Introduction to Scientific Computing in Python

Continuum Analytics and Robert Johansson

August 17, 2015

Table of Contents

Chapter 1
Introduction to scientific computing with Python

1.1 The role of computing in science

Science has traditionally been divided into experimental and theoretical disciplines, but during the last several decades computing has emerged as a very important part of science. Scientific computing is often closely related to theory, but it also has many characteristics in common with experimental work. It is therefore often viewed as a new third branch of science. In most fields of science, computational work is an important complement to both experiments and theory, and nowadays a vast majority of both experimental and theoretical papers involve some numerical calculations, simulations or computer modeling.

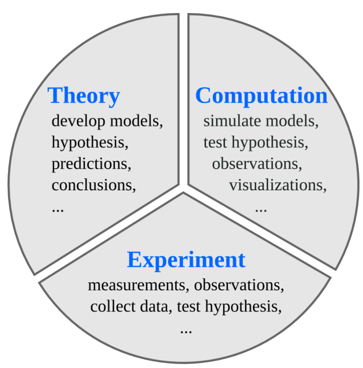


Figure 1.1: Theory, experiment, computation

In experimental and theoretical sciences there are well established codes of conduct for how results and methods are published and made available to other scientists. For example, in theoretical sciences, derivations, proofs and other results are published in full detail, or made available upon request. Likewise, in experimental sciences, the methods used and the results are published, and all experimental data should be available upon request. It is considered unscientific to withhold crucial details in a theoretical proof or experimental method, that would hinder other scientists from replicating and reproducing the results.

In computational sciences there are not yet any well established guidelines for how source code and generated data should be handled. For example, it is relatively rare that source code used in simulations for published papers are provided to readers, in contrast to the open nature of experimental and theoretical work. And it is not uncommon that source code for simulation software is withheld and considered a competitive advantage (or unnecessary to publish).

However, this issue has recently started to attract increasing attention, and a number of editorials in high-profile journals have called for increased openness in computational sciences. Some prestigious journals, including Science, have even started to demand of authors to provide the source code for simulation software used in publications to readers upon request.

Discussions are also ongoing on how to facilitate distribution of scientific software, for example as supplementary materials to scientific papers.

1.1.1 References

- Reproducible Research in Computational Science, Roger D. Peng, Science 334, 1226 (2011).
- Shining Light into Black Boxes, A. Morin et al., Science 336, 159-160 (2012).
- The case for open computer programs, D.C. Ince, Nature 482, 485 (2012).

1.2 Requirements on scientific computing

Replication and **reproducibility** are two of the cornerstones of the scientific method. With respect to numerical work, complying with these concepts have the following practical implications:

- Replication: An author of a scientific paper that involves numerical calculations should be able to rerun the simulations and replicate the results upon request. Other scientists should also be able to perform the same calculations and obtain the same results, given the information about the methods used in a publication.
- Reproducibility: The results obtained from numerical simulations should be reproducible with an independent implementation of the method, or using a different method altogether.

In summary: A sound scientific result should be reproducible, and a sound scientific study should be replicable.

To achieve these goals, we need to:

- Keep and take note of *exactly* which source code and version were used to produce data and figures in published papers.
- Record information of which version of external software was used. Keep access to the environment that was used.
- Make sure that old codes and notes are backed up and kept for future reference.
- Be ready to give additional information about the methods used, and perhaps also the simulation codes, to an interested reader who requests it (even years after the paper was published!).
- Ideally codes should be published online, to make it easier for other scientists interested in the codes to access them.

1.2.1 Tools for managing source code

Ensuring replicability and reprodicibility of scientific simulations is a *complicated problem*, but there are good tools to help with this:

- Revision Control System (RCS) software.
  - Good choices include:

        * git - http://git-scm.com
        * mercurial - http://mercurial.selenic.com. Also known as `hg`.
        * subversion - http://subversion.apache.org. Also known as `svn`.

- Online repositories for source code. Available as both private and public repositories.
    - Some good alternatives are
        * Github - http://www.github.com
        * Bitbucket - http://www.bitbucket.com
        * Privately hosted repositories on the university's or department's servers.

Note Repositories are also excellent for version controlling manuscripts, figures, thesis files, data files, lab logs, etc. — basically any digital content that must be preserved and is frequently updated. Again, both public and private repositories are readily available. They are also excellent collaboration tools!

1.3 What is Python?

Python is a modern, general-purpose, object-oriented, high-level programming language.

General characteristics of Python:

- **clean and simple language:** Easy-to-read and intuitive code, easy-to-learn minimalistic syntax, maintainability scales well with size of projects.
- **expressive language:** Fewer lines of code, fewer bugs, easier to maintain.

Technical details:

- **dynamically typed:** No need to define the type of variables, function arguments or return types.
- **automatic memory management:** No need to explicitly allocate and deallocate memory for variables and data arrays. No memory leak bugs.
- **interpreted:** No need to compile the code. The Python interpreter reads and executes the python code directly.

Advantages:

- The main advantage is ease of programming, minimizing the time required to develop, debug and maintain the code.
- Well designed language that encourage many good programming practices:
- Modular and object-oriented programming, good system for packaging and re-use of code. This often results in more transparent, maintainable and bug-free code.
- Documentation tightly integrated with the code.
- A large standard library, and a large collection of add-on packages.

Disadvantages:

- Since Python is an interpreted and dynamically typed programming language, the execution of Python code can be slow compared to compiled statically typed programming languages, such as C and Fortran.
- Somewhat decentralized, with different environment, packages and documentation spread out at different places. Can make it harder to get started.

1.4 What makes Python suitable for scientific computing?

- Python has a strong position in scientific computing:
    - Large community of users, easy to find help and documentation.

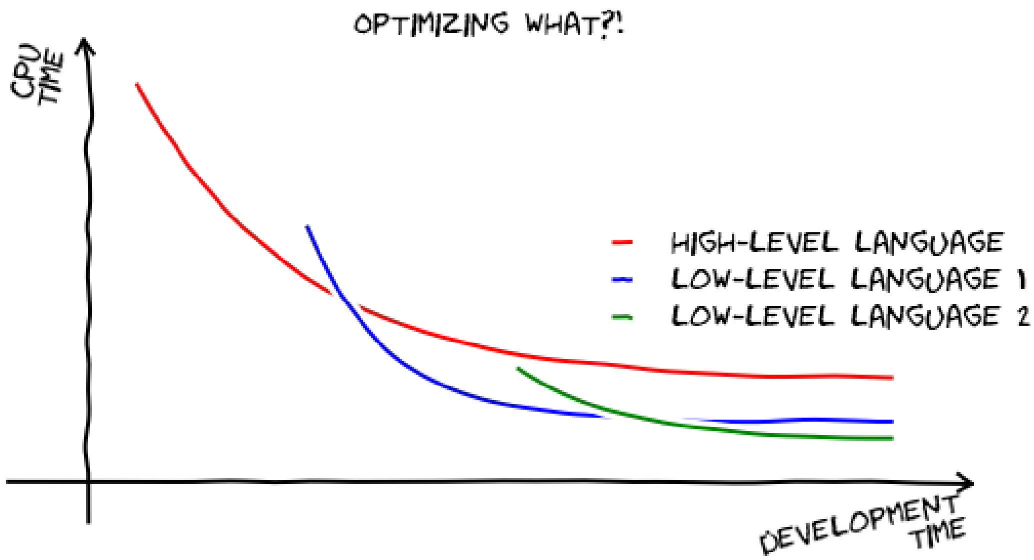


Figure 1.2: Optimizing what

- Extensive ecosystem of scientific libraries and environments
  - numpy: http://numpy.scipy.org - Numerical Python
  - scipy: http://www.scipy.org - Scientific Python
  - matplotlib: http://www.matplotlib.org - graphics library
- Great performance due to close integration with time-tested and highly optimized codes written in C and Fortran:
  - blas, altas blas, lapack, arpack, Intel MKL, ...
- Good support for
  - Parallel processing with processes and threads
  - Interprocess communication (MPI)
  - GPU computing (OpenCL and CUDA)
- Readily available and suitable for use on high-performance computing clusters.
- No license costs, no unnecessary use of research budget.

1.4.1 The scientific Python software stack

1.4.2 Python environments

Python is not only a programming language, but often also refers to the standard implementation of the interpreter (technically referred to as CPython) that actually runs the Python code on a computer.

There are also many different environments through which the Python interpreter can be used. Each environment has different advantages and is suitable for different workflows. One strength of Python is that it is versatile and can be used in complementary ways, but it can be confusing for beginners so we will start with a brief survey of Python environments that are useful for scientific computing.

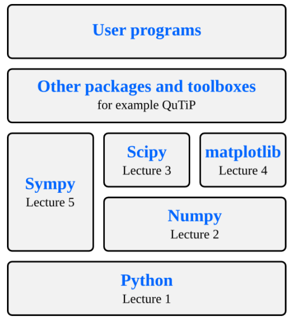


Figure 1.3: Scientific Python Stack

1.4.3 Python interpreter

The standard way to use the Python programming language is to use the Python interpreter to run Python code. The python interpreter is a program that reads and execute the Python code in files passed to it as arguments. At the command prompt, the command `python` is used to invoke the Python interpreter.

For example, to run a file `my-program.py` that contains Python code from the command prompt, use::

```
$ python my-program.py
```

We can also start the interpreter by simply typing `python` at the command line, and interactively type Python code into the interpreter.

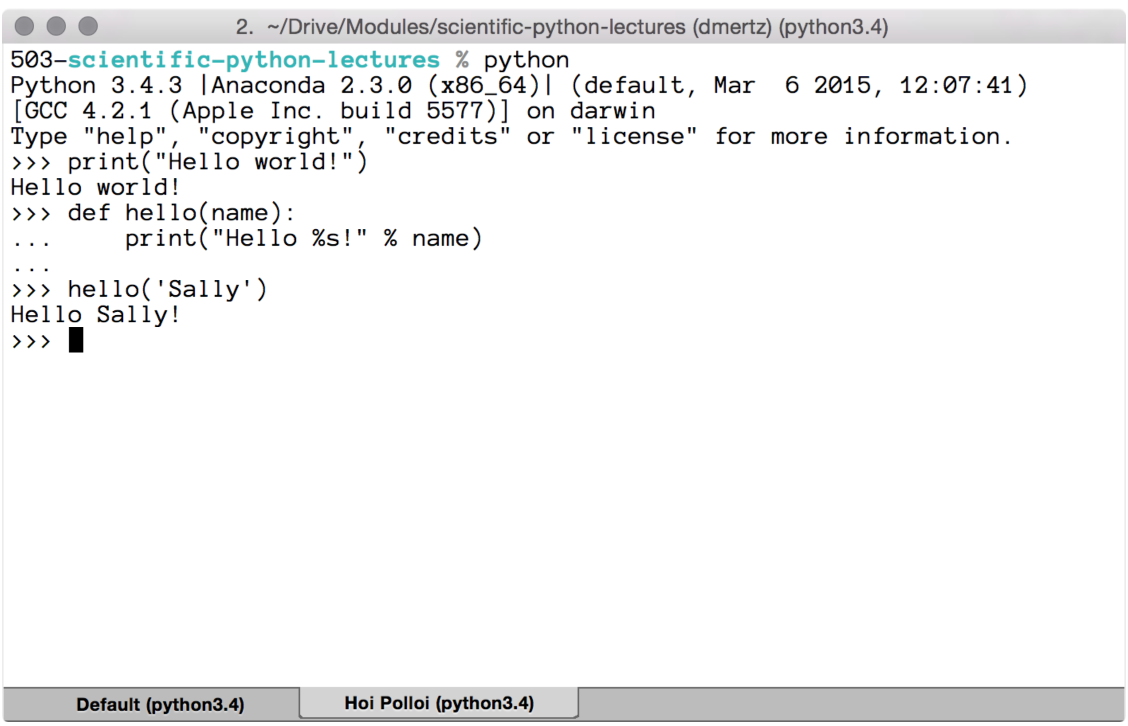


Figure 1.4: Python screenshot

This is often how we want to work when developing scientific applications, or when doing small calculations. But the standard Python interpreter is not very convenient for this kind of work, due to a number of limitations.

1.4.4 IPython

IPython is an interactive shell that addresses the limitation of the standard Python interpreter, and it is a work-horse for scientific use of python. It provides an interactive prompt to the Python interpreter with a greatly improved user-friendliness.

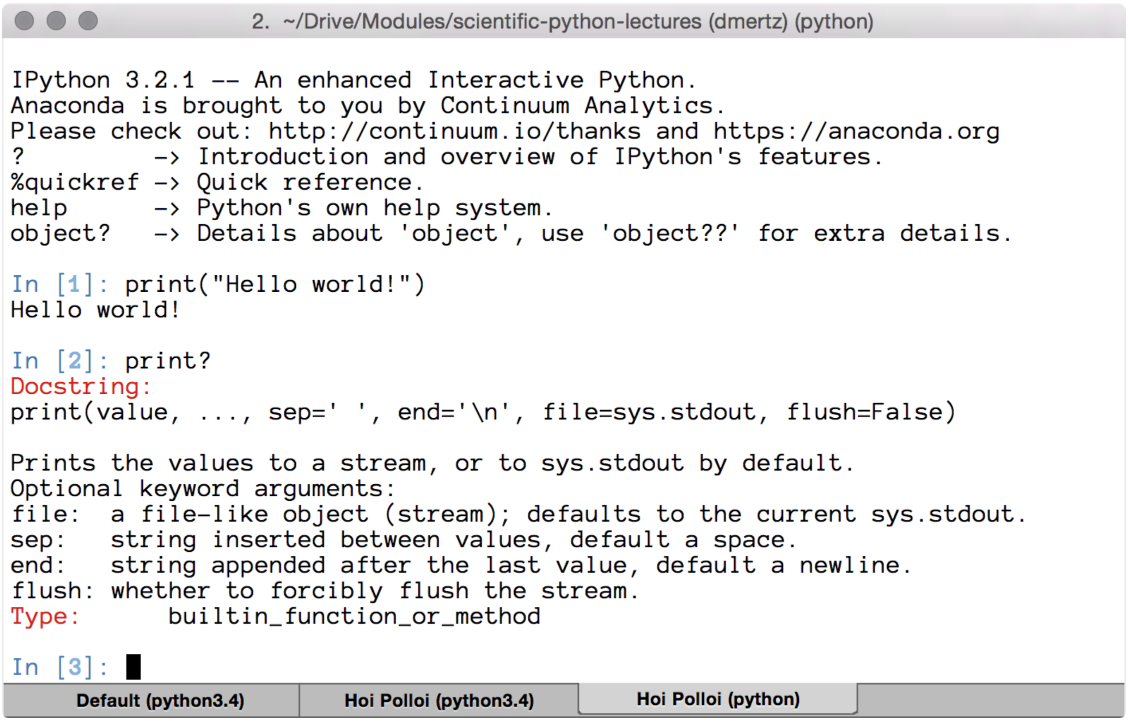


Figure 1.5: IPython screenshot

Some of the many useful features of IPython includes:

- Command history, which can be browsed with the up and down arrows on the keyboard.
- Tab auto-completion.
- In-line editing of code.
- Object introspection, and automatic extract of documentation strings from Python objects like classes and functions.
- Good interaction with operating system shell.
- Support for multiple parallel back-end processes, that can run on computing clusters or cloud services like Amazon EC2.

1.4.5 IPython notebook

IPython notebook is an HTML-based notebook environment for Python, similar to Mathematica or Maple. It is based on the IPython shell, but provides a cell-based environment with great interactivity, where calculations can be organized and documented in a structured way.

Although using a web browser as graphical interface, IPython notebooks are usually run locally, from the same computer that run the browser. To start a new IPython notebook session, run the following command:

```
$ ipython notebook
```

from a directory where you want the notebooks to be stored. This will open a new browser window (or a new tab in an existing window) with an index page where existing notebooks are shown and from which new notebooks can be created.

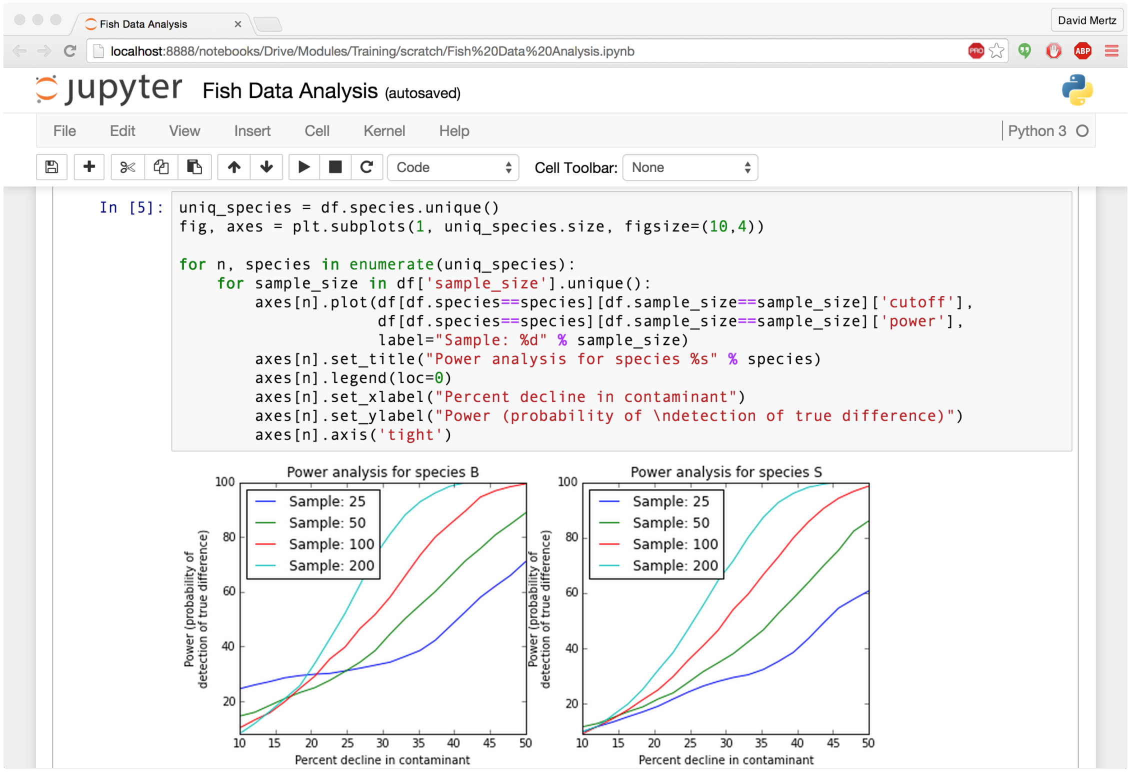


Figure 1.6: IPython notebook

1.4.6 Spyder

Spyder is a MATLAB-like IDE for scientific computing with python. It has the many advantages of a traditional IDE environment, for example that everything from code editing, execution and debugging is carried out in a single environment, and work on different calculations can be organized as projects in the IDE environment.

Some advantages of Spyder:

- Powerful code editor, with syntax high-lighting, dynamic code introspection and integration with the python debugger.
- Variable explorer, IPython command prompt.
- Integrated documentation and help.

1.5 Versions of Python

There are two currently maintained families of python: Python 2 and Python 3. Python 3 will eventually supercede Python 2, but it is not fully backward-compatible. For these lectures either version will be work, since most features are compatible between versions.

To see which version of Python you have, run

```
% python --version
Python 3.4.3 :: Anaconda 2.3.0 (x86_64)
% python2 --version
Python 2.7.10
```

Several versions of Python can be installed in parallel, as shown above.

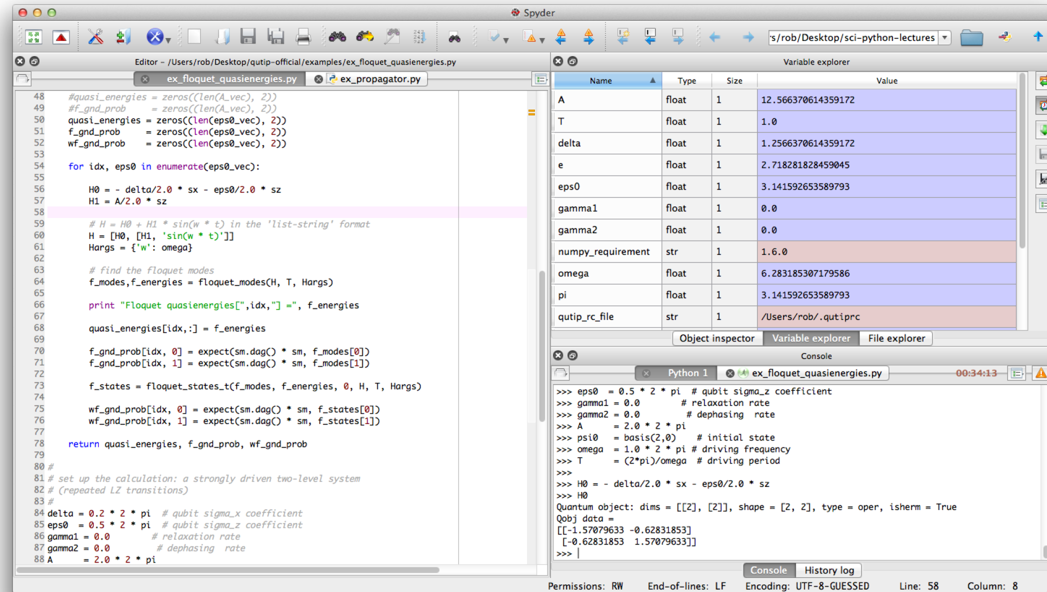


Figure 1.7: Spyder screenshot

1.6 Installation

1.6.1 Linux

In Ubuntu Linux, to installing python and all the requirements of these lectures run:

```
% sudo apt-get install python ipython ipython-notebook
% sudo apt-get install python-numpy python-scipy python-matplotlib python-sympy
% sudo apt-get install spyder
```

To use the Anaconda Python distribution that includes a very large range of scientific libraries pre-compiled (including all of those required for these lectures), you can run:

```
% wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
% bash Miniconda-latest-Linux-x86_64.sh
% conda install anaconda
```

1.6.2 MacOS X

Python is included by default in Mac OS X, but OS releases typically lag the latest Python versions. To use the Anaconda Python distribution that includes a very large range of scientific libraries pre-compiled, you can run these commands in a terminal:

```
% wget https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
% bash Miniconda3-latest-MacOSX-x86_64.sh
% conda install anaconda
```

1.6.3 Windows

Windows lacks a good packaging system, so the easiest way to set up a Python environment is to install a pre-packaged distribution. Some good alternatives are:

- Anaconda. The Anaconda Python distribution comes with many scientific computing and data science packages and is free, including for commercial use and redistribution. It also has add-on products such as Accelerate, IOPro, and MKL Optimizations, which have free trials and are free for academic use.
- Python.org. Official distribution from the creators of Python. The tools `pip` (included with recent versions) or `conda` may be used to install additional packages.
- Enthought Python Distribution. EPD is a commercial product but is available free for academic use.

1.7 Further reading

- Python. The official Python website.
- Python tutorials. The official Python tutorials.
- Think Python. A free book on Python.

Chapter 2

Introduction to Python programming

This curriculum builds on material by J. Robert Johansson from his "Introduction to scientific computing with Python," generously made available under a Creative Commons Attribution 3.0 Unported License at https://github.com/jrjohansson/scientific-python-lectures. The Continuum Analytics enhancements use the Creative Commons Attribution-NonCommercial 4.0 International License.

2.1 Python program files

- Python code is usually stored in text files with the file ending "`.py`":

  ```
  myprogram.py
  ```

- Every line in a Python program file is assumed to be a Python statement, or part thereof.
  - The only exception is comment lines, which start with the character `#` (optionally preceded by an arbitrary number of white-space characters, i.e., tabs or spaces). Comment lines are usually ignored by the Python interpreter.

- To run our Python program from the command line, we use:

  ```
  $ python myprogram.py
  ```

- On UNIX systems, it is common to define the path to the interpreter on the first line of the program. Note that this is a comment line as far as the Python interpreter is concerned:

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

If we do, and if we additionally set the file script to be executable, we can run the program like this:

```
$ myprogram.py
```

2.1.1 Example:

```
In [1]: ls scripts/hello-world*.py
```

```
scripts/hello-world-in-swedish.py  scripts/hello-world.py
```

```
In [2]: cat scripts/hello-world.py
```

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

print("Hello world!")
```

```
In [3]: !python scripts/hello-world.py
```

```
Hello world!
```

2.1.2 Character encoding

The standard character encoding is ASCII, but we can use any other encoding; for example, UTF-8. To specify that UTF-8 is used, we include the special line

```
# -*- coding: UTF-8 -*-
```

at the top of the file.

```
In [4]: cat scripts/hello-world-in-swedish.py
```

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

print("Hej världen!")
```

```
In [5]: !python scripts/hello-world-in-swedish.py
```

```
Hej världen!
```

Other than these two *optional* lines in the beginning of a Python code file, no additional code is required for initializing a program.

2.2 IPython notebooks

This file - an IPython notebook - does not follow the standard pattern with Python code in a text file. Instead, an IPython notebook is stored as a file in the JSON format. The advantage is that we can mix formatted text, Python code and code output. It requires the IPython notebook server to run it though, and therefore isn't a standalone Python program as described above. Other than that, there is no difference between the Python code that goes into a program file or an IPython notebook.

2.3 Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more.

2.3.1 References

- The Python Language Reference: http://docs.python.org/2/reference/index.html
- The Python Standard Library: http://docs.python.org/2/library/

To use a module in a Python program, it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

```
In [6]: import math
```

This includes the whole module and makes it available for use later in the program. For example, we can do:

```
In [7]: import math
        x = math.cos(2 * math.pi)
        print(x)
```

```
1.0
```

Alternatively, we can choose to import all symbols (functions and variables) in a module to the current namespace (so that we don't need to use the prefix "`math.`" every time we use something from the `math` module:

```
In [8]: from math import *
        x = cos(2 * pi)
        print(x)
```

```
1.0
```

This pattern can be very convenient, but in large programs that include many modules, it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would eliminate potentially confusing problems with namespace collisions.

As a third alternative, we can choose to import only a few selected symbols from a module by explicitly listing which ones we want to import instead of using the wildcard character `*`:

```
In [9]: from math import cos, pi
        x = cos(2 * pi)
        print(x)
```

```
1.0
```

2.3.2 Looking at what a module contains, and its documentation

Once a module is imported, we can list the symbols it provides using the `dir` function:

```
In [10]: import math
         for name in dir(math):
             print(name)
```

```
__doc__
__file__
__loader__
__name__
__package__
__spec__
acos
acosh
asin
asinh
atan
atan2
atanh
ceil
copysign
cos
cosh
```

```
degrees
e
erf
erfc
exp
expm1
fabs
factorial
floor
fmod
frexp
fsum
gamma
hypot
isfinite
isinf
isnan
ldexp
lgamma
log
log10
log1p
log2
modf
pi
pow
radians
sin
sinh
sqrt
tan
tanh
trunc
```

And using the function **help**, we can get a description of each function (almost; not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

```
In [11]: help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.
```

```
In [12]: log(10)
```

```
Out[12]: 2.302585092994046
```

```
In [13]: log(10, 2)
```

```
Out[13]: 3.3219280948873626
```

We can also use the **help** function directly on modules: Try

```
help(math)
```

Some very useful modules from the Python standard library are `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

A complete list of standard modules for Python 2 and Python 3 are available at http://docs.python.org/2/library/ and http://docs.python.org/3/library/, respectively.

2.4 Variables and types

2.4.1 Symbol names

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some special characters such as _. Normal variable names must start with a letter.

By convention, variable names start with a lowercase letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

2.4.2 Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

```
In [14]: # variable assignments
         x = 1.0
         my_variable = 12.2
```

Although not explicitly specified, a variable does have a type associated with it. The type is derived from the value that was assigned to it.

```
In [15]: type(x)

Out[15]: float
```

If we assign a new value to a variable, its type can change.

```
In [16]: x = 1

In [17]: type(x)

Out[17]: int
```

If we try to use a variable that has not yet been defined, we get an `NameError`:

```
In [18]: try:
             print(y)
         except NameError as e:
             print(repr(e))

NameError("name 'y' is not defined",)
```

2.4.3 Fundamental types

```
In [19]: # integers
         x = 1
         type(x)
```

```
Out[19]: int
```

```
In [20]: # float
         x = 1.0
         type(x)
```

```
Out[20]: float
```

```
In [21]: # boolean
         b1 = True
         b2 = False

         type(b1)
```

```
Out[21]: bool
```

```
In [22]: # complex numbers: note the use of `j` to specify the imaginary part
         x = 1.0 - 1.0j
         type(x)
```

```
Out[22]: complex
```

```
In [23]: print(x)
```

```
(1-1j)
```

```
In [24]: print(x.real, x.imag)
```

```
1.0 -1.0
```

2.4.4 Type utility functions

The module `types` contains a number of type name definitions that can be used to test if variables are of certain types:

```
In [25]: import types
         # print all types defined in the `types` module
         for name in dir(types):
             print(name)
```

```
BuiltinFunctionType
BuiltinMethodType
CodeType
DynamicClassAttribute
FrameType
FunctionType
GeneratorType
GetSetDescriptorType
LambdaType
MappingProxyType
MemberDescriptorType
MethodType
```

```
ModuleType
SimpleNamespace
TracebackType
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
__package__
__spec__
_calculate_meta
new_class
prepare_class
```

```
In [26]: x = 1.0
         # check if the variable x is a float
         type(x) is float

Out[26]: True

In [27]: # check if the variable x is an int
         type(x) is int

Out[27]: False
```

We can also use the `isinstance` method for testing types of variables:

```
In [28]: isinstance(x, float)

Out[28]: True
```

2.4.5 Type casting

```
In [29]: x = 1.5
         print(x, type(x))

1.5 <class 'float'>

In [30]: x = int(x)
         print(x, type(x))

1 <class 'int'>

In [31]: z = complex(x)
         print(z, type(z))

(1+0j) <class 'complex'>

In [32]: try:
             x = float(z)
         except TypeError as e:
             print(repr(e))

TypeError("can't convert complex to float",)
```

Complex variables cannot be cast to floats or integers. We need to use `z.real` or `z.imag` to extract the part of the complex number we want:

```
In [33]: y = bool(z.real)
         print(z.real, " -> ", y, type(y))
         y = bool(z.imag)
         print(z.imag, " -> ", y, type(y))

1.0  ->  True <class 'bool'>
0.0  ->  False <class 'bool'>
```

2.5 Operators and comparisons

Most operators and comparisons in Python work as one would expect:

- Arithmetic operators `+`, `-`, `*`, `/`, `//` (integer division), '**' power

```
In [34]: 1 + 2, 1 - 2, 1 * 2, 1 / 2

Out[34]: (3, -1, 2, 0.5)

In [35]: 1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0

Out[35]: (3.0, -1.0, 2.0, 0.5)

In [36]: # Integer division of float numbers
         3.0 // 2.0

Out[36]: 1.0

In [37]: # Note! The power operators in python isn't ^, but **
         2 ** 2

Out[37]: 4
```

Note: The `/` operator always performs a floating point division in Python 3.x. This is not true in Python 2.x, where the result of `/` is always an integer if the operands are integers. To be more specific, `1/2 = 0.5` (`float`) in Python 3.x, and `1/2 = 0` (`int`) in Python 2.x (but `1.0/2 = 0.5` in Python 2.x).

- The boolean operators are spelled out as the words `and`, `not`, `or`.

```
In [38]: True and False

Out[38]: False

In [39]: not False

Out[39]: True

In [40]: True or False

Out[40]: True
```

- Comparison operators >, <, >= (greater or equal), <= (less or equal), `==` equality, `is` identical.

```
In [41]: 2 > 1, 2 < 1

Out[41]: (True, False)

In [42]: 2 > 2, 2 < 2

Out[42]: (False, False)
```

```
In [43]: 2 >= 2, 2 <= 2
```

```
Out[43]: (True, True)
```

```
In [44]: # equality
         [1,2] == [1,2]
```

```
Out[44]: True
```

```
In [45]: # objects identical?
         l1 = l2 = [1,2]
         l1 is l2
```

```
Out[45]: True
```

2.6 Compound types: Strings, List and dictionaries

2.6.1 Strings

Strings are the variable type that is used for storing text messages.

```
In [46]: s = "Hello world"
         type(s)
```

```
Out[46]: str
```

```
In [47]: # length of the string: the number of characters
         len(s)
```

```
Out[47]: 11
```

```
In [48]: # replace a substring in a string with somethign else
         s2 = s.replace("world", "test")
         print(s2)
```

```
Hello test
```

We can index a character in a string using `[]`:

```
In [49]: s[0]
```

```
Out[49]: 'H'
```

Heads up, MATLAB users: Indexing start at 0!

We can extract a part of a string using the syntax `[start:stop]`, which extracts characters between index `start` and `stop` -1 (the character at index `stop` is not included):

```
In [50]: s[0:5]
```

```
Out[50]: 'Hello'
```

```
In [51]: s[4:5]
```

```
Out[51]: 'o'
```

If we omit either (or both) of `start` or `stop` from `[start:stop]`, the default is the beginning and the end of the string, respectively:

```
In [52]: s[:5]
```

```
Out[52]: 'Hello'
```

```
In [53]: s[6:]
```

```
Out[53]: 'world'
```

```
In [54]: s[:]
```

```
Out[54]: 'Hello world'
```

We can also define the step size using the syntax `[start:end:step]` (the default value for `step` is 1, as we saw above):

```
In [55]: s[::1]
```

```
Out[55]: 'Hello world'
```

```
In [56]: s[::2]
```

```
Out[56]: 'Hlowrd'
```

This technique is called *slicing*. Read more about the syntax here: http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice

Python has a very rich set of functions for text processing. See for example http://docs.python.org/2/library/string.html for more information.

String formatting examples

```
In [57]: print("str1", "str2", "str3")  # The print statement concatenates strings with a space
```

```
str1 str2 str3
```

```
In [58]: print("str1", 1.0, False, -1j)  # The print statement converts all arguments to strings
```

```
str1 1.0 False (-0-1j)
```

```
In [59]: print("str1" + "str2" + "str3") # strings added with + are concatenated without space
```

```
str1str2str3
```

```
In [60]: print("value = %f" % 1.0)       # we can use C-style string formatting
```

```
value = 1.000000
```

```
In [61]: # this formatting creates a string
         s2 = "value1 = %.2f. value2 = %d" % (3.1415, 1.5)

         print(s2)
```

```
value1 = 3.14. value2 = 1
```

```
In [62]: # alternative, more intuitive way of formatting a string
         s3 = 'value1 = {0}, value2 = {1}'.format(3.1415, 1.5)
         print(s3)
```

```
value1 = 3.1415, value2 = 1.5
```

2.6.2 List

Lists are very similar to strings, except that each element can be of any type.

The syntax for creating lists in Python is `[...]`:

```
In [63]: l = [1,2,3,4]
         print(type(l))
         print(l)
```

```
<class 'list'>
[1, 2, 3, 4]
```

We can use the same slicing techniques to manipulate lists as we could use on strings:

```
In [64]: print(l)
         print(l[1:3])
         print(l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Heads up, MATLAB users: Indexing starts at 0!

```
In [65]: l[0]
```

```
Out[65]: 1
```

Elements in a list do not all have to be of the same type:

```
In [66]: l = [1, 'a', 1.0, 1-1j]

         print(l)
```

```
[1, 'a', 1.0, (1-1j)]
```

Python lists do not have to be homogeneous and may be arbitrarily nested:

```
In [67]: nested_list = [1, [2, [3, [4, [5]]]]]
         nested_list
```

```
Out[67]: [1, [2, [3, [4, [5]]]]]
```

Lists play a very important role in Python. For example, they are used in loops and other flow control structures (discussed below). There are a number of convenient functions for generating lists of various types; for example, the `range` function:

```
In [68]: start = 10
         stop = 30
         step = 2
         range(start, stop, step)
```

```
Out[68]: range(10, 30, 2)
```

```
In [69]: # in Python 3 range generates an iterator, which can be converted to a list using 'list(...)'.
         # It has no effect in Python 2
         list(range(start, stop, step))
```

```
Out[69]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

```
In [70]: list(range(-10, 10))
```

```
Out[70]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [71]: s
```

```
Out[71]: 'Hello world'
```

```
In [72]: # convert a string to a list by type casting:
         s2 = list(s)
         s2
```

```
Out[72]: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

```
In [73]: # sorting lists
         s2.sort()
         print(s2)
```

```
[' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

Adding, inserting, modifying, and removing elements from lists

```
In [74]: # create a new empty list
         l = []
         # add an elements using `append`
         l.append("A")
         l.append("d")
         l.append("d")
         print(l)
```

```
['A', 'd', 'd']
```

We can modify lists by assigning new values to elements in the list. In technical jargon, lists are *mutable*.

```
In [75]: l[1] = "p"
         l[2] = "p"
         print(l)
```

```
['A', 'p', 'p']
```

```
In [76]: l[1:3] = ["d", "d"]
         print(l)
```

```
['A', 'd', 'd']
```

Insert an element at an specific index using `insert`.

```
In [77]: l.insert(0, "i")
         l.insert(1, "n")
         l.insert(2, "s")
         l.insert(3, "e")
         l.insert(4, "r")
         l.insert(5, "t")
         print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Remove first element with specific value using 'remove'.

```
In [78]: l.remove("A")
         print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

Remove an element at a specific location using `del`.

```
In [79]: del l[7]
         del l[6]
         print(l)
```

```
['i', 'n', 's', 'e', 'r', 't']
```

See `help(list)` for more details, or read the online documentation.

2.6.3 Tuples

Tuples are like lists, except that they cannot be modified once created; that is, they are *immutable*.

In Python, tuples are created using the syntax `(..., ..., ...)`, or even `..., ...`:

```
In [80]: point = (10, 20)
         print(point, type(point))
```

```
(10, 20) <class 'tuple'>
```

```
In [81]: point = 10, 20
         print(point, type(point))
```

```
(10, 20) <class 'tuple'>
```

We can unpack a tuple by assigning it to a comma-separated list of variables:

```
In [82]: x, y = point
         print("x =", x)
         print("y =", y)
```

```
x = 10
y = 20
```

If we try to assign a new value to an element in a tuple we get an error:

```
In [83]: try:
             point[0] = 20
         except TypeError as e:
             print(repr(e))
```

```
TypeError("'tuple' object does not support item assignment",)
```

2.6.4 Dictionaries

Dictionaries are also like lists, except that each element is a key-value pair. The syntax for dictionaries is `{key1 : value1, ...}`:

```
In [84]: params = {"parameter1" : 1.0,
                   "parameter2" : 2.0,
                   "parameter3" : 3.0,}

         print(type(params))
         print(params)
```

```
<class 'dict'>
{'parameter3': 3.0, 'parameter2': 2.0, 'parameter1': 1.0}
```

```
In [85]: print("parameter1 = " + str(params["parameter1"]))
         print("parameter2 = " + str(params["parameter2"]))
         print("parameter3 = " + str(params["parameter3"]))
```

```
parameter1 = 1.0
parameter2 = 2.0
parameter3 = 3.0
```

```
In [86]: params["parameter1"] = "A"
         params["parameter2"] = "B"

         # add a new entry
         params["parameter4"] = "D"

         print("parameter1 = " + str(params["parameter1"]))
         print("parameter2 = " + str(params["parameter2"]))
         print("parameter3 = " + str(params["parameter3"]))
         print("parameter4 = " + str(params["parameter4"]))
```

```
parameter1 = A
parameter2 = B
parameter3 = 3.0
parameter4 = D
```

2.7 Control Flow

2.7.1 Conditional statements: if, elif, else

The Python syntax for conditional execution of code uses the keywords `if`, `elif` (else if), `else`:

```
In [87]: statement1 = False
         statement2 = False

         if statement1:
             print("statement1 is True")
         elif statement2:
             print("statement2 is True")
         else:
             print("statement1 and statement2 are False")
```

```
statement1 and statement2 are False
```

For the first time, here we encounter a peculiar and unusual aspect of the Python programming language: Program blocks are defined by their indentation level.

Compare to the equivalent C code:

```
if (statement1)
{
    printf("statement1 is True\n");
}
else if (statement2)
{
```

```
    printf("statement2 is True\n");
}
else
{
    printf("statement1 and statement2 are False\n");
}
```

In C, blocks are defined by enclosing them in curly braces { and }. The level of indentation (spaces or a tab before the code statements) does not have an effect; it's just optional formatting.

But in Python, the extent of a code block is defined by its indentation level — denoted with a tab or 4-5 spaces. This means that we have to be careful to indent our code correctly, or else we will get syntax errors.

Examples:

```
In [88]: statement1 = statement2 = True

         if statement1:
             if statement2:
                 print("both statement1 and statement2 are True")

both statement1 and statement2 are True
```

```
# Bad indentation!
if statement1:
    if statement2: # next line is not properly indented
    print("both statement1 and statement2 are True")

  File "<ipython-input-17-d663108bdb86>", line 4
    print("both statement1 and statement2 are True")
        ^
IndentationError: expected an indented block
```

```
In [89]: statement1 = False

         if statement1:
             print("printed if statement1 is True")

             print("still inside the if block")

In [90]: if statement1:
             print("printed if statement1 is True")

         print("now outside the if block")

now outside the if block
```

2.8 Loops

In Python, loops can be programmed in a number of different ways. The most common is the `for` loop, which is used together with iterable objects, such as lists. The basic syntax is:

2.8.1 `for` loops:

```
In [91]: for x in [1,2,3]:
             print(x)
```

```
1
2
3
```

The `for` loop iterates over the elements of the supplied list, and executes the containing block once for each element. Any kind of list can be used in the `for` loop. For example:

```
In [92]: for x in range(4): # by default range start at 0
             print(x)
```

```
0
1
2
3
```

Note: `range(4)` does not include 4 !

```
In [93]: for x in range(-3,3):
             print(x)
```

```
-3
-2
-1
0
1
2
```

```
In [94]: for word in ["scientific", "computing", "with", "python"]:
             print(word)
```

```
scientific
computing
with
python
```

To iterate over key-value pairs of a dictionary:

```
In [95]: for key, value in params.items():
             print(key + " = " + str(value))
```

```
parameter3 = 3.0
parameter4 = D
parameter2 = B
parameter1 = A
```

Sometimes it is useful to have access to the indices of the values when iterating over a list. We can use the `enumerate` function for this:

```
In [96]: for idx, x in enumerate(range(-3,3)):
             print(idx, x)
```

```
0 -3
1 -2
2 -1
3 0
4 1
5 2
```

2.8.2 Using Lists: Creating lists using `for` loops:

A convenient and compact way to initialize lists:

```
In [97]: l1 = [x**2 for x in range(0,5)]
         print(l1)
```

```
[0, 1, 4, 9, 16]
```

2.8.3 `while` loops:

```
In [98]: i = 0
         while i < 5:
             print(i)
             i = i + 1
         print("done")
```

```
0
1
2
3
4
done
```

Note that the `print("done")` statement is not part of the `while` loop body because of the difference in indentation.

2.9 Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

```
In [99]: def func0():
             print("test")
```

```
In [100]: func0()
```

```
test
```

Optional, but highly recommended: Define a so-called "docstring" — a description of the function's purpose and behavior. The docstring should follow directly after the function definition, before the code in the function body.

```
In [101]: def func1(s):
              """
              Print a string 's' and tell how many characters it has
              """
              print(s + " has " + str(len(s)) + " characters")
```

```
In [102]: help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Print a string 's' and tell how many characters it has
```

```
In [103]: func1("test")
```

```
test has 4 characters
```

Functions that return a value use the `return` keyword:

```
In [104]: def square(x):
              """
              Return the square of x.
              """
              return x ** 2
```

```
In [105]: square(4)
```

```
Out[105]: 16
```

We can return multiple values from a function using tuples (see above):

```
In [106]: def powers(x):
              """
              Return a few powers of x.
              """
              return x ** 2, x ** 3, x ** 4
```

```
In [107]: powers(3)
```

```
Out[107]: (9, 27, 81)
```

```
In [108]: x2, x3, x4 = powers(3)
          print(x3)
```

```
27
```

2.9.1 Default argument and keyword arguments

when defining a function, we can give default values to the arguments the function takes:

```
In [109]: def myfunc(x, p=2, debug=False):
              if debug:
                  print("evaluating myfunc for x = " + str(x) + " using exponent p = " + str(p))
              return x**p
```

If we don't provide a value of the `debug` argument when calling the function `myfunc`, it defaults to the value provided in the function definition:

```
In [110]: myfunc(5)
```

```
Out[110]: 25
```

```
In [111]: myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

```
Out[111]: 25
```

If we explicitly list the names of the arguments in the function calls, they do not need to come in the same order as in the function definition. This is called *keyword* arguments, and is often very useful in functions that take a lot of optional arguments.

```
In [112]: myfunc(p=3, debug=True, x=7)
```

```
evaluating myfunc for x = 7 using exponent p = 3
```

```
Out[112]: 343
```

2.9.2 Unnamed functions (lambda function)

In Python we can also create unnamed functions using the `lambda` keyword:

```
In [113]: f1 = lambda x: x**2

          # is equivalent to
          def f2(x):
              return x**2
```

```
In [114]: f1(2), f2(2)
```

```
Out[114]: (4, 4)
```

This technique is useful, for example, when we want to pass a simple function as an argument to another function, like this:

```
In [115]: # map is a built-in python function
          map(lambda x: x**2, range(-3,4))
```

```
Out[115]: <map at 0x106b57128>
```

```
In [116]: # in python 3 we can use `list(...)` to convert the iterator to an explicit list
          list(map(lambda x: x**2, range(-3,4)))
```

```
Out[116]: [9, 4, 1, 0, 1, 4, 9]
```

2.10 Classes

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.

In Python, a class can contain *attributes* (variables) and *methods* (functions).

A class is defined almost like a function, but using the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).

- Each class method should have an argument `self` as its first argument. This object is a self-reference.
- Some class method names have special meaning; for example:
  - `__init__`: The name of the method that is invoked when the object is first created.
  - `__str__` : A method that is invoked when a simple string representation of the class is needed, as for example when printed.
  - There are many more; see http://docs.python.org/2/reference/datamodel.html#special-method-names

```
In [117]: class Point:
              """
              Simple class for representing a point in a Cartesian coordinate system.
              """
              def __init__(self, x, y):
                  """
                  Create a new Point at x, y.
                  """
                  self.x = x
                  self.y = y
              def translate(self, dx, dy):
```

```
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy
    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))
```

To create a new instance of a class:

```
In [118]: p1 = Point(0, 0) # this will invoke the __init__ method in the Point class

          print(p1)         # this will invoke the __str__ method
```

```
Point at [0.000000, 0.000000]
```

To invoke a class method in the class instance `p`:

```
In [119]: p2 = Point(1, 1)
          p1.translate(0.25, 1.5)
          print(p1)
          print(p2)
```

```
Point at [0.250000, 1.500000]
Point at [1.000000, 1.000000]
```

Note that calling class methods can modifiy the state of that particular class instance, but does not affect other class instances or any global variables.

That is one of the nice things about object-oriented design: code such as functions and related variables are grouped in separate and independent entities.

2.11 Modules

One of the most important concepts in good programming is to reuse code and avoid repetitions.

The idea is to write functions and classes with a well-defined purpose and scope, and reuse these instead of repeating similar code in different parts of a program (modular programming). This improves readability and maintainability of your programs. In practice, your programs have fewer bugs, and are easier to extend and debug/troubleshoot.

Python supports modular programming at different levels. Functions and classes are examples of tools for low-level modular programming. Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module. A Python module is defined in a Python file (with file-ending `.py`), and can be made accessible to other Python modules and programs using the `import` statement.

The following example, `mymodule.py`, contains simple example implementations of a variable, function and a class:

```
In [120]: %%file mymodule.py
          """
          Example of a Python module. Contains a variable called my_variable,
          a function called my_function, and a class called MyClass.
          """

          my_variable = 0

          def my_function():
```

```
        """
        Example function
        """
        return my_variable

    class MyClass:
        """
        Example class.
        """

        def __init__(self):
            self.variable = my_variable
        def set_variable(self, new_value):
            """
            Set self.variable to a new value
            """
            self.variable = new_value
        def get_variable(self):
            return self.variable
```

```
Writing mymodule.py
```

We can import the module `mymodule` into our Python program using `import`:

```
In [121]: import mymodule
```

Use `help(module)` to get a summary of what the module provides:

```
In [122]: help(mymodule)
```

```
Help on module mymodule:

NAME
    mymodule

DESCRIPTION
    Example of a Python module. Contains a variable called my_variable,
    a function called my_function, and a class called MyClass.

CLASSES
    builtins.object
        MyClass

    class MyClass(builtins.object)
     |  Example class.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |
     |  get_variable(self)
     |
     |  set_variable(self, new_value)
     |      Set self.variable to a new value
     |
```

```
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FUNCTIONS
    my_function()
        Example function

DATA
    my_variable = 0

FILE
    /Users/dmertz/Drive/Modules/scientific-python-lectures/mymodule.py
```

```
In [123]: mymodule.my_variable

Out[123]: 0

In [124]: mymodule.my_function()

Out[124]: 0

In [125]: my_class = mymodule.MyClass()
          my_class.set_variable(10)
          my_class.get_variable()

Out[125]: 10
```

If we make changes to the code in `mymodule.py`, we need to reload it using `reload`:

```
In [126]: from imp import reload
          reload(mymodule)

Out[126]: <module 'mymodule' from '/Users/dmertz/Drive/Modules/scientific-python-lectures/mymodule.py'>
```

2.12 Exceptions

In Python, errors are managed with a special language construct called "Exceptions". When errors occur, exceptions can be raised, which interrupts the normal program flow and falls back to the point in the code where the closest try-except statement is defined.

To generate an exception, we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseException` or a class derived from it.

```
In [127]: try:
              raise Exception("description of the error")
          except Exception as e:
              print(repr(e))

Exception('description of the error',)
```

A typical use of exceptions is to abort functions when some error condition occurs. For example:

```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # rest of the code goes here
```

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generated an error
```

For example:

```
In [128]: try:
              print("test")
              # generate an error: the variable test is not defined
              print(test)
          except:
              print("Caught an exception")
```

```
test
Caught an exception
```

To get information about the error, we can access the `Exception` class instance that describes the exception by using, for example:

```
except Exception as e:
```

```
In [129]: try:
              print("test")
              # generate an error: the variable test is not defined
              print(test)
          except Exception as e:
              print("Caught an exception:" + str(e))
```

```
test
Caught an exception:name 'test' is not defined
```

2.13 Further reading

- http://www.python.org - The official web page of the Python programming language.
- http://www.python.org/dev/peps/pep-0008 - Style guide for Python programming. Highly recommended.
- http://www.greenteapress.com/thinkpython/ - A free book on Python programming.
- Python Essential Reference - A good reference book on Python programming.

Chapter 3
Numpy - multidimensional data arrays

In [1]:
```
# what is this line all about?!? Answer in lecture 4
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

3.1 Introduction

The `numpy` package (module) is used in almost all numerical computation using Python. Numpy provides high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran, so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

To use `numpy`, you need to import the module. For example:

In [2]:
```
from numpy import *
```

In the `numpy` package, the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

3.2 Creating `numpy` arrays

There are a number of ways to initialize new numpy arrays, including:

- from a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

3.2.1 From lists

To create new vector and matrix arrays from Python lists, we can use the `numpy.array` function.

In [3]:
```
# a vector: the argument to the array function is a Python list
v = array([1,2,3,4])
v
```

```
Out[3]: array([1, 2, 3, 4])

In [4]: # a matrix: the argument to the array function is a nested Python list
        M = array([[1, 2], [3, 4]])
        M

Out[4]: array([[1, 2],
               [3, 4]])
```

The v and M objects are both of the type ndarray that the numpy module provides.

```
In [5]: type(v), type(M)

Out[5]: (numpy.ndarray, numpy.ndarray)
```

The difference between the v and M arrays is only their shapes. We can get information about the shape of an array by using the ndarray.shape property.

```
In [6]: v.shape

Out[6]: (4,)

In [7]: M.shape

Out[7]: (2, 2)
```

The number of elements in the array is available through the ndarray.size property:

```
In [8]: M.size

Out[8]: 4
```

Or we could use the function numpy.shape and numpy.size

```
In [9]: shape(M)

Out[9]: (2, 2)

In [10]: size(M)

Out[10]: 4
```

So far the numpy.ndarray looks very much like a Python list (or nested list). Why not simply use Python lists for computations, instead of creating a new array type?

There are several reasons:

- Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications. Implementing such functions for Python lists would not be very efficient because of the dynamic typing.
- Numpy arrays are **statically typed** and **homogeneous**. The type of the elements is determined when the array is created.
- Numpy arrays are memory efficient.
- Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of numpy arrays can be implemented in a compiled language (C and Fortran is used).

Using the dtype (data type) property of an ndarray, we can see what type the data of an array has:

```
In [11]: M.dtype
```

```
Out[11]: dtype('int64')
```

We get an error if we try to assign a value of the wrong type to an element in a numpy array:

```
In [12]: try:
             M[0,0] = "hello"
         except ValueError as e:
             print(repr(e))

ValueError("invalid literal for int() with base 10: 'hello'",)
```

If we want, we can explicitly define the type of the array data when we create it, using the dtype keyword argument:

```
In [13]: M = array([[1, 2], [3, 4]], dtype=complex)
         M

Out[13]: array([[ 1.+0.j,  2.+0.j],
                [ 3.+0.j,  4.+0.j]])
```

Common data types that can be used with dtype are: int, float, complex, bool, object, etc.

We can also explicitly define the bit size of the data types, for example: int64, int16, float128, complex128.

3.2.2 Using array-generating functions

For larger arrays, it is impractical to initialize the data manually, using explicit Python lists. Instead we can use one of the many functions in numpy that generate arrays of different forms. Some of the more common are:

arange

```
In [14]: # create a range
         x = arange(0, 10, 1) # arguments: start, stop, step
         x

Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: x = arange(-1, 1, 0.1)
         x

Out[15]: array([ -1.00000000e+00,  -9.00000000e-01,  -8.00000000e-01,
                 -7.00000000e-01,  -6.00000000e-01,  -5.00000000e-01,
                 -4.00000000e-01,  -3.00000000e-01,  -2.00000000e-01,
                 -1.00000000e-01,  -2.22044605e-16,   1.00000000e-01,
                  2.00000000e-01,   3.00000000e-01,   4.00000000e-01,
                  5.00000000e-01,   6.00000000e-01,   7.00000000e-01,
                  8.00000000e-01,   9.00000000e-01])
```

linspace and logspace

```
In [16]: # using linspace, both end points ARE included
         linspace(0, 10, 25)

Out[16]: array([  0.        ,   0.41666667,   0.83333333,   1.25      ,
                  1.66666667,   2.08333333,   2.5       ,   2.91666667,
                  3.33333333,   3.75      ,   4.16666667,   4.58333333,
                  5.        ,   5.41666667,   5.83333333,   6.25      ,
                  6.66666667,   7.08333333,   7.5       ,   7.91666667,
                  8.33333333,   8.75      ,   9.16666667,   9.58333333,  10.        ])
```

```
In [17]: logspace(0, 10, 10, base=math.e)

Out[17]: array([  1.00000000e+00,   3.03773178e+00,   9.22781435e+00,
                  2.80316249e+01,   8.51525577e+01,   2.58670631e+02,
                  7.85771994e+02,   2.38696456e+03,   7.25095809e+03,
                  2.20264658e+04])
```

mgrid

```
In [18]: x, y = mgrid[0:5, 0:5] # similar to meshgrid in MATLAB

In [19]: x

Out[19]: array([[0, 0, 0, 0, 0],
                [1, 1, 1, 1, 1],
                [2, 2, 2, 2, 2],
                [3, 3, 3, 3, 3],
                [4, 4, 4, 4, 4]])

In [20]: y

Out[20]: array([[0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4],
                [0, 1, 2, 3, 4]])
```

random data

```
In [21]: from numpy import random

In [22]: # uniform random numbers in [0,1]
         random.rand(5,5)

Out[22]: array([[ 0.03636076,  0.44514759,  0.66518293,  0.75835268,  0.59207674],
                [ 0.28141048,  0.20111081,  0.00572901,  0.98049814,  0.59304213],
                [ 0.34027591,  0.62526429,  0.4914911 ,  0.18098194,  0.84354665],
                [ 0.29046621,  0.07903848,  0.18036557,  0.69015598,  0.66410549],
                [ 0.72311202,  0.67740137,  0.12136326,  0.93869056,  0.91733199]])

In [23]: # standard normal distributed random numbers
         random.randn(5,5)

Out[23]: array([[ 1.1801927 , -1.25957295, -0.371572  ,  0.45980427, -1.87468801],
                [-0.59000628,  0.81853806,  0.70351193, -0.65519014,  0.40280782],
                [-0.3760094 , -0.22502101, -0.90825071, -0.12257355, -0.99241568],
                [-0.51218034, -0.60630681,  1.72870919,  0.87370964, -1.63287707],
                [-0.64165892,  0.58476853,  0.35584857,  0.50899941, -0.63282126]])
```

diag

```
In [24]: # a diagonal matrix
         diag([1,2,3])

Out[24]: array([[1, 0, 0],
                [0, 2, 0],
                [0, 0, 3]])
```

```
In [25]: # diagonal with offset from the main diagonal
         diag([1,2,3], k=1)

Out[25]: array([[0, 1, 0, 0],
                [0, 0, 2, 0],
                [0, 0, 0, 3],
                [0, 0, 0, 0]])
```

zeros and ones

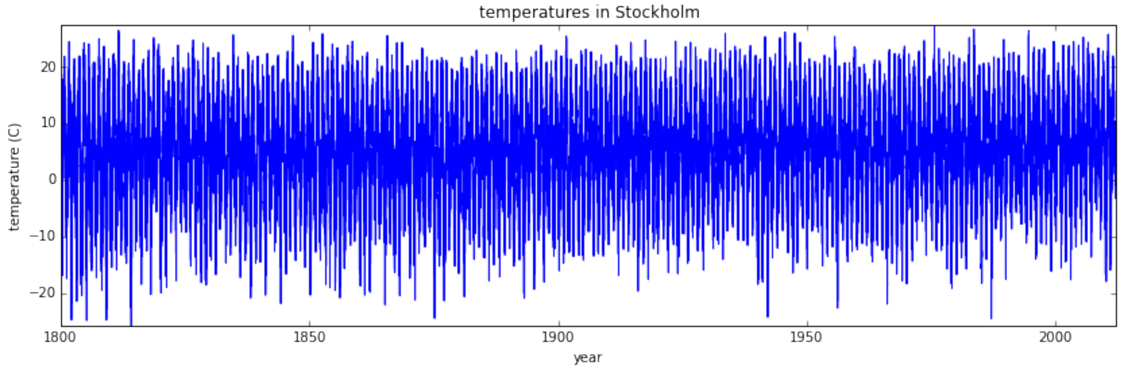
```
In [26]: zeros((3,3))

Out[26]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])

In [27]: ones((3,3))

Out[27]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])
```

3.3 File I/O

3.3.1 Comma-separated values (CSV)

A very common file format for data files is comma-separated values (CSV), or related formats such as TSV (tab-separated values). To read data from such files into Numpy arrays, we can use the `numpy.genfromtxt` function. For example:

```
In [28]: !head stockholm_td_adj.dat

1800  1  1    -6.1    -6.1    -6.1 1
1800  1  2   -15.4   -15.4   -15.4 1
1800  1  3   -15.0   -15.0   -15.0 1
1800  1  4   -19.3   -19.3   -19.3 1
1800  1  5   -16.8   -16.8   -16.8 1
1800  1  6   -11.4   -11.4   -11.4 1
1800  1  7    -7.6    -7.6    -7.6 1
1800  1  8    -7.1    -7.1    -7.1 1
1800  1  9   -10.1   -10.1   -10.1 1
1800  1 10    -9.5    -9.5    -9.5 1

In [29]: data = genfromtxt('stockholm_td_adj.dat')

In [30]: data.shape

Out[30]: (77431, 7)

In [31]: fig, ax = subplots(figsize=(14,4))
         ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,5])
         ax.axis('tight')
         ax.set_title('temperatures in Stockholm')
         ax.set_xlabel('year')
         ax.set_ylabel('temperature (C)');
```



Using `numpy.savetxt`, we can store a Numpy array to a file in CSV format:

```
In [32]: M = rand(3,3)

         M

Out[32]: array([[ 0.98166471,  0.95956617,  0.29935288],
                [ 0.84995869,  0.98118065,  0.60505178],
                [ 0.55596909,  0.89157525,  0.42748148]])

In [33]: savetxt("random-matrix.csv", M)

In [34]: !cat random-matrix.csv

9.816647072251577510e-01 9.595661701598807714e-01 2.993528772395587234e-01
8.499586857036480758e-01 9.811806531134164011e-01 6.050517821305401167e-01
5.559690906506703501e-01 8.915752478165345218e-01 4.274814801450405044e-01

In [35]: savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format

         !cat random-matrix.csv

0.98166 0.95957 0.29935
0.84996 0.98118 0.60505
0.55597 0.89158 0.42748
```

3.3.2 Numpy's native file format

Useful when storing and reading back numpy array data. Use the functions `numpy.save` and `numpy.load`:

```
In [36]: save("random-matrix.npy", M)

         !file random-matrix.npy

random-matrix.npy: data

In [37]: load("random-matrix.npy")

Out[37]: array([[ 0.98166471,  0.95956617,  0.29935288],
                [ 0.84995869,  0.98118065,  0.60505178],
                [ 0.55596909,  0.89157525,  0.42748148]])
```

3.4 More properties of the numpy arrays

```
In [38]: M.itemsize # bytes per element
```

```
Out[38]: 8
```

```
In [39]: M.nbytes # number of bytes
```

```
Out[39]: 72
```

```
In [40]: M.ndim # number of dimensions
```

```
Out[40]: 2
```

3.5 Manipulating arrays

3.5.1 Indexing

We can index elements in an array using square brackets and indices:

```
In [41]: # v is a vector, and has only one dimension, taking one index
         v[0]
```

```
Out[41]: 1
```

```
In [42]: # M is a matrix, or a 2-dimensional array, taking two indices
         M[1,1]
```

```
Out[42]: 0.9811806531134164
```

If we omit an index of a multidimensional array, it returns the whole row (or, in general, a N-1 dimensional array)

```
In [43]: M
```

```
Out[43]: array([[ 0.98166471,  0.95956617,  0.29935288],
                [ 0.84995869,  0.98118065,  0.60505178],
                [ 0.55596909,  0.89157525,  0.42748148]])
```

```
In [44]: M[1]
```

```
Out[44]: array([ 0.84995869,  0.98118065,  0.60505178])
```

The same thing can be achieved with using `:` instead of an index:

```
In [45]: M[1,:] # row 1
```

```
Out[45]: array([ 0.84995869,  0.98118065,  0.60505178])
```

```
In [46]: M[:,1] # column 1
```

```
Out[46]: array([ 0.95956617,  0.98118065,  0.89157525])
```

We can assign new values to elements in an array using indexing:

```
In [47]: M[0,0] = 1
```

```
In [48]: M
```

```
Out[48]: array([[ 1.        ,  0.95956617,  0.29935288],
                [ 0.84995869,  0.98118065,  0.60505178],
                [ 0.55596909,  0.89157525,  0.42748148]])
```

```
In [49]: # also works for rows and columns
         M[1,:] = 0
         M[:,2] = -1
```

```
In [50]: M
```

```
Out[50]: array([[ 1.        ,  0.95956617, -1.        ],
                [ 0.        ,  0.        , -1.        ],
                [ 0.55596909,  0.89157525, -1.        ]])
```

3.5.2 Index slicing

Index slicing is the technical name for the syntax `M[lower:upper:step]` to extract part of an array:

```
In [51]: A = array([1,2,3,4,5])
         A
```

```
Out[51]: array([1, 2, 3, 4, 5])
```

```
In [52]: A[1:3]
```

```
Out[52]: array([2, 3])
```

Array slices are *mutable*: if they are assigned a new value, the original array from which the slice was extracted is modified:

```
In [53]: A[1:3] = [-2,-3]

         A
```

```
Out[53]: array([ 1, -2, -3,  4,  5])
```

We can omit any of the three parameters in `M[lower:upper:step]`:

```
In [54]: A[::] # lower, upper, step all take the default values
```

```
Out[54]: array([ 1, -2, -3,  4,  5])
```

```
In [55]: A[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

```
Out[55]: array([ 1, -3,  5])
```

```
In [56]: A[:3] # first three elements
```

```
Out[56]: array([ 1, -2, -3])
```

```
In [57]: A[3:] # elements from index 3
```

```
Out[57]: array([4, 5])
```

Negative indices count from the end of the array (positive index from the beginning):

```
In [58]: A = array([1,2,3,4,5])
```

```
In [59]: A[-1] # the last element in the array
```

```
Out[59]: 5
```

```
In [60]: A[-3:] # the last three elements
```

```
Out[60]: array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

```
In [61]: A = array([[n+m*10 for n in range(5)] for m in range(5)])

         A
```

```
Out[61]: array([[ 0,  1,  2,  3,  4],
                [10, 11, 12, 13, 14],
                [20, 21, 22, 23, 24],
                [30, 31, 32, 33, 34],
                [40, 41, 42, 43, 44]])
```

```
In [62]: # a block from the original array
         A[1:4, 1:4]
```

```
Out[62]: array([[11, 12, 13],
                [21, 22, 23],
                [31, 32, 33]])
```

```
In [63]: # strides
         A[::2, ::2]
```

```
Out[63]: array([[ 0,  2,  4],
                [20, 22, 24],
                [40, 42, 44]])
```

3.5.3 Fancy indexing

Fancy indexing is the name for when an array or list is used in place of an index:

```
In [64]: row_indices = [1, 2, 3]
         A[row_indices]
```

```
Out[64]: array([[10, 11, 12, 13, 14],
                [20, 21, 22, 23, 24],
                [30, 31, 32, 33, 34]])
```

```
In [65]: col_indices = [1, 2, -1] # remember, index -1 means the last element
         A[row_indices, col_indices]
```

```
Out[65]: array([11, 22, 34])
```

We can also use index masks: If the index mask is an Numpy array of data type `bool`, then an element is selected (True) or not (False) depending on the value of the index mask at the position of each element:

```
In [66]: B = array([n for n in range(5)])
         B
```

```
Out[66]: array([0, 1, 2, 3, 4])
```

```
In [67]: row_mask = array([True, False, True, False, False])
         B[row_mask]
```

```
Out[67]: array([0, 2])
```

```
In [68]: # same thing
         row_mask = array([1,0,1,0,0], dtype=bool)
         B[row_mask]
```

```
Out[68]: array([0, 2])
```

This feature is very useful to conditionally select elements from an array, for example, by using comparison operators:

```
In [69]: x = arange(0, 10, 0.5)
         x
```

```
Out[69]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
                 5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

```
In [70]: mask = (5 < x) * (x < 7.5)
         mask
```

```
Out[70]: array([False, False, False, False, False, False, False, False, False,
                False, False,  True,  True,  True,  True, False, False, False,
                False, False], dtype=bool)
```

```
In [71]: x[mask]
```

```
Out[71]: array([ 5.5,  6. ,  6.5,  7. ])
```

3.6 Functions for extracting data from arrays and creating arrays

3.6.1 where

The index mask can be converted to position index using the `where` function:

```
In [72]: indices = where(mask)
         indices
```

```
Out[72]: (array([11, 12, 13, 14]),)
```

```
In [73]: x[indices] # this indexing is equivalent to the fancy indexing x[mask]
```

```
Out[73]: array([ 5.5,  6. ,  6.5,  7. ])
```

3.6.2 diag

With the `diag` function we can also extract the diagonal and subdiagonals of an array:

```
In [74]: diag(A)
```

```
Out[74]: array([ 0, 11, 22, 33, 44])
```

```
In [75]: diag(A, -1)
```

```
Out[75]: array([10, 21, 32, 43])
```

3.6.3 take

The `take` function is similar to the fancy indexing described above:

```
In [76]: v2 = arange(-3,3)
         v2

Out[76]: array([-3, -2, -1,  0,  1,  2])

In [77]: row_indices = [1, 3, 5]
         v2[row_indices] # fancy indexing

Out[77]: array([-2,  0,  2])

In [78]: v2.take(row_indices)

Out[78]: array([-2,  0,  2])
```

But `take` also works on lists and other objects:

```
In [79]: take([-3, -2, -1,  0,  1,  2], row_indices)

Out[79]: array([-2,  0,  2])
```

3.6.4 choose

Constructs an array by picking elements from several arrays:

```
In [80]: which = [1, 0, 1, 0]
         choices = [[-2,-2,-2,-2], [5,5,5,5]]

         choose(which, choices)

Out[80]: array([ 5, -2,  5, -2])
```

3.7 Linear algebra

Vectorizing code is the key to writing efficient numerical calculation with Python/Numpy. That means that as much as possible of a program should be formulated in terms of matrix and vector operations, like matrix-matrix multiplication.

3.7.1 Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

```
In [81]: v1 = arange(0, 5)

In [82]: v1 * 2

Out[82]: array([0, 2, 4, 6, 8])

In [83]: v1 + 2

Out[83]: array([2, 3, 4, 5, 6])

In [84]: A * 2, A + 2
```

```
Out[84]: (array([[ 0,  2,  4,  6,  8],
                [20, 22, 24, 26, 28],
                [40, 42, 44, 46, 48],
                [60, 62, 64, 66, 68],
                [80, 82, 84, 86, 88]]), array([[ 2,  3,  4,  5,  6],
                [12, 13, 14, 15, 16],
                [22, 23, 24, 25, 26],
                [32, 33, 34, 35, 36],
                [42, 43, 44, 45, 46]]))
```

3.7.2 Element-wise array-array operations

When we add, subtract, multiply and divide arrays using other arrays, the default behavior is **element-wise** operations:

```
In [85]: A * A # element-wise multiplication

Out[85]: array([[   0,    1,    4,    9,   16],
               [ 100,  121,  144,  169,  196],
               [ 400,  441,  484,  529,  576],
               [ 900,  961, 1024, 1089, 1156],
               [1600, 1681, 1764, 1849, 1936]])

In [86]: v1 * v1

Out[86]: array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

```
In [87]: A.shape, v1.shape

Out[87]: ((5, 5), (5,))

In [88]: A * v1

Out[88]: array([[  0,   1,   4,   9,  16],
               [  0,  11,  24,  39,  56],
               [  0,  21,  44,  69,  96],
               [  0,  31,  64,  99, 136],
               [  0,  41,  84, 129, 176]])
```

3.7.3 Matrix algebra

What about matrix multiplication? There are two ways. We can either use the `dot` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

```
In [89]: dot(A, A)

Out[89]: array([[ 300,  310,  320,  330,  340],
               [1300, 1360, 1420, 1480, 1540],
               [2300, 2410, 2520, 2630, 2740],
               [3300, 3460, 3620, 3780, 3940],
               [4300, 4510, 4720, 4930, 5140]])

In [90]: dot(A, v1)

Out[90]: array([ 30, 130, 230, 330, 430])
```

```
In [91]: dot(v1, v1)

Out[91]: 30
```

Or we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+, -, *` to use matrix algebra.

```
In [92]: M = matrix(A)
         v = matrix(v1).T # make it a column vector

In [93]: v

Out[93]: matrix([[0],
                 [1],
                 [2],
                 [3],
                 [4]])

In [94]: M * M

Out[94]: matrix([[ 300,  310,  320,  330,  340],
                 [1300, 1360, 1420, 1480, 1540],
                 [2300, 2410, 2520, 2630, 2740],
                 [3300, 3460, 3620, 3780, 3940],
                 [4300, 4510, 4720, 4930, 5140]])

In [95]: try:
             M * v
         except ValueError as e:
             print(repr(e))

In [96]: # inner product
         v.T * v

Out[96]: matrix([[30]])

In [97]: # with matrix objects, standard matrix algebra applies
         try:
             v + M*v
         except ValueError as e:
             print(repr(e))
```

If we try to add, subtract or multiply objects with incompatible shapes, we get an error:

```
In [98]: v = matrix([1,2,3,4,5,6]).T

In [99]: shape(M), shape(v)

Out[99]: ((5, 5), (6, 1))

In [100]: try:
              M * v
          except ValueError as e:
              print(repr(e))

ValueError('shapes (5,5) and (6,1) not aligned: 5 (dim 1) != 6 (dim 0)',)
```

Explore these related functions: `inner`, `outer`, `cross`, `kron`, `tensordot` using the help function. For example: `help(kron)`.

3.7.4 Array/Matrix transformations

Above we used the `.T` to transpose the matrix object `v`. We could have used the `transpose` function to accomplish the same thing.

Other mathematical functions that transform matrix objects are:

```
In [101]: C = matrix([[1j, 2j], [3j, 4j]])
          C

Out[101]: matrix([[ 0.+1.j,  0.+2.j],
                  [ 0.+3.j,  0.+4.j]])

In [102]: conjugate(C)

Out[102]: matrix([[ 0.-1.j,  0.-2.j],
                  [ 0.-3.j,  0.-4.j]])
```

Hermitian conjugate: transpose + conjugate:

```
In [103]: C.H

Out[103]: matrix([[ 0.-1.j,  0.-3.j],
                  [ 0.-2.j,  0.-4.j]])
```

We can extract the real and imaginary parts of complex-valued arrays using `real` and `imag`:

```
In [104]: real(C) # same as: C.real

Out[104]: matrix([[ 0.,  0.],
                  [ 0.,  0.]])

In [105]: imag(C) # same as: C.imag

Out[105]: matrix([[ 1.,  2.],
                  [ 3.,  4.]])
```

Or the complex argument and absolute value:

```
In [106]: angle(C+1) # heads up MATLAB Users, angle is used instead of arg

Out[106]: array([[ 0.78539816,  1.10714872],
                 [ 1.24904577,  1.32581766]])

In [107]: abs(C)

Out[107]: matrix([[ 1.,  2.],
                  [ 3.,  4.]])
```

3.7.5 Matrix computations

Inverse

```
In [108]: inv(C) # equivalent to C.I

Out[108]: matrix([[ 0.+2.j ,  0.-1.j ],
                  [ 0.-1.5j,  0.+0.5j]])

In [109]: C.I * C

Out[109]: matrix([[  1.00000000e+00+0.j,   4.44089210e-16+0.j],
                  [  0.00000000e+00+0.j,   1.00000000e+00+0.j]])
```

Determinant

```
In [110]: det(C)
```

```
Out[110]: (2.0000000000000004+0j)
```

```
In [111]: det(C.I)
```

```
Out[111]: (0.50000000000000011+0j)
```

3.7.6 Data processing

Often it is useful to store datasets in Numpy arrays. Numpy provides a number of functions to calculate statistics of datasets in arrays.

For example, let's calculate some properties from the Stockholm temperature dataset used above.

```
In [112]: # reminder, the tempeature dataset is stored in the data variable:
          shape(data)
```

```
Out[112]: (77431, 7)
```

mean

```
In [113]: # the temperature data is in column 3
          mean(data[:,3])
```

```
Out[113]: 6.1971096847515854
```

The daily mean temperature in Stockholm over the last 200 years has been about 6.2 C.

standard deviations and variance

```
In [114]: std(data[:,3]), var(data[:,3])
```

```
Out[114]: (8.2822716213405734, 68.596023209663414)
```

min and max

```
In [115]: # lowest daily average temperature
          data[:,3].min()
```

```
Out[115]: -25.800000000000001
```

```
In [116]: # highest daily average temperature
          data[:,3].max()
```

```
Out[116]: 28.300000000000001
```

sum, prod, and trace

```
In [117]: d = arange(0, 10)
          d
```

```
Out[117]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [118]: # sum up all elements
          sum(d)
```

```
Out[118]: 45
```

```
In [119]: # product of all elements
          prod(d+1)

Out[119]: 3628800

In [120]: # cummulative sum
          cumsum(d)

Out[120]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])

In [121]: # cummulative product
          cumprod(d+1)

Out[121]: array([      1,       2,       6,      24,     120,     720,    5040,
            40320,  362880, 3628800])

In [122]: # same as: diag(A).sum()
          trace(A)

Out[122]: 110
```

3.7.7 Computations on subsets of arrays

We can compute with subsets of the data in an array using indexing, fancy indexing, and the other methods of extracting data from an array (described above).

For example, let's go back to the temperature dataset:

```
In [123]: !head -n 3 stockholm_td_adj.dat

1800  1  1    -6.1    -6.1    -6.1 1
1800  1  2   -15.4   -15.4   -15.4 1
1800  1  3   -15.0   -15.0   -15.0 1
```

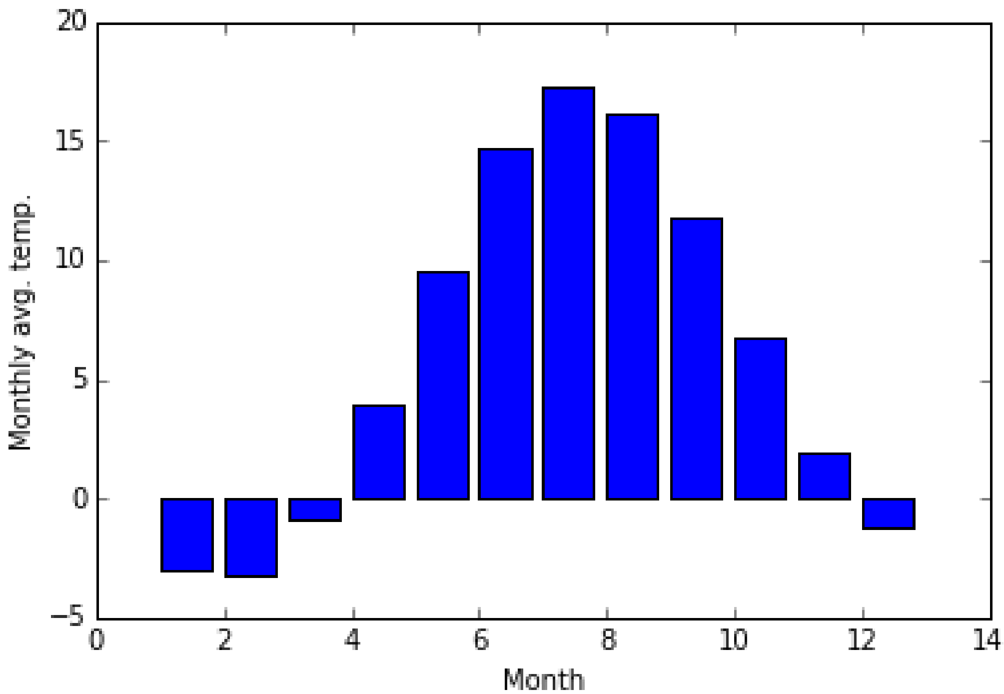
The dataformat is: year, month, day, daily average temperature, low, high, location.

If we are interested in the average temperature only in a particular month, say February, then we can create a index mask and use it to select only the data for that month using:

```
In [124]: unique(data[:,1]) # the month column takes values from 1 to 12

Out[124]: array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
           12.])

In [125]: mask_feb = data[:,1] == 2

In [126]: # the temperature data is in column 3
          mean(data[mask_feb,3])

Out[126]: -3.2121095707365961
```

With these tools we have very powerful data processing capabilities at our disposal. For example, to extract the average monthly average temperatures for each month of the year only takes a few lines of code:

```
In [127]: months = arange(1,13)
          monthly_mean = [mean(data[data[:,1] == month, 3]) for month in months]

          fig, ax = subplots()
          ax.bar(months, monthly_mean)
          ax.set_xlabel("Month")
          ax.set_ylabel("Monthly avg. temp.");
```



3.7.8 Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to a multidimensional arrays, it is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the `axis` argument we can specify how these functions should behave:

```
In [128]: m = rand(3,3)
          m

Out[128]: array([[ 0.90335299,  0.89597617,  0.04058584],
                 [ 0.09775568,  0.04431795,  0.59755417],
                 [ 0.13687355,  0.62502204,  0.72665469]])

In [129]: # global max
          m.max()

Out[129]: 0.90335299212908093

In [130]: # max in each column
          m.max(axis=0)

Out[130]: array([ 0.90335299,  0.89597617,  0.72665469])

In [131]: # max in each row
          m.max(axis=1)

Out[131]: array([ 0.90335299,  0.59755417,  0.72665469])
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

3.8 Reshaping, resizing and stacking arrays

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

```
In [132]: A

Out[132]: array([[ 0,  1,  2,  3,  4],
                 [10, 11, 12, 13, 14],
                 [20, 21, 22, 23, 24],
                 [30, 31, 32, 33, 34],
                 [40, 41, 42, 43, 44]])

In [133]: n, m = A.shape

In [134]: B = A.reshape((1,n*m))
          B

Out[134]: array([[ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
                  32, 33, 34, 40, 41, 42, 43, 44]])

In [135]: B[0,0:5] = 5 # modify the array
          B

Out[135]: array([[ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
                  32, 33, 34, 40, 41, 42, 43, 44]])

In [136]: A # and the original variable is also changed. B is only a different view of the same data

Out[136]: array([[ 5,  5,  5,  5,  5],
                 [10, 11, 12, 13, 14],
                 [20, 21, 22, 23, 24],
                 [30, 31, 32, 33, 34],
                 [40, 41, 42, 43, 44]])
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function create a copy of the data.

```
In [137]: B = A.flatten()
          B

Out[137]: array([ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
                 32, 33, 34, 40, 41, 42, 43, 44])

In [138]: B[0:5] = 10
          B

Out[138]: array([10, 10, 10, 10, 10, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
                 32, 33, 34, 40, 41, 42, 43, 44])

In [139]: A # now A has not changed, because B's data is a copy of A's, not refering to the same data

Out[139]: array([[ 5,  5,  5,  5,  5],
                 [10, 11, 12, 13, 14],
                 [20, 21, 22, 23, 24],
                 [30, 31, 32, 33, 34],
                 [40, 41, 42, 43, 44]])
```

3.9 Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array; for example, converting a vector to a column or row matrix:

```
In [140]: v = array([1,2,3])

In [141]: shape(v)

Out[141]: (3,)

In [142]: # make a column matrix of the vector v
          v[:, newaxis]

Out[142]: array([[1],
                 [2],
                 [3]])

In [143]: # column matrix
          v[:,newaxis].shape

Out[143]: (3, 1)

In [144]: # row matrix
          v[newaxis,:].shape

Out[144]: (1, 3)
```

3.10 Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate`, we can create larger vectors and matrices from smaller ones:

3.10.1 tile and repeat

```
In [145]: a = array([[1, 2], [3, 4]])

In [146]: # repeat each element 3 times
          repeat(a, 3)

Out[146]: array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])

In [147]: # tile the matrix 3 times
          tile(a, 3)

Out[147]: array([[1, 2, 1, 2, 1, 2],
                 [3, 4, 3, 4, 3, 4]])
```

3.10.2 concatenate

```
In [148]: b = array([[5, 6]])

In [149]: concatenate((a, b), axis=0)

Out[149]: array([[1, 2],
                 [3, 4],
                 [5, 6]])

In [150]: concatenate((a, b.T), axis=1)

Out[150]: array([[1, 2, 5],
                 [3, 4, 6]])
```

3.10.3 hstack and vstack

```
In [151]: vstack((a,b))

Out[151]: array([[1, 2],
                 [3, 4],
                 [5, 6]])

In [152]: hstack((a,b.T))

Out[152]: array([[1, 2, 5],
                 [3, 4, 6]])
```

3.11 Copy and "deep copy"

To achieve high performance, assignments in Python usually do not copy the underlying objects. This is important, for example, when objects are passed between functions, to avoid an excessive amount of memory copying when it is not necessary (technical term: pass by reference).

```
In [153]: A = array([[1, 2], [3, 4]])

          A

Out[153]: array([[1, 2],
                 [3, 4]])

In [154]: # now B is referring to the same array data as A
          B = A

In [155]: # changing B affects A
          B[0,0] = 10

          B

Out[155]: array([[10,  2],
                 [ 3,  4]])

In [156]: A

Out[156]: array([[10,  2],
                 [ 3,  4]])
```

If we want to avoid this behavior, so that when we get a new completely independent object `B` copied from `A`, then we need to do a so-called "deep copy" using the function `copy`:

```
In [157]: B = copy(A)

In [158]: # now, if we modify B, A is not affected
          B[0,0] = -5
          B

Out[158]: array([[-5,  2],
                 [ 3,  4]])

In [159]: A

Out[159]: array([[10,  2],
                 [ 3,  4]])
```

3.12 Iterating over array elements

Generally, it's best to avoid iterating over the elements of arrays whenever we can. Why? In a interpreted language like Python (or MATLAB), iterations are really slow compared to vectorized operations.

However, sometimes iterations are unavoidable. For such cases, the Python `for` loop is the most convenient way to iterate over an array:

```
In [160]: v = array([1,2,3,4])

          for element in v:
              print(element)
```

```
1
2
3
4
```

```
In [161]: M = array([[1,2], [3,4]])

          for row in M:
              print("row", row)

              for element in row:
                  print(element)
```

```
row [1 2]
1
2
row [3 4]
3
4
```

When we need to iterate over each element of an array and modify its elements, it is convenient to use the `enumerate` function to obtain both the element and its index in the `for` loop:

```
In [162]: for row_idx, row in enumerate(M):
              print("row_idx", row_idx, "row", row)

              for col_idx, element in enumerate(row):
                  print("col_idx", col_idx, "element", element)

                  # update the matrix M: square each element
                  M[row_idx, col_idx] = element ** 2
```

```
row_idx 0 row [1 2]
col_idx 0 element 1
col_idx 1 element 2
row_idx 1 row [3 4]
col_idx 0 element 3
col_idx 1 element 4
```

```
In [163]: # each element in M is now squared
          M
```

```
Out[163]: array([[ 1,  4],
                 [ 9, 16]])
```

3.13 Vectorizing functions

As mentioned several times, to get good performance we should try to avoid looping over elements in our vectors and matrices, and instead use vectorized algorithms. The first step in converting a scalar algorithm to a vectorized algorithm is to make sure that the functions we write work with vector inputs.

```
In [164]: def Theta(x):
              """
              Scalar implemenation of the Heaviside step function.
              """
              if x >= 0:
                  return 1
              else:
                  return 0

In [165]: try:
              Theta(array([-3,-2,-1,0,1,2,3]))
          except ValueError as e:
              print(repr(e))

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

OK, that didn't work because we didn't write the `Theta` function so that it can handle a vector input.

To get a vectorized version of Theta, we can use the Numpy function `vectorize`. In many cases it can automatically vectorize a function:

```
In [166]: Theta_vec = vectorize(Theta)

In [167]: Theta_vec(array([-3,-2,-1,0,1,2,3]))

Out[167]: array([0, 0, 0, 1, 1, 1, 1])
```

We can also implement the function to accept a vector input from the beginning (requires more effort but might give better performance):

```
In [168]: def Theta(x):
              """
              Vector-aware implementation of the Heaviside step function.
              """
              return 1 * (x >= 0)

In [169]: Theta(array([-3,-2,-1,0,1,2,3]))

Out[169]: array([0, 0, 0, 1, 1, 1, 1])

In [170]: # still works for scalars as well
          Theta(-1.2), Theta(2.6)

Out[170]: (0, 1)
```

3.14 Using arrays in conditions

When using arrays in conditions, for example in `if` statements and other boolean expressions, one needs to use `any` or `all`, which requires that any or all elements in the array evaluates to `True`:

```
In [171]: M
```

```
Out[171]: array([[ 1,  4],
                 [ 9, 16]])

In [172]: if (M > 5).any():
              print("at least one element in M is larger than 5")
          else:
              print("no element in M is larger than 5")

at least one element in M is larger than 5

In [173]: if (M > 5).all():
              print("all elements in M are larger than 5")
          else:
              print("all elements in M are not larger than 5")

all elements in M are not larger than 5
```

3.15 Type casting

Since Numpy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions (see also the similar `asarray` function). This always create a new array of new type:

```
In [174]: M.dtype

Out[174]: dtype('int64')

In [175]: M2 = M.astype(float)
          M2

Out[175]: array([[  1.,   4.],
                 [  9.,  16.]])

In [176]: M2.dtype

Out[176]: dtype('float64')

In [177]: M3 = M.astype(bool)
          M3

Out[177]: array([[ True,  True],
                 [ True,  True]], dtype=bool)
```

3.16 Further reading

- http://numpy.scipy.org
- http://scipy.org/Tentative_NumPy_Tutorial
- http://scipy.org/NumPy_for_Matlab_Users - A Numpy guide for MATLAB users.

Chapter 4
SciPy - Library of scientific algorithms for Python

```
In [1]: # what is this line all about? Answer in lecture 4
        %pylab inline
        from IPython.display import Image
```

```
Populating the interactive namespace from numpy and matplotlib
```

4.1 Introduction

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

In this lecture, we will look at how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module.

```
In [2]: from scipy import *
```

If we only need to use part of the SciPy framework, we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la`, we can do:

```
In [3]: import scipy.linalg as la
```

4.2 Special functions

A large number of mathematical special functions are important for many computional physics problems. SciPy provides implementations of a very extensive set of special functions. For details, see the list of functions in the reference documentation at http://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special.

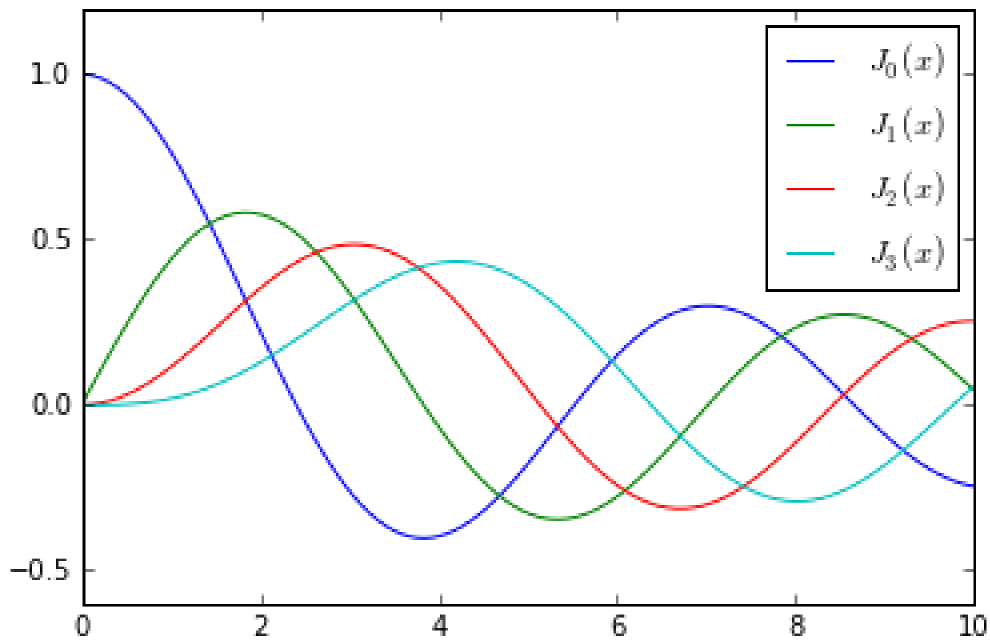
To demonstrate the typical usage of special functions, we will look in more detail at the Bessel functions:

```
In [4]: #
        # The scipy.special module includes a large number of Bessel functions
        # Here we will use the functions jn and yn, which are the Bessel functions
        # of the first and second kind and real-valued order. We also include the
        # function jn_zeros and yn_zeros that gives the zeroes of the functions jn
        # and yn.
        #
        from scipy.special import jn, yn, jn_zeros, yn_zeros
```

```
In [5]: n = 0    # order
        x = 0.0

        # Bessel function of first kind
        print("J_%d(%f) = %f" % (n, x, jn(n, x)))

        x = 1.0
        # Bessel function of second kind
        print("Y_%d(%f) = %f" % (n, x, yn(n, x)))
```

```
J_0(0.000000) = 1.000000
Y_0(1.000000) = 0.088257
```

```
In [6]: x = linspace(0, 10, 100)

        fig, ax = subplots()
        for n in range(4):
            ax.plot(x, jn(n, x), label=r"$J_%d(x)$" % n)
        ax.legend();
```




```
In [7]: # zeros of Bessel functions
        n = 0 # order
        m = 4 # number of roots to compute
        jn_zeros(n, m)
```

```
Out[7]: array([  2.40482556,   5.52007811,   8.65372791,  11.79153444])
```

4.3 Integration

4.3.1 Numerical integration: quadrature

Numerical evaluation of a function of the type

$$\int_a^b f(x)dx$$

is called *numerical quadrature*, or simply *quadrature*. SciPy provides a series of functions for different kind of quadrature, for example the `quad`, `dblquad` and `tplquad` for single, double and triple integrals, respectively.

```
In [8]: from scipy.integrate import quad, dblquad, tplquad
```

The `quad` function takes a large number of optional arguments which can be used to fine-tune the behavior of the function (try `help(quad)` for details).

The basic usage is as follows:

```
In [9]: # define a simple function for the integrand
        def f(x):
            return x
```

```
In [10]: x_lower = 0 # the lower limit of x
         x_upper = 1 # the upper limit of x
```

```
val, abserr = quad(f, x_lower, x_upper)

print("integral value =", val, ", absolute error =", abserr )
```

```
integral value = 0.5 , absolute error = 5.551115123125783e-15
```

If we need to pass extra arguments to the integrand function, we can use the **args** keyword argument:

```
In [11]: def integrand(x, n):
             """
             Bessel function of first kind and order n.
             """
             return jn(n, x)


         x_lower = 0  # the lower limit of x
         x_upper = 10 # the upper limit of x

         val, abserr = quad(integrand, x_lower, x_upper, args=(3,))

         print(val, abserr)
```

```
0.7366751370811073 9.389126882496403e-13
```

For simple functions, we can use a lambda function (nameless function) instead of explicitly defining a function for the integrand:

```
In [12]: val, abserr = quad(lambda x: exp(-x ** 2), -Inf, Inf)

         print("numerical  =", val, abserr)

         analytical = sqrt(pi)
         print("analytical =", analytical)
```

```
numerical  = 1.7724538509055159 1.4202636780944923e-08
analytical = 1.77245385091
```

As shown in the example above, we can also use 'Inf' or '-Inf' as integral limits.
Higher-dimensional integration works in the same way:

```
In [13]: def integrand(x, y):
             return exp(-x**2-y**2)

         x_lower = 0
         x_upper = 10
         y_lower = 0
         y_upper = 10

         val, abserr = dblquad(integrand, x_lower, x_upper, lambda x : y_lower, lambda x: y_upper)

         print(val, abserr)
```

```
0.7853981633974476 1.638229942140971e-13
```

Note how we had to pass lambda functions for the limits for the y integration, since these in general can be functions of x.

4.4 Ordinary differential equations (ODEs)

SciPy provides two different ways to solve ODEs: An API based on the function `odeint`, and an object-oriented API based on the class `ode`. Usually `odeint` is easier to get started with, but the `ode` class offers a finer level of control.

Here we will use the `odeint` functions. For more information about the class `ode`, try `help(ode)`. It does pretty much the same thing as `odeint`, but in an object-oriented fashion.

To use `odeint`, first import it from the `scipy.integrate` module.

```
In [14]: from scipy.integrate import odeint, ode
```

A system of ODEs are usually formulated on standard form before it is attacked numerically. The standard form is:

$y' = f(y, t)$

where

$y = [y_1(t), y_2(t), ..., y_n(t)]$

and f is some function that gives the derivatives of the function $y_i(t)$. To solve an ODE, we need to know the function f and an initial condition, $y(0)$.

Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives.

Once we have defined the Python function `f` and array `y_0` (that is f and $y(0)$ in the mathematical formulation), we can use the `odeint` function as:

```
y_t = odeint(f, y_0, t)
```

where `t` is and array with time-coordinates for which to solve the ODE problem. `y_t` is an array with one row for each point in time in `t`, where each column corresponds to a solution `y_i(t)` at that point in time.

We will see how we can implement `f` and `y_0` in Python code in the examples below.

Example: double pendulum Let's consider a physical example: The double compound pendulum, described in some detail here: http://en.wikipedia.org/wiki/Double_pendulum

```
In [15]: Image(url='http://upload.wikimedia.org/wikipedia/commons/c/c9/Double-compound-pendulum-dimensi

Out[15]: <IPython.core.display.Image object>
```

The equations of motion of the pendulum are given on the wiki page:

$\dot{\theta}_1 = \frac{6}{m\ell^2} \frac{2p_{\theta_1} - 3\cos(\theta_1-\theta_2)p_{\theta_2}}{16-9\cos^2(\theta_1-\theta_2)}$

$\dot{\theta}_2 = \frac{6}{m\ell^2} \frac{8p_{\theta_2} - 3\cos(\theta_1-\theta_2)p_{\theta_1}}{16-9\cos^2(\theta_1-\theta_2)}.$

$\dot{p}_{\theta_1} = -\frac{1}{2}m\ell^2 \left[\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + 3\frac{g}{\ell}\sin\theta_1\right]$

$\dot{p}_{\theta_2} = -\frac{1}{2}m\ell^2 \left[-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + \frac{g}{\ell}\sin\theta_2\right]$

To make the Python code simpler to follow, let's introduce new variable names and the vector notation: $x = [\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}]$

$\dot{x}_1 = \frac{6}{m\ell^2} \frac{2x_3 - 3\cos(x_1-x_2)x_4}{16-9\cos^2(x_1-x_2)}$

$\dot{x}_2 = \frac{6}{m\ell^2} \frac{8x_4 - 3\cos(x_1-x_2)x_3}{16-9\cos^2(x_1-x_2)}$

$\dot{x}_3 = -\frac{1}{2}m\ell^2 \left[\dot{x}_1\dot{x}_2\sin(x_1-x_2) + 3\frac{g}{\ell}\sin x_1\right]$

$\dot{x}_4 = -\frac{1}{2}m\ell^2 \left[-\dot{x}_1\dot{x}_2\sin(x_1-x_2) + \frac{g}{\ell}\sin x_2\right]$

```
In [16]: g = 9.82
         L = 0.5
         m = 0.1
```

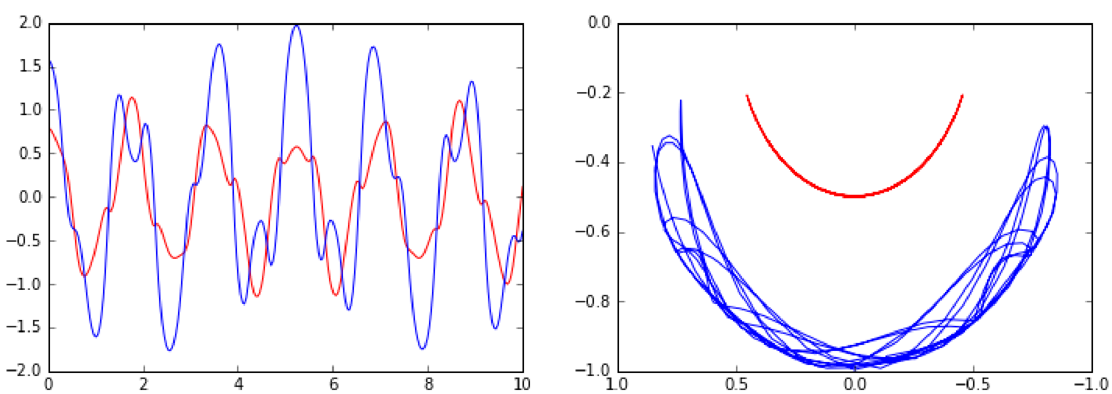
```
def dx(x, t):
    """
    The right-hand side of the pendulum ODE
    """
    x1, x2, x3, x4 = x[0], x[1], x[2], x[3]

    dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * cos(x1-x2) * x4)/(16 - 9 * cos(x1-x2)**2)
    dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * cos(x1-x2) * x3)/(16 - 9 * cos(x1-x2)**2)
    dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * sin(x1-x2) + 3 * (g/L) * sin(x1))
    dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * sin(x1-x2) + (g/L) * sin(x2))

    return [dx1, dx2, dx3, dx4]
```

In [17]:
```
# choose an initial state
x0 = [pi/4, pi/2, 0, 0]
```

In [18]:
```
# time coordinate to solve the ODE for: from 0 to 10 seconds
t = linspace(0, 10, 250)
```

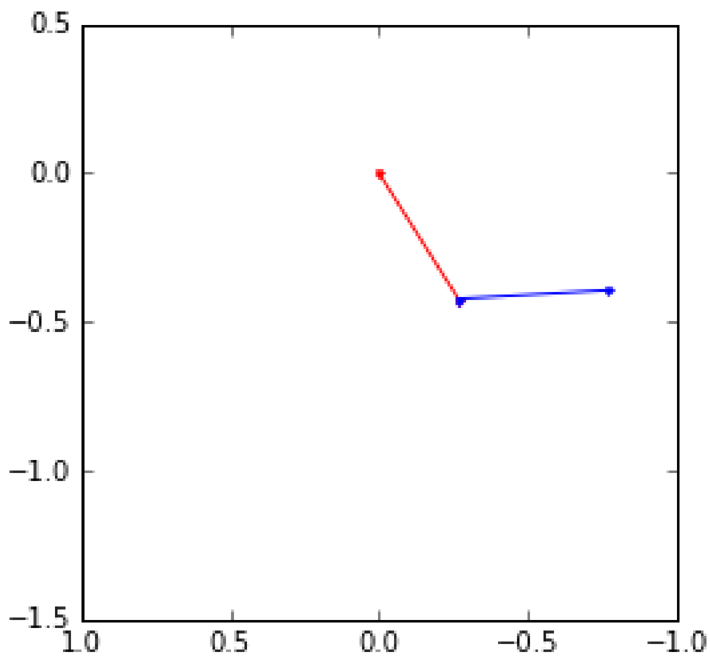
In [19]:
```
# solve the ODE problem
x = odeint(dx, x0, t)
```

In [20]:
```
# plot the angles as a function of time

fig, axes = subplots(1,2, figsize=(12,4))
axes[0].plot(t, x[:, 0], 'r', label="theta1")
axes[0].plot(t, x[:, 1], 'b', label="theta2")


x1 = + L * sin(x[:, 0])
y1 = - L * cos(x[:, 0])

x2 = x1 + L * sin(x[:, 1])
y2 = y1 - L * cos(x[:, 1])

axes[1].plot(x1, y1, 'r', label="pendulum1")
axes[1].plot(x2, y2, 'b', label="pendulum2")
axes[1].set_ylim([-1, 0])
axes[1].set_xlim([1, -1]);
```



Simple animation of the pendulum motion. We will see how to make a better animation in Lecture 4.

```
In [21]: from IPython.display import clear_output
         import time

In [22]: fig, ax = subplots(figsize=(4,4))

         for t_idx, tt in enumerate(t[:200]):

             x1 = + L * sin(x[t_idx, 0])
             y1 = - L * cos(x[t_idx, 0])

             x2 = x1 + L * sin(x[t_idx, 1])
             y2 = y1 - L * cos(x[t_idx, 1])

             ax.cla()
             ax.plot([0, x1], [0, y1], 'r.-')
             ax.plot([x1, x2], [y1, y2], 'b.-')
             ax.set_ylim([-1.5, 0.5])
             ax.set_xlim([1, -1])

             display(fig)
             clear_output()

             time.sleep(0.1)
```



Example: Damped harmonic oscillator ODE problems are important in computational physics, so we will look at one more example: the damped harmonic oscillation. This problem is well described on the wiki page: http://en.wikipedia.org/wiki/Damping

The equation of motion for the damped oscillator is:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + 2\zeta\omega_0\frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

where x is the position of the oscillator, ω_0 is the frequency, and ζ is the damping ratio. To write this second-order ODE on standard form, we introduce $p = \frac{\mathrm{d}x}{\mathrm{d}t}$:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$
$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

In the implementation of this example, we will add extra arguments to the RHS function for the ODE, rather than using global variables as we did in the previous example. As a consequence of the extra arguments to the RHS, we need to pass an keyword argument `args` to the `odeint` function:

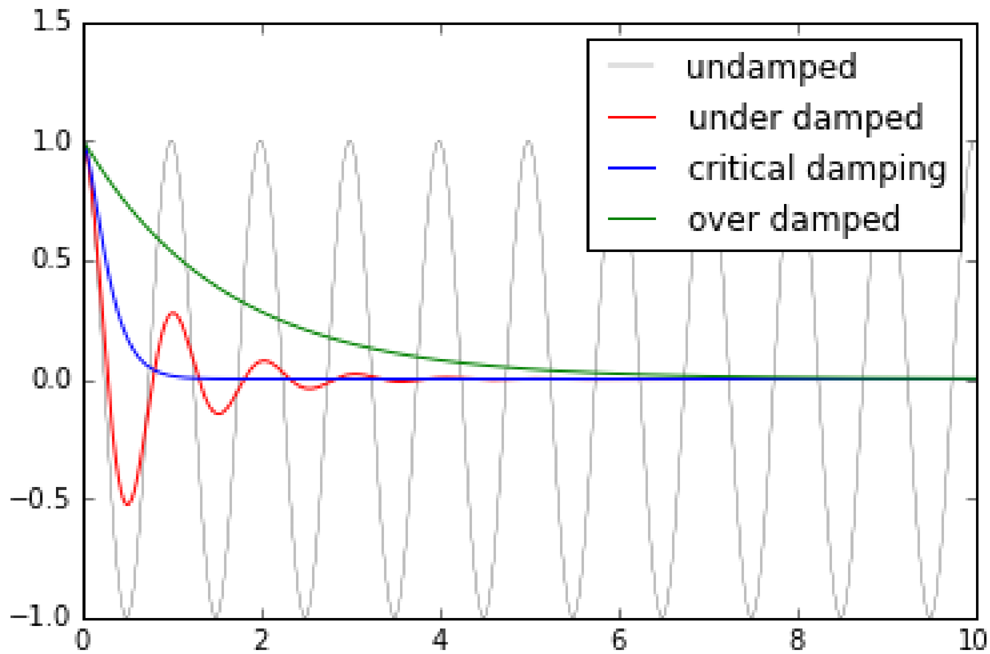
```
In [23]: def dy(y, t, zeta, w0):
             """
             The right-hand side of the damped oscillator ODE
             """
             x, p = y[0], y[1]

             dx = p
             dp = -2 * zeta * w0 * p - w0**2 * x

             return [dx, dp]
```

```
In [24]: # initial state:
         y0 = [1.0, 0.0]
```

```
In [25]: # time coordinate to solve the ODE for
         t = linspace(0, 10, 1000)
         w0 = 2*pi*1.0
```

```
In [26]: # solve the ODE problem for three different values of the damping ratio

         y1 = odeint(dy, y0, t, args=(0.0, w0)) # undamped
         y2 = odeint(dy, y0, t, args=(0.2, w0)) # under damped
         y3 = odeint(dy, y0, t, args=(1.0, w0)) # critial damping
         y4 = odeint(dy, y0, t, args=(5.0, w0)) # over damped
```

```
In [27]: fig, ax = subplots()
         ax.plot(t, y1[:,0], 'k', label="undamped", linewidth=0.25)
         ax.plot(t, y2[:,0], 'r', label="under damped")
         ax.plot(t, y3[:,0], 'b', label=r"critical damping")
         ax.plot(t, y4[:,0], 'g', label="over damped")
         ax.legend();
```



4.5 Fourier transform

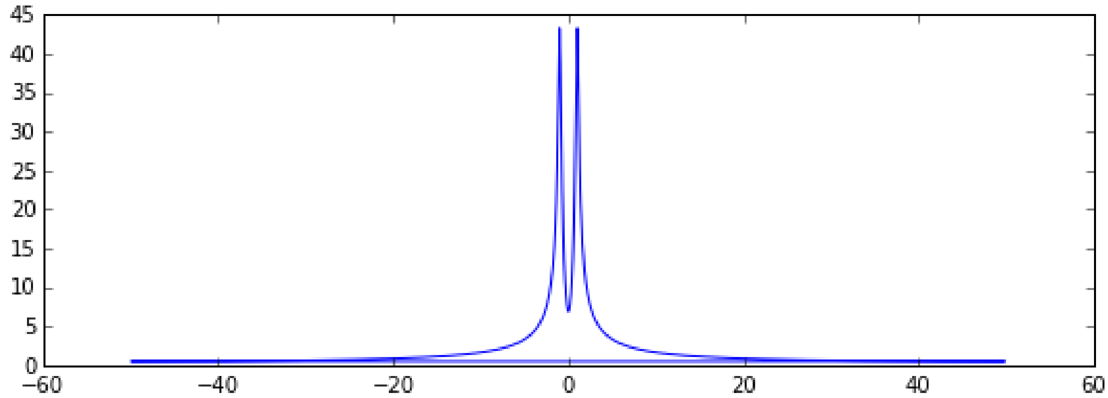
Fourier transforms are one of the universal tools in computational physics; they appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK library from NetLib, an efficient and well tested FFT library written in FORTRAN. The SciPy API has a few additional convenience functions, but overall the API is closely related to the original FORTRAN library.

To use the `fftpack` module in a python program, include it using:

```
In [28]: from scipy.fftpack import *
```

To demonstrate how to do a fast Fourier transform with SciPy, let's look at the FFT of the solution to the damped oscillator from the previous section:

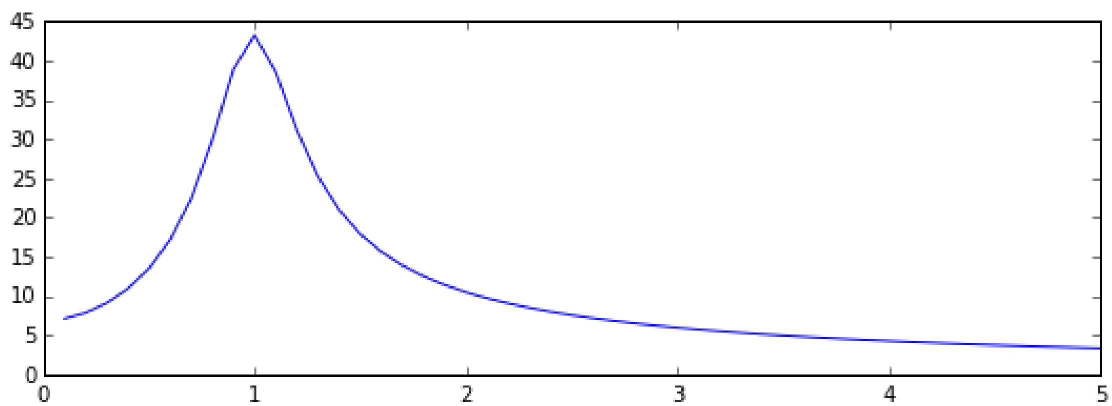
```
In [29]: N = len(t)
         dt = t[1]-t[0]

         # calculate the fast fourier transform
         # y2 is the solution to the under-damped oscillator from the previous section
         F = fft(y2[:,0])

         # calculate the frequencies for the components in F
         w = fftfreq(N, dt)

In [30]: fig, ax = subplots(figsize=(9,3))
         ax.plot(w, abs(F));
```



Since the signal is real, the spectrum is symmetric. We therefore only need to plot the part that corresponds to the postive frequencies. To extract that part of the `w` and `F`, we can use some of the indexing tricks for NumPy arrays we saw in Lecture 2:

```
In [31]: indices = where(w > 0) # select only indices for elements that corresponds to positive frequen
         w_pos = w[indices]
         F_pos = F[indices]
```

```
In [32]: fig, ax = subplots(figsize=(9,3))
         ax.plot(w_pos, abs(F_pos))
         ax.set_xlim(0, 5);
```



As expected, we now see a peak in the spectrum that is centered around 1, which is the frequency we used in the damped oscillator example.

4.6 Linear algebra

The linear algebra module contains a lot of matrix-related functions, including linear equation solving, eigenvalue solvers, matrix functions (for example matrix-exponentiation), a number of different decompositions (SVD, LU, cholesky), etc.

Detailed documetation is available at: http://docs.scipy.org/doc/scipy/reference/linalg.html

Here we will look at how to use some of these functions:

4.6.1 Linear equation systems

Linear equation systems on the matrix form

$Ax = b$

where A is a matrix and x, b are vectors can be solved like:

```
In [33]: A = array([[1,2,3], [4,5,6], [7,8,9]])
         b = array([1,2,3])

In [34]: x = solve(A, b)

         x

Out[34]: array([-0.33333333,  0.66666667,  0.        ])

In [35]: # check
         dot(A, x) - b

Out[35]: array([ -1.11022302e-16,   0.00000000e+00,   0.00000000e+00])
```

We can also do the same with

$AX = B$

where A, B, X are matrices:

```
In [36]: A = rand(3,3)
         B = rand(3,3)

In [37]: X = solve(A, B)

In [38]: X

Out[38]: array([[-1.18576771,  2.95011285, -4.18272141],
                [ 1.58452628, -1.78546257,  4.93954657],
                [-0.31708076,  1.69250712, -2.29522332]])

In [39]: # check
         norm(dot(A, X) - B)

Out[39]: 3.5975337699988621e-16
```

4.6.2 Eigenvalues and eigenvectors

The eigenvalue problem for a matrix A:

$Av_n = \lambda_n v_n$

where v_n is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals` and for calculating both eigenvalues and eigenvectors, use the function `eig`:

```
In [40]: evals = eigvals(A)

In [41]: evals

Out[41]: array([ 1.18476874+0.j        , -0.00767866+0.12773283j,
                -0.00767866-0.12773283j])

In [42]: evals, evecs = eig(A)

In [43]: evals
```

Out[43]: array([1.18476874+0.j   , -0.00767866+0.12773283j,
               -0.00767866-0.12773283j])

In [44]: evecs

Out[44]: array([[0.69060036+0.j   , 0.64727156+0.j   , 0.64727156-0.j   ],
               [0.42998846+0.j   , -0.37526303-0.5011759j ,
                -0.37526303+0.5011759j],
               [0.58153338+0.j   , 0.15609273+0.40580162j,
                 0.15609273-0.40580162j]])

The eigenvectors corresponding to the nth eigenvalue (stored in `evals[n]`) is the nth *column* in `evecs`, i.e., `evecs[:,n]`. To verify this, let's try mutiplying eigenvectors with the matrix and compare to the product of the eigenvector and the eigenvalue:

```
In [45]: n = 1

         norm(dot(A, evecs[:,n]) - evals[n] * evecs[:,n])

Out[45]: 1.6293708262254898e-16
```

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

4.6.3 Matrix operations

```
In [46]: # the matrix inverse
         inv(A)

Out[46]: array([[ 3.80599293,  4.52024913, -6.85975449],
                [-7.41381008,  0.82839291,  8.8158496 ],
                [ 5.32680964, -1.01919526, -4.72821215]])

In [47]: # determinant
         det(A)

Out[47]: 0.019400158815669057

In [48]: # norms of various orders
         norm(A, ord=2), norm(A, ord=Inf)

Out[48]: (1.4646872727298801, 1.5319294534693251)
```

4.6.4 Sparse matrices

Sparse matrices are often useful in numerical simulations dealing with large systems, if the problem can be described in matrix form where the matrices or vectors mostly contains zeroes. Scipy has good support for sparse matrices, with basic linear algebra operations (such as equation solving, eigenvalue calculations, etc).

There are many possible strategies for storing sparse matrices in an efficient way. Some of the most common are the so-called coordinate form (COO), list of list (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has advantanges and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc.) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. Often a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before being used in real calculations.

For more information about these sparse formats, see http://en.wikipedia.org/wiki/Sparse_matrix

When we create a sparse matrix, we have to choose which format it should be stored in. For example:

```
In [49]: from scipy.sparse import *
```

```
In [50]: # dense matrix
         M = array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]]); M
```

```
Out[50]: array([[1, 0, 0, 0],
                [0, 3, 0, 0],
                [0, 1, 1, 0],
                [1, 0, 0, 1]])
```

```
In [51]: # convert from dense to sparse
         A = csr_matrix(M); A
```

```
Out[51]: <4x4 sparse matrix of type '<class 'numpy.int64'>'
                 with 6 stored elements in Compressed Sparse Row format>
```

```
In [52]: # convert from sparse to dense
         A.todense()
```

```
Out[52]: matrix([[1, 0, 0, 0],
                 [0, 3, 0, 0],
                 [0, 1, 1, 0],
                 [1, 0, 0, 1]], dtype=int64)
```

More efficient way to create sparse matrices: create an empty matrix and populate it using matrix indexing (avoids creating a potentially large dense matrix):

```
In [53]: A = lil_matrix((4,4)) # empty 4x4 sparse matrix
         A[0,0] = 1
         A[1,1] = 3
         A[2,2] = A[2,1] = 1
         A[3,3] = A[3,0] = 1
         A
```

```
Out[53]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in LInked List format>
```

```
In [54]: A.todense()
```

```
Out[54]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.],
                 [ 0.,  1.,  1.,  0.],
                 [ 1.,  0.,  0.,  1.]])
```

Converting between different sparse matrix formats:

```
In [55]: A
```

```
Out[55]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in LInked List format>
```

```
In [56]: A = csr_matrix(A); A
```

```
Out[56]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in Compressed Sparse Row format>
```

```
In [57]: A = csc_matrix(A); A
```

```
Out[57]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in Compressed Sparse Column format>
```

We can compute with sparse matrices like we do with dense matrices:

```
In [58]: A.todense()

Out[58]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.],
                 [ 0.,  1.,  1.,  0.],
                 [ 1.,  0.,  0.,  1.]])

In [59]: (A * A).todense()

Out[59]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  9.,  0.,  0.],
                 [ 0.,  4.,  1.,  0.],
                 [ 2.,  0.,  0.,  1.]])

In [60]: try:
             dot(A, A).todense()
         except ValueError as e:
             print(repr(e))

ValueError('Cannot find a common data type.',)

In [61]: v = array([1,2,3,4])[:,newaxis]; v

Out[61]: array([[1],
                [2],
                [3],
                [4]])

In [62]: # sparse matrix - dense vector multiplication
         A * v

Out[62]: array([[ 1.],
                [ 6.],
                [ 5.],
                [ 5.]])

In [63]: # same result with dense matrix - dense vector multiplcation
         A.todense() * v

Out[63]: matrix([[ 1.],
                 [ 6.],
                 [ 5.],
                 [ 5.]])
```

4.7 Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved. Here we will only look at a few very simple cases. For a more detailed introduction to optimization with SciPy, see: http://scipy-lectures.github.com/advanced/mathematical_optimization/index.html

To use the optimization module in SciPy, first include the `optimize` module:

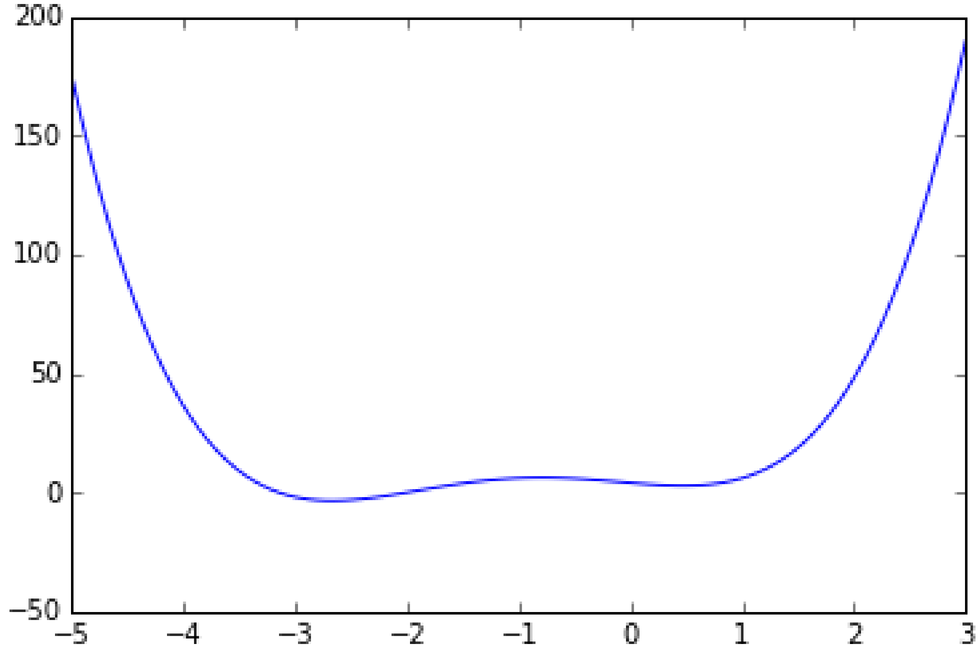
```
In [64]: from scipy import optimize
```

4.7.1 Finding a minima

First, let's find the minima of a simple function of a single variable:

```
In [65]: def f(x):
             return 4*x**3 + (x-2)**2 + x**4
```

```
In [66]: fig, ax  = subplots()
         x = linspace(-5, 3, 100)
         ax.plot(x, f(x));
```



We can use the fmin_bfgs function to find the minima of a function:

```
In [67]: x_min = optimize.fmin_bfgs(f, -2)
         x_min

Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 6
         Function evaluations: 30
         Gradient evaluations: 10

Out[67]: array([-2.67298167])
```

```
In [68]: optimize.fmin_bfgs(f, 0.5)

Optimization terminated successfully.
         Current function value: 2.804988
         Iterations: 3
         Function evaluations: 15
         Gradient evaluations: 5
```

```
Out[68]: array([ 0.46961745])
```

We can also use the brent or fminbound functions. They have slightly different syntax and use different algorithms.

```
In [69]: optimize.brent(f)
```

```
Out[69]: 0.46961743402759754
```

```
In [70]: optimize.fminbound(f, -4, 2)
```

```
Out[70]: -2.6729822917513886
```

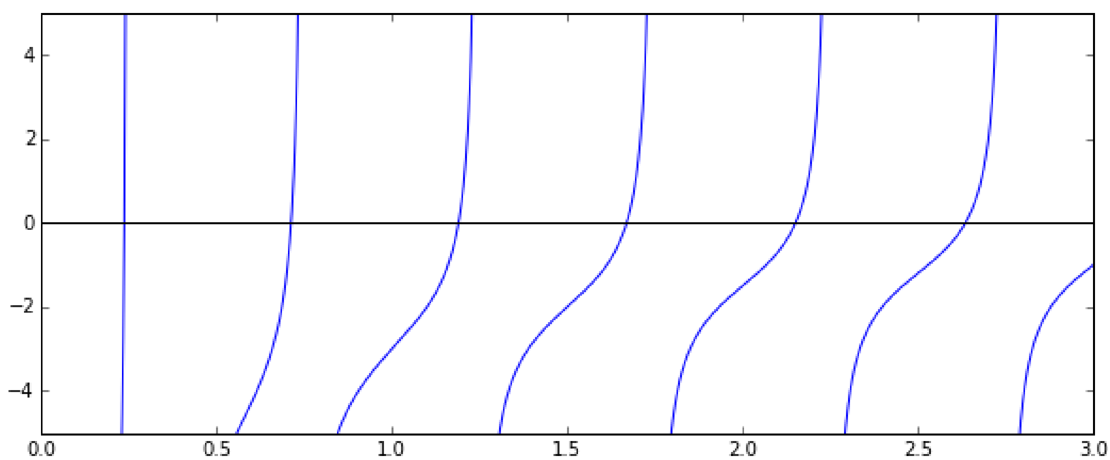
4.7.2 Finding a solution to a function

To find the root for a function of the form $f(x) = 0$, we can use the fsolve function. It requires an initial guess:

```
In [71]: omega_c = 3.0
         def f(omega):
             # a transcendental equation: resonance frequencies of a low-Q SQUID terminated microwave r
             return tan(2*pi*omega) - omega_c/omega
```

```
In [72]: fig, ax  = subplots(figsize=(10,4))
         x = linspace(0, 3, 1000)
         y = f(x)
         mask = where(abs(y) > 50)
         x[mask] = y[mask] = NaN # get rid of vertical line when the function flip sign
         ax.plot(x, y)
         ax.plot([0, 3], [0, 0], 'k')
         ax.set_ylim(-5,5);
```

```
/Users/dmertz/anaconda/lib/python3.4/site-packages/IPython/kernel/__main__.py:4: RuntimeWarning: divide b
```




```
In [73]: optimize.fsolve(f, 0.1)
```

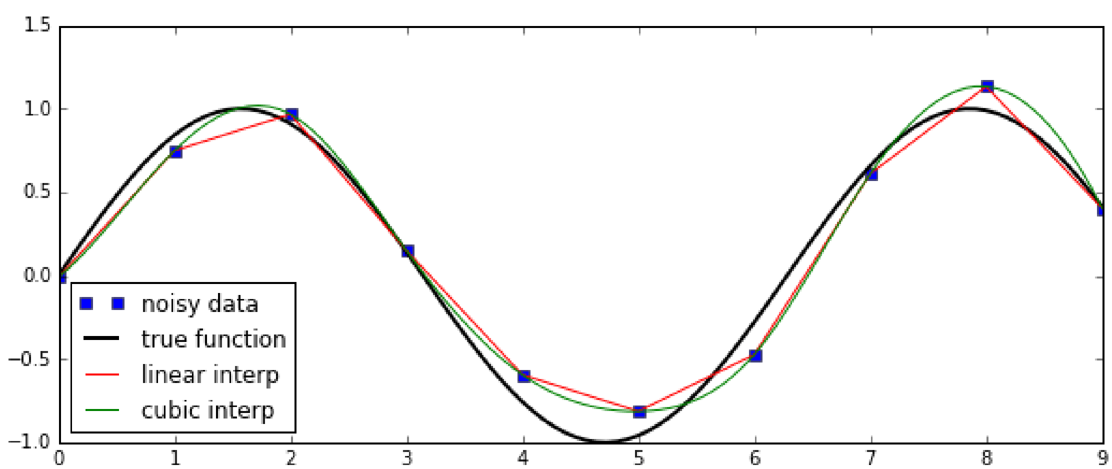
```
Out[73]: array([ 0.23743014])
```

```
In [74]: optimize.fsolve(f, 0.6)

Out[74]: array([ 0.71286972])

In [75]: optimize.fsolve(f, 1.1)

Out[75]: array([ 1.18990285])
```

4.8 Interpolation

Interpolation is simple and convenient in SciPy: The `interp1d` function, when given arrays describing X and Y data, returns an object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X). It returns the corresponding interpolated y value:

```
In [76]: from scipy.interpolate import *

In [77]: def f(x):
             return sin(x)

In [78]: n = arange(0, 10)
         x = linspace(0, 9, 100)

         y_meas = f(n) + 0.1 * randn(len(n)) # simulate measurement with noise
         y_real = f(x)

         linear_interpolation = interp1d(n, y_meas)
         y_interp1 = linear_interpolation(x)

         cubic_interpolation = interp1d(n, y_meas, kind='cubic')
         y_interp2 = cubic_interpolation(x)

In [79]: fig, ax = subplots(figsize=(10,4))
         ax.plot(n, y_meas, 'bs', label='noisy data')
         ax.plot(x, y_real, 'k', lw=2, label='true function')
         ax.plot(x, y_interp1, 'r', label='linear interp')
         ax.plot(x, y_interp2, 'g', label='cubic interp')
         ax.legend(loc=3);
```



4.9 Statistics

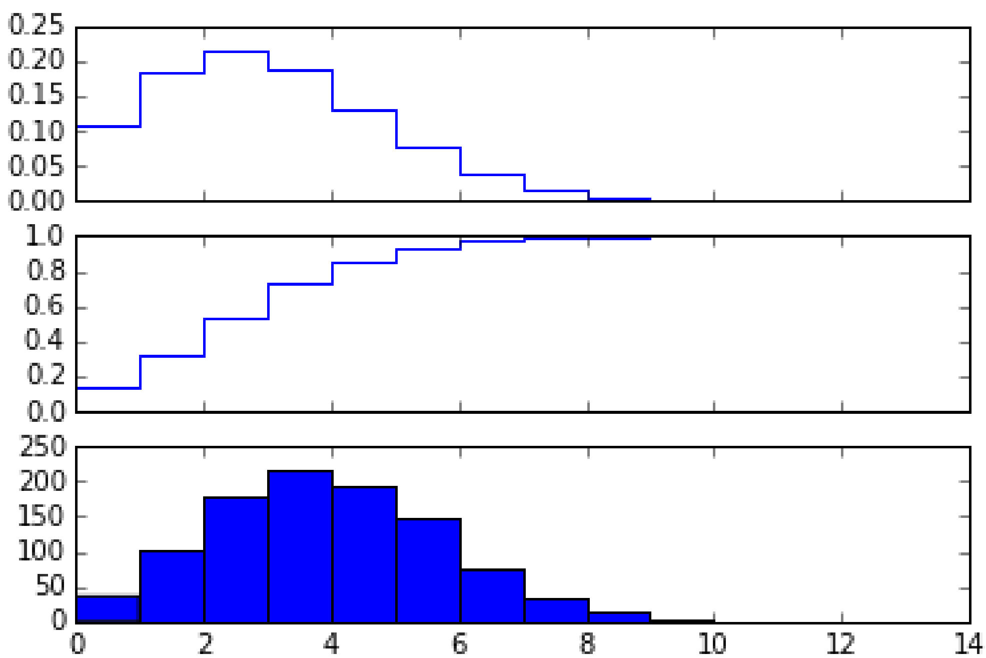
The `scipy.stats` module contains a large number of statistical distributions, statistical functions and tests. For a complete documentation of its features, see http://docs.scipy.org/doc/scipy/reference/stats.html.

There is also a very powerful Python package for statistical modeling called statsmodels. See http://statsmodels.sourceforge.net for more details.

```
In [80]: from scipy import stats
```

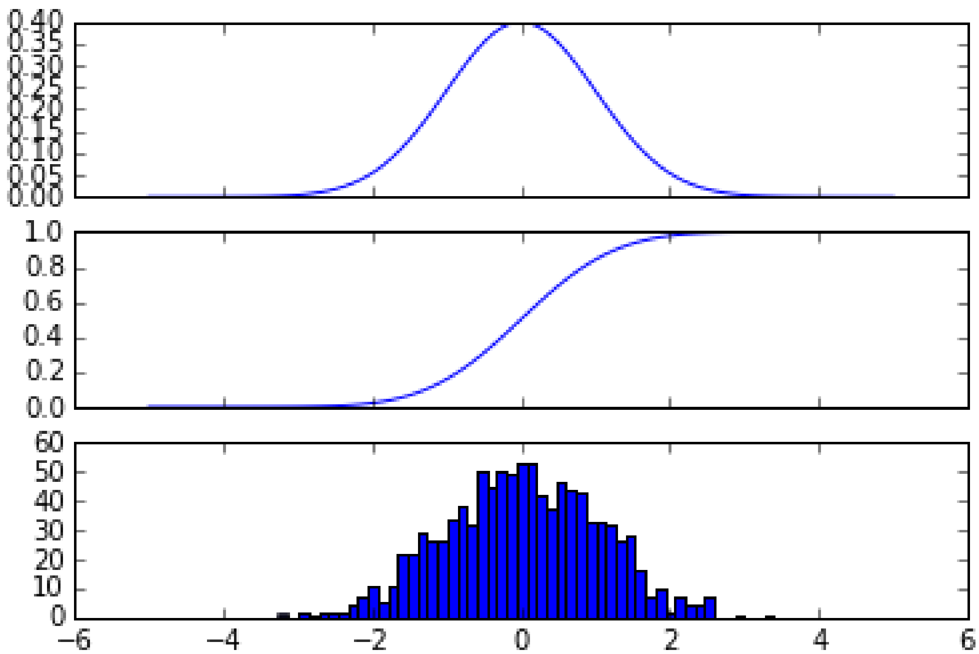
```
In [81]: # create a (discrete) random variable with poissionian distribution

         X = stats.poisson(3.5) # photon distribution for a coherent state with n=3.5 photons
```

```
In [82]: n = arange(0,15)

         fig, axes = subplots(3,1, sharex=True)

         # plot the probability mass function (PMF)
         axes[0].step(n, X.pmf(n))

         # plot the commulative distribution function (CDF)
         axes[1].step(n, X.cdf(n))

         # plot histogram of 1000 random realizations of the stochastic variable X
         axes[2].hist(X.rvs(size=1000));
```




```
In [83]: # create a (continous) random variable with normal distribution
         Y = stats.norm()
```

```
In [84]: x = linspace(-5,5,100)

         fig, axes = subplots(3,1, sharex=True)

         # plot the probability distribution function (PDF)
         axes[0].plot(x, Y.pdf(x))

         # plot the commulative distributin function (CDF)
         axes[1].plot(x, Y.cdf(x));

         # plot histogram of 1000 random realizations of the stochastic variable Y
         axes[2].hist(Y.rvs(size=1000), bins=50);
```



Statistics:

```
In [85]: X.mean(), X.std(), X.var() # poission distribution

Out[85]: (3.5, 1.8708286933869707, 3.5)

In [86]: Y.mean(), Y.std(), Y.var() # normal distribution

Out[86]: (0.0, 1.0, 1.0)
```

4.9.1 Statistical tests

Test whether two sets of (independent) random data comes from the same distribution:

```
In [87]: t_statistic, p_value = stats.ttest_ind(X.rvs(size=1000), X.rvs(size=1000))

         print("t-statistic =", t_statistic)
         print("p-value =", p_value)
```

```
t-statistic = 1.13574794292
p-value = 0.256198284814
```

Since the p value is very large, we cannot reject the hypothesis that the two sets of random data have *different* means.

To test whether the mean of a single sample of data has mean 0.1 (the true mean is 0.0):

```
In [88]: stats.ttest_1samp(Y.rvs(size=1000), 0.1)
```

```
Out[88]: (-3.9277846437083679, 9.1622807001296135e-05)
```

Low p-value means that we can reject the hypothesis that the mean of Y is 0.1.

```
In [89]: Y.mean()
```

```
Out[89]: 0.0
```

```
In [90]: stats.ttest_1samp(Y.rvs(size=1000), Y.mean())
```

```
Out[90]: (-0.5796402331344479, 0.56228794580881125)
```

4.10 Further reading

- http://www.scipy.org - The official web page for the SciPy project.
- http://docs.scipy.org/doc/scipy/reference/tutorial/index.html - A tutorial on how to get started using SciPy.
- https://github.com/scipy/scipy/ - The SciPy source code.

Chapter 5

matplotlib - 2D and 3D plotting in Python

```
In [1]: # This line configures matplotlib to show figures embedded in the notebook,
        # instead of opening a new window for each figure. More about that later.
        # If you are using an old version of IPython, try using '%pylab inline' instead.
        %matplotlib inline
```

5.1 Introduction

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeXformatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

One of the of the key features of matplotlib that I would like to emphasize, and that I think makes matplotlib highly suitable for generating figures for scientific publications is that all aspects of the figure can be controlled *programmatically*. This is important for reproducibility and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page: http://matplotlib.org/

To get started using Matplotlib in a Python program, either include the symbols from the `pylab` module (the easy way):

```
In [2]: from pylab import *
```

or import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

```
In [3]: import matplotlib.pyplot as plt
```

5.2 MATLAB-like API

The easiest way to get started with plotting using matplotlib is often to use the MATLAB-like API provided by matplotlib.

It is designed to be compatible with MATLAB's plotting functions, so it is easy to get started with if you are familiar with MATLAB.

To use this API from matplotlib, we need to include the symbols in the `pylab` module:

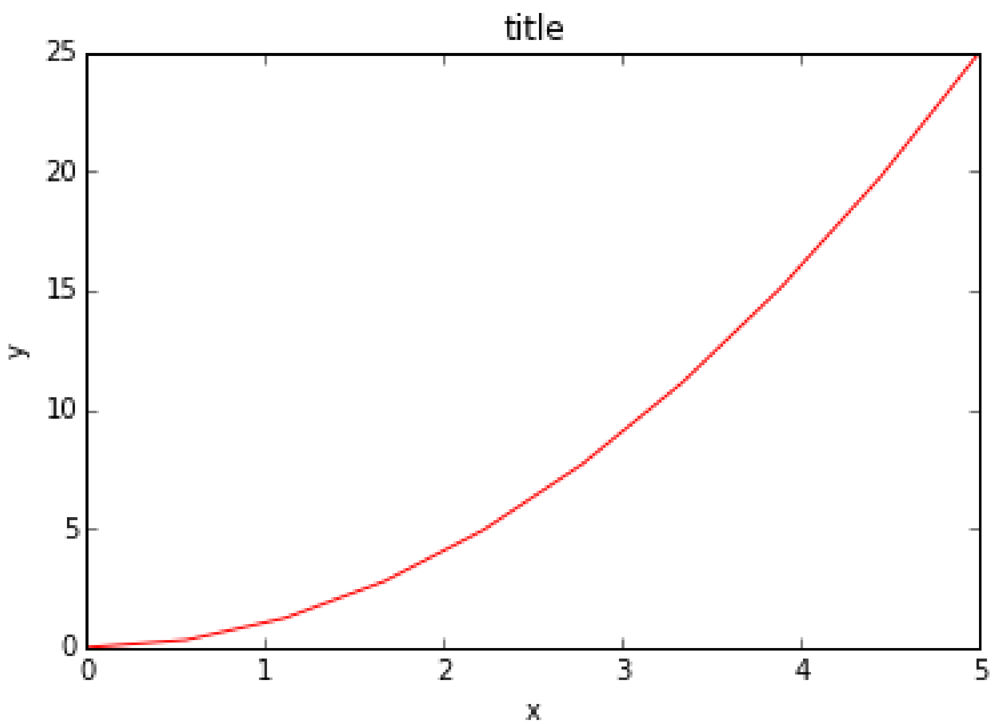
```
In [4]: from pylab import *
```

5.2.1 Example

A simple figure with MATLAB-like plotting API:

```
In [5]: x = linspace(0, 5, 10)
        y = x ** 2
```

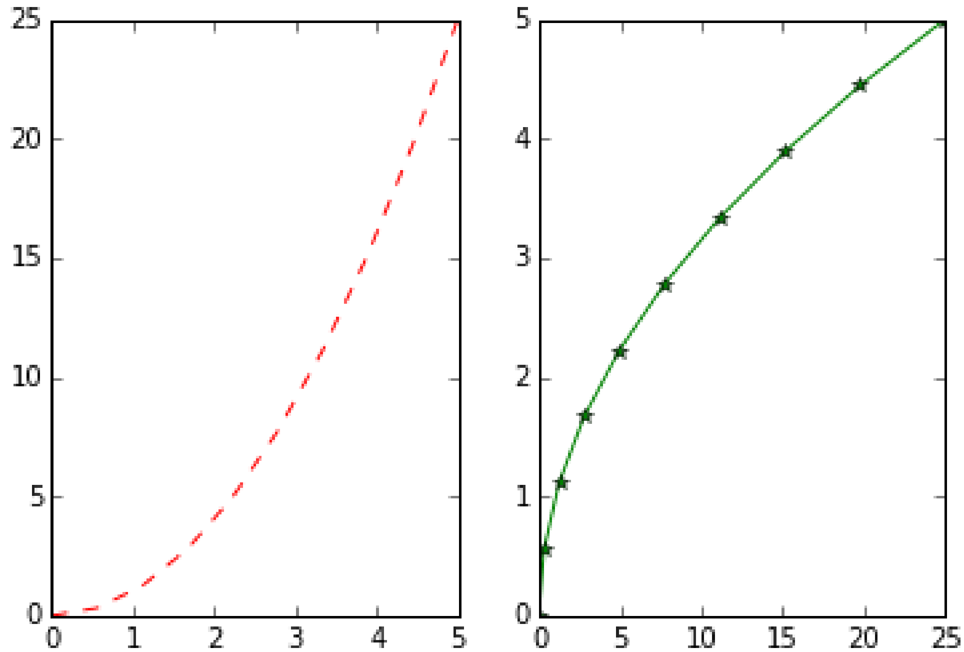
```
In [6]: figure()
        plot(x, y, 'r')
        xlabel('x')
        ylabel('y')
        title('title')
        show()
```



Most of the plotting related functions in MATLAB are covered by the `pylab` module. For example, subplot and color/symbol selection:

```
In [7]: subplot(1,2,1)
        plot(x, y, 'r--')
```

```
subplot(1,2,2)
plot(y, x, 'g*-');
```



The good thing about the pylab MATLAB-style API is that it is easy to get started with if you are familiar with MATLAB, and it has a minumum of coding overhead for simple plots.

However, I'd encourrage not using the MATLAB compatible API for anything but the simplest figures.

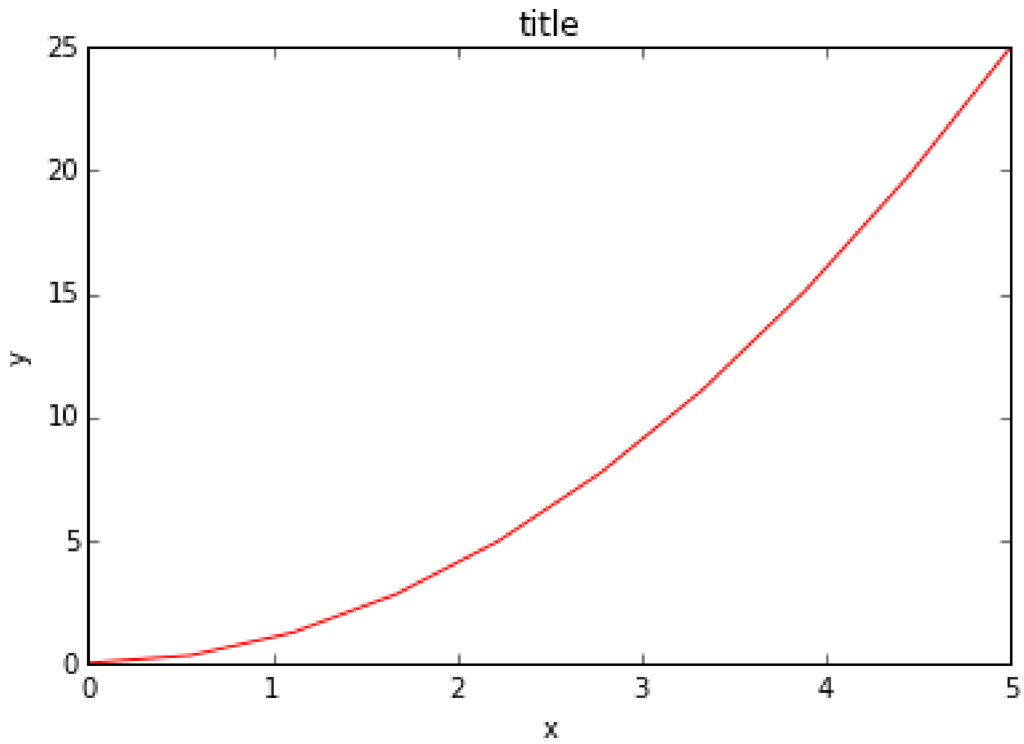
Instead, I recommend learning and using matplotlib's object-oriented plotting API. It is remarkably powerful. For advanced figures with subplots, insets and other components it is very nice to work with.

5.3 The matplotlib object-oriented API

The main idea with object-oriented programming is to have objects that one can apply functions and actions on, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

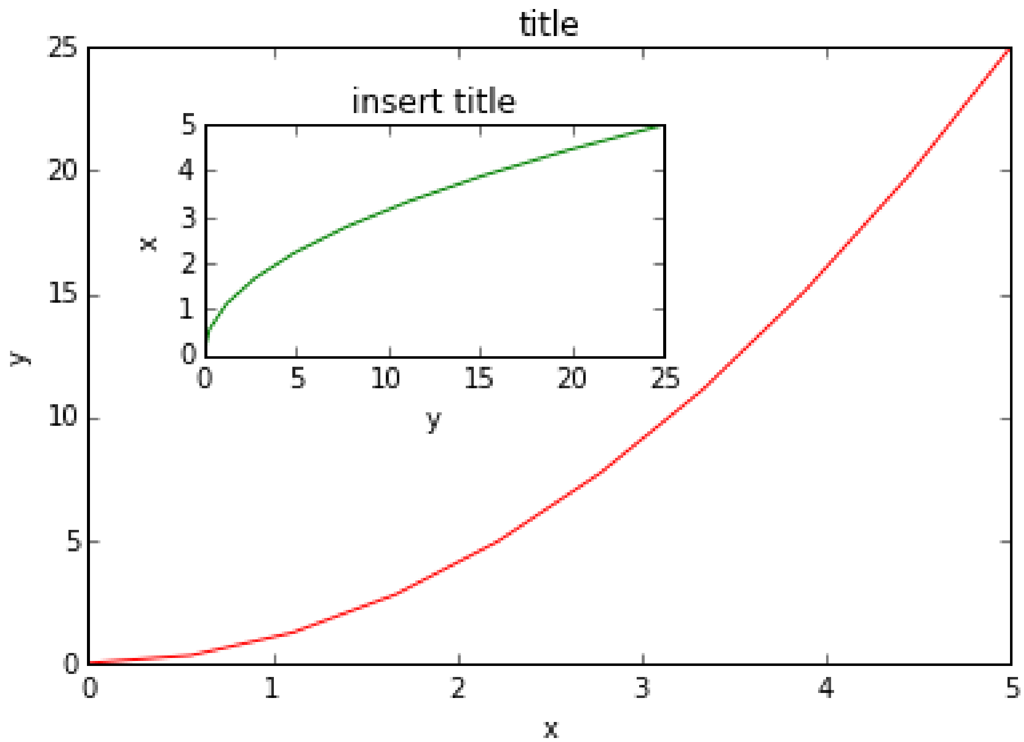
To use the object-oriented API we start out very much like in the previous example, but instead of creating a new global figure instance we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

```
In [8]: fig = plt.figure()

        axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)

        axes.plot(x, y, 'r')

        axes.set_xlabel('x')
        axes.set_ylabel('y')
        axes.set_title('title');
```



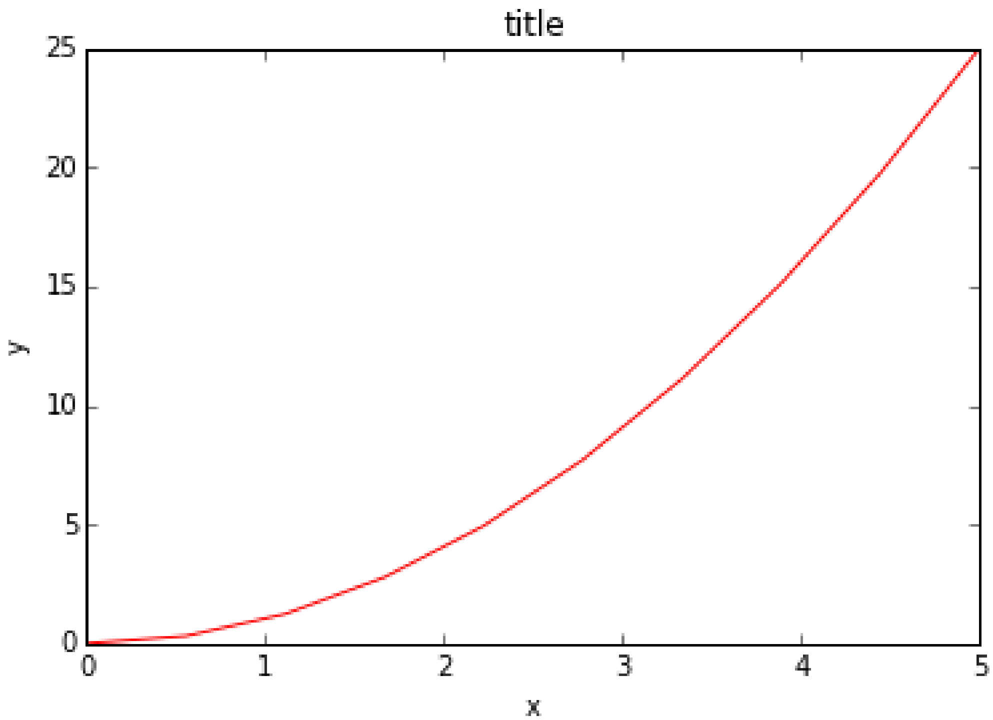
Although a little bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```
In [9]: fig = plt.figure()

        axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
        axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

        # main figure
        axes1.plot(x, y, 'r')
        axes1.set_xlabel('x')
        axes1.set_ylabel('y')
        axes1.set_title('title')

        # insert
        axes2.plot(y, x, 'g')
        axes2.set_xlabel('y')
        axes2.set_ylabel('x')
        axes2.set_title('insert title');
```

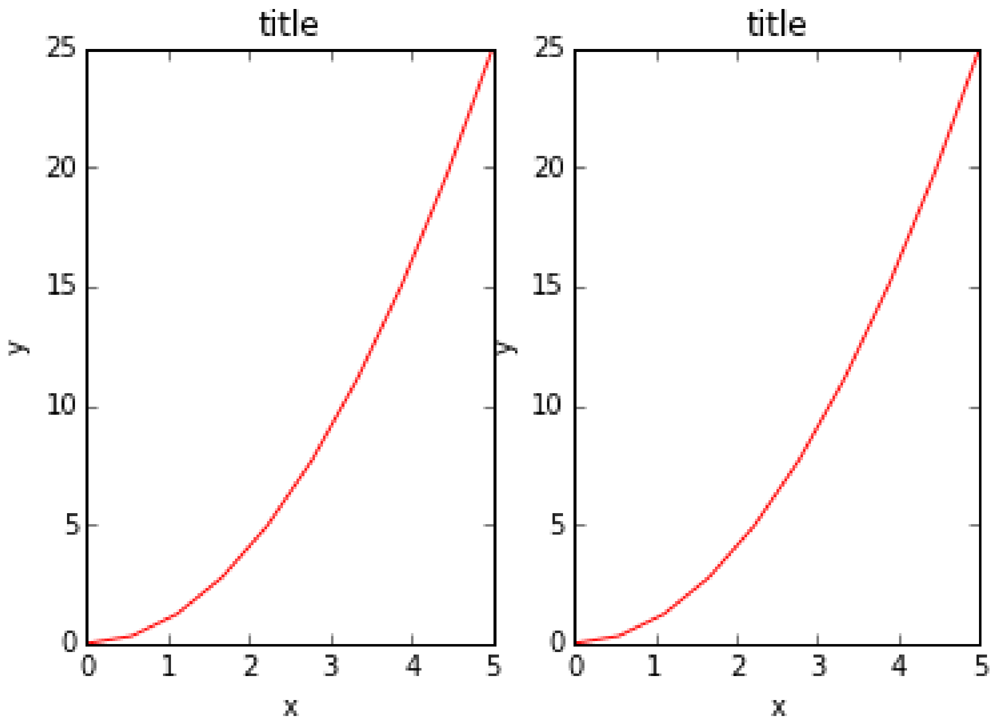


If we don't care about being explicit about where our plot axes are placed in the figure canvas, then we can use one of the many axis layout managers in matplotlib. My favorite is `subplots`, which can be used like this:

```
In [10]: fig, axes = plt.subplots()

         axes.plot(x, y, 'r')
         axes.set_xlabel('x')
         axes.set_ylabel('y')
         axes.set_title('title');
```



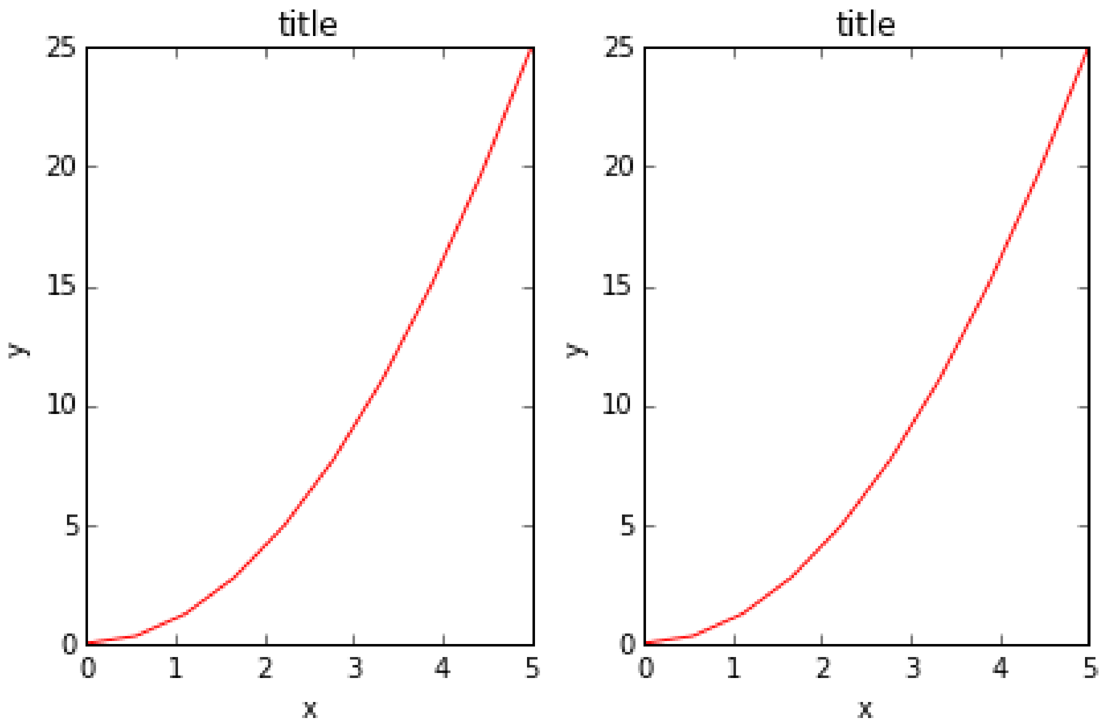

```
In [11]: fig, axes = plt.subplots(nrows=1, ncols=2)

         for ax in axes:
             ax.plot(x, y, 'r')
             ax.set_xlabel('x')
             ax.set_ylabel('y')
             ax.set_title('title')
```



That was easy, but it isn't so pretty with overlapping figure axes and labels, right?

We can deal with that by using the `fig.tight_layout` method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

```
In [12]: fig, axes = plt.subplots(nrows=1, ncols=2)

         for ax in axes:
             ax.plot(x, y, 'r')
             ax.set_xlabel('x')
             ax.set_ylabel('y')
             ax.set_title('title')

         fig.tight_layout()
```



5.3.1 Figure size, aspect ratio and DPI

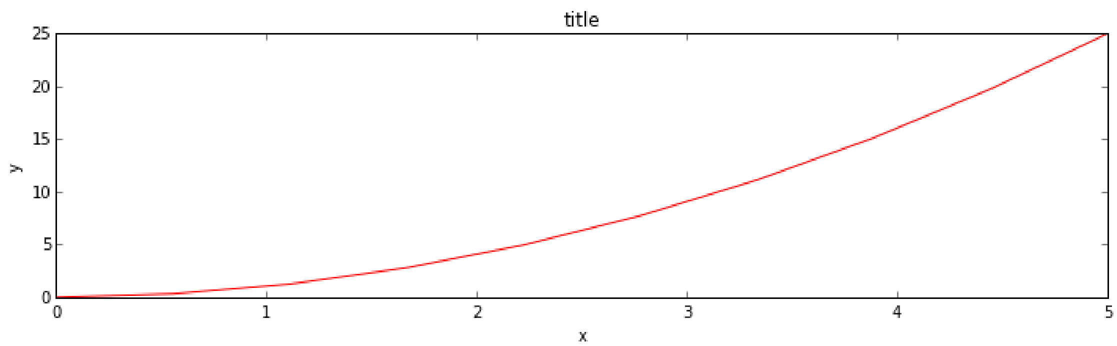
Matplotlib allows the aspect ratio, DPI and figure size to be specified when the `Figure` object is created, using the `figsize` and `dpi` keyword arguments. `figsize` is a tuple of the width and height of the figure in inches, and `dpi` is the dots-per-inch (pixel per inch). To create an 800x400 pixel, 100 dots-per-inch figure, we can do:

```
In [13]: fig = plt.figure(figsize=(8,4), dpi=100)
```

```
<matplotlib.figure.Figure at 0x109225d68>
```

The same arguments can also be passed to layout managers, such as the `subplots` function:

```
In [14]: fig, axes = plt.subplots(figsize=(12,3))

         axes.plot(x, y, 'r')
         axes.set_xlabel('x')
         axes.set_ylabel('y')
         axes.set_title('title');
```



5.3.2 Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class:

```
In [15]: fig.savefig("filename.png")
```

Here we can also optionally specify the DPI and choose between different output formats:

```
In [16]: fig.savefig("filename.png", dpi=200)
```

What formats are available and which ones should be used for best quality? Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. For scientific papers, I recommend using PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command). In some cases, PGF can also be good alternative.

5.3.3 Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

```
In [17]: ax.set_title("title");
```

Axis labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

```
In [18]: ax.set_xlabel("x")
         ax.set_ylabel("y");
```

Legends

Legends for curves in a figure can be added in two ways. One method is to use the `legend` method of the axis object and pass a list/tuple of legend texts for the previously defined curves:

```
In [19]: ax.legend(["curve1", "curve2", "curve3"]);
```

The method described above follows the MATLAB API. It is somewhat prone to errors and unflexible if curves are added to or removed from the figure (resulting in a wrongly labelled curve).

A better method is to use the `label="label text"` keyword argument when plots or other objects are added to the figure, and then using the `legend` method without arguments to add the legend to the figure:

```
In [20]: ax.plot(x, x**2, label="curve1")
         ax.plot(x, x**3, label="curve2")
         ax.legend();
```

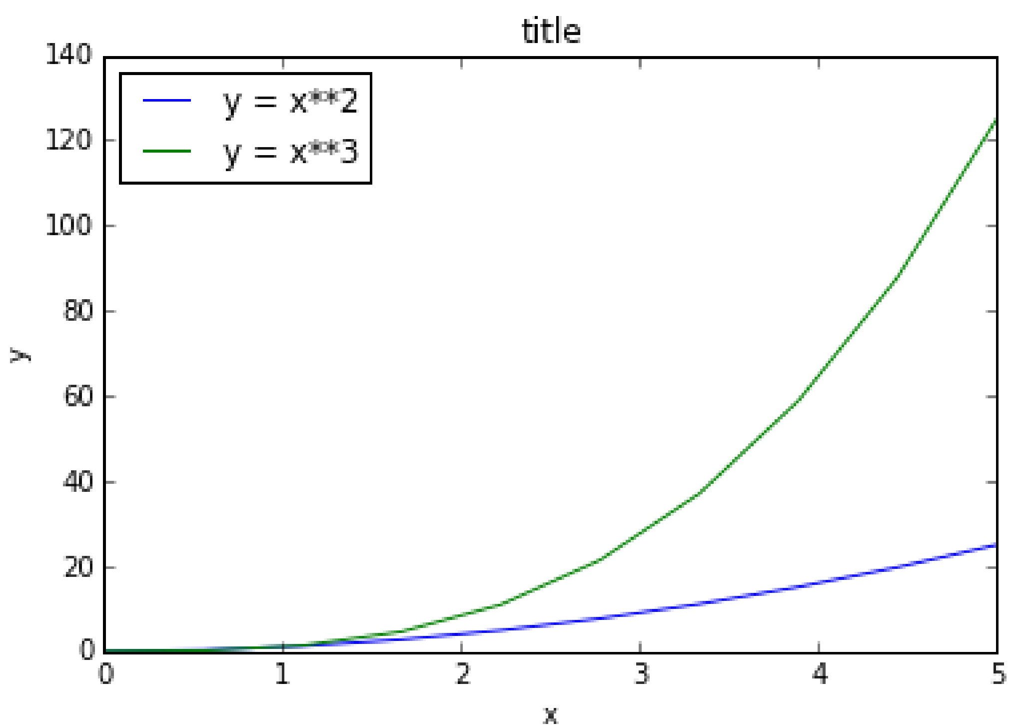
The advantage with this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly.

The `legend` function takes an optional keyword argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of `loc` are numerical codes for the various places the legend can be drawn. See http://matplotlib.org/users/legend_guide.html#legend-location for details. Some of the most common `loc` values are:

```
In [21]: ax.legend(loc=0) # let matplotlib decide the optimal location
         ax.legend(loc=1) # upper right corner
         ax.legend(loc=2) # upper left corner
         ax.legend(loc=3) # lower left corner
         ax.legend(loc=4) # lower right corner
         # .. many more options are available

Out[21]: <matplotlib.legend.Legend at 0x107a4ccc0>
```

The following figure shows how to use the figure title, axis labels and legends described above:

```
In [22]: fig, ax = plt.subplots()

         ax.plot(x, x**2, label="y = x**2")
         ax.plot(x, x**3, label="y = x**3")
         ax.legend(loc=2); # upper left corner
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_title('title');
```



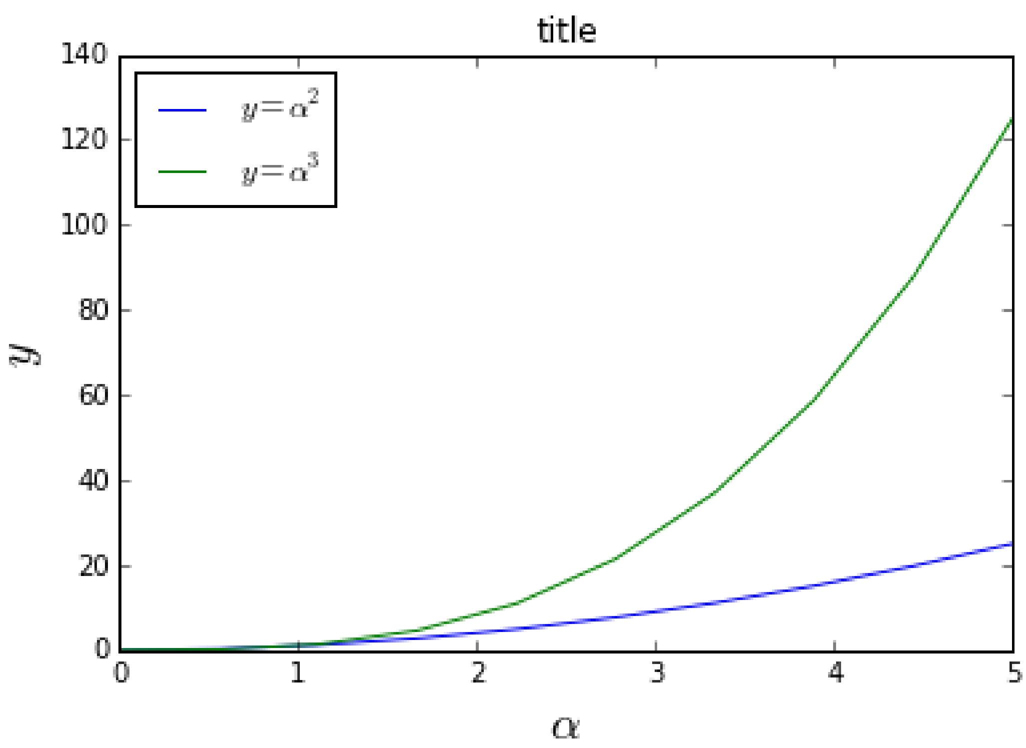
5.3.4 Formatting text: LaTeX, fontsize, font family

The figure above is functional, but it does not (yet) satisfy the criteria for a figure used in a publication. First and foremost, we need to have LaTeX formatted text, and second, we need to be able to adjust the font size to appear right in a publication.

Matplotlib has great support for LaTeX. All we need to do is to use dollar signs encapsulate LaTeX in any text (legend, title, label, etc.). For example, `"$y=x^3$"`.

But here we can run into a slightly subtle problem with LaTeX code and Python text strings. In LaTeX, we frequently use the backslash in commands, for example `\alpha` to produce the symbol α. But the backslash already has a meaning in Python strings (the escape code character). To avoid Python messing up our latex code, we need to use "raw" text strings. Raw text strings are prepended with an 'r', like `r"\alpha"` or `r'\alpha'` instead of `"\alpha"` or `'\alpha'`:

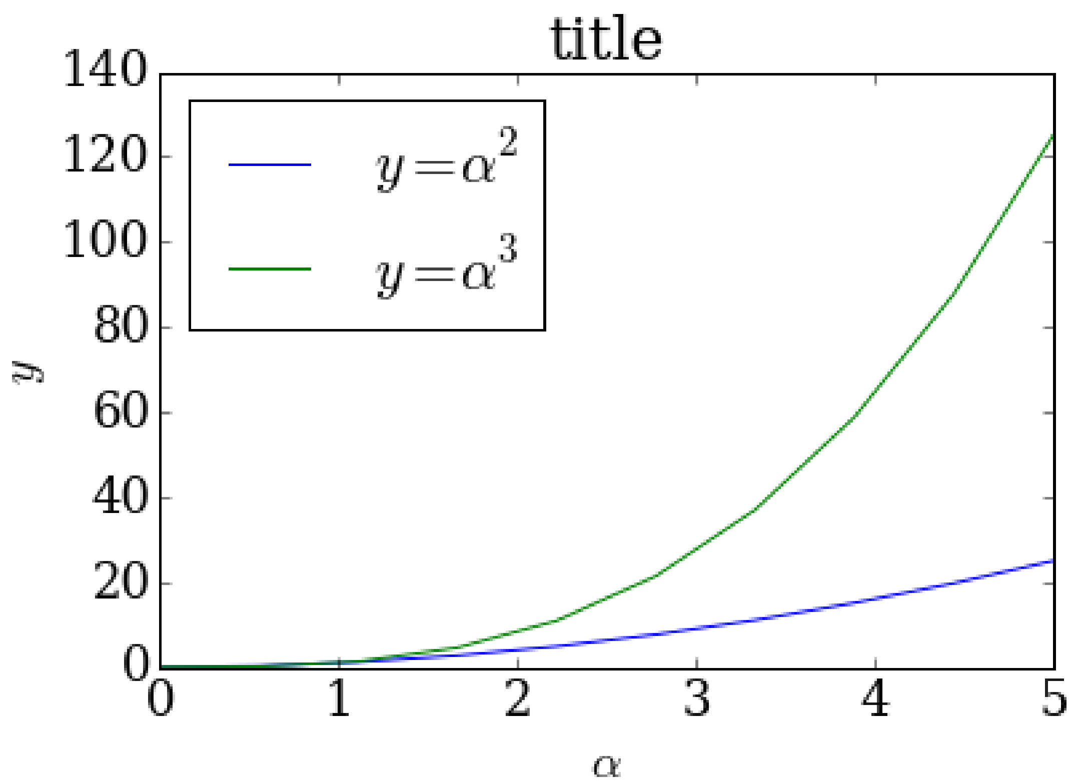
```
In [23]: fig, ax = plt.subplots()

         ax.plot(x, x**2, label=r"$y = \alpha^2$")
         ax.plot(x, x**3, label=r"$y = \alpha^3$")
         ax.legend(loc=2) # upper left corner
         ax.set_xlabel(r'$\alpha$', fontsize=18)
         ax.set_ylabel(r'$y$', fontsize=18)
         ax.set_title('title');
```



We can also change the global font size and font family, which applies to all text elements in a figure (tick labels, axis labels and titles, legends, etc.):

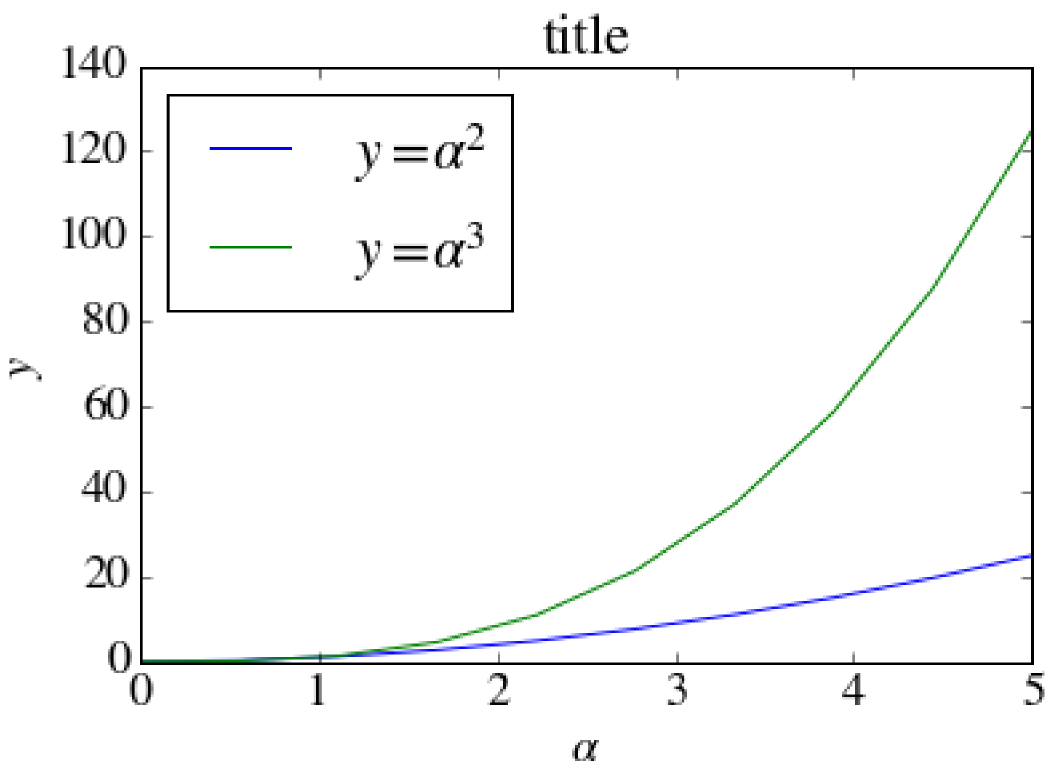
```
In [24]: # Update the matplotlib configuration parameters:
         matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

```
In [25]: fig, ax = plt.subplots()

         ax.plot(x, x**2, label=r"$y = \alpha^2$")
         ax.plot(x, x**3, label=r"$y = \alpha^3$")
         ax.legend(loc=2) # upper left corner
         ax.set_xlabel(r'$\alpha$')
         ax.set_ylabel(r'$y$')
         ax.set_title('title');
```



A good choice of global fonts are the STIX fonts:

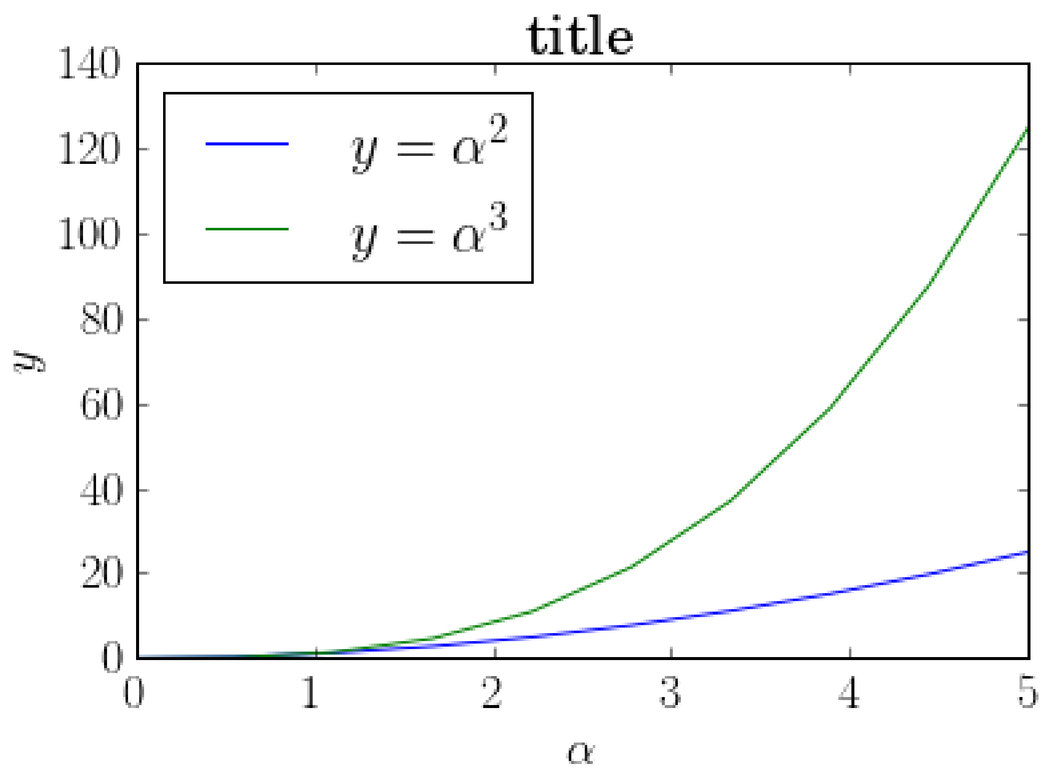
```
In [26]: # Update the matplotlib configuration parameters:
         matplotlib.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.fontset':

In [27]: fig, ax = plt.subplots()

         ax.plot(x, x**2, label=r"$y = \alpha^2$")
         ax.plot(x, x**3, label=r"$y = \alpha^3$")
         ax.legend(loc=2) # upper left corner
         ax.set_xlabel(r'$\alpha$')
         ax.set_ylabel(r'$y$')
         ax.set_title('title');
```



Or, alternatively, we can request that matplotlib uses LaTeX to render the text elements in the figure:

```
In [28]: matplotlib.rcParams.update({'font.size': 18, 'text.usetex': True})
```

```
In [29]: fig, ax = plt.subplots()

         ax.plot(x, x**2, label=r"$y = \alpha^2$")
         ax.plot(x, x**3, label=r"$y = \alpha^3$")
         ax.legend(loc=2) # upper left corner
         ax.set_xlabel(r'$\alpha$')
         ax.set_ylabel(r'$y$')
         ax.set_title('title');
```




```
In [30]: # restore
         matplotlib.rcParams.update({'font.size': 12, 'font.family': 'sans', 'text.usetex': False})
```

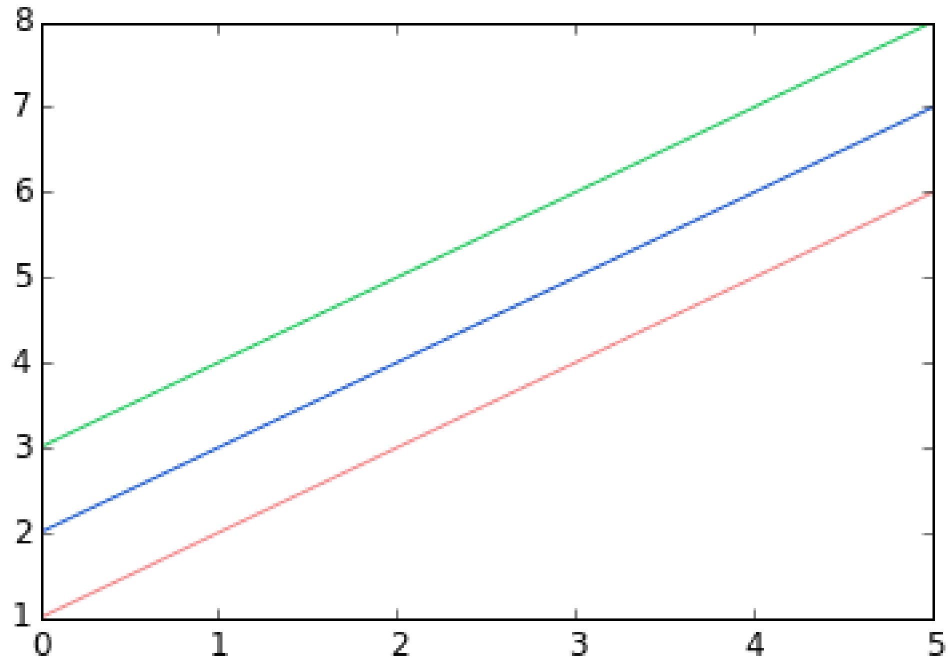
5.3.5 Setting colors, linewidths, linetypes

Colors With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

```
In [31]: # MATLAB style line color and style
         ax.plot(x, x**2, 'b.-') # blue line with dots
         ax.plot(x, x**3, 'g--') # green dashed line

Out[31]: [<matplotlib.lines.Line2D at 0x1090691d0>]
```

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments:

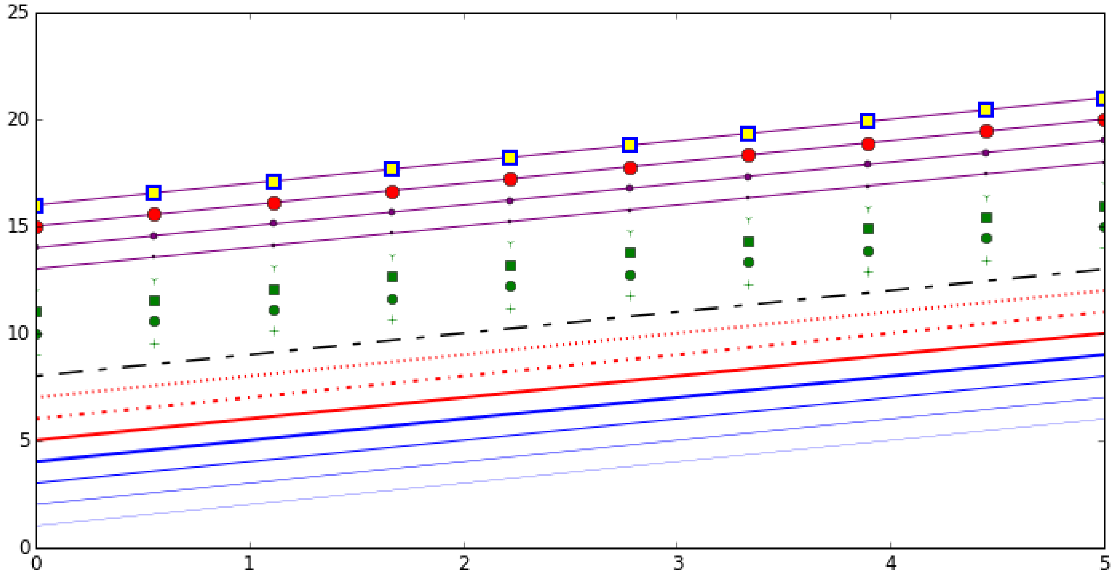
```
In [32]: fig, ax = plt.subplots()

         ax.plot(x, x+1, color="red", alpha=0.5) # half-transparant red
         ax.plot(x, x+2, color="#1155dd")        # RGB hex code for a bluish color
         ax.plot(x, x+3, color="#15cc55")        # RGB hex code for a greenish color

Out[32]: [<matplotlib.lines.Line2D at 0x10920a1d0>]
```



Line and marker styles To change the line width, we can use the `linewidth` or `lw` keyword argument. The line style can be selected using the `linestyle` or `ls` keyword arguments:

```
In [33]: fig, ax = plt.subplots(figsize=(12,6))

         ax.plot(x, x+1, color="blue", linewidth=0.25)
         ax.plot(x, x+2, color="blue", linewidth=0.50)
         ax.plot(x, x+3, color="blue", linewidth=1.00)
         ax.plot(x, x+4, color="blue", linewidth=2.00)

         # possible linestype options ‘-‘, ‘{’, ‘-.’, ‘:’, ‘steps’
         ax.plot(x, x+5, color="red", lw=2, linestyle=’-’)
         ax.plot(x, x+6, color="red", lw=2, ls=’-.’)
         ax.plot(x, x+7, color="red", lw=2, ls=’:’)

         # custom dash
         line, = ax.plot(x, x+8, color="black", lw=1.50)
         line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

         # possible marker symbols: marker = ’+’, ’o’, ’*’, ’s’, ’,’, ’.’, ’1’, ’2’, ’3’, ’4’, ...
         ax.plot(x, x+ 9, color="green", lw=2, ls=’*’, marker=’+’)
         ax.plot(x, x+10, color="green", lw=2, ls=’*’, marker=’o’)
         ax.plot(x, x+11, color="green", lw=2, ls=’*’, marker=’s’)
         ax.plot(x, x+12, color="green", lw=2, ls=’*’, marker=’1’)

         # marker size and color
         ax.plot(x, x+13, color="purple", lw=1, ls=’-’, marker=’o’, markersize=2)
         ax.plot(x, x+14, color="purple", lw=1, ls=’-’, marker=’o’, markersize=4)
```

```
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue");
```

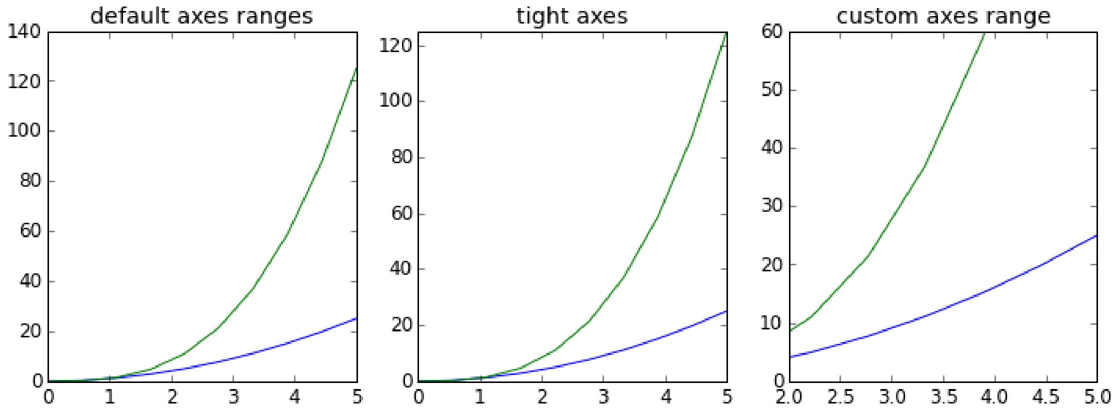


5.3.6 Control over axis appearance

The appearance of the axes is an important aspect of a figure that we often need to modify to make a publication quality graphics. We need to be able to control where the ticks and labels are placed, modify the font size and possibly the labels used on the axes. In this section we will look at controling those properties in a matplotlib figure.

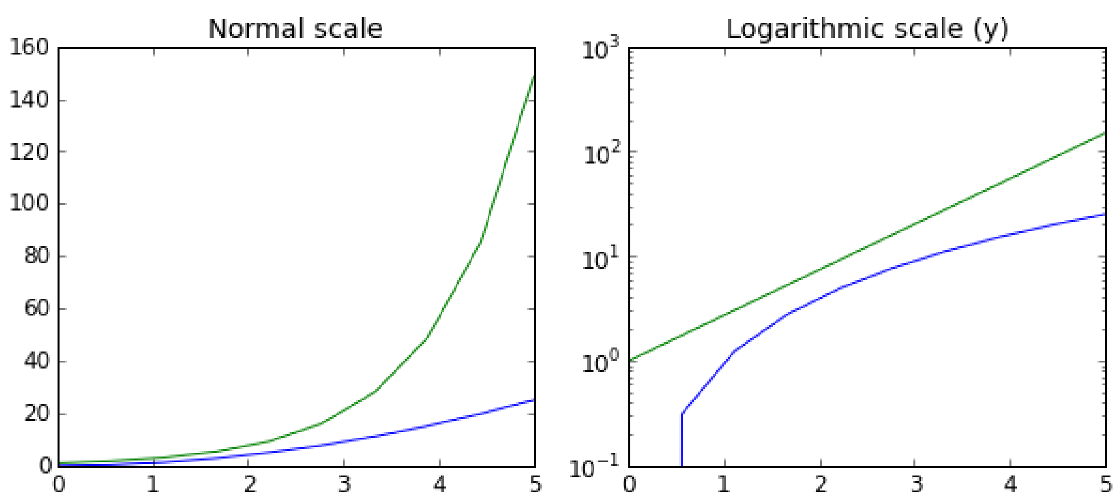
Plot range The first thing we might want to configure is the ranges of the axes. We can do this using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatrically getting "tightly fitted" axes ranges:

```
In [34]: fig, axes = plt.subplots(1, 3, figsize=(12, 4))

         axes[0].plot(x, x**2, x, x**3)
         axes[0].set_title("default axes ranges")

         axes[1].plot(x, x**2, x, x**3)
         axes[1].axis('tight')
         axes[1].set_title("tight axes")

         axes[2].plot(x, x**2, x, x**3)
         axes[2].set_ylim([0, 60])
         axes[2].set_xlim([2, 5])
         axes[2].set_title("custom axes range");
```



Logarithmic scale It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

```
In [35]: fig, axes = plt.subplots(1, 2, figsize=(10,4))

         axes[0].plot(x, x**2, x, exp(x))
         axes[0].set_title("Normal scale")

         axes[1].plot(x, x**2, x, exp(x))
         axes[1].set_yscale("log")
         axes[1].set_title("Logarithmic scale (y)");
```

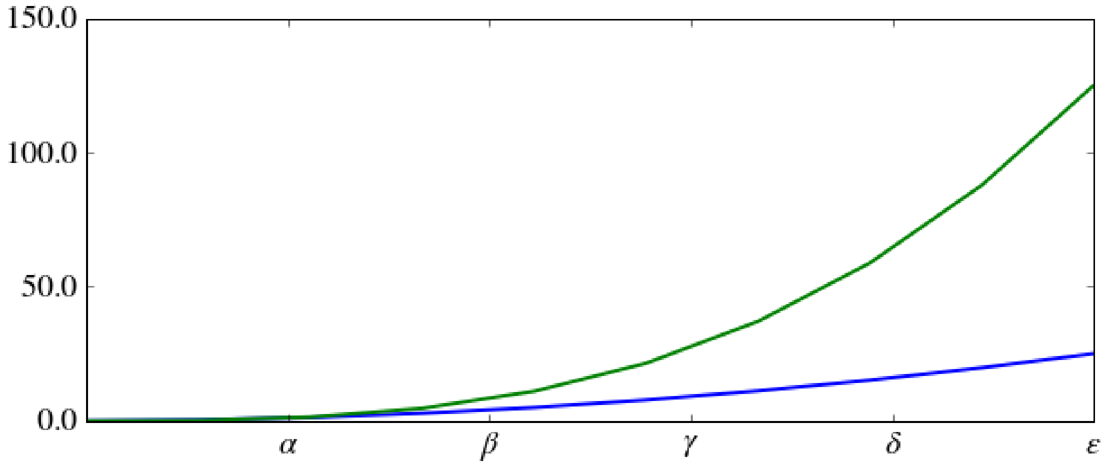


5.3.7 Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

```
In [36]: fig, ax = plt.subplots(figsize=(10, 4))

         ax.plot(x, x**2, x, x**3, lw=2)

         ax.set_xticks([1, 2, 3, 4, 5])
         ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$', r'$\epsilon$'], fontsize

         yticks = [0, 50, 100, 150]
         ax.set_yticks(yticks)
         ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX formatted labels
```

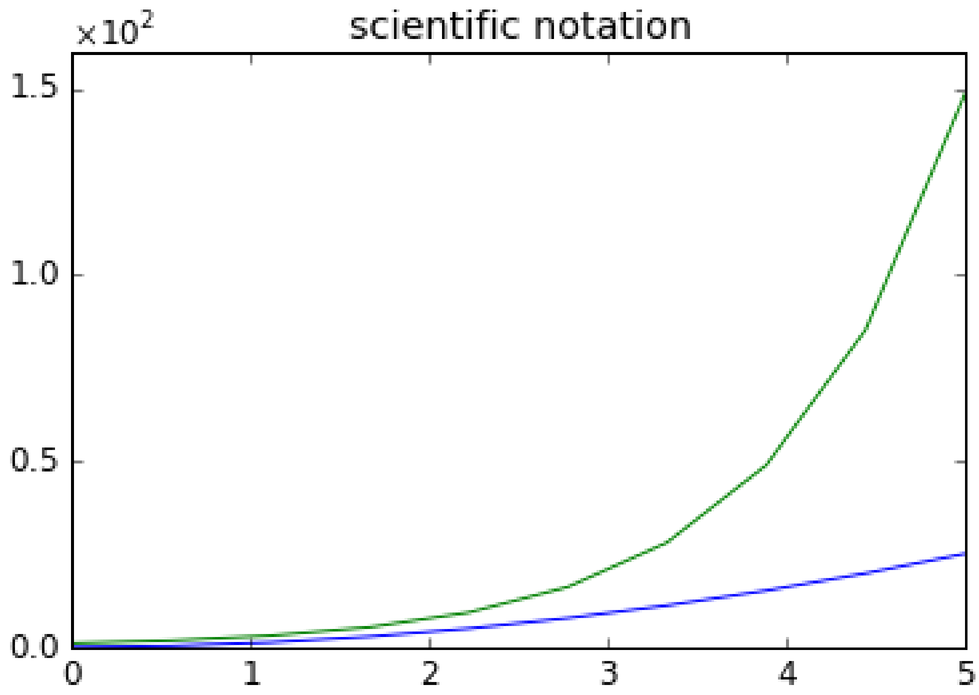
```
Out[36]: [<matplotlib.text.Text at 0x107e76940>,
          <matplotlib.text.Text at 0x107e84438>,
          <matplotlib.text.Text at 0x107e740b8>,
          <matplotlib.text.Text at 0x107e609b0>]
```



There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

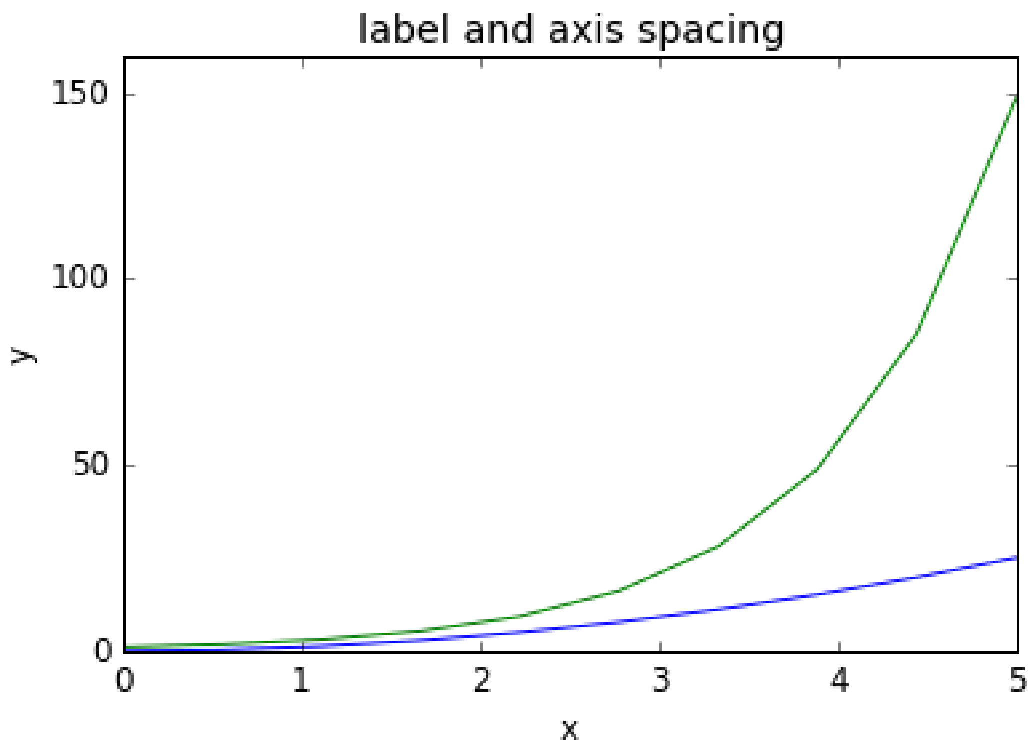
Scientific notation With large numbers on axes, it is often better use scientific notation:

```
In [37]: fig, ax = plt.subplots(1, 1)

         ax.plot(x, x**2, x, exp(x))
         ax.set_title("scientific notation")

         ax.set_yticks([0, 50, 100, 150])

         from matplotlib import ticker
         formatter = ticker.ScalarFormatter(useMathText=True)
         formatter.set_scientific(True)
         formatter.set_powerlimits((-1,1))
         ax.yaxis.set_major_formatter(formatter)
```



5.3.8 Axis number and axis label spacing

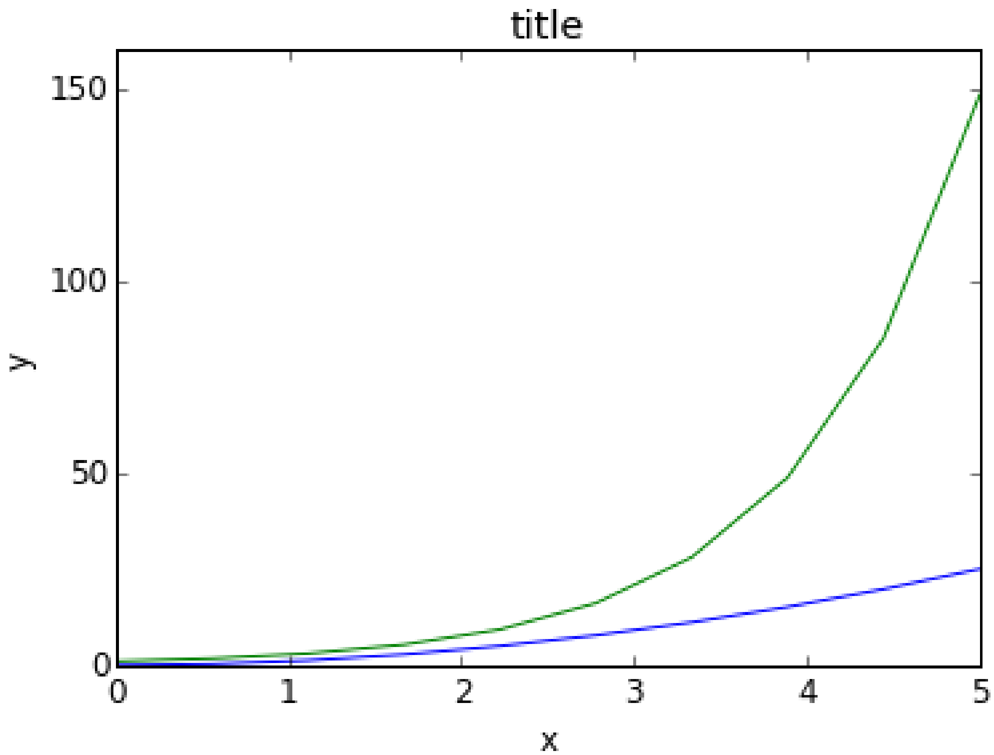
In [38]:
```
# distance between x and y axis and the numbers on the axes
rcParams['xtick.major.pad'] = 5
rcParams['ytick.major.pad'] = 5

fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```




```
In [39]: # restore defaults
         rcParams['xtick.major.pad'] = 3
         rcParams['ytick.major.pad'] = 3
```

Axis position adjustments Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust`:

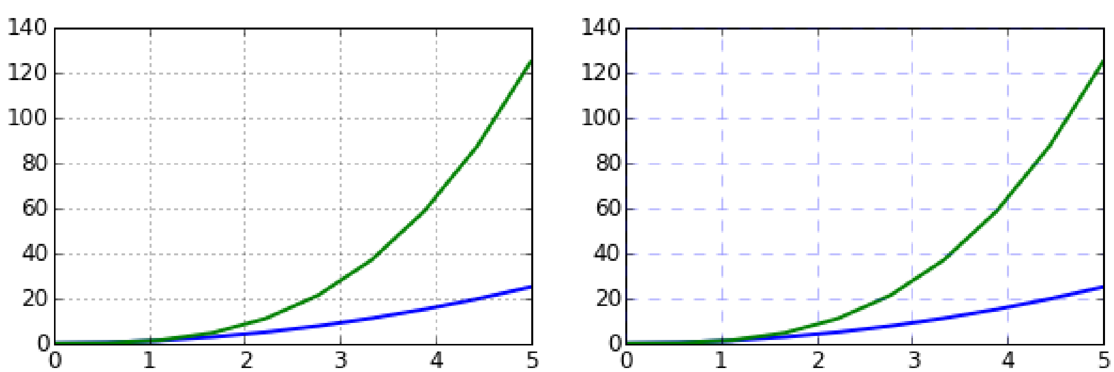
```
In [40]: fig, ax = plt.subplots(1, 1)

         ax.plot(x, x**2, x, exp(x))
         ax.set_yticks([0, 50, 100, 150])

         ax.set_title("title")
         ax.set_xlabel("x")
         ax.set_ylabel("y")

         fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



5.3.9 Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

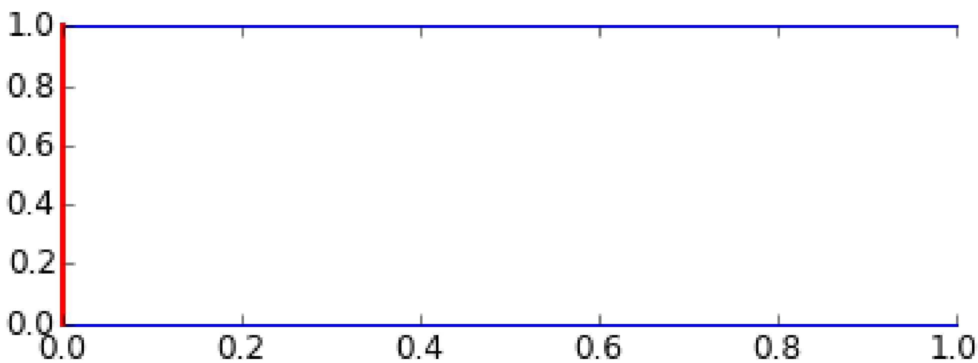
```
In [41]: fig, axes = plt.subplots(1, 2, figsize=(10,3))

         # default grid appearance
         axes[0].plot(x, x**2, x, x**3, lw=2)
         axes[0].grid(True)

         # custom grid appearance
         axes[1].plot(x, x**2, x, x**3, lw=2)
         axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```



5.3.10 Axis spines

We can also change the properties of axis spines:

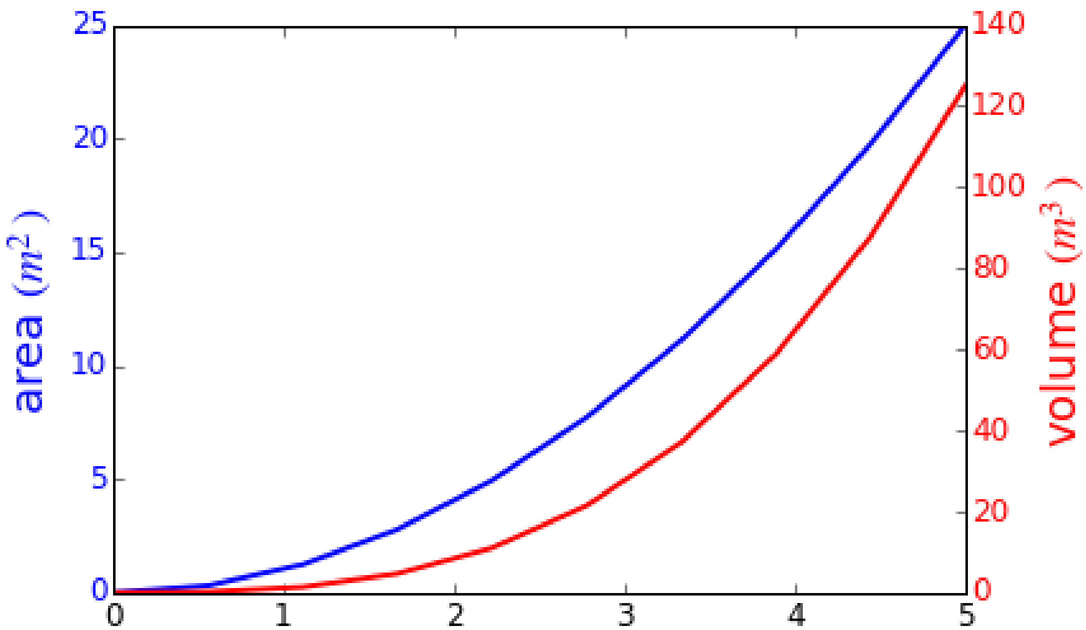
In [42]:
```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```



5.3.11 Twin axes

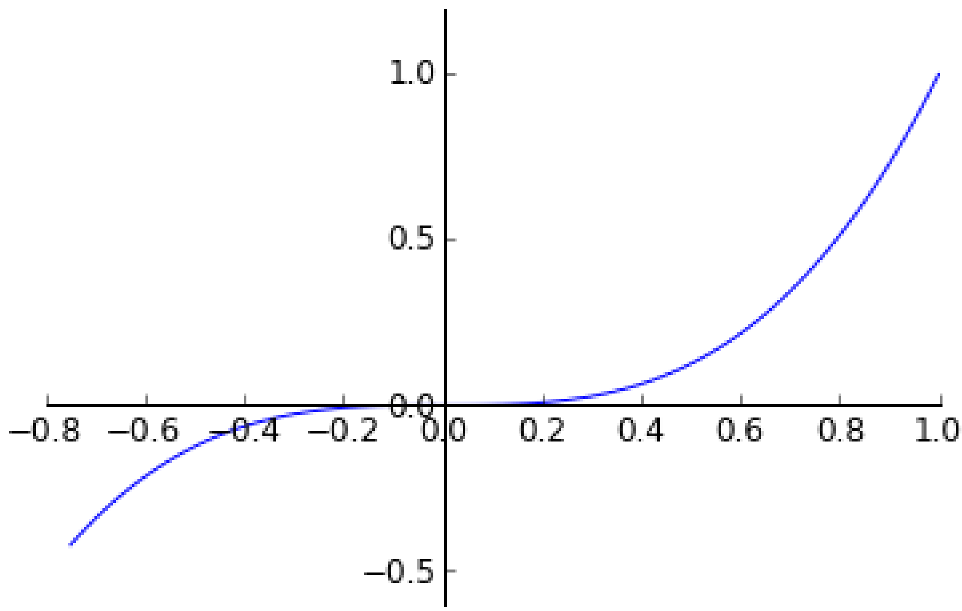
Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the `twinx` and `twiny` functions:

In [43]:
```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



5.3.12 Axes where x and y is zero

In [44]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

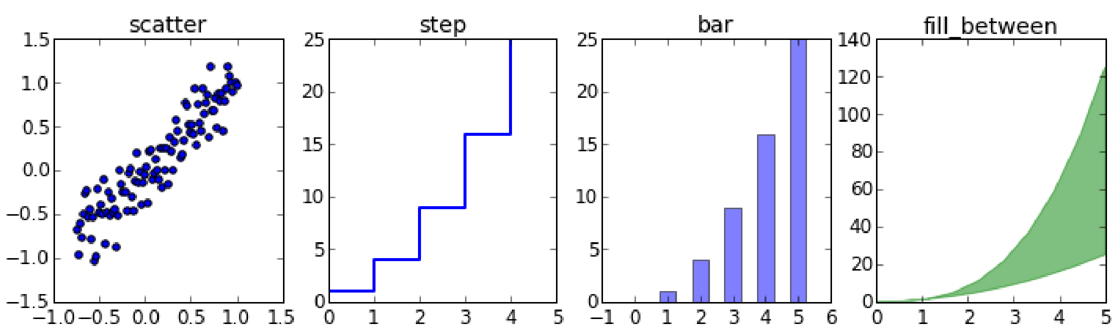


5.3.13 Other 2D plot styles

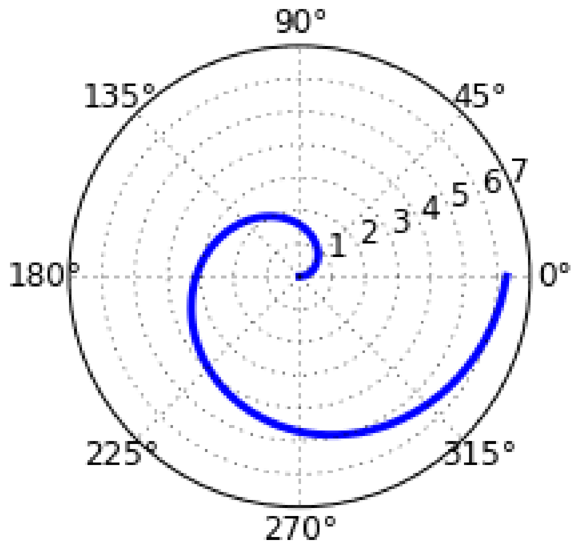
In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

```
In [45]: n = array([0,1,2,3,4,5])
```

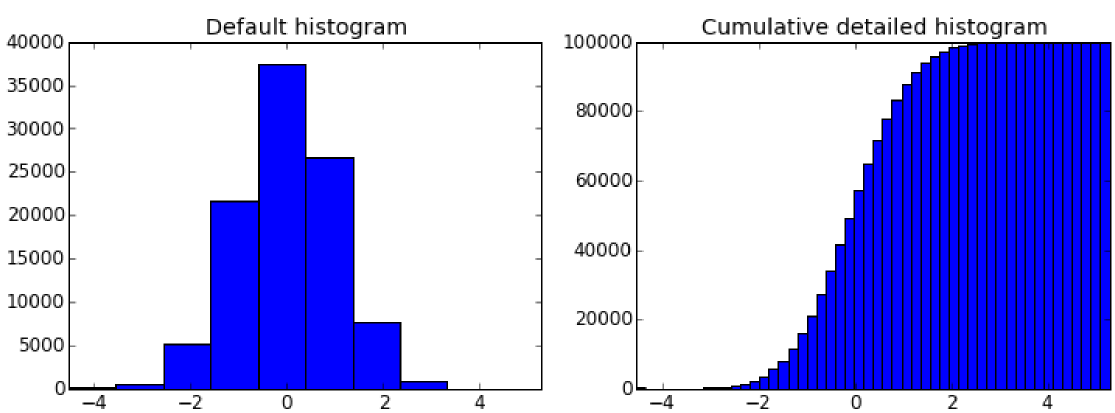
```
In [46]: fig, axes = plt.subplots(1, 4, figsize=(12,3))

         axes[0].scatter(xx, xx + 0.25*randn(len(xx)))
         axes[0].set_title("scatter")

         axes[1].step(n, n**2, lw=2)
         axes[1].set_title("step")

         axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
         axes[2].set_title("bar")

         axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
         axes[3].set_title("fill_between");
```



In [47]:
```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = linspace(0, 2 * pi, 100)
ax.plot(t, t, color='blue', lw=3);
```



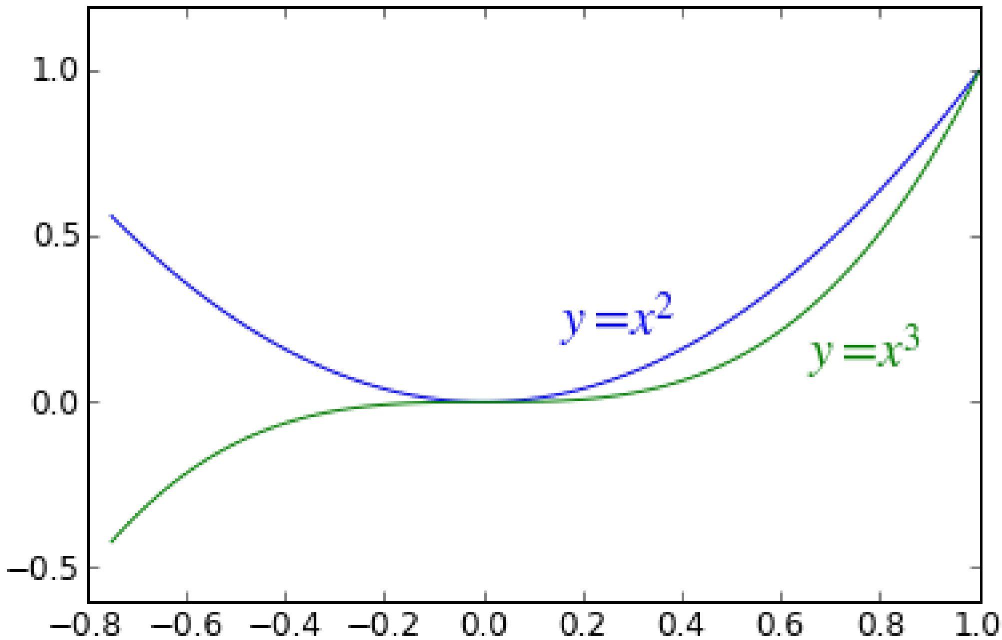
In [48]:
```
# A histogram
n = np.random.randn(100000)
fig, axes = plt.subplots(1, 2, figsize=(12,4))

axes[0].hist(n)
axes[0].set_title("Default histogram")
axes[0].set_xlim((min(n), max(n)))

axes[1].hist(n, cumulative=True, bins=50)
axes[1].set_title("Cumulative detailed histogram")
axes[1].set_xlim((min(n), max(n)));
```



5.3.14 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

```
In [49]: fig, ax = plt.subplots()

         ax.plot(xx, xx**2, xx, xx**3)

         ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
         ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

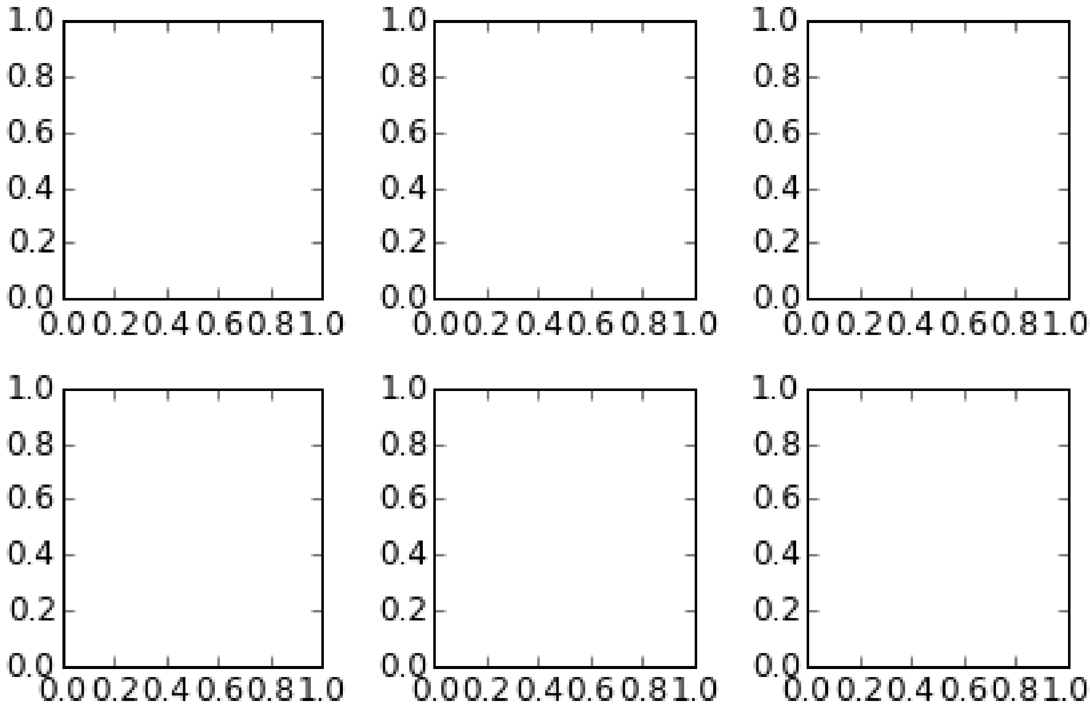


5.3.15 Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

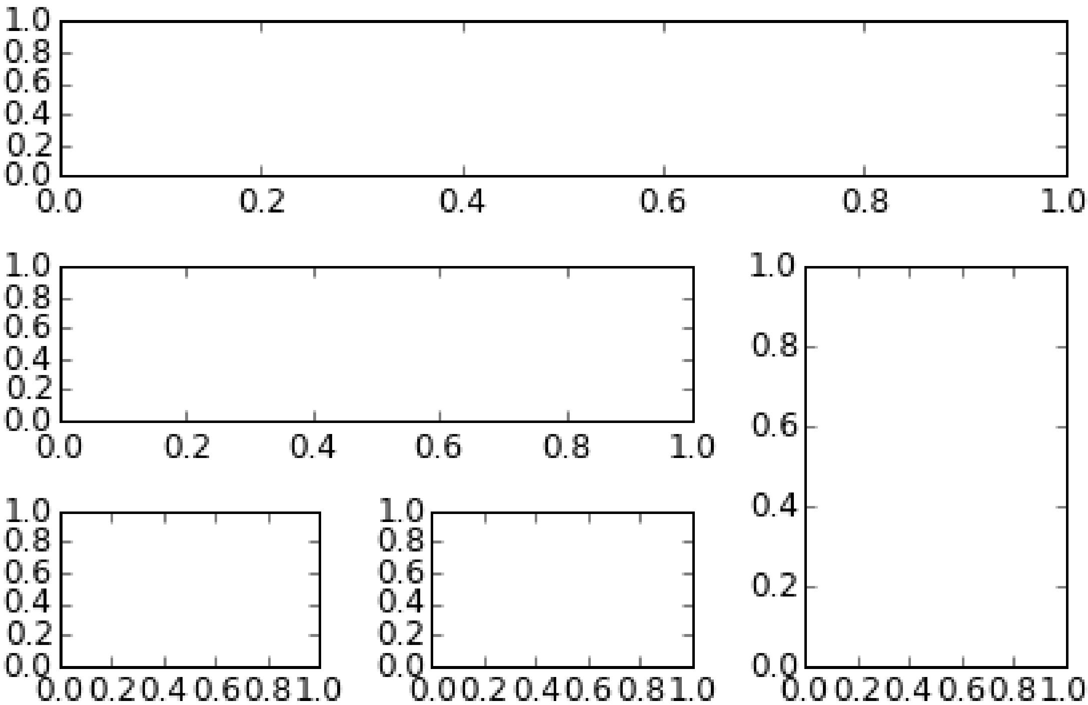
subplots

```
In [50]: fig, ax = plt.subplots(2, 3)
         fig.tight_layout()
```



subplot2grid

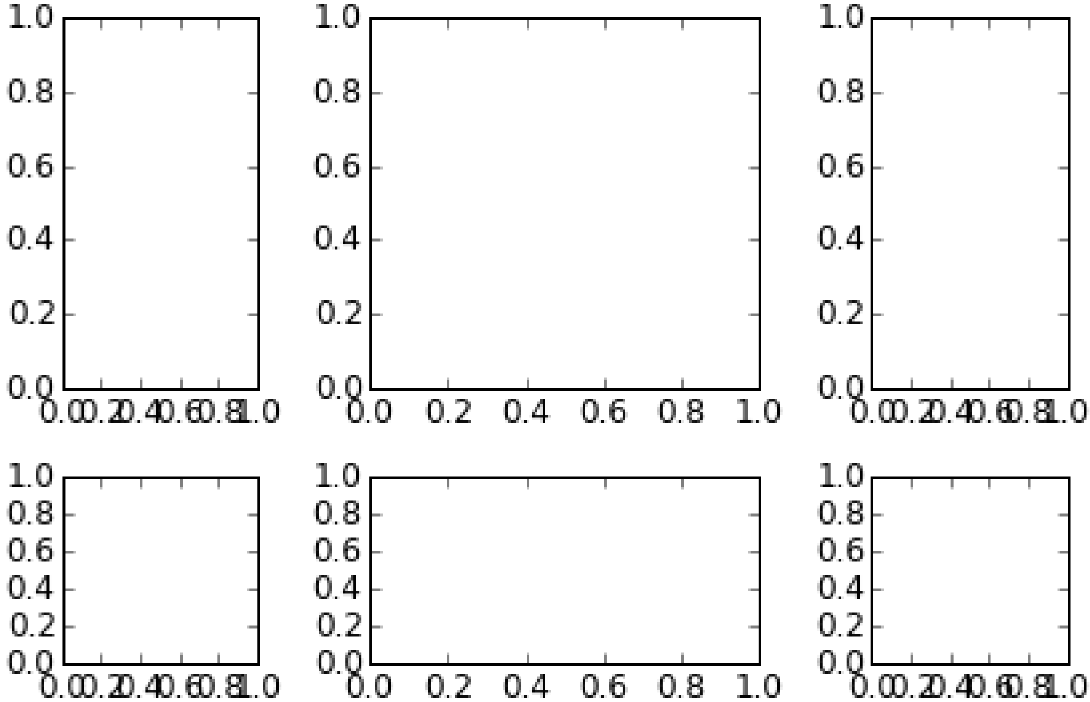
```
In [51]: fig = plt.figure()
         ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
         ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
         ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
         ax4 = plt.subplot2grid((3,3), (2,0))
         ax5 = plt.subplot2grid((3,3), (2,1))
         fig.tight_layout()
```



gridspec

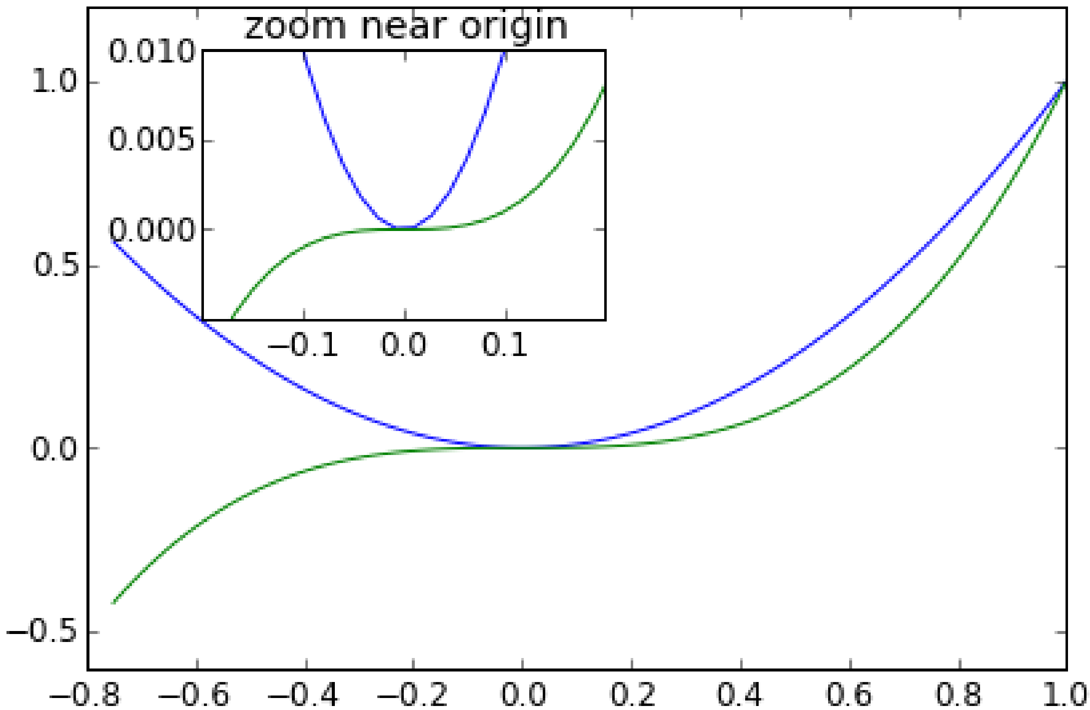
```
In [52]: import matplotlib.gridspec as gridspec
```

```
In [53]: fig = plt.figure()

         gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
         for g in gs:
             ax = fig.add_subplot(g)

         fig.tight_layout()
```



add_axes Manually adding axes with `add_axes` is useful for adding insets to figures:

```
In [54]: fig, ax = plt.subplots()

         ax.plot(xx, xx**2, xx, xx**3)
         fig.tight_layout()

         # inset
         inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

         inset_ax.plot(xx, xx**2, xx, xx**3)
         inset_ax.set_title('zoom near origin')

         # set axis range
         inset_ax.set_xlim(-.2, .2)
         inset_ax.set_ylim(-.005, .01)

         # set axis tick locations
         inset_ax.set_yticks([0, 0.005, 0.01])
         inset_ax.set_xticks([-0.1,0,.1]);
```



5.3.16 Colormap and contour figures

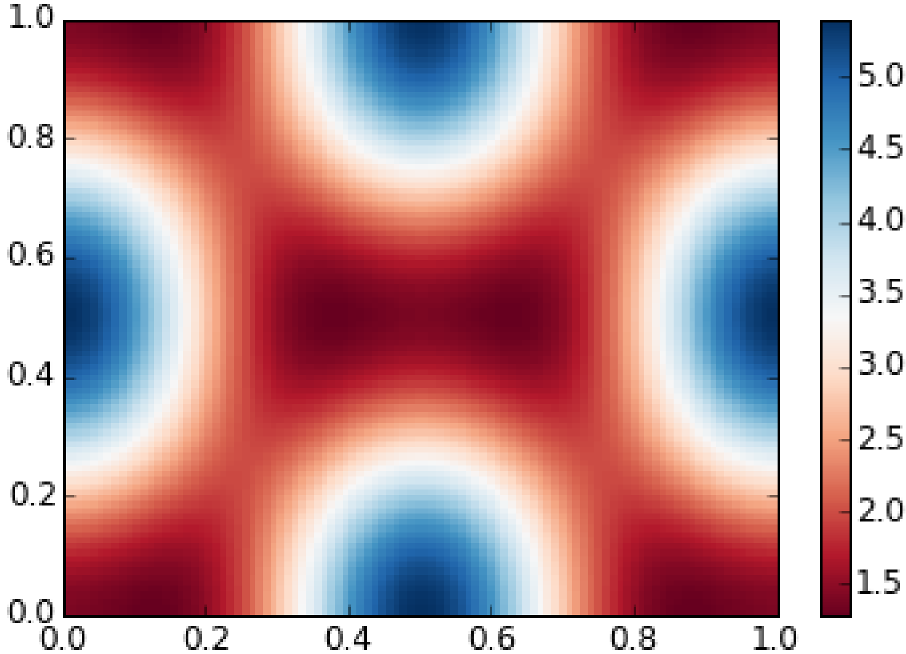
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

In [55]:
```
alpha = 0.7
phi_ext = 2 * pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * cos(phi_p)*cos(phi_m) - alpha * cos(phi_ext - 2*phi_p)
```

In [56]:
```
phi_m = linspace(0, 2*pi, 100)
phi_p = linspace(0, 2*pi, 100)
X,Y = meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

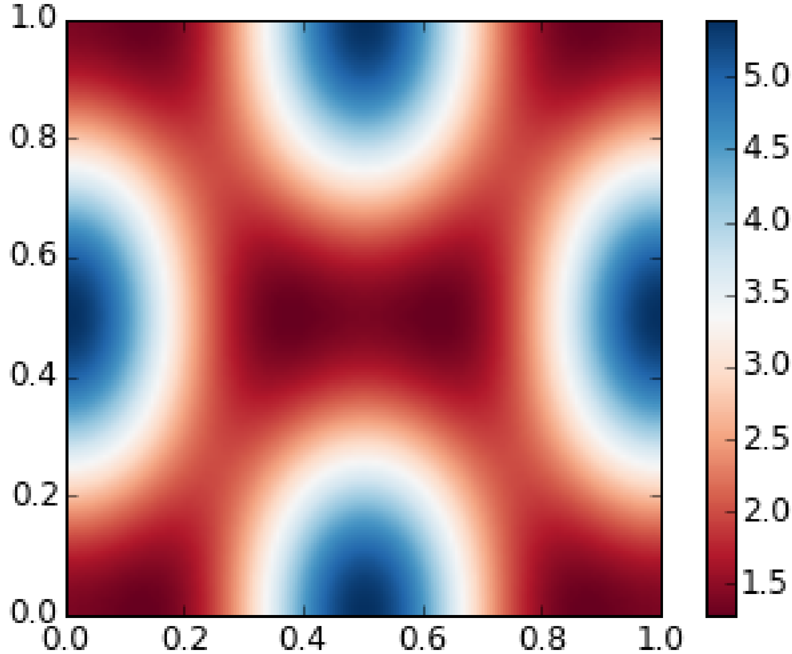
pcolor

In [57]:
```
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*pi), Y/(2*pi), Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```



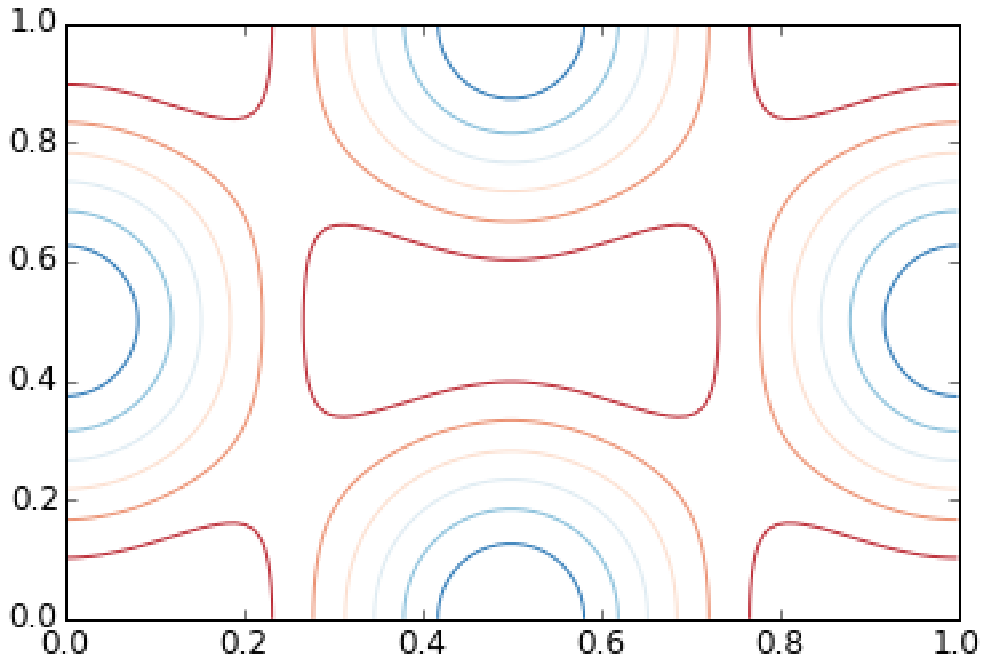
imshow

```
In [58]: fig, ax = plt.subplots()

         im = ax.imshow(Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[0, 1, 0, 1])
         im.set_interpolation('bilinear')

         cb = fig.colorbar(im, ax=ax)
```



contour

```
In [59]: fig, ax = plt.subplots()

         cnt = ax.contour(Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[0, 1, 0, 1])
```



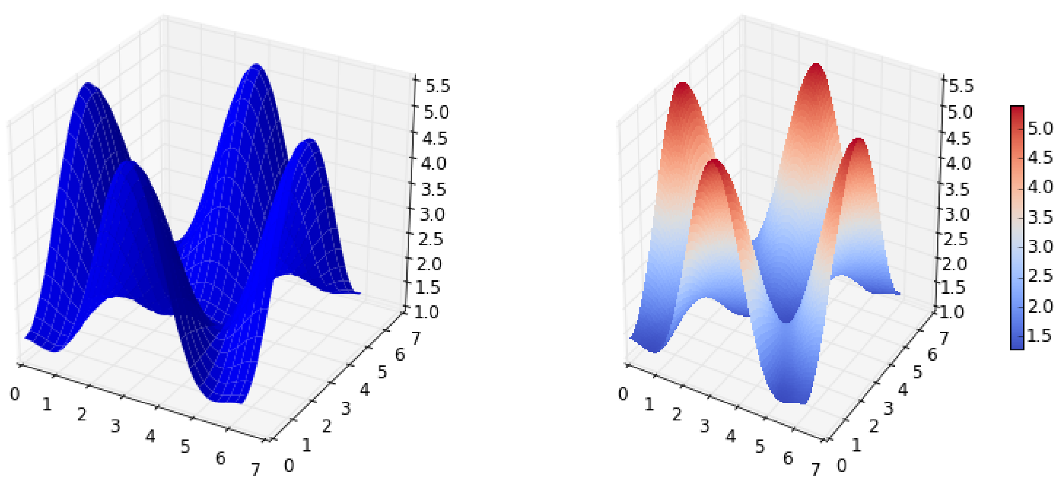
5.4 3D figures

To use 3D graphics in matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

```
In [60]: from mpl_toolkits.mplot3d.axes3d import Axes3D
```

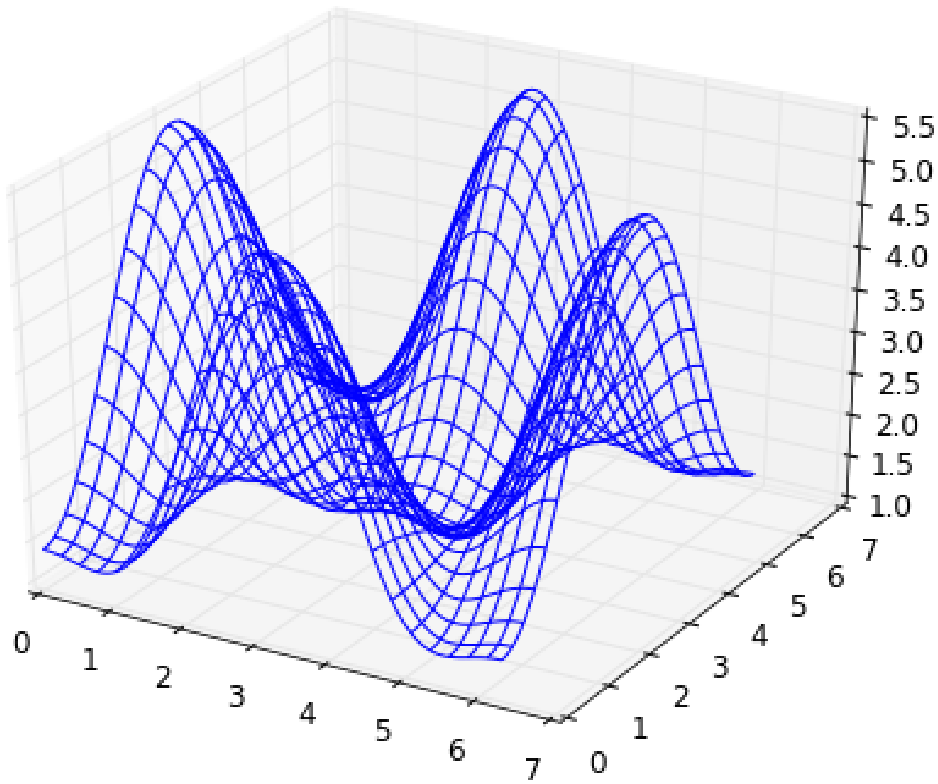
Surface plots

```
In [61]: fig = plt.figure(figsize=(14,6))

         # `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument to add_subp
         ax = fig.add_subplot(1, 2, 1, projection='3d')

         p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

         # surface_plot with color grading and color bar
         ax = fig.add_subplot(1, 2, 2, projection='3d')
         p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0, antialiased=
         cb = fig.colorbar(p, shrink=0.5)
```



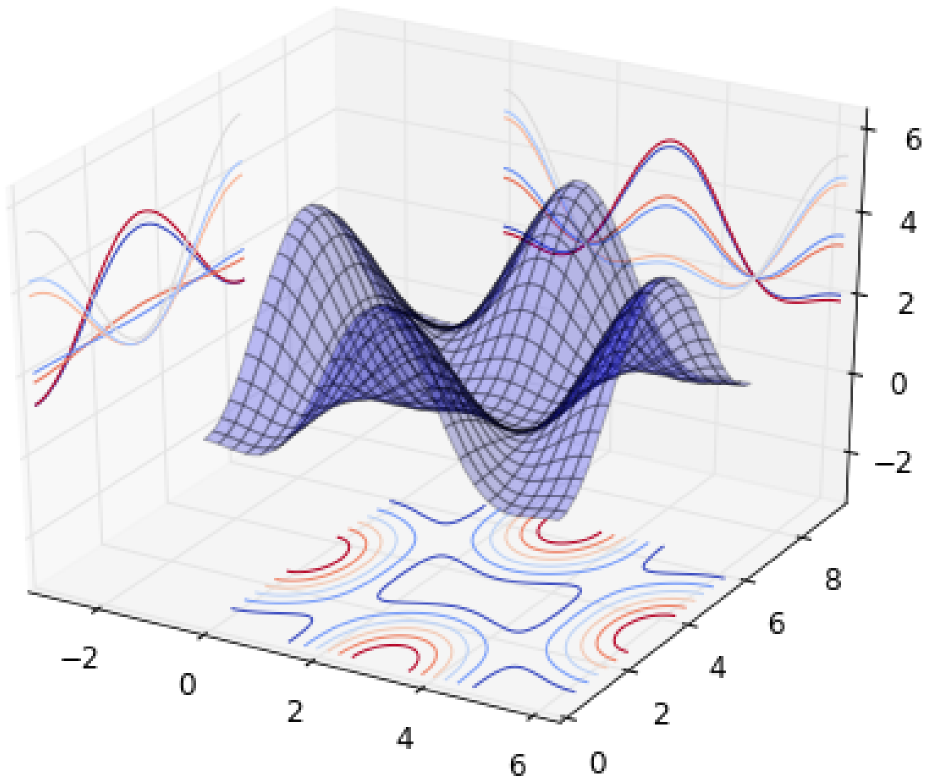
Wire-frame plot

```
In [62]: fig = plt.figure(figsize=(8,6))

         ax = fig.add_subplot(1, 1, 1, projection='3d')

         p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```



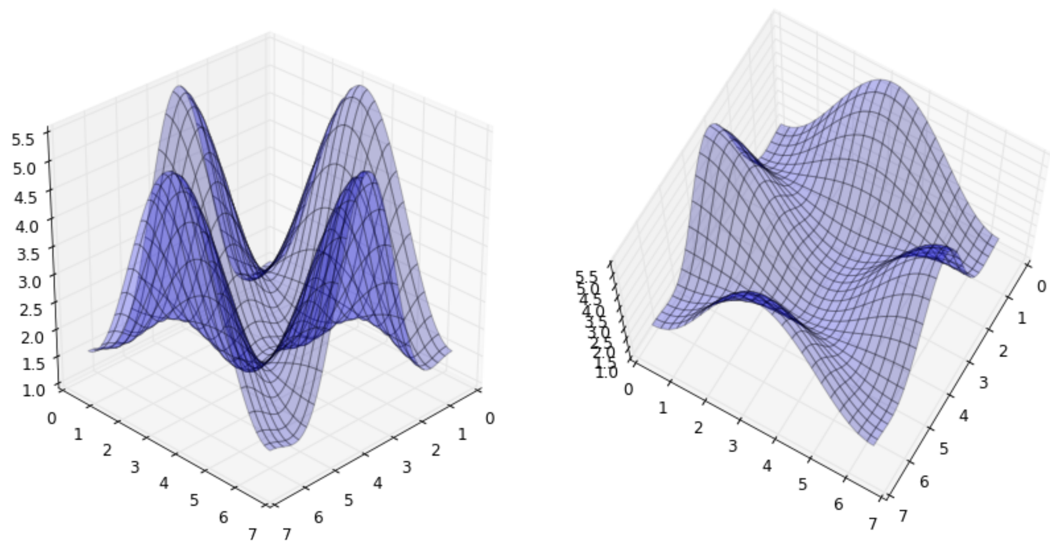
Coutour plots with projections

In [63]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-pi, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=cm.coolwarm)

ax.set_xlim3d(-pi, 2*pi);
ax.set_ylim3d(0, 3*pi);
ax.set_zlim3d(-pi, 2*pi);
```



Change the view angle We can change the perspective of a 3D plot using the `view_init` method, which takes two arguments: `elevation` and `azimuth` angle (in degrees):

```
In [64]: fig = plt.figure(figsize=(12,6))

         ax = fig.add_subplot(1,2,1, projection='3d')
         ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
         ax.view_init(30, 45)

         ax = fig.add_subplot(1,2,2, projection='3d')
         ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
         ax.view_init(70, 30)

         fig.tight_layout()
```



5.4.1 Animations

Matplotlib also includes a simple API for generating animations for sequences of figures. With the `FuncAnimation` function we can generate a movie file from sequences of figures. The function takes the following arguments: `fig`, a figure canvas, `func`, a function that we provide which updates the figure, `init_func`, a function we provide to setup the figure, `frame`, the number of frames to generate, and `blit`, which tells the animation function to only update parts of the frame which have changed (for smoother animations):

```
def init():
    # setup figure

def update(frame_counter):
    # update figure for new frame

anim = animation.FuncAnimation(fig, update, init_func=init, frames=200, blit=True)

anim.save('animation.mp4', fps=30) # fps = frames per second
```

To use the animation features in matplotlib we first need to import the module `matplotlib.animation`:

```
In [65]: from matplotlib import animation
```

```
In [66]: # solve the ode problem of the double compound pendulum again

         from scipy.integrate import odeint

         g = 9.82; L = 0.5; m = 0.1

         def dx(x, t):
             x1, x2, x3, x4 = x[0], x[1], x[2], x[3]

             dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * cos(x1-x2) * x4)/(16 - 9 * cos(x1-x2)**2)
             dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * cos(x1-x2) * x3)/(16 - 9 * cos(x1-x2)**2)
```

```
        dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * sin(x1-x2) + 3 * (g/L) * sin(x1))
        dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * sin(x1-x2) + (g/L) * sin(x2))
        return [dx1, dx2, dx3, dx4]

    x0 = [pi/2, pi/2, 0, 0]  # initial state
    t = linspace(0, 10, 250) # time coordinates
    x = odeint(dx, x0, t)    # solve the ODE problem
```

Generate an animation that shows the positions of the pendulums as a function of time:

```
In [67]: fig, ax = plt.subplots(figsize=(5,5))

         ax.set_ylim([-1.5, 0.5])
         ax.set_xlim([1, -1])

         pendulum1, = ax.plot([], [], color="red", lw=2)
         pendulum2, = ax.plot([], [], color="blue", lw=2)

         def init():
             pendulum1.set_data([], [])
             pendulum2.set_data([], [])

         def update(n):
             # n = frame counter
             # calculate the positions of the pendulums
             x1 = + L * sin(x[n, 0])
             y1 = - L * cos(x[n, 0])
             x2 = x1 + L * sin(x[n, 1])
             y2 = y1 - L * cos(x[n, 1])

             # update the line data
             pendulum1.set_data([0 ,x1], [0 ,y1])
             pendulum2.set_data([x1,x2], [y1,y2])

         anim = animation.FuncAnimation(fig, update, init_func=init, frames=len(t), blit=True)

         # anim.save can be called in a few different ways, some which might or might not work
         # on different platforms and with different versions of matplotlib and video encoders
         #anim.save('animation.mp4', fps=20, extra_args=['-vcodec', 'libx264'],
         #           writer=animation.FFMpegWriter())
         anim.save('animation.mp4', fps=20, extra_args=['-vcodec', 'libx264'])
         #anim.save('animation.mp4', fps=20, writer="ffmpeg", codec="libx264")
         #anim.save('animation.mp4', fps=20, writer="avconv", codec="libx264")

         plt.close(fig)
```

Note: To generate the movie file we need to have either `ffmpeg` or `avconv` installed. Install it on Ubuntu using:

```
$ sudo apt-get install ffmpeg
```

or (newer versions)

```
$ sudo apt-get install libav-tools
```

On MacOSX, try:

```
$ sudo port install ffmpeg
```

```
In [68]: from IPython.display import HTML
         import codecs
         video = open("animation.mp4", "rb").read()
         video_encoded = codecs.encode(video, "base64")
         video_tag = '<video controls alt="test" src="data:video/x-m4v;base64,{0}">'.format(
             video_encoded)
         HTML(video_tag)

Out[68]: <IPython.core.display.HTML object>

In [69]: #!open animation.mp4
```

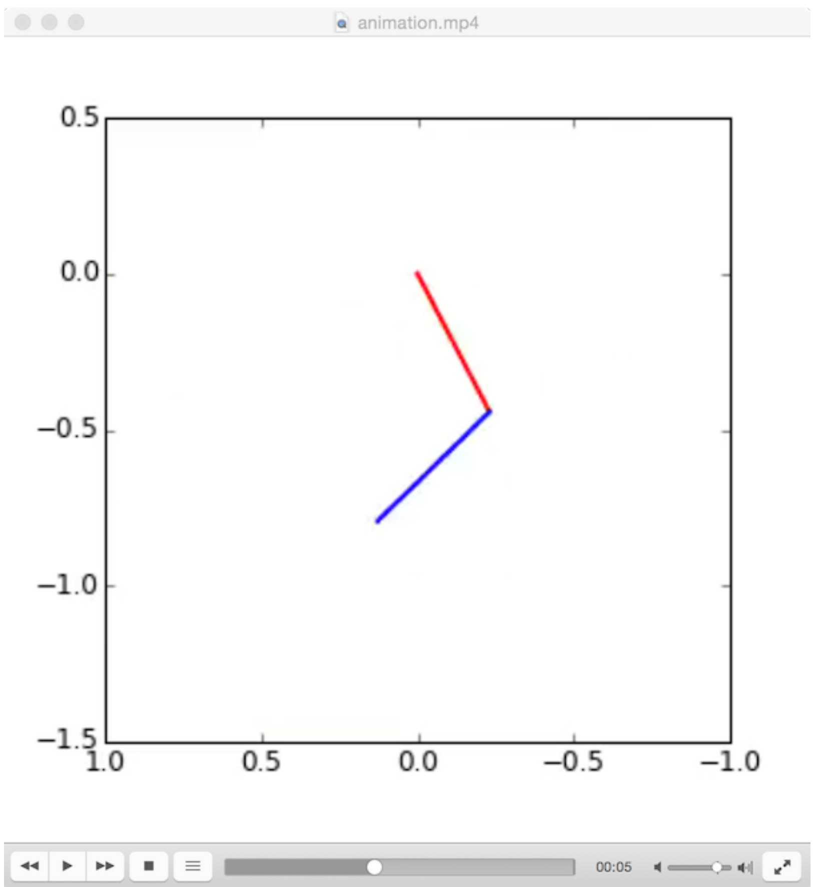


Figure 5.1: Double pendulum animation

5.4.2 Backends

Matplotlib has a number of “backends” which are responsible for rendering graphs. The different backends are able to generate graphics with different formats and display/event loops. There is a distinction between noninteractive backends (such as ‘agg’, ‘svg’, ‘pdf’, etc.) that are only used to generate image files (e.g. with the `savefig` function), and interactive backends (such as Qt4Agg, GTK, MaxOSX) that can display a GUI window for interactively exploring figures.

A list of available backends are:

```
In [70]: print(matplotlib.rcsetup.all_backends)
```

```
['GTK', 'GTKAgg', 'GTKCairo', 'MacOSX', 'Qt4Agg', 'Qt5Agg', 'TkAgg', 'WX', 'WXAgg', 'CocoaAgg', 'GTK3Ca
```

The default backend, called agg, is based on a library for raster graphics which is great for generating raster formats like PNG.

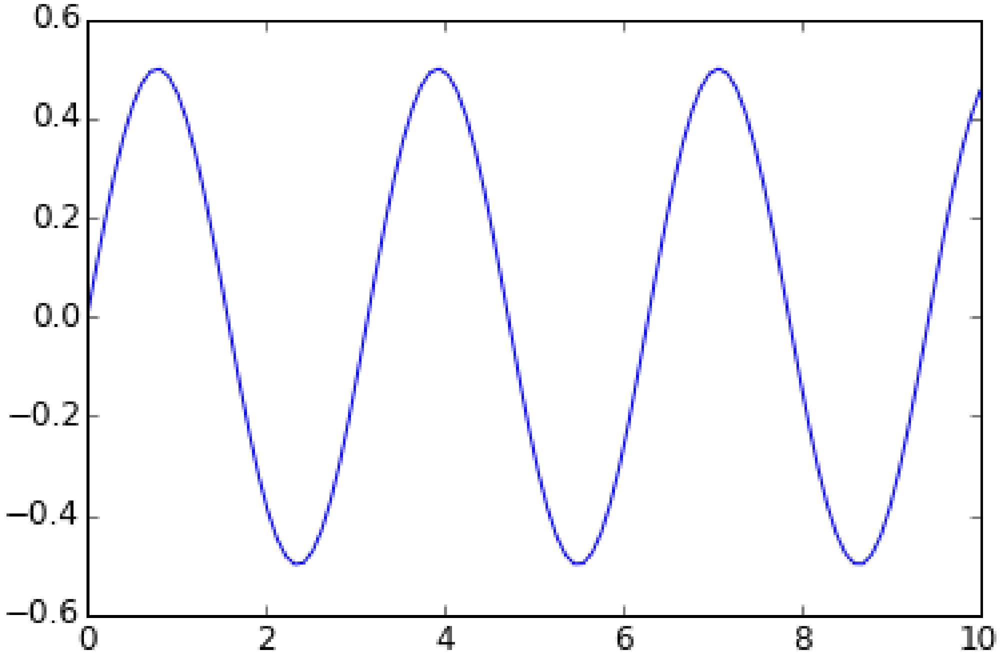
Normally we don't need to bother with changing the default backend; but sometimes it can be useful to switch to, for example, PDF or GTKCairo (if you are using Linux) to produce high-quality vector graphics instead of raster based graphics.

Generating SVG with the svg backend

```
In [71]: #
         # RESTART THE NOTEBOOK: the matplotlib backend can only be selected before pylab is imported!
         # (e.g. Kernel > Restart)
         #
         import matplotlib
         matplotlib.use('svg')
         import matplotlib.pylab as plt
         import numpy
         from IPython.display import Image, SVG
```

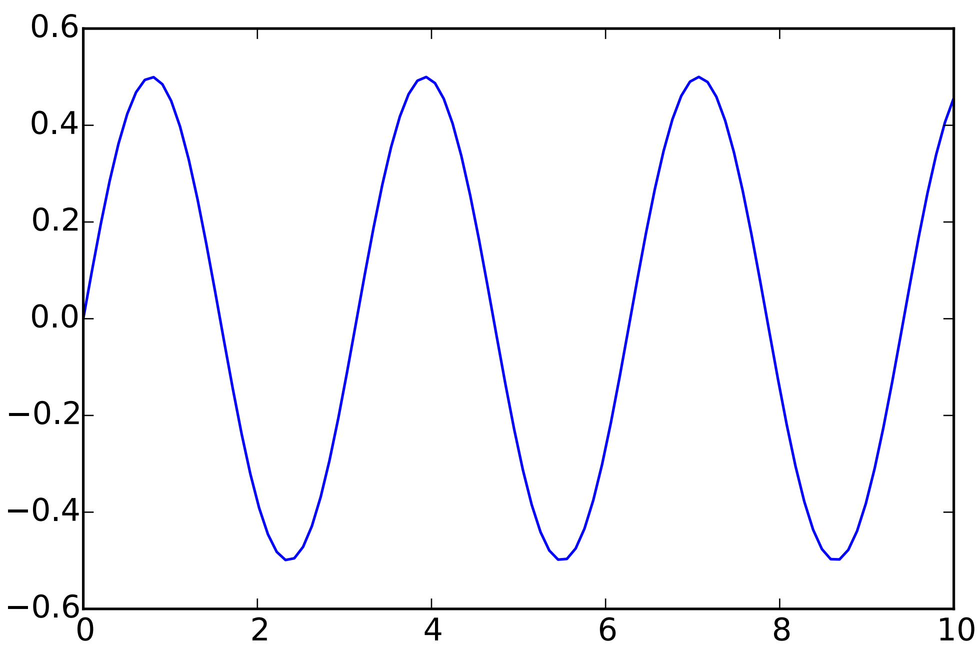
```
/Users/dmertz/anaconda/lib/python3.4/site-packages/matplotlib/__init__.py:1318: UserWarning:  This call t
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

  warnings.warn(_use_error_msg)
```

```
In [72]: #
         # Now we are using the svg backend to produce SVG vector graphics
         #
         fig, ax = plt.subplots()
         t = numpy.linspace(0, 10, 100)
         ax.plot(t, numpy.cos(t)*numpy.sin(t))
         plt.savefig("test.svg")
```




```
In [73]: #
         # Show the produced SVG file.
         #
         SVG(filename="test.svg")
```

Out[73]:



The IPython notebook inline backend When we use IPython notebook it is convenient to use a matplotlib backend that outputs the graphics embedded in the notebook file. To activate this backend, somewhere in the beginning on the notebook, we add:

```
%matplotlib inline
```

It is also possible to activate inline matplotlib plotting with:

```
%pylab inline
```

The difference is that `%pylab inline` imports a number of packages into the global address space (scipy, numpy), while `%matplotlib inline` only sets up inline plotting. In new notebooks created for IPython 1.0+, I would recommend using `%matplotlib inline`, since it is tidier and you have more control over which packages are imported and how. Commonly, scipy and numpy are imported separately with:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

The inline backend has a number of configuration options that can be set by using the IPython magic command `%config` to update settings in `InlineBackend`. For example, we can switch to SVG figures or higher resolution figures with either:

```
%config InlineBackend.figure_format='svg'
```

or:

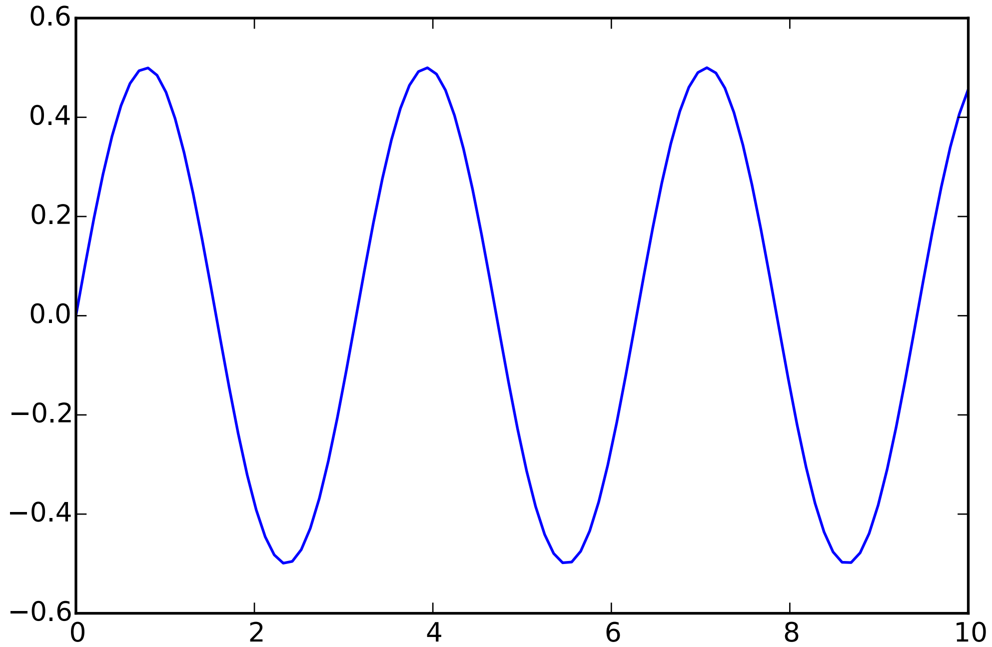
```
%config InlineBackend.figure_format='retina'
```

For more information, type:

```
%config InlineBackend
```

```
In [74]: %matplotlib inline
         %config InlineBackend.figure_format='svg'

         import matplotlib.pylab as plt
         import numpy
```

```
In [75]: #
         # Now we are using the SVG vector graphics displaced inline in the notebook
         #
         fig, ax = plt.subplots()
         t = numpy.linspace(0, 10, 100)
         ax.plot(t, numpy.cos(t)*numpy.sin(t))
         plt.savefig("test.svg")
```

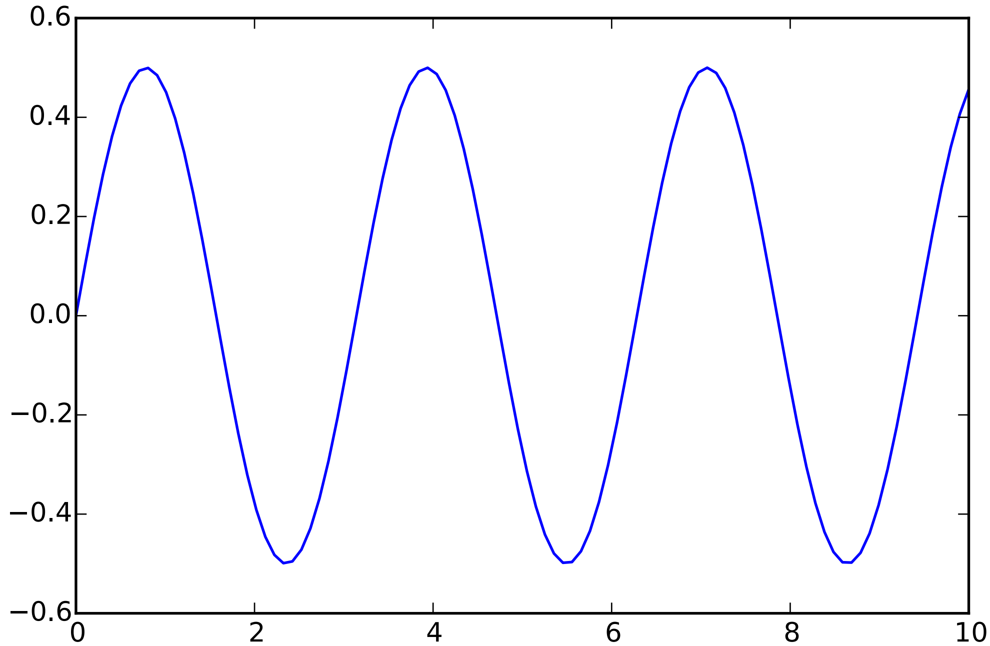


Interactive backend (this makes more sense in a python script file)

```
In [76]: #
         # RESTART THE NOTEBOOK: the matplotlib backend can only be selected before pylab is imported!
         # (e.g. Kernel > Restart)
         #
         import matplotlib
         matplotlib.use('Qt4Agg') # or for example MacOSX
         import matplotlib.pylab as plt
         import numpy
```

```
/Users/dmertz/anaconda/lib/python3.4/site-packages/matplotlib/__init__.py:1318: UserWarning:  This call t
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

  warnings.warn(_use_error_msg)
```

```
In [77]: # Now, open an interactive plot window with the Qt4Agg backend
         fig, ax = plt.subplots()
         t = numpy.linspace(0, 10, 100)
         ax.plot(t, numpy.cos(t)*numpy.sin(t))
         plt.show()
```



Note that when we use an interactive backend, we must call `plt.show()` to make the figure appear on the screen.

5.5 Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.
- http://matplotlib.org/gallery.html - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib - A good matplotlib tutorial.
- http://scipy-lectures.github.io/matplotlib/matplotlib.html - Another good matplotlib reference.

Chapter 6

Sympy - Symbolic algebra in Python

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

6.1 Introduction

There are two notable Computer Algebra Systems (CAS) for Python:

- SymPy - A python module that can be used in any Python program, or in an IPython session, that provides powerful CAS features.
- Sage - Sage is a full-featured and very powerful CAS enviroment that aims to provide an open source system that competes with Mathematica and Maple. Sage is not a regular Python module, but rather a CAS environment that uses Python as its programming language.

Sage is in some aspects more powerful than SymPy, but both offer very comprehensive CAS functionality. The advantage of SymPy is that it is a regular Python module and integrates well with the IPython notebook.

In this lecture we will therefore look at how to use SymPy with IPython notebooks. If you are interested in an open source CAS environment I also recommend to read more about Sage.

To get started using SymPy in a Python program or notebook, import the module `sympy`:

```
In [2]: from sympy import *
```

To get nice-looking LaTeXformatted output run:

```
In [3]: init_printing()

        # or with older versions of sympy/ipython, load the IPython extension
        #%load_ext sympy.interactive.ipythonprinting
        # or
        #%load_ext sympyprinting
```

6.2 Symbolic variables

In SymPy we need to create symbols for the variables we want to work with. We can create a new symbol using the `Symbol` class:

```
In [4]: x = Symbol('x')
```

```
In [5]: (pi + x)**2
```

Out[5]:

$$(x + \pi)^2$$

```
In [6]: # alternative way of defining symbols
        a, b, c = symbols("a, b, c")
```

```
In [7]: type(a)
```

```
Out[7]: sympy.core.symbol.Symbol
```

We can add assumptions to symbols when we create them:

```
In [8]: x = Symbol('x', real=True)
```

```
In [9]: x.is_imaginary
```

```
Out[9]: False
```

```
In [10]: x = Symbol('x', positive=True)
```

```
In [11]: x > 0
```

Out[11]:

True

6.2.1 Complex numbers

The imaginary unit is denoted `I` in Sympy.

```
In [12]: 1+1*I
```

Out[12]:

$$1 + i$$

```
In [13]: I**2
```

Out[13]:

$$-1$$

```
In [14]: (x * I + 1)**2
```

Out[14]:

$$(ix + 1)^2$$

6.2.2 Rational numbers

There are three different numerical types in SymPy: `Real`, `Rational`, `Integer`:

```
In [15]: r1 = Rational(4,5)
         r2 = Rational(5,4)
```

```
In [16]: r1
```

Out[16]:

$$\frac{4}{5}$$

```
In [17]: r1+r2
```

Out[17]:

$$\frac{41}{20}$$

```
In [18]: r1/r2
```

Out[18]:

$$\frac{16}{25}$$

6.3 Numerical evaluation

SymPy uses a library for artitrary precision as numerical backend, and has predefined SymPy expressions for a number of mathematical constants, such as: `pi`, `e`, `oo` for infinity.

To evaluate an expression numerically we can use the `evalf` function (or `N`). It takes an argument `n` which specifies the number of significant digits.

```
In [19]: pi.evalf(n=50)
```

Out[19]:

$$3.1415926535897932384626433832795028841971693993751$$

```
In [20]: y = (x + pi)**2
```

```
In [21]: N(y, 5) # same as evalf
```

Out[21]:

$$(x + 3.1416)^2$$

When we numerically evaluate algebraic expressions we often want to substitute a symbol with a numerical value. In SymPy we do that using the `subs` function:

```
In [22]: y.subs(x, 1.5)
```

Out[22]:

$$(1.5 + \pi)^2$$

```
In [23]: N(y.subs(x, 1.5))
```

Out[23]:

$$21.5443823618587$$

The `subs` function can of course also be used to substitute Symbols and expressions:

```
In [24]: y.subs(x, a+pi)
```

Out[24]:

$$(a + 2\pi)^2$$

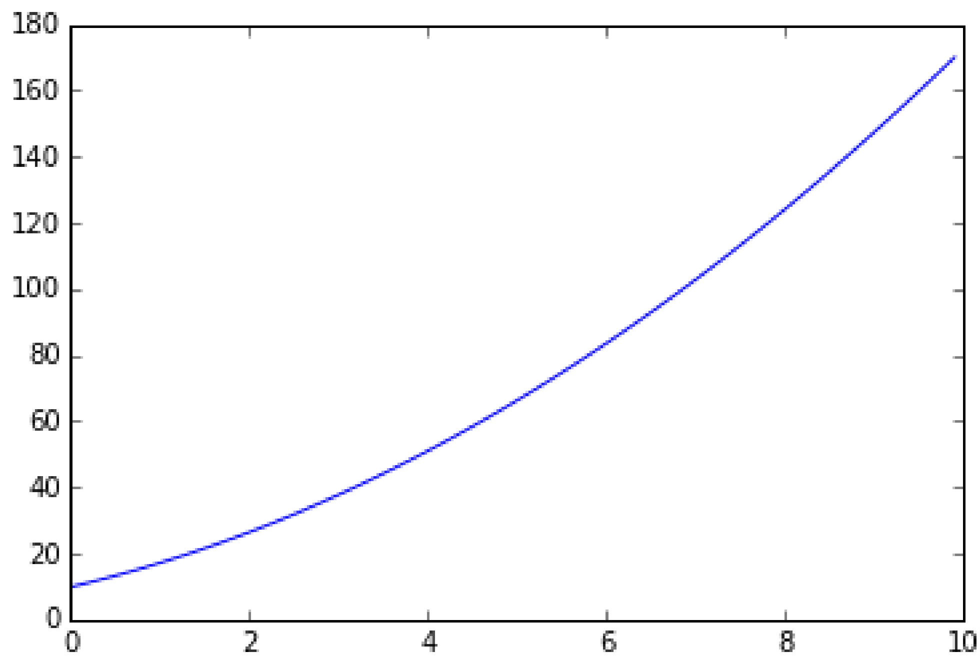
We can also combine numerical evolution of expressions with NumPy arrays:

```
In [25]: import numpy
```

```
In [26]: x_vec = numpy.arange(0, 10, 0.1)
```

```
In [27]: y_vec = numpy.array([N(((x + pi)**2).subs(x, xx)) for xx in x_vec])
```

```
In [28]: fig, ax = subplots()
         ax.plot(x_vec, y_vec);
```



However, this kind of numerical evolution can be very slow, and there is a much more efficient way to do it: Use the function `lambdify` to "compile" a Sympy expression into a function that is much more efficient to evaluate numerically:

```
In [29]: f = lambdify([x], (x + pi)**2, 'numpy')  # the first argument is a list of variables that
                                                   # f will be a function of: in this case only x -> f(x
```

```
In [30]: y_vec = f(x_vec)  # now we can directly pass a numpy array and f(x) is efficiently evaluated
```

The speedup when using "lambdified" functions instead of direct numerical evaluation can be significant, often several orders of magnitude. Even in this simple example we get a significant speed up:

In [31]: %%timeit

```
y_vec = numpy.array([N(((x + pi)**2).subs(x, xx)) for xx in x_vec])
```

```
100 loops, best of 3: 16.4 ms per loop
```

In [32]: %%timeit

```
y_vec = f(x_vec)
```

```
The slowest run took 14.09 times longer than the fastest. This could mean that an intermediate result is
1000000 loops, best of 3: 1.49 µs per loop
```

6.4 Algebraic manipulations

One of the main uses of an CAS is to perform algebraic manipulations of expressions. For example, we might want to expand a product, factor an expression, or simply an expression. The functions for doing these basic operations in SymPy are demonstrated in this section.

6.4.1 Expand and factor

The first steps in an algebraic manipulation

In [33]: (x+1)*(x+2)*(x+3)

Out[33]:

$$(x+1)(x+2)(x+3)$$

In [34]: expand((x+1)*(x+2)*(x+3))

Out[34]:

$$x^3 + 6x^2 + 11x + 6$$

The `expand` function takes a number of keywords arguments which we can tell the functions what kind of expansions we want to have performed. For example, to expand trigonometric expressions, use the `trig=True` keyword argument:

In [35]: sin(a+b)

Out[35]:

$$\sin(a+b)$$

In [36]: expand(sin(a+b), trig=True)

Out[36]:

$$\sin(a)\cos(b) + \sin(b)\cos(a)$$

See `help(expand)` for a detailed explanation of the various types of expansions the `expand` functions can perform.

The opposite a product expansion is of course factoring. The factor an expression in SymPy use the `factor` function:

In [37]: factor(x**3 + 6 * x**2 + 11*x + 6)

Out[37]:

$$(x+1)(x+2)(x+3)$$

6.4.2 Simplify

The `simplify` tries to simplify an expression into a nice looking expression, using various techniques. More specific alternatives to the `simplify` functions also exists: `trigsimp`, `powsimp`, `logcombine`, etc.

The basic usages of these functions are as follows:

```
In [38]: # simplify expands a product
         simplify((x+1)*(x+2)*(x+3))
```

Out[38]:

$$(x+1)(x+2)(x+3)$$

```
In [39]: # simplify uses trigonometric identities
         simplify(sin(a)**2 + cos(a)**2)
```

Out[39]:

$$1$$

```
In [40]: simplify(cos(x)/sin(x))
```

Out[40]:

$$\frac{1}{\tan(x)}$$

6.4.3 apart and together

To manipulate symbolic expressions of fractions, we can use the `apart` and `together` functions:

```
In [41]: f1 = 1/((a+1)*(a+2))
```

```
In [42]: f1
```

Out[42]:

$$\frac{1}{(a+1)(a+2)}$$

```
In [43]: apart(f1)
```

Out[43]:

$$-\frac{1}{a+2}+\frac{1}{a+1}$$

```
In [44]: f2 = 1/(a+2) + 1/(a+3)
```

```
In [45]: f2
```

Out[45]:

$$\frac{1}{a+3}+\frac{1}{a+2}$$

```
In [46]: together(f2)
```

Out[46]:

$$\frac{2a+5}{(a+2)(a+3)}$$

Simplify usually combines fractions but does not factor:

```
In [47]: simplify(f2)
```

Out[47]:

$$\frac{2a+5}{(a+2)(a+3)}$$

6.5 Calculus

In addition to algebraic manipulations, the other main use of CAS is to do calculus, like derivatives and integrals of algebraic expressions.

6.5.1 Differentiation

Differentiation is usually simple. Use the `diff` function. The first argument is the expression to take the derivative of, and the second argument is the symbol by which to take the derivative:

```
In [48]: y
```

Out[48]:

$$(x+\pi)^2$$

```
In [49]: diff(y**2, x)
```

Out[49]:

$$4(x+\pi)^3$$

For higher order derivatives we can do:

```
In [50]: diff(y**2, x, x)
```

Out[50]:

$$12(x+\pi)^2$$

```
In [51]: diff(y**2, x, 2) # same as above
```

Out[51]:

$$12(x+\pi)^2$$

To calculate the derivative of a multivariate expression, we can do:

```
In [52]: x, y, z = symbols("x,y,z")
```

```
In [53]: f = sin(x*y) + cos(y*z)
```

$\frac{d^3f}{dxdy^2}$

```
In [54]: diff(f, x, 1, y, 2)
```

Out[54]:

$$-x\left(xy\cos(xy)+2\sin(xy)\right)$$

6.6 Integration

Integration is done in a similar fashion:

```
In [55]: f
```

Out[55]:

$$\sin(xy) + \cos(yz)$$

```
In [56]: integrate(f, x)
```

Out[56]:

$$x\cos(yz) + \begin{cases} 0 & \text{for } y = 0 \\ -\frac{1}{y}\cos(xy) & \text{otherwise} \end{cases}$$

By providing limits for the integration variable we can evaluate definite integrals:

```
In [57]: integrate(f, (x, -1, 1))
```

Out[57]:

$$2\cos(yz)$$

and also improper integrals

```
In [58]: integrate(exp(-x**2), (x, -oo, oo))
```

Out[58]:

$$\sqrt{\pi}$$

Remember, `oo` is the SymPy notation for inifinity.

6.6.1 Sums and products

We can evaluate sums and products using the functions: 'Sum'

```
In [59]: n = Symbol("n")
```

```
In [60]: Sum(1/n**2, (n, 1, 10))
```

Out[60]:

$$\sum_{n=1}^{10} \frac{1}{n^2}$$

```
In [61]: Sum(1/n**2, (n,1, 10)).evalf()
```

Out[61]:

$$1.54976773116654$$

```
In [62]: Sum(1/n**2, (n, 1, oo)).evalf()
```

Out[62]:

$$1.64493406684823$$

Products work much the same way:

```
In [63]: Product(n, (n, 1, 10)) # 10!
```

Out[63]:

$$\prod_{n=1}^{10} n$$

6.7 Limits

Limits can be evaluated using the `limit` function. For example,

```
In [64]: limit(sin(x)/x, x, 0)
```

Out[64]:

$$1$$

We can use 'limit' to check the result of derivation using the `diff` function:

```
In [65]: f
```

Out[65]:

$$\sin(xy) + \cos(yz)$$

```
In [66]: diff(f, x)
```

Out[66]:

$$y\cos(xy)$$

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}x} = \frac{f(x+h,y) - f(x,y)}{h}$$

```
In [67]: h = Symbol("h")

In [68]: limit((f.subs(x, x+h) - f)/h, h, 0)
```

Out[68]:

$$y\cos(xy)$$

OK!

We can change the direction from which we approach the limiting point using the `dir` keywork argument:

```
In [69]: limit(1/x, x, 0, dir="+")
```

Out[69]:

$$\infty$$

```
In [70]: limit(1/x, x, 0, dir="-")
```

Out[70]:

$$-\infty$$

6.8 Series

Series expansion is also one of the most useful features of a CAS. In SymPy we can perform a series expansion of an expression using the `series` function:

```
In [71]: series(exp(x), x)
```

Out[71]:

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \mathcal{O}\left(x^6\right)$$

By default it expands the expression around $x = 0$, but we can expand around any value of x by explicitly include a value in the function call:

```
In [72]: series(exp(x), x, 1)
```

Out[72]:

$$e + e\left(x - 1\right) + \frac{e}{2}\left(x - 1\right)^2 + \frac{e}{6}\left(x - 1\right)^3 + \frac{e}{24}\left(x - 1\right)^4 + \frac{e}{120}\left(x - 1\right)^5 + \mathcal{O}\left(\left(x - 1\right)^6; x \rightarrow 1\right)$$

And we can explicitly define to which order the series expansion should be carried out:

```
In [73]: series(exp(x), x, 1, 10)
```

Out[73]:

$$e + e\left(x - 1\right) + \frac{e}{2}\left(x - 1\right)^2 + \frac{e}{6}\left(x - 1\right)^3 + \frac{e}{24}\left(x - 1\right)^4 + \frac{e}{120}\left(x - 1\right)^5 + \frac{e}{720}\left(x - 1\right)^6 + \frac{e}{5040}\left(x - 1\right)^7 + \frac{e}{40320}\left(x - 1\right)^8 + \frac{e}{362880}\left(x - 1\right)^9 + \mathcal{O}\left(\left(x - 1\right)^{10}; x \rightarrow 1\right)$$

The series expansion includes the order of the approximation, which is very useful for keeping track of the order of validity when we do calculations with series expansions of different order:

```
In [74]: s1 = cos(x).series(x, 0, 5)
         s1
```

Out[74]:

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + \mathcal{O}\left(x^5\right)$$

```
In [75]: s2 = sin(x).series(x, 0, 2)
         s2
```

Out[75]:

$$x + \mathcal{O}\left(x^2\right)$$

```
In [76]: expand(s1 * s2)
```

Out[76]:

$$x + \mathcal{O}\left(x^2\right)$$

If we want to get rid of the order information we can use the `removeO` method:

```
In [77]: expand(s1.removeO() * s2.removeO())
```

Out[77]:

$$\frac{x^5}{24} - \frac{x^3}{2} + x$$

But note that this is not the correct expansion of $\cos(x)\sin(x)$ to 5th order:

```
In [78]: (cos(x)*sin(x)).series(x, 0, 6)
```

Out[78]:

$$x - \frac{2x^3}{3} + \frac{2x^5}{15} + \mathcal{O}\left(x^6\right)$$

6.9 Linear algebra

6.9.1 Matrices

Matrices are defined using the `Matrix` class:

In [79]:
```
m11, m12, m21, m22 = symbols("m11, m12, m21, m22")
b1, b2 = symbols("b1, b2")
```

In [80]:
```
A = Matrix([[m11, m12],[m21, m22]])
A
```

Out[80]:

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

In [81]:
```
b = Matrix([[b1], [b2]])
b
```

Out[81]:

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

With `Matrix` class instances we can do the usual matrix algebra operations:

In [82]:
```
A**2
```

Out[82]:

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix}$$

In [83]:
```
A * b
```

Out[83]:

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix}$$

And calculate determinants and inverses, and the like:

In [84]:
```
A.det()
```

Out[84]:

$$m_{11}m_{22} - m_{12}m_{21}$$

In [85]:
```
A.inv()
```

Out[85]:

$$\begin{bmatrix} \frac{1}{m_{11}} + \frac{m_{12}m_{21}}{m_{11}^2\left(m_{22} - \frac{m_{12}m_{21}}{m_{11}}\right)} & -\frac{m_{12}}{m_{11}\left(m_{22} - \frac{m_{12}m_{21}}{m_{11}}\right)} \\ -\frac{m_{21}}{m_{11}\left(m_{22} - \frac{m_{12}m_{21}}{m_{11}}\right)} & \frac{1}{m_{22} - \frac{m_{12}m_{21}}{m_{11}}} \end{bmatrix}$$

6.10 Solving equations

For solving equations and systems of equations we can use the `solve` function:

```
In [86]: solve(x**2 - 1, x)
```

Out[86]:

$$[-1, \quad 1]$$

```
In [87]: solve(x**4 - x**2 - 1, x)
```

Out[87]:

$$\left[-i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}}, \quad i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}}, \quad -\sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}}, \quad \sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}}\right]$$

System of equations:

```
In [88]: solve([x + y - 1, x - y - 1], [x,y])
```

Out[88]:

$$\{x : 1, \quad y : 0\}$$

In terms of other symbolic expressions:

```
In [89]: solve([x + y - a, x - y - c], [x,y])
```

Out[89]:

$$\left\{x : \frac{a}{2} + \frac{c}{2}, \quad y : \frac{a}{2} - \frac{c}{2}\right\}$$

6.11 Quantum mechanics: noncommuting variables

How about non-commuting symbols? In quantum mechanics we need to work with noncommuting operators, and SymPy has a nice support for noncommuting symbols and even a subpackage for quantum mechanics related calculations!

```
In [90]: from sympy.physics.quantum import *
```

6.12 States

We can define symbol states, kets and bras:

```
In [91]: Ket('psi')
```

Out[91]:

$$|\psi\rangle$$

```
In [92]: Bra('psi')
```

Out[92]:

$$\langle\psi|$$

In [93]:
```
u = Ket('0')
d = Ket('1')

a, b = symbols('alpha beta', complex=True)
```

In [94]:
```
phi = a * u + sqrt(1-abs(a)**2) * d; phi
```

Out[94]:

$$\alpha|0\rangle + \sqrt{-\left|\alpha\right|^{2} + 1}|1\rangle$$

In [95]:
```
Dagger(phi)
```

Out[95]:

$$\overline{\alpha}\langle 0| + \overline{\sqrt{-\left|\alpha\right|^{2} + 1}}\langle 1|$$

In [96]:
```
Dagger(phi) * d
```

Out[96]:

$$\left(\overline{\alpha}\langle 0| + \overline{\sqrt{-\left|\alpha\right|^{2} + 1}}\langle 1|\right)|1\rangle$$

Use `qapply` to distribute a mutiplication:

In [97]:
```
qapply(Dagger(phi) * d)
```

Out[97]:

$$\overline{\alpha}\left\langle 0 \right. \left|1\right\rangle + \overline{\sqrt{-\left|\alpha\right|^{2} + 1}}\left\langle 1 \right. \left|1\right\rangle$$

In [98]:
```
qapply(Dagger(phi) * u)
```

Out[98]:

$$\overline{\alpha}\left\langle 0 \right. \left|0\right\rangle + \overline{\sqrt{-\left|\alpha\right|^{2} + 1}}\left\langle 1 \right. \left|0\right\rangle$$

6.12.1 Operators

In [99]:
```
A = Operator('A')
B = Operator('B')
```

Check if they are commuting!

In [100]:
```
A * B == B * A
```

Out[100]: False

In [101]:
```
expand((A+B)**3)
```

Out[101]:

$$ABA + A\left(B\right)^{2} + \left(A\right)^{2}B + \left(A\right)^{3} + BAB + B\left(A\right)^{2} + \left(B\right)^{2}A + \left(B\right)^{3}$$

In [102]:
```
c = Commutator(A,B)
c
```

Out[102]:

$$[A, B]$$

We can use the `doit` method to evaluate the commutator:

```
In [103]: c.doit()
```

Out[103]:

$$AB - BA$$

We can mix quantum operators with C-numbers:

```
In [104]: c = Commutator(a * A, b * B)
          c
```

Out[104]:

$$\alpha\beta [A, B]$$

To expand the commutator, use the `expand` method with the `commutator=True` keyword argument:

```
In [105]: c = Commutator(A+B, A*B)
          c.expand(commutator=True)
```

Out[105]:

$$-[A, B] B + A [A, B]$$

```
In [106]: Dagger(Commutator(A, B))
```

Out[106]:

$$-\left[A^\dagger, B^\dagger\right]$$

```
In [107]: ac = AntiCommutator(A,B)
```

```
In [108]: ac.doit()
```

Out[108]:

$$AB + BA$$

Example: Quadrature commutator Let's look at the commutator of the electromagnetic field quadatures x and p. We can write the quadrature operators in terms of the creation and annihilation operators as:

$x = (a + a^\dagger)/\sqrt{2}$
$p = -i(a - a^\dagger)/\sqrt{2}$

```
In [109]: X = (A + Dagger(A))/sqrt(2)
          X
```

Out[109]:

$$\frac{\sqrt{2}}{2} \left(A^\dagger + A\right)$$

```
In [110]: P = -I * (A - Dagger(A))/sqrt(2)
          P
```

Out[110]:

$$-\frac{\sqrt{2}i}{2}\left(-A^{\dagger}+A\right)$$

Let's expand the commutator $[x,p]$

```
In [111]: Commutator(X, P).expand(commutator=True).expand(commutator=True)
```

Out[111]:

$$-i\left[A^{\dagger},A\right]$$

Here we see directly that the well known commutation relation for the quadratures
$[x,p]=i$
is a directly related to
$[A,A^{\dagger}]=1$
(which SymPy does not know about, and does not simplify).
For more details on the quantum module in SymPy, see:

- http://docs.sympy.org/0.7.2/modules/physics/quantum/index.html
- http://nbviewer.ipython.org/urls/raw.github.com/ipython/ipython/master/docs/examples/notebooks/sympy_quantum

6.13 Further reading

- http://sympy.org/en/index.html - The SymPy projects web page.
- https://github.com/sympy/sympy - The source code of SymPy.
- http://live.sympy.org - Online version of SymPy for testing and demonstrations.

Figure 6.1: Continuum Logo

Chapter 7
Using Fortran and C code with Python

In [147]:
```
%pylab inline
from IPython.display import Image
```

```
Populating the interactive namespace from numpy and matplotlib
```

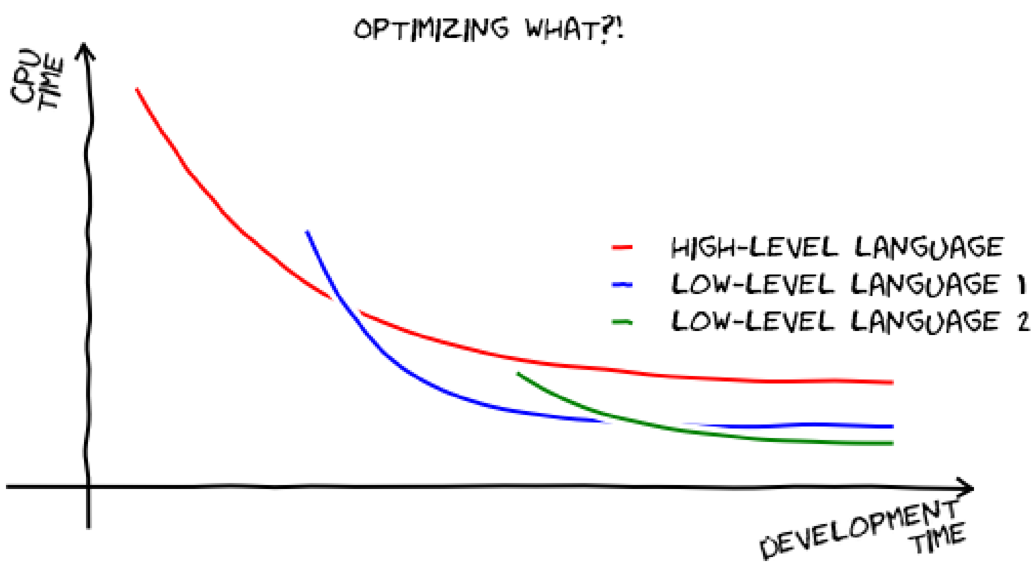
The advantage of Python is that it is flexible and easy to program. The time it takes to setup a new calulation is therefore short. But for certain types of calculations Python (and any other interpreted language) can be very slow. It is particularly iterations over large arrays that is difficult to do efficiently.

Such calculations may be implemented in a compiled language such as C or Fortran. In Python it is relatively easy to call out to libraries with compiled C or Fortran code. In this lecture we will look at how to do that.

But before we go ahead and work on optimizing anything, it is always worthwhile to ask. . . .

In [148]:
```
Image(filename='images/optimizing-what.png')
```

Out[148]:



7.1 Fortran

7.1.1 F2PY

F2PY is a program that (almost) automatically wraps fortran code for use in Python: By using the `f2py` program we can compile fortran code into a module that we can import in a Python program.

F2PY is a part of NumPy, but you will also need to have a fortran compiler to run the examples below.

7.1.2 Example 0: scalar input, no output

```
In [149]: %%file hellofortran.f
          C File  hellofortran.f
                  subroutine hellofortran (n)
                  integer n

                  do 100 i=0, n
                      print *, "Fortran says hello"
          100     continue
                  end
```

```
Overwriting hellofortran.f
```

Generate a python module using `f2py`:

```
In [150]: !f2py3 -c -m hellofortran hellofortran.f
```

```
running build
running config_cc
unifing config_cc, config, build_clib, build_ext, build commands --compiler options
running config_fc
unifing config_fc, config, build_clib, build_ext, build commands --fcompiler options
running build_src
build_src
building extension "hellofortran" sources
f2py options: []
f2py:> /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/hellofortranmodule.c
creating /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4
Reading fortran codes...
        Reading file 'hellofortran.f' (format:fix,strict)
Post-processing...
        Block: hellofortran
                        Block: hellofortran
Post-processing (stage 2)...
Building modules...
        Building module "hellofortran"...
                Constructing wrapper function "hellofortran"...
                  hellofortran(n)
        Wrote C/API module "hellofortran" to file "/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/hellofortranmod
  adding '/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/fortranobject.c' to sources.
  adding '/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4' to include_dirs.
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.c -> /tmp/tmp_6
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.h -> /tmp/tmp_6
build_src: building npy-pkg config files
running build_ext
customize UnixCCompiler
```

```
customize UnixCCompiler using build_ext
customize Gnu95FCompiler
Found executable /usr/bin/gfortran
customize Gnu95FCompiler
customize Gnu95FCompiler using build_ext
building 'hellofortran' extension
compiling C sources
C compiler: gcc -pthread -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC

creating /tmp/tmp_6mh2wh9/tmp
creating /tmp/tmp_6mh2wh9/tmp/tmp_6mh2wh9
creating /tmp/tmp_6mh2wh9/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4
compile options: '-I/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gcc: /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/hellofortranmodule.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/array
                 from /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/hellofortranmodule.c:17:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
gcc: /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/fortranobject.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/array
                 from /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/fortranobject.c:2:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
compiling Fortran sources
Fortran f77 compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -fPIC -O3 -funroll-
Fortran f90 compiler: /usr/bin/gfortran -Wall -g -fno-second-underscore -fPIC -O3 -funroll-loops
Fortran fix compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -Wall -g -fno-secon
compile options: '-I/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gfortran:f77: hellofortran.f
/usr/bin/gfortran -Wall -g -Wall -g -shared /tmp/tmp_6mh2wh9/tmp/tmp_6mh2wh9/src.linux-x86_64-3.4/hellof
Removing build directory /tmp/tmp_6mh2wh9
```

Example of a python script that use the module:

```
In [151]: %%file hello.py
          import hellofortran

          hellofortran.hellofortran(5)
```

```
Overwriting hello.py
```

```
In [152]: # run the script
          !python hello.py
```

```
Fortran says hello
 Fortran says hello
 Fortran says hello
```

```
 Fortran says hello
 Fortran says hello
 Fortran says hello
```

7.1.3 Example 1: vector input and scalar output

```
In [153]: %%file dprod.f

                subroutine dprod(x, y, n)

                double precision x(n), y
                y = 1.0

                do 100 i=1, n
                    y = y * x(i)
          100   continue
                end
```

```
Overwriting dprod.f
```

```
In [154]: !rm -f dprod.pyf
          !f2py3 -m dprod -h dprod.pyf dprod.f
```

```
Reading fortran codes...
        Reading file 'dprod.f' (format:fix,strict)
Post-processing...
        Block: dprod
{}
In: :dprod:dprod.f:dprod
vars2fortran: No typespec for argument "n".
                        Block: dprod
Post-processing (stage 2)...
Saving signatures to file "./dprod.pyf"
```

The f2py program generated a module declaration file called dsum.pyf. Let's look what's in it:

```
In [155]: !cat dprod.pyf
```

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module dprod ! in
    interface  ! in :dprod
        subroutine dprod(x,y,n) ! in :dprod:dprod.f
            double precision dimension(n) :: x
            double precision :: y
            integer, optional,check(len(x)>=n),depend(x) :: n=len(x)
        end subroutine dprod
    end interface
end python module dprod

! This file was auto-generated with f2py (version:2).
! See http://cens.ioc.ee/projects/f2py2e/
```

The module does not know what Fortran subroutine arguments is input and output, so we need to manually edit the module declaration files and mark output variables with `intent(out)` and input variable with `intent(in)`:

```
In [156]: %%file dprod.pyf
          python module dprod ! in
              interface  ! in :dprod
                  subroutine dprod(x,y,n) ! in :dprod:dprod.f
                      double precision dimension(n), intent(in) :: x
                      double precision, intent(out) :: y
                      integer, optional,check(len(x)>=n),depend(x),intent(in) :: n=len(x)
                  end subroutine dprod
              end interface
          end python module dprod
```

```
Overwriting dprod.pyf
```

Compile the fortran code into a module that can be included in python:

```
In [157]: !f2py3 -c dprod.pyf dprod.f
```

```
running build
running config_cc
unifing config_cc, config, build_clib, build_ext, build commands --compiler options
running config_fc
unifing config_fc, config, build_clib, build_ext, build commands --fcompiler options
running build_src
build_src
building extension "dprod" sources
creating /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4
f2py options: []
f2py: dprod.pyf
Reading fortran codes...
        Reading file 'dprod.pyf' (format:free)
Post-processing...
        Block: dprod
                        Block: dprod
Post-processing (stage 2)...
Building modules...
        Building module "dprod"...
                Constructing wrapper function "dprod"...
                  y = dprod(x,[n])
        Wrote C/API module "dprod" to file "/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/dprodmodule.c"
  adding '/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/fortranobject.c' to sources.
  adding '/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4' to include_dirs.
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.c -> /tmpo
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.h -> /tmpo
build_src: building npy-pkg config files
running build_ext
customize UnixCCompiler
customize UnixCCompiler using build_ext
customize Gnu95FCompiler
Found executable /usr/bin/gfortran
customize Gnu95FCompiler
customize Gnu95FCompiler using build_ext
building 'dprod' extension
compiling C sources
C compiler: gcc -pthread -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC
```

```
creating /tmp/tmpo0ulkq7s/tmp
creating /tmp/tmpo0ulkq7s/tmp/tmpo0ulkq7s
creating /tmp/tmpo0ulkq7s/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4
compile options: '-I/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gcc: /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/fortranobject.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/arra
                 from /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/fortranobject.c:2:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
gcc: /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/dprodmodule.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/arra
                 from /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/dprodmodule.c:18:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/dprodmodule.c:111:12: warning: 'f2py_size' defined but not used [-
 static int f2py_size(PyArrayObject* var, ...)
            ^
compiling Fortran sources
Fortran f77 compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -fPIC -O3 -funroll-
Fortran f90 compiler: /usr/bin/gfortran -Wall -g -fno-second-underscore -fPIC -O3 -funroll-loops
Fortran fix compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -Wall -g -fno-secon
compile options: '-I/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gfortran:f77: dprod.f
/usr/bin/gfortran -Wall -g -Wall -g -shared /tmp/tmpo0ulkq7s/tmp/tmpo0ulkq7s/src.linux-x86_64-3.4/dprodm
Removing build directory /tmp/tmpo0ulkq7s
```

Using the module from Python

```
In [158]: import dprod
```

```
In [159]: help(dprod)
```

```
Help on module dprod:

NAME
    dprod

DESCRIPTION
    This module 'dprod' is auto-generated with f2py (version:2).
    Functions:
      y = dprod(x,n=len(x))
    .

DATA
    dprod = <fortran object>

VERSION
```

```
    b'$Revision: $'

FILE
    /home/dhavide/repositories/scientific-python-lectures/dprod.cpython-34m.so
```

```
In [160]: dprod.dprod(arange(1,50))

Out[160]: 6.082818640342675e+62

In [161]: # compare to numpy
          prod(arange(1.0,50.0))

Out[161]: 6.0828186403426752e+62

In [162]: dprod.dprod(arange(1,10), 5) # only the 5 first elements

Out[162]: 120.0
```

Compare performance:

```
In [163]: xvec = rand(500)

In [164]: timeit dprod.dprod(xvec)

The slowest run took 5.61 times longer than the fastest. This could mean that an intermediate result is
1000000 loops, best of 3: 1.63 µs per loop

In [165]: timeit xvec.prod()

The slowest run took 6.64 times longer than the fastest. This could mean that an intermediate result is
100000 loops, best of 3: 8.46 µs per loop
```

7.1.4 Example 2: cumulative sum, vector input and vector output

The cummulative sum function for an array of data is a good example of a loop intense algorithm: Loop through a vector and store the cummulative sum in another vector.

```
In [166]: # simple python algorithm: example of a SLOW implementation
          # Why? Because the loop is implemented in python.
          def py_dcumsum(a):
              b = empty_like(a)
              b[0] = a[0]
              for n in range(1,len(a)):
                  b[n] = b[n-1]+a[n]
              return b
```

Fortran subroutine for the same thing: here we have added the `intent(in)` and `intent(out)` as comment lines in the original fortran code, so we do not need to manually edit the fortran module declaration file generated by `f2py`.

```
In [167]: %%file dcumsum.f
          c File dcumsum.f
                 subroutine dcumsum(a, b, n)
                 double precision a(n)
                 double precision b(n)
                 integer n
          cf2py  intent(in) :: a
          cf2py  intent(out) :: b
```

```
cf2py  intent(hide) :: n

       b(1) = a(1)
       do 100 i=2, n
           b(i) = b(i-1) + a(i)
100    continue
       end
```

```
Overwriting dcumsum.f
```

We can directly compile the fortran code to a python module:

```
In [168]: !f2py3 -c dcumsum.f -m dcumsum
```

```
running build
running config_cc
unifing config_cc, config, build_clib, build_ext, build commands --compiler options
running config_fc
unifing config_fc, config, build_clib, build_ext, build commands --fcompiler options
running build_src
build_src
building extension "dcumsum" sources
f2py options: []
f2py:> /tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsummodule.c
creating /tmp/tmpe46xtmge/src.linux-x86_64-3.4
Reading fortran codes...
        Reading file 'dcumsum.f' (format:fix,strict)
Post-processing...
        Block: dcumsum
                        Block: dcumsum
Post-processing (stage 2)...
Building modules...
        Building module "dcumsum"...
                Constructing wrapper function "dcumsum"...
                  b = dcumsum(a)
        Wrote C/API module "dcumsum" to file "/tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsummodule.c"
  adding '/tmp/tmpe46xtmge/src.linux-x86_64-3.4/fortranobject.c' to sources.
  adding '/tmp/tmpe46xtmge/src.linux-x86_64-3.4' to include_dirs.
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.c -> /tmp/tmpe
copying /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/f2py/src/fortranobject.h -> /tmp/tmpe
build_src: building npy-pkg config files
running build_ext
customize UnixCCompiler
customize UnixCCompiler using build_ext
customize Gnu95FCompiler
Found executable /usr/bin/gfortran
customize Gnu95FCompiler
customize Gnu95FCompiler using build_ext
building 'dcumsum' extension
compiling C sources
C compiler: gcc -pthread -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC

creating /tmp/tmpe46xtmge/tmp
creating /tmp/tmpe46xtmge/tmp/tmpe46xtmge
creating /tmp/tmpe46xtmge/tmp/tmpe46xtmge/src.linux-x86_64-3.4
```

```
compile options: '-I/tmp/tmpe46xtmge/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gcc: /tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsummodule.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/arra
                 from /tmp/tmpe46xtmge/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsummodule.c:18:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
/tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsummodule.c:111:12: warning: 'f2py_size' defined but not used
 static int f2py_size(PyArrayObject* var, ...)
            ^
gcc: /tmp/tmpe46xtmge/src.linux-x86_64-3.4/fortranobject.c
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/arra
                 from /tmp/tmpe46xtmge/src.linux-x86_64-3.4/fortranobject.h:13,
                 from /tmp/tmpe46xtmge/src.linux-x86_64-3.4/fortranobject.c:2:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
compiling Fortran sources
Fortran f77 compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -fPIC -O3 -funroll
Fortran f90 compiler: /usr/bin/gfortran -Wall -g -fno-second-underscore -fPIC -O3 -funroll-loops
Fortran fix compiler: /usr/bin/gfortran -Wall -g -ffixed-form -fno-second-underscore -Wall -g -fno-secon
compile options: '-I/tmp/tmpe46xtmge/src.linux-x86_64-3.4 -I/home/dhavide/anaconda3/lib/python3.4/site-p
gfortran:f77: dcumsum.f
/usr/bin/gfortran -Wall -g -Wall -g -shared /tmp/tmpe46xtmge/tmp/tmpe46xtmge/src.linux-x86_64-3.4/dcumsu
Removing build directory /tmp/tmpe46xtmge
```

```
In [169]: import dcumsum

In [170]: a = array([1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0])

In [171]: py_dcumsum(a)

Out[171]: array([  1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.])

In [172]: dcumsum.dcumsum(a)

Out[172]: array([  1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.])

In [173]: cumsum(a)

Out[173]: array([  1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.])
```

Benchmark the different implementations:

```
In [174]: a = rand(10000)

In [175]: timeit py_dcumsum(a)

100 loops, best of 3: 5.41 ms per loop

In [176]: timeit dcumsum.dcumsum(a)

10000 loops, best of 3: 18.5 µs per loop

In [177]: timeit a.cumsum()

The slowest run took 17.25 times longer than the fastest. This could mean that an intermediate result i
10000 loops, best of 3: 47 µs per loop
```

7.1.5 Further reading

1. http://www.scipy.org/F2py
2. http://dsnra.jpl.nasa.gov/software/Python/F2PY_tutorial.pdf
3. http://www.shocksolution.com/2009/09/f2py-binding-fortran-python/

7.2 C

7.3 ctypes

ctypes is a Python library for calling out to C code. It is not as automatic as `f2py`, and we manually need to load the library and set properties such as the functions return and argument types. On the otherhand we do not need to touch the C code at all.

```
In [178]: %%file functions.c

          #include <stdio.h>

          void hello(int n);

          double dprod(double *x, int n);

          void dcumsum(double *a, double *b, int n);

          void
          hello(int n)
          {
              int i;

              for (i = 0; i < n; i++)
              {
                  printf("C says hello\n");
              }
          }


          double
          dprod(double *x, int n)
          {
              int i;
              double y = 1.0;

              for (i = 0; i < n; i++)
              {
                  y *= x[i];
              }

              return y;
          }

          void
          dcumsum(double *a, double *b, int n)
          {
              int i;
```

```
    b[0] = a[0];
    for (i = 1; i < n; i++)
    {
        b[i] = a[i] + b[i-1];
    }
}
```

```
Overwriting functions.c
```

Compile the C file into a shared library:

```
In [179]: !gcc -c -Wall -O2 -Wall -ansi -pedantic -fPIC -o functions.o functions.c
          !gcc -o libfunctions.so -shared functions.o
```

The result is a compiled shared library `libfunctions.so`:

```
In [180]: !file libfunctions.so
```

```
libfunctions.so: ELF 64-bit LSB  shared object, x86-64, version 1 (SYSV), dynamically linked, BuildID[sh
```

Now we need to write wrapper functions to access the C library: To load the library we use the ctypes package, which included in the Python standard library (with extensions from numpy for passing arrays to C). Then we manually set the types of the argument and return values (no automatic code inspection here!).

```
In [181]: %%file functions.py

          import numpy
          import ctypes

          _libfunctions = numpy.ctypeslib.load_library('libfunctions', '.')

          _libfunctions.hello.argtypes = [ctypes.c_int]
          _libfunctions.hello.restype  =  ctypes.c_void_p

          _libfunctions.dprod.argtypes = [numpy.ctypeslib.ndpointer(dtype=numpy.float), ctypes.c_int]
          _libfunctions.dprod.restype  = ctypes.c_double

          _libfunctions.dcumsum.argtypes = [numpy.ctypeslib.ndpointer(dtype=numpy.float), numpy.ctypesli
          _libfunctions.dcumsum.restype  = ctypes.c_void_p

          def hello(n):
              return _libfunctions.hello(int(n))

          def dprod(x, n=None):
              if n is None:
                  n = len(x)
              x = numpy.asarray(x, dtype=numpy.float)
              return _libfunctions.dprod(x, int(n))

          def dcumsum(a, n):
              a = numpy.asarray(a, dtype=numpy.float)
              b = numpy.empty(len(a), dtype=numpy.float)
              _libfunctions.dcumsum(a, b, int(n))
              return b
```

```
Overwriting functions.py
```

```
In [182]: %%file run_hello_c.py

          import functions

          functions.hello(3)
```

```
Overwriting run_hello_c.py
```

```
In [183]: !python run_hello_c.py
```

```
C says hello
C says hello
C says hello
```

```
In [184]: import functions
```

7.3.1 Product function:

```
In [185]: functions.dprod([1,2,3,4,5])

Out[185]: 120.0
```

7.3.2 Cumulative sum:

```
In [186]: a = rand(100000)

In [187]: res_c = functions.dcumsum(a, len(a))

In [188]: res_fortran = dcumsum.dcumsum(a)

In [189]: res_c - res_fortran

Out[189]: array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

7.3.3 Simple benchmark

```
In [190]: timeit functions.dcumsum(a, len(a))
```

1000 loops, best of 3: 517 μs per loop

```
In [191]: timeit dcumsum.dcumsum(a)
```

1000 loops, best of 3: 247 μs per loop

```
In [192]: timeit a.cumsum()
```

1000 loops, best of 3: 517 μs per loop

7.3.4 Further reading

- http://docs.python.org/2/library/ctypes.html
- http://www.scipy.org/Cookbook/Ctypes

7.4 Cython

A hybrid between python and C that can be compiled: Basically Python code with type declarations.

```
In [193]: %%file cy_dcumsum.pyx

          cimport numpy

          def dcumsum(numpy.ndarray[numpy.float64_t, ndim=1] a, numpy.ndarray[numpy.float64_t, ndim=1] b
              cdef int i, n = len(a)
              b[0] = a[0]
              for i from 1 <= i < n:
                  b[i] = b[i-1] + a[i]
              return b
```

```
Overwriting cy_dcumsum.pyx
```

A build file for generating C code and compiling it into a Python module.

```
In [194]: %%file setup.py
          import numpy
          from distutils.core import setup
          from distutils.extension import Extension
          from Cython.Distutils import build_ext

          setup(
              cmdclass = {'build_ext': build_ext},
              ext_modules = [Extension("cy_dcumsum", ["cy_dcumsum.pyx"], include_dirs=[numpy.get_include
              ],
          )
```

```
Overwriting setup.py
```

```
In [195]: !python setup.py build_ext --inplace
```

```
running build_ext
cythoning cy_dcumsum.pyx to cy_dcumsum.c
building 'cy_dcumsum' extension
gcc -pthread -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC -I/home/dhavide/anaconda3/lib/pytho
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/array
                 from cy_dcumsum.c:257:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:15
 #warning "Using deprecated NumPy API, disable it by " \
  ^
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ndar
                 from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/array
                 from cy_dcumsum.c:257:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/__multiarray_api.h:1629:1: 
 _import_array(void)
 ^
In file included from /home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/ufunc
                 from cy_dcumsum.c:258:
/home/dhavide/anaconda3/lib/python3.4/site-packages/numpy/core/include/numpy/__ufunc_api.h:241:1: warnin
 _import_umath(void)
```

```
 ^
gcc -pthread -shared -L/home/dhavide/anaconda3/lib -Wl,-rpath=/home/dhavide/anaconda3/lib,--no-as-neede
```

```
In [196]: import cy_dcumsum
```

```
In [197]: a = array([1,2,3,4], dtype=float)
          b = empty_like(a)
          cy_dcumsum.dcumsum(a,b)
          b
```

```
Out[197]: array([  1.,   3.,   6.,  10.])
```

```
In [198]: a = array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0])
```

```
In [199]: b = empty_like(a)
          cy_dcumsum.dcumsum(a, b)
          b
```

```
Out[199]: array([  1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.])
```

```
In [200]: py_dcumsum(a)
```

```
Out[200]: array([  1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.])
```

```
In [201]: a = rand(100000)
          b = empty_like(a)
```

```
In [202]: timeit py_dcumsum(a)
```

```
10 loops, best of 3: 72.7 ms per loop
```

```
In [203]: timeit cy_dcumsum.dcumsum(a,b)
```

```
1000 loops, best of 3: 469 µs per loop
```

7.4.1 Cython in the IPython notebook

When working with the IPython (especially in the notebook), there is a more convenient way of compiling and loading Cython code. Using the `%%cython` IPython magic (command to IPython), we can simply type the Cython code in a code cell and let IPython take care of the conversion to C code, compilation and loading of the function. To be able to use the `%%cython` magic, we first need to load the extension `cythonmagic`:

```
In [204]: %load_ext Cython
```

```
The Cython extension is already loaded. To reload it, use:
  %reload_ext Cython
```

```
In [205]: %%cython

          cimport numpy

          def cy_dcumsum2(numpy.ndarray[numpy.float64_t, ndim=1] a, numpy.ndarray[numpy.float64_t, ndim=
              cdef int i, n = len(a)
              b[0] = a[0]
              for i from 1 <= i < n:
                  b[i] = b[i-1] + a[i]
              return b
```

```
In [206]: timeit cy_dcumsum2(a,b)
```

```
1000 loops, best of 3: 552 µs per loop
```

7.4.2 Further reading

- http://cython.org
- http://docs.cython.org/src/userguide/tutorial.html
- http://wiki.cython.org/tutorials/numpy

Figure 7.1: Continuum Logo

CONTINUUM ANALYTICS

Chapter 8
Tools for high-performance computing applications

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
```

8.1 multiprocessing

Python has a built-in process-based library for concurrent computing, called `multiprocessing`.

```
In [2]: import multiprocessing
        import os
        import time
        import numpy

In [3]: def task(args):
            print("PID =", os.getpid(), ", args =", args)

            return os.getpid(), args

In [4]: task("test")

PID = 11411 , args = test

Out[4]: (11411, 'test')

In [5]: pool = multiprocessing.Pool(processes=4)

In [6]: result = pool.map(task, [1,2,3,4,5,6,7,8])

PID = 11425 , args = 2
PID = 11426 , args = 3
PID = 11427 , args = 4
PID = 11424 , args = 1
PID = 11425 , args = 7
PID = 11426 , args = 6
PID = 11427 , args = 5
PID = 11427 , args = 8
```

```
In [7]: result

Out[7]: [(11424, 1),
         (11425, 2),
         (11426, 3),
         (11427, 4),
         (11427, 5),
         (11426, 6),
         (11425, 7),
         (11427, 8)]
```

The multiprocessing package is very useful for highly parallel tasks that do not need to communicate with each other, other than when sending the initial data to the pool of processes and when and collecting the results.

8.2 IPython parallel

IPython includes a very interesting and versatile parallel computing environment, which is very easy to use. It builds on the concept of ipython engines and controllers, that one can connect to and submit tasks to. To get started using this framework for parallel computing, one first have to start up an IPython cluster of engines. The easiest way to do this is to use the `ipcluster` command,

```
$ ipcluster start -n 4
```

Or, alternatively, from the "Clusters" tab on the IPython notebook dashboard page. This will start 4 IPython engines on the current host, which is useful for multicore systems. It is also possible to setup IPython clusters that spans over many nodes in a computing cluster. For more information about possible use cases, see the official documentation Using IPython for parallel computing.

To use the IPython cluster in our Python programs or notebooks, we start by creating an instance of `IPython.parallel.Client`:

```
In [8]: from IPython.parallel import Client

In [9]: cli = Client()
```

Using the 'ids' attribute we can retreive a list of ids for the IPython engines in the cluster:

```
In [10]: cli.ids

Out[10]: [0, 1, 2, 3]
```

Each of these engines are ready to execute tasks. We can selectively run code on individual engines:

```
In [11]: def getpid():
             """ return the unique ID of the current process """
             import os
             return os.getpid()

In [12]: # first try it on the notebook process
         getpid()

Out[12]: 11411

In [13]: # run it on one of the engines
         cli[0].apply_sync(getpid)

Out[13]: 11464
```

```
In [14]: # run it on ALL of the engines at the same time
         cli[:].apply_sync(getpid)
```

```
Out[14]: [11464, 11478, 11477, 11479]
```

We can use this cluster of IPython engines to execute tasks in parallel. The easiest way to dispatch a function to different engines is to define the function with the decorator:

```
@view.parallel(block=True)
```

Here, `view` is supposed to be the engine pool which we want to dispatch the function (task). Once our function is defined this way we can dispatch it to the engine using the `map` method in the resulting class (in Python, a decorator is a language construct which automatically wraps the function into another function or a class).

To see how all this works, lets look at an example:

```
In [15]: dview = cli[:]
```

```
In [16]: @dview.parallel(block=True)
         def dummy_task(delay):
             """ a dummy task that takes 'delay' seconds to finish """
             import os, time

             t0 = time.time()
             pid = os.getpid()
             time.sleep(delay)
             t1 = time.time()

             return [pid, t0, t1]
```

```
In [17]: # generate random delay times for dummy tasks
         delay_times = numpy.random.rand(4)
```

Now, to map the function `dummy_task` to the random delay time data, we use the `map` method in `dummy_task`:

```
In [18]: dummy_task.map(delay_times)
```

```
Out[18]: [[11464, 1439847252.0649436, 1439847252.9168468],
          [11478, 1439847252.0697844, 1439847252.7821367],
          [11477, 1439847252.0755847, 1439847252.5272865],
          [11479, 1439847252.0843427, 1439847252.6486437]]
```

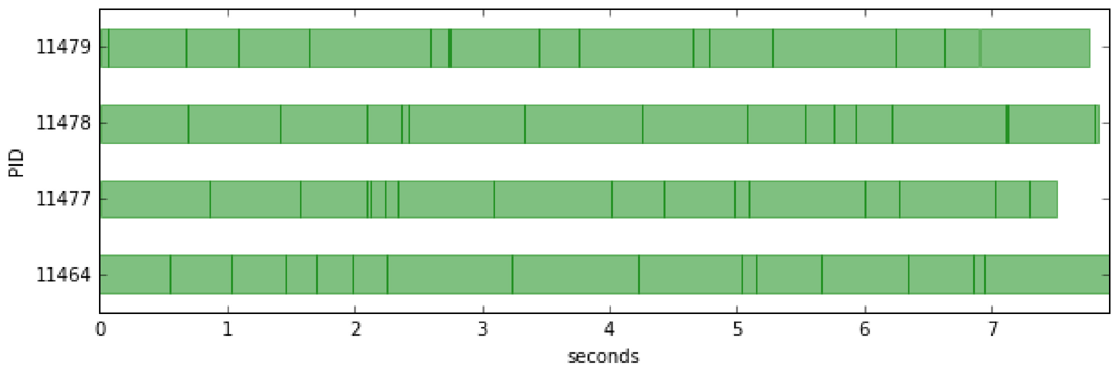
Let's do the same thing again with many more tasks and visualize how these tasks are executed on different IPython engines:

```
In [19]: def visualize_tasks(results):
             res = numpy.array(results)
             fig, ax = plt.subplots(figsize=(10, res.shape[1]))

             yticks = []
             yticklabels = []
             tmin = min(res[:,1])
             for n, pid in enumerate(numpy.unique(res[:,0])):
                 yticks.append(n)
                 yticklabels.append("%d" % pid)
                 for m in numpy.where(res[:,0] == pid)[0]:
```

```
                    ax.add_patch(plt.Rectangle((res[m,1] - tmin, n-0.25),
                                 res[m,2] - res[m,1], 0.5, color="green", alpha=0.5))

            ax.set_ylim(-.5, n+.5)
            ax.set_xlim(0, max(res[:,2]) - tmin + 0.)
            ax.set_yticks(yticks)
            ax.set_yticklabels(yticklabels)
            ax.set_ylabel("PID")
            ax.set_xlabel("seconds")
```

```
In [20]: delay_times = numpy.random.rand(64)
```

```
In [21]: result = dummy_task.map(delay_times)
         visualize_tasks(result)
```



That's a nice and easy parallelization! We can see that we utilize all four engines quite well.

But one short coming so far is that the tasks are not load balanced, so one engine might be idle while others still have more tasks to work on.

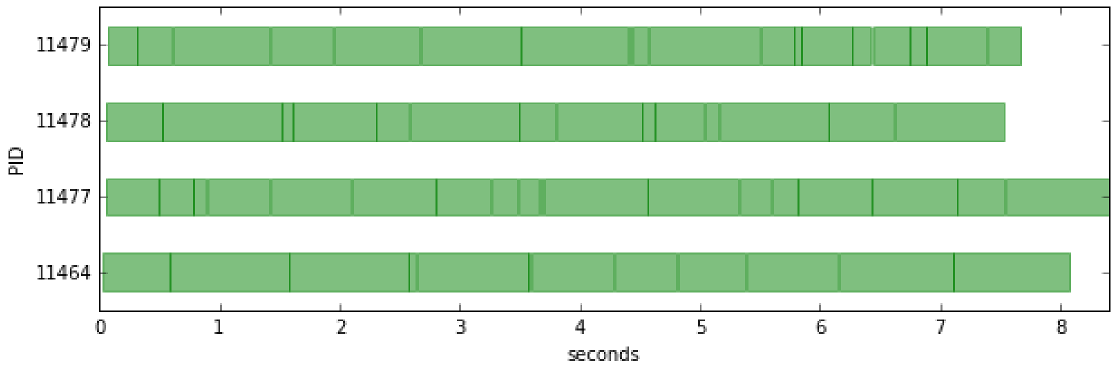
However, the IPython parallel environment provides a number of alternative "views" of the engine cluster, and there is a view that provides load balancing as well (above we have used the "direct view", which is why we called it "dview").

To obtain a load balanced view we simply use the `load_balanced_view` method in the engine cluster client instance `cli`:

```
In [22]: lbview = cli.load_balanced_view()
```

```
In [23]: @lbview.parallel(block=True)
         def dummy_task_load_balanced(delay):
             """ a dummy task that takes 'delay' seconds to finish """
             import os, time

             t0 = time.time()
             pid = os.getpid()
             time.sleep(delay)
             t1 = time.time()

             return [pid, t0, t1]
```

```
In [24]: result = dummy_task_load_balanced.map(delay_times)
         visualize_tasks(result)
```



In the example above we can see that the engine cluster is a bit more efficiently used, and the time to completion is shorter than in the previous example.

8.2.1 Further reading

There are many other ways to use the IPython parallel environment. The official documentation has a nice guide:

- http://ipython.org/ipython-doc/dev/parallel/

8.3 MPI

When more communication between processes is required, sophisticated solutions such as MPI and OpenMP are often needed. MPI is process based parallel processing library/protocol, and can be used in Python programs through the `mpi4py` package:

http://mpi4py.scipy.org/

To use the `mpi4py` package we include `MPI` from `mpi4py`:

```
from mpi4py import MPI
```

A MPI python program must be started using the `mpirun -n N` command, where `N` is the number of processes that should be included in the process group.

Note that the IPython parallel enviroment also has support for MPI, but to begin with we will use `mpi4py` and the `mpirun` in the follow examples.

8.3.1 Example 1

```
In [30]: %%file mpitest.py

         from mpi4py import MPI

         comm = MPI.COMM_WORLD
         rank = comm.Get_rank()

         if rank == 0:
            data = [1.0, 2.0, 3.0, 4.0]
            comm.send(data, dest=1, tag=11)
         elif rank == 1:
            data = comm.recv(source=0, tag=11)

         print("rank =%d, data =%s" % (rank,data))
```

```
Overwriting mpitest.py
```

```
In [31]: !mpirun -n 2 python mpitest.py
```

```
rank =0, data =[1.0, 2.0, 3.0, 4.0]
rank =1, data =[1.0, 2.0, 3.0, 4.0]
```

8.3.2 Example 2

Send a numpy array from one process to another:

```
In [34]: %%file mpi-numpy-array.py

         from mpi4py import MPI
         import numpy

         comm = MPI.COMM_WORLD
         rank = comm.Get_rank()

         if rank == 0:
            data = numpy.random.rand(10)
            comm.Send(data, dest=1, tag=13)
         elif rank == 1:
            data = numpy.empty(10, dtype=numpy.float64)
            comm.Recv(data, source=0, tag=13)

         print("rank =%d, data =%s" % (rank,data))
```

```
Overwriting mpi-numpy-array.py
```

```
In [35]: !mpirun -n 2 python mpi-numpy-array.py
```

```
rank =0, data =[ 0.36359812  0.55163426  0.02430927  0.91038764  0.79185896  0.20713193
  0.75367627  0.64960013  0.9828266   0.69653435]
rank =1, data =[ 0.36359812  0.55163426  0.02430927  0.91038764  0.79185896  0.20713193
  0.75367627  0.64960013  0.9828266   0.69653435]
```

8.3.3 Example 3: Matrix-vector multiplication

```
In [36]: # prepare some random data
         N = 16
         A = numpy.random.rand(N, N)
         numpy.save("random-matrix.npy", A)
         x = numpy.random.rand(N)
         numpy.save("random-vector.npy", x)
```

```
In [37]: %%file mpi-matrix-vector.py

         from mpi4py import MPI
         import numpy

         comm = MPI.COMM_WORLD
         rank = comm.Get_rank()
         p = comm.Get_size()

         def matvec(comm, A, x):
```

```
        m = A.shape[0] / p
        y_part = numpy.dot(A[rank * m:(rank+1)*m], x)
        y = numpy.zeros_like(x)
        comm.Allgather([y_part,  MPI.DOUBLE], [y, MPI.DOUBLE])
        return y

    A = numpy.load("random-matrix.npy")
    x = numpy.load("random-vector.npy")
    y_mpi = matvec(comm, A, x)

    if rank == 0:
        y = numpy.dot(A, x)
        print(y_mpi)
        print("sum(y - y_mpi) = %f" % (y - y_mpi).sum())
```

```
Writing mpi-matrix-vector.py
```

In [38]:
```
!mpirun -n 4 python mpi-matrix-vector.py
```

```
[ 4.92983554  4.07883074  4.2398615   4.7207529   4.56465512  4.60331962
  5.40504466  3.35139743  3.70229     3.98073722  4.18009121  4.91292805
  3.35528324  4.48848917  3.87617524  4.02399674]
sum(y - y_mpi) = 0.000000
```

8.3.4 Example 4: Sum of the elements in a vector

In [39]:
```
# prepare some random data
N = 128
a = numpy.random.rand(N)
numpy.save("random-vector.npy", a)
```

In [41]:
```
%%file mpi-psum.py

from mpi4py import MPI
import numpy as np

def psum(a):
    r = MPI.COMM_WORLD.Get_rank()
    size = MPI.COMM_WORLD.Get_size()
    m = len(a) / size
    locsum = np.sum(a[r*m:(r+1)*m])
    rcvBuf = np.array(0.0, 'd')
    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE], [rcvBuf, MPI.DOUBLE], op=MPI.SUM)
    return rcvBuf

a = np.load("random-vector.npy")
s = psum(a)

if MPI.COMM_WORLD.Get_rank() == 0:
    print("sum = %f, numpy sum = %f" % (s,a.sum()))
```

```
Overwriting mpi-psum.py
```

In [42]:
```
!mpirun -n 4 python mpi-psum.py
```

```
sum = 58.909508, numpy sum = 58.909508
```

8.3.5 Further reading

- http://mpi4py.scipy.org
- http://mpi4py.scipy.org/docs/usrman/tutorial.html
- https://computing.llnl.gov/tutorials/mpi/

8.4 OpenMP

What about OpenMP? OpenMP is a standard and widely used thread-based parallel API that unfortunaltely is **not** useful directly in Python. The reason is that the CPython implementation use a global interpreter lock, making it impossible to simultaneously run several Python threads. Threads are therefore not useful for parallel computing in Python, unless it is only used to wrap compiled code that do the OpenMP parallelization (Numpy can do something like that).

This is clearly a limitation in the Python interpreter, and as a consequence all parallelization in Python must use processes (not threads).

However, there is a way around this that is not that painful. When calling out to compiled code the GIL is released, and it is possible to write Python-like code in Cython where we can selectively release the GIL and do OpenMP computations.

```
In [43]: N_core = multiprocessing.cpu_count()

         print("This system has %d cores" % N_core)
```

```
This system has 2 cores
```

Here is a simple example that shows how OpenMP can be used via cython:

```
In [44]: %load_ext Cython
```

```
In [45]: %%cython -f -c-fopenmp --link-args=-fopenmp -c-g

         cimport cython
         cimport numpy
         from cython.parallel import prange, parallel
         cimport openmp


         def cy_openmp_test():
             cdef int n, N
             # release GIL so that we can use OpenMP
             with nogil, parallel():
                 N = openmp.omp_get_num_threads()
                 n = openmp.omp_get_thread_num()
                 with gil:
                     print("Number of threads %d: thread number %d" % (N, n))
```

```
In [46]: cy_openmp_test()
```

```
Number of threads 2: thread number 0
Number of threads 2: thread number 1
```

8.4.1 Example: matrix vector multiplication

```
In [47]: # prepare some random data
         N = 4 * N_core

         M = numpy.random.rand(N, N)
         x = numpy.random.rand(N)
         y = numpy.zeros_like(x)
```

Let's first look at a simple implementation of matrix-vector multiplication in Cython:

```
In [48]: %%cython

         cimport cython
         cimport numpy
         import numpy

         @cython.boundscheck(False)
         @cython.wraparound(False)
         def cy_matvec(numpy.ndarray[numpy.float64_t, ndim=2] M,
                       numpy.ndarray[numpy.float64_t, ndim=1] x,
                       numpy.ndarray[numpy.float64_t, ndim=1] y):

             cdef int i, j, n = len(x)

             for i from 0 <= i < n:
                 for j from 0 <= j < n:
                     y[i] += M[i, j] * x[j]

             return y
```

```
In [49]: # check that we get the same results
         y = numpy.zeros_like(x)
         cy_matvec(M, x, y)
         numpy.dot(M, x) - y
```

```
Out[49]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [50]: %timeit numpy.dot(M, x)
```

```
The slowest run took 465.61 times longer than the fastest. This could mean that an intermediate result i
1000000 loops, best of 3: 2.15 µs per loop
```

```
In [51]: %timeit cy_matvec(M, x, y)
```

```
The slowest run took 4.91 times longer than the fastest. This could mean that an intermediate result is
100000 loops, best of 3: 3.29 µs per loop
```

The Cython implementation here is a bit slower than numpy.dot, but not by much, so if we can use multiple cores with OpenMP it should be possible to beat the performance of numpy.dot.

```
In [52]: %%cython -f -c-fopenmp --link-args=-fopenmp -c-g

         cimport cython
         cimport numpy
         from cython.parallel import parallel
         cimport openmp
```

```
@cython.boundscheck(False)
@cython.wraparound(False)
def cy_matvec_omp(numpy.ndarray[numpy.float64_t, ndim=2] M,
                  numpy.ndarray[numpy.float64_t, ndim=1] x,
                  numpy.ndarray[numpy.float64_t, ndim=1] y):

    cdef int i, j, n = len(x), N, r, m

    # release GIL, so that we can use OpenMP
    with nogil, parallel():
        N = openmp.omp_get_num_threads()
        r = openmp.omp_get_thread_num()
        m = n / N

        for i from 0 <= i < m:
            for j from 0 <= j < n:
                y[r * m + i] += M[r * m + i, j] * x[j]

    return y
```

```
In [53]: # check that we get the same results
         y = numpy.zeros_like(x)
         cy_matvec_omp(M, x, y)
         numpy.dot(M, x) - y

Out[53]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [54]: %timeit numpy.dot(M, x)

The slowest run took 297.37 times longer than the fastest. This could mean that an intermediate result i
1000000 loops, best of 3: 1.93 µs per loop

In [55]: %timeit cy_matvec_omp(M, x, y)

The slowest run took 69.79 times longer than the fastest. This could mean that an intermediate result is
100000 loops, best of 3: 10.3 µs per loop
```

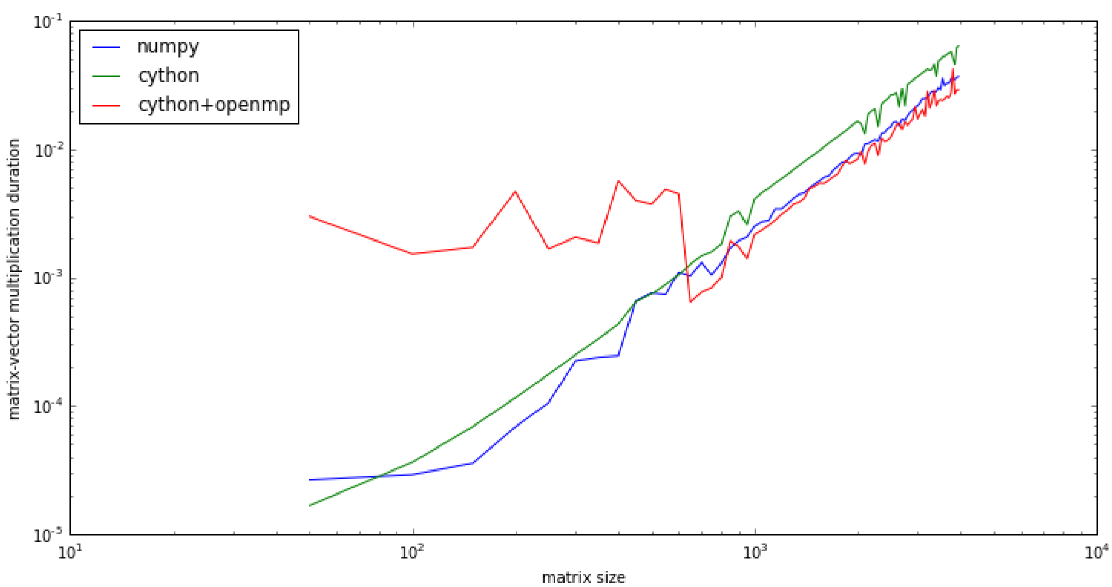
Now, this implementation is much slower than numpy.dot for this problem size, because of overhead associated with OpenMP and threading, etc. But let's look at the how the different implementations compare with larger matrix sizes:

```
In [56]: N_vec  = numpy.arange(25, 2000, 25) * N_core

In [58]: duration_ref = numpy.zeros(len(N_vec))
         duration_cy = numpy.zeros(len(N_vec))
         duration_cy_omp = numpy.zeros(len(N_vec))

         for idx, N in enumerate(N_vec):

             M = numpy.random.rand(N, N)
             x = numpy.random.rand(N)
             y = numpy.zeros_like(x)

             t0 = time.time()
             numpy.dot(M, x)
```

```
        duration_ref[idx] = time.time() - t0

        t0 = time.time()
        cy_matvec(M, x, y)
        duration_cy[idx] = time.time() - t0

        t0 = time.time()
        cy_matvec_omp(M, x, y)
        duration_cy_omp[idx] = time.time() - t0
```

In [59]:
```
fig, ax = plt.subplots(figsize=(12, 6))

ax.loglog(N_vec, duration_ref, label='numpy')
ax.loglog(N_vec, duration_cy, label='cython')
ax.loglog(N_vec, duration_cy_omp, label='cython+openmp')

ax.legend(loc=2)
ax.set_yscale("log")
ax.set_ylabel("matrix-vector multiplication duration")
ax.set_xlabel("matrix size");
```

Out[59]: <matplotlib.text.Text at 0x7f536af122b0>



For large problem sizes the the cython+OpenMP implementation is faster than numpy.dot. With this simple implementation, the speedup for large problem sizes is about:

In [60]:
```
((duration_ref / duration_cy_omp)[-10:]).mean()
```

Out[60]: 1.2483748994665467

Obviously one could do a better job with more effort, since the theoretical limit of the speed-up is:

In [61]:
```
N_core
```

Out[61]: 2

8.4.2 Further reading

- http://openmp.org
- http://docs.cython.org/src/userguide/parallelism.html

8.5 OpenCL

OpenCL is an API for heterogenous computing, for example using GPUs for numerical computations. There is a python package called `pyopencl` that allows OpenCL code to be compiled, loaded and executed on the compute units completely from within Python. This is a nice way to work with OpenCL, because the time-consuming computations should be done on the compute units in compiled code, and in this Python only server as a control language.

```
In [ ]: %%file opencl-dense-mv.py

        import pyopencl as cl
        import numpy
        import time

        # problem size
        n = 10000

        # platform
        platform_list = cl.get_platforms()
        platform = platform_list[0]

        # device
        device_list = platform.get_devices()
        device = device_list[0]

        if False:
            print("Platform name:" + platform.name)
            print("Platform version:" + platform.version)
            print("Device name:" + device.name)
            print("Device type:" + cl.device_type.to_string(device.type))
            print("Device memory: " + str(device.global_mem_size//1024//1024) + ' MB')
            print("Device max clock speed:" + str(device.max_clock_frequency) + ' MHz')
            print("Device compute units:" + str(device.max_compute_units))

        # context
        ctx = cl.Context([device]) # or we can use cl.create_some_context()

        # command queue
        queue = cl.CommandQueue(ctx)

        # kernel
        KERNEL_CODE = """
        //
        // Matrix-vector multiplication: r = m * v
        //
        #define N %(mat_size)d
        __kernel
        void dmv_cl(__global float *m, __global float *v, __global float *r)
        {
```

```
    int i, gid = get_global_id(0);

    r[gid] = 0;
    for (i = 0; i < N; i++)
    {
        r[gid] += m[gid * N + i] * v[i];
    }
}
"""

kernel_params = {"mat_size": n}
program = cl.Program(ctx, KERNEL_CODE % kernel_params).build()

# data
A = numpy.random.rand(n, n)
x = numpy.random.rand(n, 1)

# host buffers
h_y = numpy.empty(numpy.shape(x)).astype(numpy.float32)
h_A = numpy.real(A).astype(numpy.float32)
h_x = numpy.real(x).astype(numpy.float32)

# device buffers
mf = cl.mem_flags
d_A_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_A)
d_x_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_x)
d_y_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=h_y.nbytes)

# execute OpenCL code
t0 = time.time()
event = program.dmv_cl(queue, h_y.shape, None, d_A_buf, d_x_buf, d_y_buf)
event.wait()
cl.enqueue_copy(queue, h_y, d_y_buf)
t1 = time.time()

print("opencl elapsed time =", (t1-t0))

# Same calculation with numpy
t0 = time.time()
y = numpy.dot(h_A, h_x)
t1 = time.time()

print("numpy elapsed time =", (t1-t0))

# see if the results are the same
print("max deviation =", numpy.abs(y-h_y).max())
```

```
% python opencl-dense-mv.py
('opencl elapsed time =', 0.05729985237121582)
('numpy elapsed time =', 0.023556947708129883)
('max deviation =', 0.015380859)
```

8.5.1 Further reading

- http://mathema.tician.de/software/pyopencl

Chapter 9

Revision control software

```
In [1]: from IPython.display import Image
```

In any software development, one of the most important tools are revision control software (RCS).

They are used in virtually all software development and in all environments, by everyone and everywhere (no kidding!)

RCS can used on almost any digital content, so it is not only restricted to software development, and is also very useful for manuscript files, figures, data and notebooks!

9.1 There are two main purposes of RCS systems:

1. Keep track of changes in the source code.
    - Allow reverting back to an older revision if something goes wrong.
    - Work on several "branches" of the software concurrently.
    - Tags revisions to keep track of which version of the software that was used for what (for example, "release-1.0", "paper-A-final", ...)
2. Make it possible for serveral people to collaboratively work on the same code base simultaneously.
    - Allow many authors to make changes to the code.
    - Clearly communicating and visualizing changes in the code base to everyone involved.

9.2 Basic principles and terminology for RCS systems

In an RCS, the source code or digital content is stored in a **repository**.

- The repository does not only contain the latest version of all files, but the complete history of all changes to the files since they were added to the repository.
- A user can **checkout** the repository, and obtain a local working copy of the files. All changes are made to the files in the local working directory, where files can be added, removed and updated.
- When a task has been completed, the changes to the local files are **commited** (saved to the repository).

- If someone else has been making changes to the same files, a **conflict** can occur. In many cases conflicts can be **resolved** automatically by the system, but in some cases we might manually have to **merge** different changes together.
- It is often useful to create a new **branch** in a repository, or a **fork** or **clone** of an entire repository, when we doing larger experimental development. The main branch in a repository is called often **master** or **trunk**. When work on a branch or fork is completed, it can be merged in to the master branch/repository.
- With distributed RCSs such as GIT or Mercurial, we can **pull** and **push** changesets between different repositories. For example, between a local copy of there repository to a central online reposistory (for example on a community repository host site like github.com).

9.2.1 Some good RCS software

1. GIT (`git`) : http://git-scm.com/
2. Mercurial (`hg`) : http://mercurial.selenic.com/

In the rest of this lecture we will look at `git`, although `hg` is just as good and work in almost exactly the same way.

9.3 Installing git

On Linux:

```
% sudo apt-get install git
```

On Mac (with macports):

```
% sudo port install git
```

The first time you start to use git, you'll need to configure your author information:

```
% git config --global user.name 'Robert Johansson'
% git config --global user.email robert@riken.jp
```

9.4 Creating and cloning a repository

To create a brand new empty repository, we can use the command `git init repository-name`:

```
In [2]: # create a new git repository called gitdemo:
        !git init gitdemo
```

```
Initialized empty Git repository in /Users/dmertz/Drive/Modules/scientific-python-lectures/gitdemo/.git/
```

If we want to fork or clone an existing repository, we can use the command `git clone repository`:

```
In [3]: !git clone https://github.com/qutip/qutip
```

```
Cloning into 'qutip'...
remote: Counting objects: 21367, done.ote: Compressing objects: 100% (18/18), done.ote: Total 21367 (de
Receiving objects: 100% (21367/21367), 16.90 MiB | 1.93 MiB/s, done.
Resolving deltas: 100% (14319/14319), done.
Checking connectivity... done.
```

Git clone can take a URL to a public repository, like above, or a path to a local directory:

```
In [4]: !git clone gitdemo gitdemo2
```

```
Cloning into 'gitdemo2'...
warning: You appear to have cloned an empty repository.
done.
```

We can also clone private repositories over secure protocols such as SSH:

```
$ git clone ssh://myserver.com/myrepository
```

9.5 Status

Using the command `git status` we get a summary of the current status of the working directory. It shows if we have modified, added or removed files.

```
In [5]: !git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

In this case, only the current ipython notebook has been added. It is listed as an untracked file, and is therefore not in the repository yet.

9.6 Adding files and committing changes

To add a new file to the repository, we first create the file and then use the `git add filename` command:

```
In [6]: %%file README

        A file with information about the gitdemo repository.

Overwriting README
```

```
In [7]: !git status

On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README
        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

After having added the file `README`, the command `git status` list it as an *untracked* file.

```
In [8]: !git add README
```

```
In [9]: !git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex
        modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

Now that it has been added, it is listed as a *new file* that has not yet been commited to the repository.

```
In [10]: !git commit -m "Added a README file" README
```

```
[master 50c355a] Added a README file
 1 file changed, 1 insertion(+), 5 deletions(-)
```

```
In [11]: !git add Lecture-7-Revision-Control-Software.ipynb
```

```
In [12]: !git commit -m "added notebook file" Lecture-7-Revision-Control-Software.ipynb
```

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes not staged for commit:
        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
```

```
        modified:   Makefile
        modified:   Preamble.tex
        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
        .ipynb_checkpoints/
        Offline/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/

no changes added to commit
```

```
In [13]: !git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

After *committing* the change to the repository from the local working directory, `git status` again reports that working directory is clean.

9.7 Commiting changes

When files that is tracked by GIT are changed, they are listed as *modified* by `git status`:

In [14]:
```
%%file README

A file with information about the gitdemo repository.

A new line.
```

```
Overwriting README
```

In [15]:
```
!git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README
        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

Again, we can commit such changes to the repository using the `git commit -m "message"` command.

In [16]:
```
!git commit -m "added one more line in README" README
```

```
[master cb5c324] added one more line in README
 1 file changed, 3 insertions(+), 1 deletion(-)
```

In [17]: !git status

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

9.8 Removing files

To remove file that has been added to the repository, use `git rm filename`, which works similar to `git add filename`:

In [18]:
```
%%file tmpfile
A short-lived file.
```

```
Writing tmpfile
```

Add it:

```
In [19]: !git add tmpfile

In [20]: !git commit -m "adding file tmpfile" tmpfile

[master 4d74889] adding file tmpfile
 1 file changed, 1 insertion(+)
 create mode 100644 tmpfile
```

Remove it again:

```
In [21]: !git rm tmpfile

rm 'tmpfile'

In [22]: !git commit -m "remove file tmpfile" tmpfile

[master c380dc8] remove file tmpfile
 1 file changed, 1 deletion(-)
 delete mode 100644 tmpfile
```

9.9 Commit logs

The messages that are added to the commit command are supposed to give a short (often one-line) description of the changes/additions/deletions in the commit. If the `-m "message"` is omitted when invoking the `git commit` message an editor will be opened for you to type a commit message (for example useful when a longer commit message is requried).

We can look at the revision log by using the command `git log`:

```
In [23]: !git log -n 20

commit c380dc8356be86fde9b020d9d5f1a1fc00dd96ed
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:40 2015 -0700

    remove file tmpfile

commit 4d74889feb1df45411fe6bc915b559e6651b439a
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:38 2015 -0700

    adding file tmpfile

commit cb5c32499483f58ee4a7f56787f40a26d35e4109
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:37 2015 -0700

    added one more line in README

commit 50c355acc25c78d83d66a6a1e39748b68e62df09
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:33 2015 -0700

    Added a README file

commit 68103a5960f95c50833ac7db7622666226292537
```

```
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:58 2015 -0700

    added a line in expr1 branch

commit 25d440cbd963e56c912013a52307d2384fbc9042
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    remove file tmpfile

commit 493c55661ea291b8573dd1746dba4c6a91a2e502
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    adding file tmpfile

commit 0b35eb6602f1d9fd113caf770d507dbfd173fd84
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:53 2015 -0700

    added one more line in README

commit e90a8c109da701138e8455c01d3d9abc7289c079
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:51 2015 -0700

    added notebook file

commit 7771777a56f5bcf7ff5a423c04d0de52076479cf
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:50 2015 -0700

    Added a README file

commit c8eb6ef0d1e06545b276e5637aa8a783fcdedb21
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:20 2015 -0700

    added a line in expr1 branch

commit e61b95df741a4dc734b84b9453290c7e5dc3cc34
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    remove file tmpfile

commit af8e43e5a5406c9f5e01696efc3bd1d7eb2e69c4
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    adding file tmpfile

commit e3896b914fb9f9368c9ade44142448b5cddd0360
```

```
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    added one more line in README

commit cd9d289b9a9182fc3e08986dfe64cca02e0488d0
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:17 2015 -0700

    Added a README file

commit 4c349ecf589e95d1633072de1fe715a5750fa084
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:54 2015 -0700

    added lecture notebook about RCS

commit d051436004f5fd47af96ac797cc3bcc7a9b08581
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:37 2015 -0700

    added a line in expr1 branch

commit ddc57f12eca11f89c157f59595b9edda83dc3815
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:51 2015 -0700

    remove file tmpfile

commit 27e5d0fa081298c8451de903006fbba79ac0130b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:49 2015 -0700

    adding file tmpfile

commit 9783b909269b2f2c2234d00a20fd00d3b3f0bc1b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:44 2015 -0700

    added one more line in README
```

In the commit log, each revision is shown with a timestampe, a unique has tag that, and author information and the commit message.

9.10 Diffs

All commits results in a changeset, which has a "diff" describing the changes to the file associated with it. We can use `git diff` so see what has changed in a file:

```
In [24]: %%file README

         A file with information about the gitdemo repository.

         README files usually contains installation instructions, and information about how to get star
```

```
Overwriting README
```

In [25]: `!git diff README`

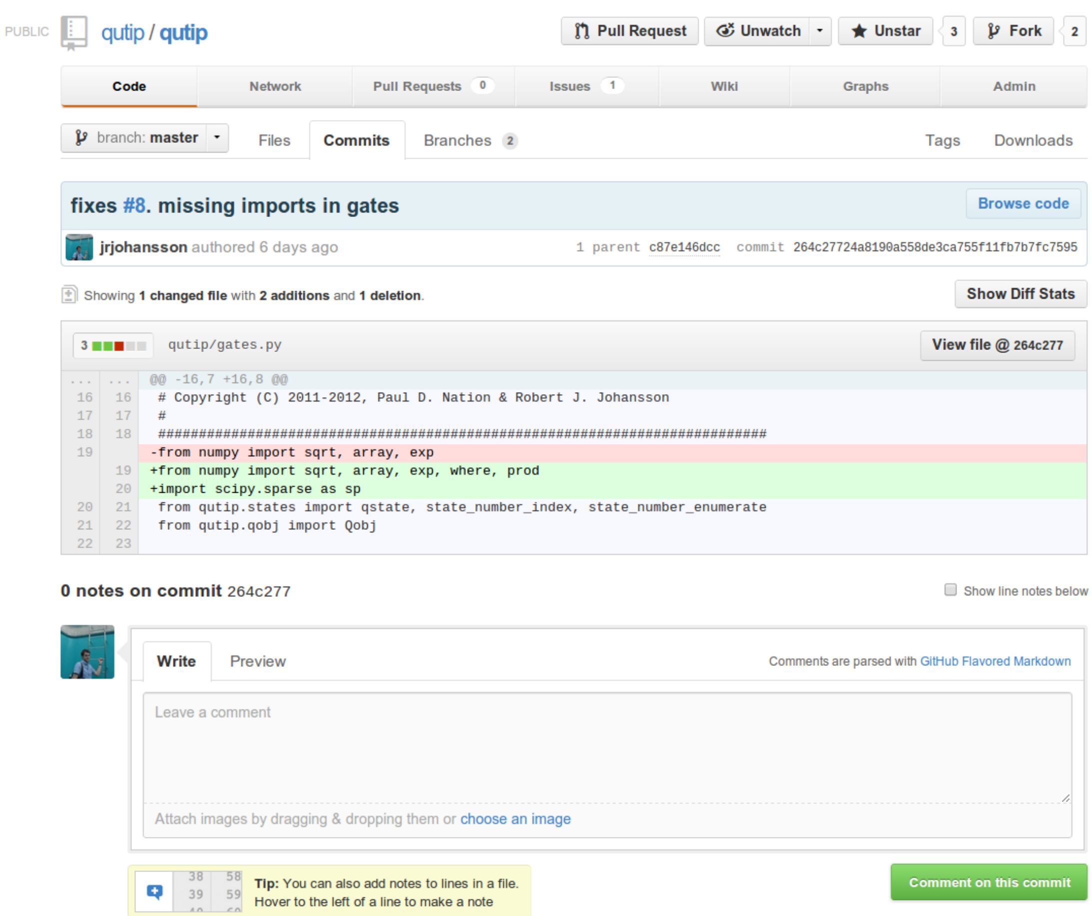
```
diff --git a/README b/README
index 4f51868..d3951c6 100644
--- a/README
+++ b/README
@@ -1,4 +1,4 @@

 A file with information about the gitdemo repository.

-A new line.
\ No newline at end of file
+README files usually contains installation instructions, and information about how to get started usin
\ No newline at end of file
```

That looks quite cryptic but is a standard form for describing changes in files. We can use other tools, like graphical user interfaces or web based systems to get a more easily understandable diff.

In github (a web-based GIT repository hosting service) it can look like this:

In [26]: `Image(filename='images/github-diff.png')`

Out[26]:



9.11 Discard changes in the working directory

To discard a change (revert to the latest version in the repository) we can use the `checkout` command like this:

```
In [27]: !git checkout -- README
```

```
In [28]: !git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

9.12 Checking out old revisions

If we want to get the code for a specific revision, we can use "git checkout" and giving it the hash code for the revision we are interested as argument:

```
In [29]: !git log -n 20
```

```
commit c380dc8356be86fde9b020d9d5f1a1fc00dd96ed
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:40 2015 -0700

    remove file tmpfile

commit 4d74889feb1df45411fe6bc915b559e6651b439a
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:38 2015 -0700

    adding file tmpfile

commit cb5c32499483f58ee4a7f56787f40a26d35e4109
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:37 2015 -0700

    added one more line in README

commit 50c355acc25c78d83d66a6a1e39748b68e62df09
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:33 2015 -0700

    Added a README file

commit 68103a5960f95c50833ac7db7622666226292537
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:58 2015 -0700

    added a line in expr1 branch

commit 25d440cbd963e56c912013a52307d2384fbc9042
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    remove file tmpfile

commit 493c55661ea291b8573dd1746dba4c6a91a2e502
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    adding file tmpfile

commit 0b35eb6602f1d9fd113caf770d507dbfd173fd84
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:53 2015 -0700

    added one more line in README

commit e90a8c109da701138e8455c01d3d9abc7289c079
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:51 2015 -0700

    added notebook file
```

```
commit 7771777a56f5bcf7ff5a423c04d0de52076479cf
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:50 2015 -0700

    Added a README file

commit c8eb6ef0d1e06545b276e5637aa8a783fcdedb21
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:20 2015 -0700

    added a line in expr1 branch

commit e61b95df741a4dc734b84b9453290c7e5dc3cc34
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    remove file tmpfile

commit af8e43e5a5406c9f5e01696efc3bd1d7eb2e69c4
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    adding file tmpfile

commit e3896b914fb9f9368c9ade44142448b5cddd0360
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    added one more line in README

commit cd9d289b9a9182fc3e08986dfe64cca02e0488d0
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:17 2015 -0700

    Added a README file

commit 4c349ecf589e95d1633072de1fe715a5750fa084
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:54 2015 -0700

    added lecture notebook about RCS

commit d051436004f5fd47af96ac797cc3bcc7a9b08581
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:37 2015 -0700

    added a line in expr1 branch

commit ddc57f12eca11f89c157f59595b9edda83dc3815
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:51 2015 -0700

    remove file tmpfile
```

```
commit 27e5d0fa081298c8451de903006fbba79ac0130b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:49 2015 -0700

    adding file tmpfile

commit 9783b909269b2f2c2234d00a20fd00d3b3f0bc1b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:44 2015 -0700

    added one more line in README
```

In [30]: `!git checkout 1f26ad648a791e266fbb951ef5c49b8d990e6461`

```
fatal: reference is not a tree: 1f26ad648a791e266fbb951ef5c49b8d990e6461
```

Now the content of all the files like in the revision with the hash code listed above (first revision)

In [31]: `!cat README`

```
A file with information about the gitdemo repository.

A new line.
```

We can move back to “the latest” (master) with the command:

In [32]: `!git checkout master`

```
M        Lecture-6A-Fortran-and-C.ipynb
M        Lecture-6B-HPC.ipynb
M        Makefile
A        "Offline/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C.ipynb
A        Offline/Lecture-6A-Fortran-and-C.tex
A        "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
A        Offline/Lecture-6B-HPC.ipynb
A        Offline/Lecture-6B-HPC.tex
A        "Offline/Lecture-6B-HPC_files/Icon\r"
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
M        Preamble.tex
M        Scientific-Computing-with-Python.pdf
Already on 'master'
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
```

In [33]: `!cat README`

```
A file with information about the gitdemo repository.

A new line.
```

In [34]: `!git status`

```
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

9.13 Tagging and branching

9.13.1 Tags

Tags are named revisions. They are useful for marking particular revisions for later references. For example, we can tag our code with the tag “paper-1-final” when when simulations for “paper-1” are finished and the paper submitted. Then we can always retreive the exactly the code used for that paper even if we continue to work on and develop the code for future projects and papers.

```
In [35]: !git log -n 20
```

```
commit c380dc8356be86fde9b020d9d5f1a1fc00dd96ed
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:40 2015 -0700

    remove file tmpfile
```

```
commit 4d74889feb1df45411fe6bc915b559e6651b439a
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:38 2015 -0700

    adding file tmpfile

commit cb5c32499483f58ee4a7f56787f40a26d35e4109
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:37 2015 -0700

    added one more line in README

commit 50c355acc25c78d83d66a6a1e39748b68e62df09
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:32:33 2015 -0700

    Added a README file

commit 68103a5960f95c50833ac7db7622666226292537
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:58 2015 -0700

    added a line in expr1 branch

commit 25d440cbd963e56c912013a52307d2384fbc9042
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    remove file tmpfile

commit 493c55661ea291b8573dd1746dba4c6a91a2e502
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:55 2015 -0700

    adding file tmpfile

commit 0b35eb6602f1d9fd113caf770d507dbfd173fd84
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:53 2015 -0700

    added one more line in README

commit e90a8c109da701138e8455c01d3d9abc7289c079
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:51 2015 -0700

    added notebook file

commit 7771777a56f5bcf7ff5a423c04d0de52076479cf
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:25:50 2015 -0700

    Added a README file
```

```
commit c8eb6ef0d1e06545b276e5637aa8a783fcdedb21
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:20 2015 -0700

    added a line in expr1 branch

commit e61b95df741a4dc734b84b9453290c7e5dc3cc34
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    remove file tmpfile

commit af8e43e5a5406c9f5e01696efc3bd1d7eb2e69c4
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    adding file tmpfile

commit e3896b914fb9f9368c9ade44142448b5cddd0360
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:18 2015 -0700

    added one more line in README

commit cd9d289b9a9182fc3e08986dfe64cca02e0488d0
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 15:10:17 2015 -0700

    Added a README file

commit 4c349ecf589e95d1633072de1fe715a5750fa084
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:54 2015 -0700

    added lecture notebook about RCS

commit d051436004f5fd47af96ac797cc3bcc7a9b08581
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:52:37 2015 -0700

    added a line in expr1 branch

commit ddc57f12eca11f89c157f59595b9edda83dc3815
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:51 2015 -0700

    remove file tmpfile

commit 27e5d0fa081298c8451de903006fbba79ac0130b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:49 2015 -0700

    adding file tmpfile
```

```
commit 9783b909269b2f2c2234d00a20fd00d3b3f0bc1b
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 14:51:44 2015 -0700

    added one more line in README
```

```
In [36]: !git tag -a demotag1 -m "Code used for this and that purpuse"
```

```
fatal: tag 'demotag1' already exists
```

```
In [37]: !git tag -l
```

```
demotag1
```

```
In [38]: !git show demotag1
```

```
tag demotag1
Tagger: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 12:16:48 2015 -0700

Code used for this and that purpuse

commit d925777dd7b7211b963d9ef141c1c2a13596e370
Author: David Mertz <dmertz@continuum.io>
Date:   Mon Aug 17 12:16:40 2015 -0700

    remove file tmpfile

diff --git a/tmpfile b/tmpfile
deleted file mode 100644
index ee4c1e7..0000000
--- a/tmpfile
+++ /dev/null
@@ -1,2 +0,0 @@
-
-A short-lived file.
\ No newline at end of file
```

To retreive the code in the state corresponding to a particular tag, we can use the `git checkout tagname` command:

```
$ git checkout demotag1
```

9.14 Branches

With branches we can create diverging code bases in the same repository. They are for example useful for experimental development that requires a lot of code changes that could break the functionality in the master branch. Once the development of a branch has reached a stable state it can always be merged back into the trunk. Branching-development-merging is a good development strategy when serveral people are involved in working on the same code base. But even in single author repositories it can often be useful to always keep the master branch in a working state, and always branch/fork before implementing a new feature, and later merge it back into the main trunk.

In GIT, we can create a new branch like this:

```
In [39]: !git branch expr1
```

We can list the existing branches like this:

```
In [40]: !git branch
```

```
expr1
* master
```

And we can switch between branches using `checkout`:

```
In [41]: !git checkout expr1
```

```
M        Lecture-6A-Fortran-and-C.ipynb
M        Lecture-6B-HPC.ipynb
M        Makefile
A        "Offline/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C.ipynb
A        Offline/Lecture-6A-Fortran-and-C.tex
A        "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
A        Offline/Lecture-6B-HPC.ipynb
A        Offline/Lecture-6B-HPC.tex
A        "Offline/Lecture-6B-HPC_files/Icon\r"
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
M        Preamble.tex
M        Scientific-Computing-with-Python.pdf
Switched to branch 'expr1'
```

Make a change in the new branch.

```
In [42]: %%file README

         A file with information about the gitdemo repository.

         README files usually contains installation instructions, and information about how to get star

         Experimental addition.
```

```
Overwriting README
```

```
In [43]: !git commit -m "added a line in expr1 branch" README
```

```
[expr1 adfa044] added a line in expr1 branch
 1 file changed, 3 insertions(+), 1 deletion(-)
```

```
In [44]: !git branch
```

```
* expr1
  master
```

```
In [45]: !git checkout master
```

```
M        Lecture-6A-Fortran-and-C.ipynb
M        Lecture-6B-HPC.ipynb
M        Makefile
A        "Offline/Icon\r"
```

```
A        Offline/Lecture-6A-Fortran-and-C.ipynb
A        Offline/Lecture-6A-Fortran-and-C.tex
A        "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
A        Offline/Lecture-6B-HPC.ipynb
A        Offline/Lecture-6B-HPC.tex
A        "Offline/Lecture-6B-HPC_files/Icon\r"
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
M        Preamble.tex
M        Scientific-Computing-with-Python.pdf
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
```

```
In [46]: !git branch
```

```
expr1
* master
```

We can merge an existing branch and all its changesets into another branch (for example the master branch) like this:

First change to the target branch:

```
In [47]: !git checkout master
```

```
M        Lecture-6A-Fortran-and-C.ipynb
M        Lecture-6B-HPC.ipynb
M        Makefile
A        "Offline/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C.ipynb
A        Offline/Lecture-6A-Fortran-and-C.tex
A        "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
A        Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
A        Offline/Lecture-6B-HPC.ipynb
A        Offline/Lecture-6B-HPC.tex
A        "Offline/Lecture-6B-HPC_files/Icon\r"
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
A        Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
M        Preamble.tex
M        Scientific-Computing-with-Python.pdf
Already on 'master'
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)
```

```
In [48]: !git merge expr1
```

```
Updating c380dc8..adfa044
Fast-forward
 README | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

```
In [49]: !git branch
```

```
expr1
* master
```

We can delete the branch `expr1` now that it has been merged into the master:

```
In [50]: !git branch -d expr1
```

```
Deleted branch expr1 (was adfa044).
```

```
In [51]: !git branch
```

```
* master
```

```
In [52]: !cat README
```

```
A file with information about the gitdemo repository.

README files usually contains installation instructions, and information about how to get started using

Experimental addition.
```

9.15 pulling and pushing changesets between repositories

If the respository has been cloned from another repository, for example on github.com, it automatically remembers the address of the parant repository (called origin):

```
In [53]: !git remote
```

```
origin
```

```
In [54]: !git remote show origin
```

```
* remote origin
  Fetch URL: git@github.com:ContinuumIO/scientific-python-lectures.git
  Push  URL: git@github.com:ContinuumIO/scientific-python-lectures.git
  HEAD branch: master
  Remote branches:
    New-branding tracked
    master       tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (fast-forwardable)
```

9.15.1 pull

We can retrieve updates from the origin repository by “pulling” changesets from “origin” to our repository:

```
In [55]: !git pull origin
```

```
Already up-to-date.
```

We can register addresses to many different repositories, and pull in different changesets from different sources, but the default source is the origin from where the repository was first cloned (and the work origin could have been omitted from the line above).

9.15.2 push

After making changes to our local repository, we can push changes to a remote repository using `git push`. Again, the default target repository is `origin`, so we can do:

```
In [56]: !git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
        new file:   "Offline/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C.ipynb
        new file:   Offline/Lecture-6A-Fortran-and-C.tex
        new file:   "Offline/Lecture-6A-Fortran-and-C_files/Icon\r"
        new file:   Offline/Lecture-6A-Fortran-and-C_files/Lecture-6A-Fortran-and-C_5_0.png
        new file:   Offline/Lecture-6B-HPC.ipynb
        new file:   Offline/Lecture-6B-HPC.tex
        new file:   "Offline/Lecture-6B-HPC_files/Icon\r"
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_33_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_37_0.png
        new file:   Offline/Lecture-6B-HPC_files/Lecture-6B-HPC_82_1.png
        modified:   Preamble.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .ipynb_checkpoints/
        Offline/.ipynb_checkpoints/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/
```

```
In [57]: !git add Lecture-7-Revision-Control-Software.ipynb
```

```
In [58]: !git commit -m "added lecture notebook about RCS" Lecture-7-Revision-Control-Software.ipynb
```

```
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)
Changes not staged for commit:
        modified:   Lecture-6A-Fortran-and-C.ipynb
        modified:   Lecture-6B-HPC.ipynb
        modified:   Makefile
```

```
        modified:   Preamble.tex
        modified:   Scientific-Computing-with-Python.pdf

Untracked files:
        .ipynb_checkpoints/
        Offline/
        __pycache__/
        gitdemo/
        mymodule.py
        qutip/

no changes added to commit
```

In [59]: !git push

```
Counting objects: 11, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 1.04 KiB | 0 bytes/s, done.
Total 11 (delta 5), reused 0 (delta 0)
To git@github.com:ContinuumIO/scientific-python-lectures.git
   68103a5..adfa044  master -> master
```

9.16 Hosted repositories

Github.com is a git repository hosting site that is very popular with both open source projects (for which it is free) and private repositories (for which a subscription might be needed).

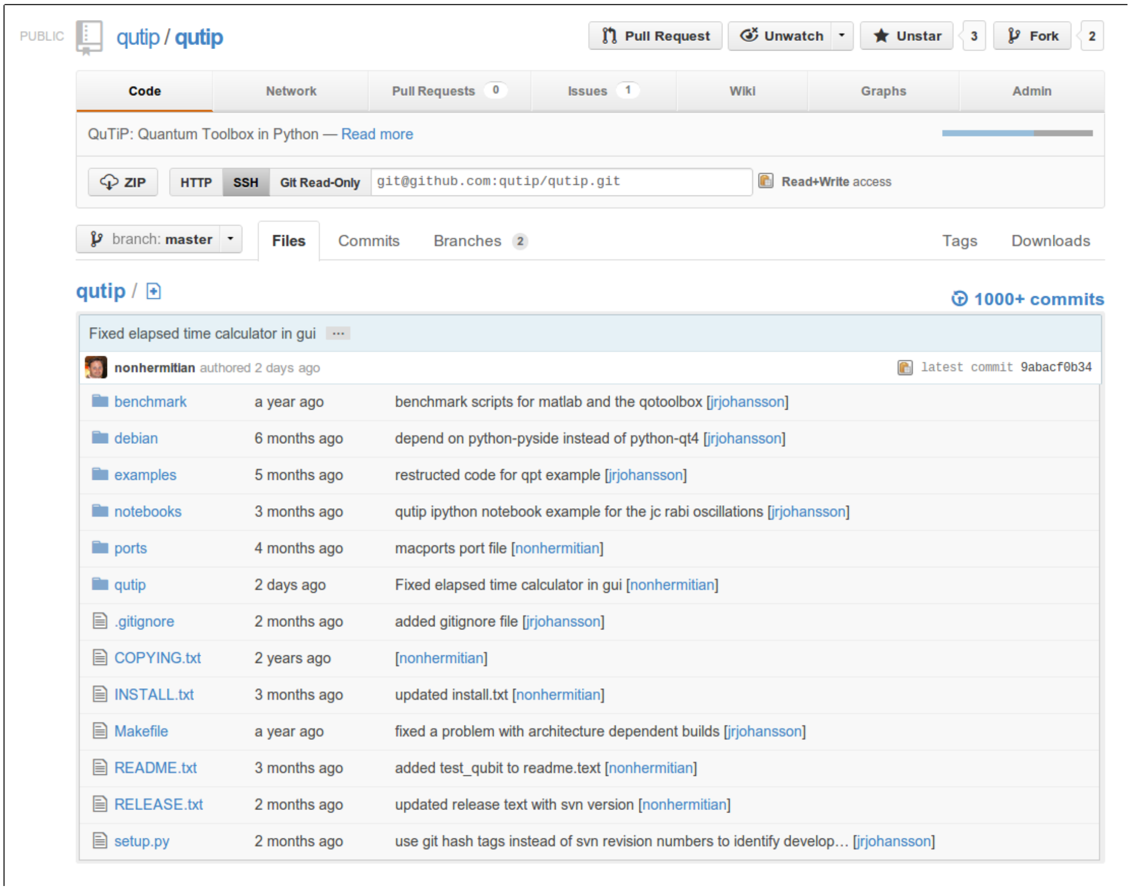
With a hosted repository it easy to collaborate with colleagues on the same code base, and you get a graphical user interface where you can browse the code and look at commit logs, track issues etc.

Some good hosted repositories are

- Github : http://www.github.com
- Bitbucket: http://www.bitbucket.org

In [60]: Image(filename='images/github-project-page.png')

Out[60]:



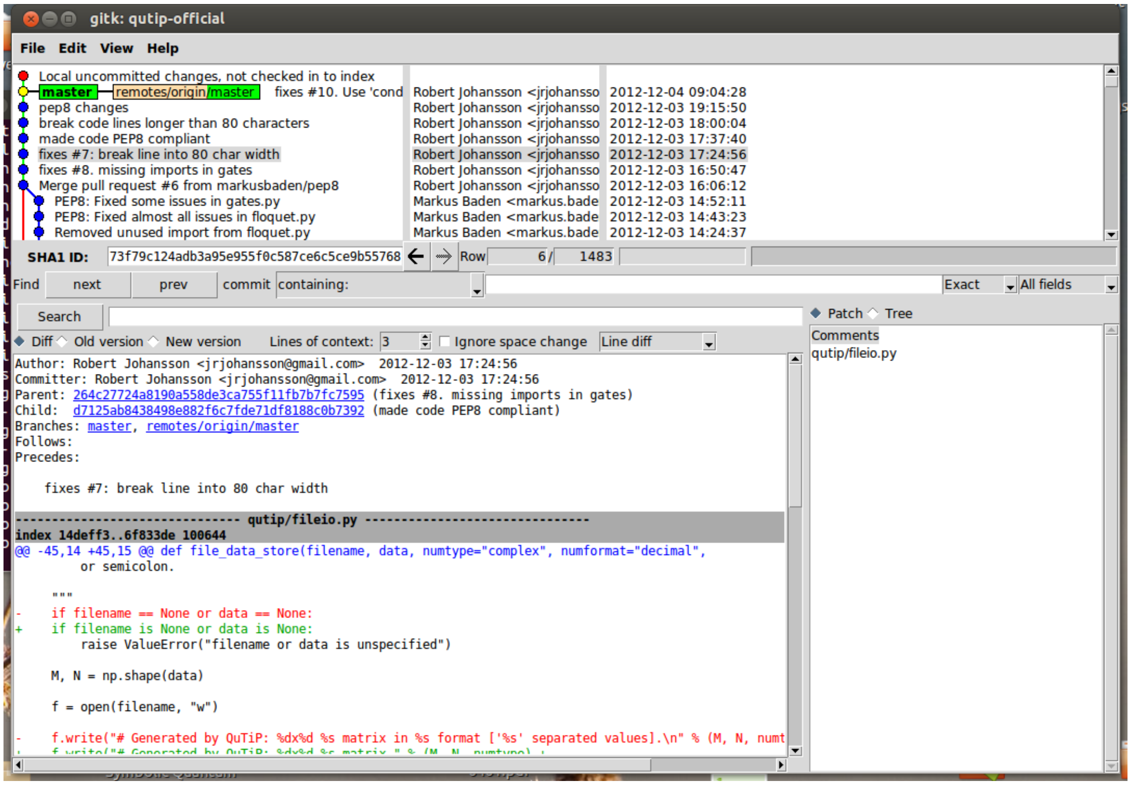
9.17 Graphical user interfaces

There are also a number of graphical users interfaces for GIT. The available options vary a little bit from platform to platform:

http://git-scm.com/downloads/guis

```
In [61]: Image(filename='images/gitk.png')
```

Out[61]:



9.18 Further reading

- http://git-scm.com/book
- http://www.vogella.com/articles/Git/article.html
- http://cheat.errtheblog.com/s/git