python Code

Good style for speed [1]:

1. Avoid explicit loops: use vectorizd numpy expressions instead. If you really need a loop, then try `list comprehensions` as they are in general much faster. Example:

```
# Slow:
x = []
for i in xrange(8):
    x.append( i**2 )

# Better:
x = [0, 0, 0, 0, 0, 0, 0, 0]
for i in xrange(len(x)):
    x[i] = i**2

# Best:
x = [ i**2 for i in xrange(8) ]
```

2. Avoid module prefix in frequently called funcions:

```
import mod
func = mod.func
for x in hugelist:
     func(x)
```

   Or even import the function into the global namespace:

```
from mod import func
for x in hugelist:
     func(x)
```

3. Plain functions are called faster than class methods: trick:

```
f = myobj.__call__
f(x)
```

4. Inlining functions speeds-up the code

[1] See also http://wiki.python.org/moin/PythonSpeed/PerformanceTips

5. Avoid usieng numpy functions with scalar arguments
6. Use xrange instead of range (but be aware that xrange only supports a small part of the interface of `range()` and similar containers. [2])
7. if-else is faster than try-except (never use exceptions for program flow!)
8. Avoid resizing of numpy arrays
9. Callbacks to Python from Fortran/C/C++ are expensive
10. Avoid unnecessary allocation of large data structures

11. Be careful with type mactching in mixed Python-Fortran software (e.g. flat/ real*8): if the array entry types do not match, arrays will be copied!

12. Use asarray with parameter order='Fortran' when sending arrays from numpy to Fortran routines!

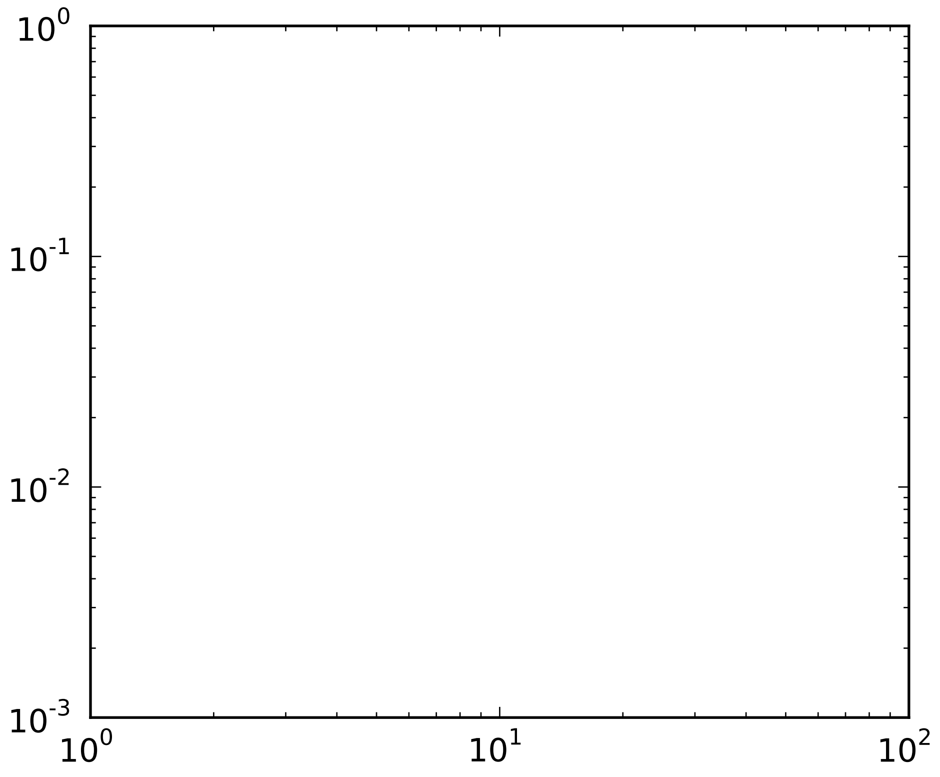
2.2 Timing Your Code

Timing your code is a more complex task you imagine. If the code to run takes long and you are not fond in very precise measurements, then the module time is a very simple way to do it. Otherwise you should use the module timeit. See the two examples below.

```
from numpy import r_, mat, vstack, eye, ones, zeros
from scipy.linalg import qr
import time
nrEXP = 4
sizes = 2**r_[2:6]
qrtimes = zeros((5,sizes.shape[0]))
k = 0
for n in sizes:
    print 'n=', n
    m = n**2
    A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
    A += vstack((eye(n), ones((m-n,n)) ))
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[0,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A, econ=True)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
```

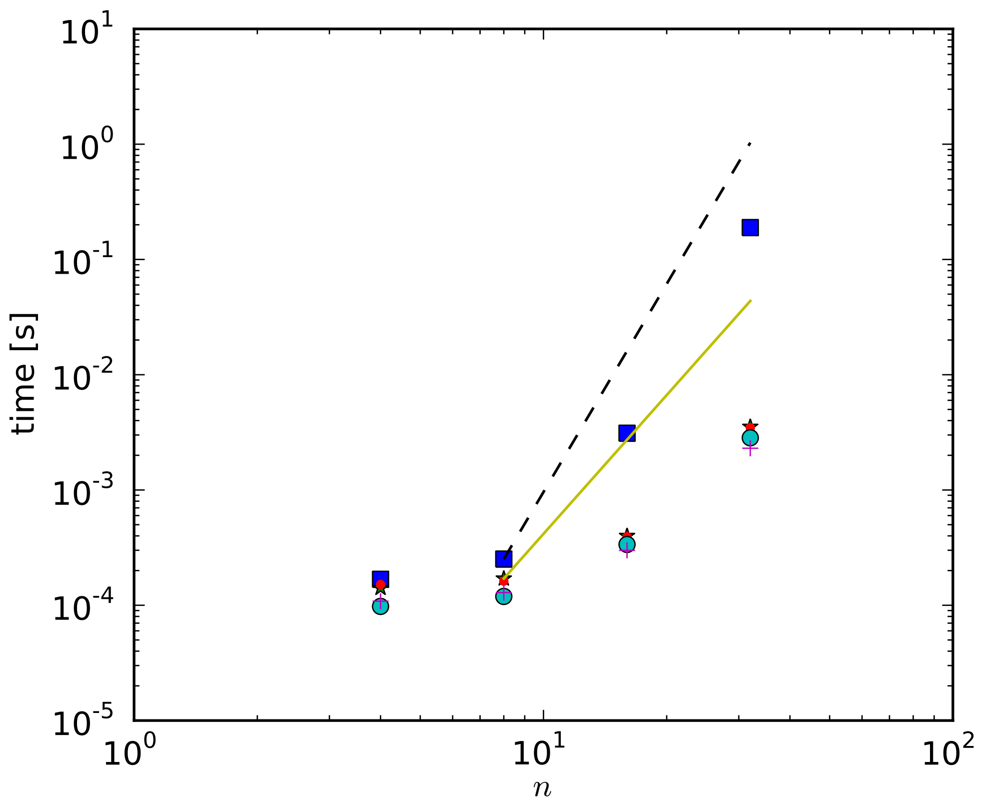
[2] Objects of type xrange don't support slicing, concatenation or repetition, xrange objects have very little behavior: they only support indexing, iteration, and the len() function. The advantage of the xrange type is that an xrange object will always take the same amount of memory, no matter the size of the range it represents. (This is some kind of lazy evaluation.) http://docs.python.org/library/functions.html#xrange

```
    qrtimes[1,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A, econ=True, overwrite_a=True)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[2,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        R = qr(A, mode='r')
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[3,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        R = qr(A, mode='r', econ=True)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[4,k] = avqr
    k += 1

import matplotlib.pyplot as plt
plt.loglog(sizes, qrtimes[0], label='qr')
plt.loglog(sizes, qrtimes[1], label="qr(econ=True)")
plt.loglog(sizes, qrtimes[2], label="qr(econ=True, overwrite_a=True)")
plt.loglog(sizes, qrtimes[3], label="qr(mode='r')")
plt.loglog(sizes, qrtimes[4], label="qr(mode='r', econ=True)")
#plt.legend()
plt.show()
```



```
from numpy import r_, mat, vstack, eye, ones, zeros
from scipy.linalg import qr
import timeit

def qr_full():
    global A
    Q, R = qr(A)

def qr_econ():
    global A
    Q, R = qr(A, econ=True)

def qr_ovecon():
    global A
    Q, R = qr(A, econ=True, overwrite_a=True)

def qr_r():
    global A
    R = qr(A, mode='r')

def qr_recon():
    global A
    R = qr(A, mode='r', econ=True)

nrEXP = 4
sizes = 2**r_[2:6]
qrtimes = zeros((5,sizes.shape[0]))
k = 0
```

```
for n in sizes:
    print 'n=', n
    m = n**2#4*n
    A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
    A += vstack((eye(n), ones((m-n,n)) ))

    t = timeit.Timer('qr_full()','from __main__ import qr_full')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[0,k] = avqr

    t = timeit.Timer('qr_econ()','from __main__ import qr_econ')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[1,k] = avqr

    t = timeit.Timer('qr_ovecon()','from __main__ import qr_ovecon')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[2,k] = avqr

    t = timeit.Timer('qr_r()','from __main__ import qr_r')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[3,k] = avqr


    t = timeit.Timer('qr_recon()','from __main__ import qr_recon')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[4,k] = avqr

    k += 1

#print qrtimes[3]
import matplotlib.pyplot as plt
plt.loglog(sizes, qrtimes[0], 's', label='qr')
plt.loglog(sizes, qrtimes[1], '*', label="qr(econ=True)")
plt.loglog(sizes, qrtimes[2], '.', label="qr(econ=True, overwrite_a=True)")
plt.loglog(sizes, qrtimes[3], 'o', label="qr(mode='r')")
plt.loglog(sizes, qrtimes[4], '+', label="qr(mode='r', econ=True)")
v4 =  qrtimes[1,1]* (sizes/sizes[1])**4
v6 =  qrtimes[0,1]* (sizes/sizes[1])**6
plt.loglog(sizes, v4, label='$O(n^4)$')
plt.loglog(sizes, v6, '--',label='$O(n^6)$')
#plt.legend(loc=2)
plt.xlabel('$n$')
plt.ylabel('time [s]')
#plt.savefig('qrtiming.eps')
plt.show()
```



2.3 Gluing Stand-Alone Applications

This section follows closely the approach of Langtangen: Python Scripting for Computational Science and Engineering, chapter 2.3.

2.3.1 Scripting

Suppose you have a stand-alone code for solving the ODE:

$m\frac{d^2u}{dt^2} + b\frac{du}{dt} + cf(u) = A\cos\omega t$

and you wish to have simulation results for varoius choices of the parameters. For instance, the provided

oscillator.f inp run

(gfortran -O3 -o oscillator oscillator.f)

reads the following parameters from the standard input, in the listed order: $m, b, c,$ name of f(y) function, A, $\omega, y_0, t_{stop}, \Delta t$

The values of the parameters may be placed in a file inp: and the program can be run as

oscillator < inp

The output consists of data points $t_i u(t_i)$, a suitable format to plot, e.g. with gnuplot.

Under the link you'll find fortran, C and python versions of this stand-alon code.

Our first goal is to simplify the user's interaction with the program; with the script simviz1.py it is possible to adjust the parameters through command-line options:

```
python simviz1.py -m 3.3 -b 0.9 -dt 0.05
```

The result should be png-files and an otional plot on the screen. The files for each such experiment should go in a subdirectory.

```python
#!/usr/bin/env python
import sys, math

# default values of input parameters:
m = 1.0; b = 0.7; c = 5.0; func = 'y'; A = 5.0; w = 2*math.pi
y0 = 0.2; tstop = 30.0; dt = 0.05; case = 'tmp1'; screenplot = 1

# read variables from the command line, one by one:
while len(sys.argv) > 1:
    option = sys.argv[1];            del sys.argv[1]
    if   option == '-m':
        m = float(sys.argv[1]);      del sys.argv[1]
    elif option == '-b':
        b = float(sys.argv[1]);      del sys.argv[1]
    elif option == '-c':
        c = float(sys.argv[1]);      del sys.argv[1]
    elif option == '-func':
        func = sys.argv[1];          del sys.argv[1]
    elif option == '-A':
        A = float(sys.argv[1]);      del sys.argv[1]
    elif option == '-w':
        w = float(sys.argv[1]);      del sys.argv[1]
    elif option == '-y0':
        y0 = float(sys.argv[1]);     del sys.argv[1]
    elif option == '-tstop':
        tstop = float(sys.argv[1]); del sys.argv[1]
    elif option == '-dt':
        dt = float(sys.argv[1]);     del sys.argv[1]
    elif option == '-noscreenplot':
        screenplot = 0
    elif option == '-case':
        case = sys.argv[1];          del sys.argv[1]
    else:
        print sys.argv[0],': invalid option',option
        sys.exit(1)

# create a subdirectory:
d = case                  # name of subdirectory
import os, shutil
if os.path.isdir(d):      # does d exist?
    shutil.rmtree(d)      # yes, remove old directory
os.mkdir(d)               # make new directory d
os.chdir(d)               # move to new directory d

# make input file to the program:
f = open('%s.i' % case, 'w')
# write a multi-line (triple-quoted) string with
# variable interpolation:
f.write("""
        %(m)g
        %(b)g
        %(c)g
        %(func)s
        %(A)g
        %(w)g
        %(y0)g
        %(tstop)g
        %(dt)g
        """ % vars())
f.close()
# run simulator:
cmd = '../oscillator < %s.i' % case  # command to run
failure = os.system(cmd)
if failure:
    print 'running the oscillator code failed'; sys.exit(1)

# make file with gnuplot commands:
f = open(case + '.gnuplot', 'w')
f.write("""
set title '%s: m=%g b=%g c=%g f(y)=%s A=%g w=%g y0=%g dt=%g';
""" % (case, m, b, c, func, A, w, y0, dt))
if screenplot:
    f.write("plot 'sim.dat' title 'y(t)' with lines;\n")
f.write("""
set size ratio 0.3 1.5, 1.0;
# define the postscript output format:
set term postscript eps monochrome dashed 'Times-Roman' 28;
# output file containing the plot:
set output '%s.ps';
# basic plot command:
plot 'sim.dat' title 'y(t)' with lines;
# make a plot in PNG format:
set term png small;
set output '%s.png';
plot 'sim.dat' title 'y(t)' with lines;
""" % (case, case))
f.close()
# make plot:
cmd = 'gnuplot -geometry 800x200 -persist ' + case + '.gnuplot'
failure = os.system(cmd)
if failure:
    print 'running gnuplot failed'; sys.exit(1)
```

One can easily generate an automatic LaTeX-report or HTML-report containing the values of the parameters and the corresponding pictures.

2.3.2 Conducting Numerical Experiments

Suppose we want to keep most of the parameters fixed and vary one parameter between a minimum and a maximum value; the results should be collected in a HTML-report:

```
python loop4simviz1.py m_min m_max m_increment [ simviz1.py options ]
```

```
#!/usr/bin/env python
"""calls simviz1.py with different m values (in a loop)"""
import sys, os
usage = 'Usage: %s m_min m_max m_increment [ simviz1.py options ]' \
        % sys.argv[0]
```

```

try:
    m_min = float(sys.argv[1])
    m_max = float(sys.argv[2])
    dm    = float(sys.argv[3])
except IndexError:
    print usage;  sys.exit(1)

simviz1_options = ' '.join(sys.argv[4:])

html = open('tmp_mruns.html', 'w')
html.write('<HTML><BODY BGCOLOR="white">\n')
psfiles = []  # plot files in PostScript format

m = m_min
while m <= m_max:
    case = 'tmp_m_%g' % m
    cmd = 'python simviz1.py %s -m %g -case %s' % \
          (simviz1_options, m, case)
    print 'running', cmd
    os.system(cmd)
    html.write('<H1>m=%g</H1> <IMG SRC="%s">\n' \
               % (m,os.path.join(case,case+'.png')))
    psfiles.append(os.path.join(case,case+'.ps'))
    m += dm
html.write('</BODY></HTML>\n')
```

or let vary any parameter:

```
loop4simviz2.py parameter min max increment [ simviz2.py options ]
```

```
#!/usr/bin/env python
"""
As loop4simviz1.py, but here we call simviz2.py, make movies,
and also allow any simviz2.py option to be varied in a loop.
"""
import sys, os
usage = 'Usage: %s parameter min max increment '\
        '[ simviz2.py options ]' % sys.argv[0]
try:
    option_name = sys.argv[1]
    min = float(sys.argv[2])
    max = float(sys.argv[3])
    incr = float(sys.argv[4])
except:
    print usage;  sys.exit(1)

simviz2_options = ' '.join(sys.argv[5:])

html = open('tmp_%s_runs.html' % option_name, 'w')
html.write('<HTML><BODY BGCOLOR="white">\n')
psfiles = []    # plot files in PostScript format
pngfiles = []   # plot files in PNG format

value = min
while value <= max:
    case = 'tmp_%s_%g' % (option_name,value)
    cmd = 'python simviz2.py %s -%s %g -case %s' % \
          (simviz2_options, option_name, value, case)
    print 'running', cmd
    os.system(cmd)
    psfile = os.path.join(case,case+'.ps')
    pngfile = os.path.join(case,case+'.png')
    html.write('<H1>%s=%g</H1> <IMG SRC="%s">\n' \
               % (option_name, value, pngfile))
    psfiles.append(psfile)
    pngfiles.append(pngfile)
    value += incr
cmd = 'convert -delay 50 -loop 1000 %s tmp_%s.gif' \
      % (' '.join(pngfiles), option_name)
print 'converting PNG files to animated GIF:\n', cmd
os.system(cmd)
html.write('<H1>Movie</H1> <IMG SRC="tmp_%s.gif">\n' % option_name)
html.write('</BODY></HTML>\n')
html.close()
```

2.4 Symbolical calculations

Even if we want to deal with numerical computations mainly, I'd like to give a short introduction into symbolic calculation. There are several reasons why we want to include some symbolic manipulations in a numerical code. For example we want to do an error analysis and would like to be able to evaluate the known exact solution on various different grids. Or we have a closed form solution for some right hand side. Beyond the toy examples there exist also industrial strength codes that use symbolic computation in a preparation step before going into the numerics [3].

In any case, we could put the symbolic expression literally into the source code. But sometimes it would be useful if we could do a few more things with the expression beside numerical evaluation, for example we may need the first derivative. The benefit of symbolic calculation is now that we can compute this derivation without explicitly put it into the source code or falling back to numerical techniques.

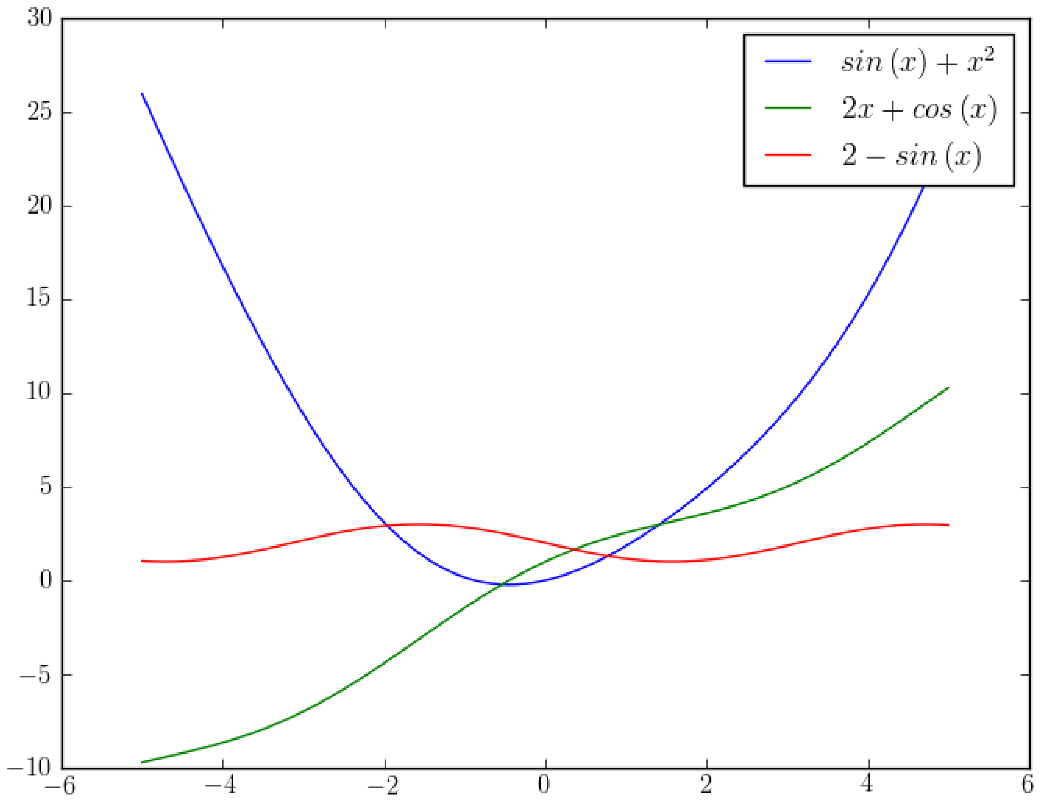
In the following section I'll show a really simple example of how to use the power of the `sympy` python module. This module acts as a library for symbolic calculation and is quite easy to use yet surprisingly powerful for it's complexity.

```
# Use explicit namespaces to make clear from which package a function comes.
import sympy as sp
import numpy as np
import pylab as pl

# Define a symbol
x = sp.Symbol("x")

# A symbolic expression
expr = sp.sin(x) + x**2

# We can evaluate it for any value of x:
print(expr({x:sp.pi**2}))


# A function for evaluating the expression by symbolic
# manipulations. We have to assign the function parameter
# y to the symbolic variable x of the expression.
fs = lambda y: expr({x:y})

# Again we can evaluate the expression for any value of x easily:
```

[3] See fore example `SyFi` which is part of the FEniCS project: http://www.fenicsproject.org/wiki/FEniCS_Project

```
print(fs(2))
print(fs(sp.pi))


# But we want the function to work with the arrays from numpy too.
# Thus we have to lambdify the expression.
fn = sp.lambdify(x, expr, "numpy")

xvals = np.linspace(-5,5,100)
yvals = fn(xvals)


# Compute the derivative of our expression
dexpr = sp.diff(expr, x)

# And again lambdify it
fnd = sp.lambdify(x, dexpr, "numpy")

# Evaluate the derivative on the given grid
dyvals = fnd(xvals)


# Compute the second derivative of our expression
ddexpr = sp.diff(expr, x, 2)

# And again lambdify it
fndd = sp.lambdify(x, ddexpr, "numpy")

# Evaluate the derivative on the given grid
ddyvals = fndd(xvals)


# And plot the function sampled at the grid points xvals
# Note how we reuse the symbolic expressions for generating the plot labels too!
pl.figure()
pl.plot(xvals, yvals, label=sp.latex(expr))
pl.plot(xvals, dyvals, label=sp.latex(dexpr))
pl.plot(xvals, ddyvals, label=sp.latex(ddexpr))
pl.legend()
pl.savefig("symbolics_demo.png")
```

And this is how the output looks like



For many other simple examples, please look at the introductory level `sympy` tutorial [4].

Note that you should use the **computationaly expensive symbolical calculations** only in the preparation phase of a numerical simulation program and never in a main computation loop.

2.5 Making of: animations

Plots are nice but sometimes an animation is even better. Thus we take the opportunity and explain in this section how to generate simple animations. An animation is nothing else than a pile of images which are shown one after the other.

This is also exactly the way we will create animations: in a first step we draw each image (we call this a frame) and after we got all the frames, we will glue them together.

There is more than one way to make animations. We describe here two possibilities which probably are used most.

- Animated images: use the common `.gif` image file format (so called *animgifs*).
- Videos: use some video compression algorithm and are placed inside a data container.

Both of these options have their respective advantages and disadvantages. Animated gif images tend to be larger in file size, but will yield good results without much technicalities. If you only have a couple of images, say up to 50 frames, then animgifs are probably the easiest way to go. (The file *animwave.gif* from the example below has a size of 750 KB)

[4] http://docs.sympy.org/tutorial.html#first-steps-with-sympy

For longer animations you should of course use the advantages of modern video compression techniques. But mature video encoding tools usually have a huge number of options interacting in a complex manner. It's not easy to find the parameters which give you the best result. For example the video compression algorithms are tuned for areas and can introduce bad artefacts if the image has only a bunch sharp lines in front of a uniform background. But this is what plots look like most of the time! (By the way, the file *animwave.flv* from the example below has a size of 700 KB.)

2.5.1 External tools

We additionaly need two programs which are not part of the python installation as proposed at the beginning of this tutorial. But they may be already preinstalled on a common Linux system today.

The first of these tools is called `convert` which is itself part of the well known `ImageMagic` batch image processor. We use this tool for creating animated gif images.

- ImageMagic can be found at http://www.imagemagick.org/

(This is not the only tool that can compose gif files, you may also try `gifsicle` which can be found at http://www.lcdf.org/gifsicle/. My own experience is that this tool sometimes is faster, uses less memory and the resulting files are smaller in size. But don't take this as a given.)

To encode videos we use the famous `mplayer` / `mencoder` toolbox. This is one of the most versatile tools for working with video data. Mplayer can play almost any file format and has a huge set of options. Mencoder is responsible for encoding videos and does a very good job.

- Mplayer / Mencoder can be found at http://www.mplayerhq.hu/

Describing these tools is far beyond the scope of this document. You should take a look at the documentation available on the webpages mentioned above. The *man pages* also provide a clean description of the parameters and show some useful examples at the very end.

2.5.2 Command lines

For creating the animations we relay on these external tools. We just call them from inside the python scripts. You may also call them directly at your favourite shell. The commands used in the example below are for animated gifs:

```
convert -delay 20 filebasename*.png animfilename.gif
```

where `filebasename` is a prefix that is common to all files you want to include in the animation. Instead of `filebasename*.png` you can plug in any regular expression here. Similar `animfilename` if the name the output file will have. The parameter `delay` specifies the frame rate, a value of 20 means a time delay of 20 ms between each two frames.

The command for encoding videos is much more complex (note that this is all **one** single line):

```
mencoder mf://./ filenamebase*.png  -fps 5 -ofps 5 -o animfilename.flv -of lavf
-oac mp3lame -lameopts abr:br=64 -srate 22050 -ovc lavc
-lavcopts vcodec=flv:keyint=50:vbitrate=700:mbd=2:mv0:trell:v4mv:cbp:last_pred=3
-vf yadif -sws 9
```

where `filebasename` and `animfilename` and as above. The resulting video is of file type `.flv` (flash video) [5]. You should change the other parameters only if you really know what you do. (These values here are passed on since the early universe.)

[5] For more information on this particular video format see http://en.wikipedia.org/wiki/Flv.

2.5.3 Example Code

The following example shows a simple wave packet propagating in a one-dimensional space.

The function `make_frames()` is responsible for plotting the frames. Notice how we start a new figure with the command `figure()` in each iteration of the for loop. This is to avoid plotting into the last frame. Also we need some bits of intelligence while saving the images. Because we use regular expressions for selecting all images that will be part of the animation, we are subject to the way the shell sorts filenames. To avoid wrong sorting we use a zero padding here:

```
"wave_"+(5-len(str(i)))*"0"+str(i)+".png"
```

results in file names like the following ones:

```
wave_00000.png wave_00001.png wave_00002.png wave_00003.png ...
```

and assures the files will be in the correct order. The other two functions:

```
make_animation_animgif(filebasename, animfilename)
```

and:

```
make_animation_video(filenamebase, animfilename)
```

just compose a string containing the shell commands shown above. Then they will execute this command on the system's shell. You can copy these two functions to your own code and use them to generate animations. The rest of the code should be self-explanatory.

By the way, with code similar to the one shown here you can dynamically compose and execute **any** command on the system's shell! Therefore you should be very careful with this powerful mechanism.

Note: this example do not work on windows (as long as "convert" and "mencoder" are not installed)

```
from numpy import *
from pylab import *

import os
import subprocess as sp


def make_frames():
    """This function generates the images (frames) of the animation.
    """

    wave = lambda x, f: sin(f*x)
    envelope = lambda x, l: exp(-(x-l)**2)

    x = linspace(-5,5,2000)
    positions = linspace(-5,5,50)

    # Plot all the individual frames
    for i, pos in enumerate(positions):
        y = wave(x,10) * envelope(x, pos)
        z = envelope(x, pos)

        # Plot the frame
        figure()
        plot(x, z, color=(0.5,0,0))
        plot(x,-z, color=(0.5,0,0))
        plot(x, y, color=(0,0,1))
        ylim(-1.3, 1.3)

        # Save the with a filename that ensures correct order on the filesystem
        savefig("wave_"+(5-len(str(i)))*"0"+str(i)+".png")



def make_animation_video(filenamebase, animfilename):
    """A function that takes frames and composes them into a video.
    """

    # The shell command controlling the 'mencoder' tool.
    # Please refer to the man page for the parameters.
    # Warning: Use of multiline strings with EOL escaping to reduce line length!
    command = """mencoder mf://./""" + filenamebase + """*.png\
    -fps 5 -ofps 5 -o """ + animfilename + """.flv\
    -of lavf -oac mp3lame -lameopts abr:br=64 -srate 22050 -ovc lavc\
    -lavcopts vcodec=flv:keyint=50:vbitrate=700:mbd=2:mv0:trell:v4mv:cbp:last_pred=3\
    -vf yadif -sws 9"""

    # Execute the command on the system's shell
    proc = sp.Popen(command, shell=True)
    os.waitpid(proc.pid, 0)



def make_animation_animgif(filebasename, animfilename):
    """Take a couple of images and compose them into an animated gif image.
    """

    # The shell command controlling the 'convert' tool.
    # Please refer to the man page for the parameters.
    command = "convert -delay 20 " + filebasename + "*.png " + animfilename + ".gif"

    # Execute the command on the system's shell
    proc = sp.Popen(command, shell=True)
    os.waitpid(proc.pid, 0)



if __name__ == "__main__":
    make_frames()
    make_animation_video("wave_", "animwave")
    make_animation_animgif("wave_", "animwave")
```

2.6 Input / Output of numerical data

In this section we would like to focus on a last but very important point once you will do real world numerical simulations. It's all about how to store numerical simulation data into files in an efficient and portable way. We will give a short introduction to a general purpose file format called HDF, the *Hierarchical Data Format*, which is carefully designed to store and organize large amounts of numerical data [6].

[6] The people responsible for taking care of the HDF standard have their home at http://www.hdfgroup.org/.

2.6.1 Introduction into HDF

When we said that HDF is a file format this is not the whole truth. HDF is rather some kind of scheme for creating file formats. Maybe we can say that in some aspects it's similar to e.g. html which is not a file format but a language for describing data.

A HDF file has the following file structure and includes only two major types of objects:

- Datasets, which are multidimensional arrays of a homogenous type
- Groups, which are container structures which can hold datasets and other groups

This results in a truly hierarchical, filesystem-like data format. In fact, resources (for now read: data sets) in an HDF file are even accessed using the POSIX-like syntax "/path/to/resource". Metadata is stored in the form of user-defined, named attributes attached to groups and datasets.

In addition HDF includes an type system for specifying data types (are the numbers here 32 bit real floats or 64 bit complex floats or ...), and dataspace objects which represent selections over dataset regions. The API is object-oriented with respect to datasets, groups, attributes, types, dataspaces and property lists.

By the way, currently there exist two major versions of HDF, HDF4 and HDF5, which differ significantly in design and API. We will only deal with HDF5 here.

2.6.2 Handling HDF files with python

Until now we just talked about the ideas of HDF. Let's now look how to work with HDF files. The library that handles HDF files has interfaces to many programing languages and there are python bindings too. Here we will present the `h5py` python package. Let's just cite the official description from the `h5py` developers:

"The HDF5 library is a versatile, mature library designed for the storage of numerical data. The h5py package provides a simple, Pythonic interface to HDF5. A straightforward high-level interface allows the manipulation of HDF5 files, groups and datasets using established Python and NumPy metaphors."

The `h5py` project is located at http://h5py.alfven.org/ where you can find the python package as well as a very detailed user reference manual and many examples of usage.

2.6.3 A first example

Ok, enough theory by now, let's go to a simple hands-on example. Hopefully the code is self-explanatory with all these comments in place.

```python
import numpy as np
import scipy as sp
import pylab as pl
import h5py as hdf

# A grid
x = np.linspace(-sp.pi,sp.pi,100)
X, Y = np.meshgrid(x,x)

# Some values
U = sp.sin(X)
V = sp.cos(X)


# Save the data into a hdf5 file
filename = "data.hdf5"

```

```python
# Create and open the file with the given name
outfile = hdf.File(filename)

# Store some data under /axisgrid with direct assignment of the data x
outfile.create_dataset("axisgridx", data=x)

# Create a group for storing further data under /grid/*
grp_grid = outfile.create_group("grid")

# Store some data under /grid/grid* with direct assignment of the data X and Y
grp_grid.create_dataset("gridx", data=X)
grp_grid.create_dataset("gridy", data=Y)

# Store the vector field values under another group called /values
grp_vals = outfile.create_group("values")

# Prepare space for storing the values but do not assign values yet
grp_vals.create_dataset("valsx", U.shape, np.floating)
grp_vals.create_dataset("valsy", V.shape, np.floating)

# Now assign the data for U and V and show slicing via ":" and ellipsis "..."
outfile["/values/valsx"][:] = U
outfile["/values/valsy"][...] = V

# And close the file
outfile.close()


# Open the file again, now in read only mode (we don't want to write data anymore!)
infile = hdf.File(filename, "r")

# Get the data, but omit every second value, just to show off some slicing
a = infile["/grid/gridx"][::2,::2]
b = infile["/grid/gridy"][::2,::2]
c = infile["/values/valsx"][::2,::2]
d = infile["/values/valsy"][::2,::2]

# Plot the data as usual
pl.figure()
pl.quiver(a,b, c,d)
pl.savefig("hdf_example.png")
```

2.6.4 HDFView

The program HDFView is a tool for browsing and editing HDF files. Using HDFView, you can view the internal file hierarchy in a tree structure, create new files, add or delete groups and datasets, view and modify the content of a dataset and much more. There are even some basic plotting facilities which can serve for a first glance at the numerical data.

You can download `hdfview` from the URL http://www.hdfgroup.org/hdf-java-html/hdfview/. The installation should pose no problems, the software is written in the Java language and available for various operating systems.

If we open the file created by the above example with the program `hdfview` then we will see something like the following image. On the left there is the structural tree shown with its two groups `grid` and *values‘* and the five leaves containing the data. On the right the actual values of the data node `valsx` is shown as a two-dimensional table (remember that the data array was two-dimensional). And on the bottom we see some information about the data currently examined node, for example the data type (64 bit floats in this case) and the size of the array (here 100 times 100 entries).

Of course you can do much more with the HDF data format and we can only show the tip of this iceberg. For example, HDF supports transparent compression of the data, further descriptive attributes for nodes, special sorting and indexing of data for fast retrieval and many things more. For more information you should read the `h5py` user guide http://h5py.alfven.org/docs/guide/index.html.

2.7 Combining Python with Fortran, C and C++

This section follows closely the approach of Langtangen: Python Scripting for Computational Science and Engineering, chapter 5.

When is integration of Python with Fortran, C, or C++ of interest?

- migration of slow parts of a Python code to Fortran or C/C++
- access to existing numerical code (from Python)

Typically: user interfaces, i/O, report generation and management is in Python, while the computationally intensive functions are in Fortran/C/C++

A source of troubles: Fortran/C/C++ has strong typing rules, while in Python variables are typeless...

Needs writing wrapper code: each function need a Python wrapper. Automatic generation: SWIG for C/C++, f2py for Fortran

2.7.1 Scientific Hello World Examples

Pure Python implementation:

```python
#!/usr/bin/env python
"""Pure Python Scientific Hello World module."""
import math, sys

def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s

def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1,r2,s)
```

with application script:

```python
#!/usr/bin/env python
"""Scientific Hello World script using the module hw."""
import sys
from hw import hw1, hw2
try:
    r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
except IndexError:
    print 'Usage:', sys.argv[0], 'r1 r2'; sys.exit(1)
print 'hw1, result:', hw1(r1, r2)
print 'hw2, result: ',
hw2(r1, r2)
```

We intend to migrate the functions hw1 and hw2 in the hw module to Fortran/C/C++. We eill also onvolve a third function hw3, which is a version of hw1 where s is an output argument, in call by reference style, and not a return variable. A pure implementation of hw3 has no meaning.

2.7.2 Combining Python and Fortran

```
      program hwtest
      real*8 r1, r2 ,s
      r1 = 1.0
      r2 = 0.0
      s = hw1(r1, r2)
      write(*,*) 'hw1, result:',s
      write(*,*) 'hw2, result:'
      call hw2(r1, r2)
      call hw3(r1, r2, s)
      write(*,*) 'hw3, result:', s
      end

      real*8 function hw1(r1, r2)
      real*8 r1, r2
      hw1 = sin(r1 + r2)
      return
      end

      subroutine hw2(r1, r2)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      return
      end

C     special version of hw1 where the result is
C     returned as an argument:

      subroutine hw3_v1(r1, r2, s)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      return
      end

C     F2py treats s as input arg. in hw3_v1; fix this:

      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end

C     test case sensitivity:

      subroutine Hw4(R1, R2, s)
      real*8 R1, r2, S
Cf2py intent(out) s
      s = SIN(r1 + r2)
      ReTuRn
      end

C end of F77 file
```

f2py comes automaticaly with numpy; we make a subdirectory f2py-hw and run f2py in this subdirectory:

```
f2py -m hw -c ../hw.f
```

The reuslt is the module hw.so which may be loaded into Python by an ordinary import statement.

- -m option specifies the name of the extension module
- -c indicates that f2py should compile and link the module
- –fcompiler=gfortran specifies the compiler to use (less error messages)
- f2py -c –help-fcompiler to see a list of Fortran compilers installed
- f2py -c –help-compiler to see a list of C compilers installed

Test if everything went fine:

```
python -c 'import hw'
```

We can test now the new module with the previous hwa.py program, who does not know if the module hw is written in Python or Fortran.

When dealing with more complicated libraries, one may want to create Python interfaces to only some of the functions, e.g.:

```
f2py -m hw -c --fcompiler=gfortran ../hw.f only: hw1 hw2 :
```

-h hw.pyf –overwrite-signature options makes f2py to write the module interfaces to a file hw.pyf that you can adjust to your needs:

```
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f only: hw1 hw2 :
```

The function hw3_v1 enables us to see one difficulty of mixing Fortran and Python. The rsult of the computations is stored in the output variable s. Since Fortran employs the call by reference for all the arguments, any change to an argument is visible in the calling code. Let us see how the interface looks like:

Bei dfault, f2py treates r1, r2, and s as input argumnets; calling hw3_v1 shows that the value of the Fortran s variable is not returned to the Python s variable in the call. The remdey is to tell f2py that s is an output parameter. We do this by replacing in the hw.pyf file

```
real*8 :: s
```

with:

```
real*8 intent(out) :: s
```

as in subroutine hw3. The directives intent(in) and intent(out) specify input or output variables, while intent(in,out) are employed for variables for both input and output.

Compiling and linking the hw module with th emodified interface specification in hw.pyf are now performed by:

```
f2py -c --fcompiler=gfortran hw.pyf ../hw.f
```

Note thet f2py changes the interface to become more Pythonic, i.e. we write in Python:

```
s = hw3(r1,r2)
```

For a Fortran routine:

```
subroutine somef(in1, in2, o1, o2, o3, o4, io1)
```

where in1, in2 are input variables, o1, o2, o3, o4 are output variables and io1 is an input/output variable, the generated Python interface will have i1, i2, io1 as input arguments and o1, o2, o3, o4, io1 is returend as a tuple:

```
o1, o2, o3, o4, io1 = somef(i1, i2, io1)
```

As an alternative to the modification of the .pyf file, we may insert an intent specification as a special Cf2py comment in the Fortran source code file. The safest way of writing hw3 is to specify the input/output nature of all the function arguments:

```
      subroutine hw3(r1,r2,s)
      real*8 :: r1, r2, s
Cf2py intent(in) r1
Cf2py intent(in) r2
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end
```

2.7.3 Building an Extension Module with Distutils

The standard way for buiilding and installing Python modules uses Python's Distutils tool which comes with the standard Python distribution. An enhanced version of Distutils with better support for Fortran code comes with numpy. We have to create a script setup.py which calls a function steup in Distutils. Then, in order to build the extension module:

```
python setup.py build
```

or to build and install:

```
python steup.py install
```

To build in the current working directory:

```
python setup.py build build_ext --inplace
```

An example of steup.py file

```python
from numpy.distutils.core import Extension, setup

setup(name='hw',
       ext_modules=[Extension(name='hw', sources=['../hw.f'])],
       )
```

This document as pdf

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*
- *fenics*

INDEX