pyengr Documentation

Release 0.0.1

Yung-Yu Chen

June 27, 2013

CONTENTS

This document is a collection of class notes for my official or unofficial Python training.

The skill to program digital computers is important for modern engineers. We routinely use computers to process data, perform numerical analysis and simulations, and control devices. We need a programming language. In this document, we are going to show that Python is such a good choice, and how to use it to solve technical problems.

CHAPTER

ONE

WHAT IS PYTHON?

The programming language Python was first made public in 1991. Python is a multi-paradigm and batteries-included programming language. It supports imperative, structural, object-oriented, and functional programming. It contains a wide spectrum of standard libraries, and has more than 10,000 3rd-party packages available online. The flexibility in programming paradigms allows the users to attack a problem with a suitable approach. The versatility of libraries further enriches our armament. Moreover, Python allows straight-forward extension to its core implementation via the C API. The interpreter itself can be easily incorporated into another host system. Regarding problem-solving, Python is much more than a programming language. It's more like an extensible runtime environment with rich programmability.

Python is an interpreted language with a strong and dynamic typing system. In most Unix-based computers, Python is pre-installed and one can enter its interactive mode in a terminal:

```
$ python
Python 2.7.3rc2 (default, Apr 22 2012, 22:30:17)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

to perform calculation:

```
>>> import sys, math
>>> sys.stdout.write('%g\n' % math.pi)
3.14159
>>> sys.stdout.write('%g\n' % math.cos(45./180.*math.pi))
0.707107
>>>
```

CHAPTER

TWO

WHY PYTHON?

Indeed Python is both powerful and easy-to-use. But what makes Python great for technical applications is its compatibility to engineering and scientific discipline. See The Zen of Python (Python Enhancement Proposal (PEP) 20):

```
$ python -c 'import this'
The Zen of Python, by Tim Peters

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

These proverbs are the general guidelines for Python programmers. It promotes several points favorable for engineers and scientists:

- Simplicity. Engineers and scientists want Occam's razor. Simplification is our job. We know a trustworthy solution is usually simple and beautiful.
- Disambiguation. Although expressions can differ, facts are facts. Uncertainty is acceptable, but anything true should never be taken as false, and vice versa.
- Practicality. Given infinite amount of time, anything can be done. For engineers, constraints are needed to deliver meaningful products or solutions.
- Collaboration. Not all programming languages emphasize on readability, but Python does.

The more I write Python, the more I like it. Although there are many good programming languages (or environments), and some can be more convenient than Python in specific areas, only Python and its community have a value system so close to the training I received as a computational scientist.

2.1 Idiomatic Programming

The Zen of Python is very insightful to programming Python. Breaking the Zen means not writing "Pythonic" code. Python programmers like to establish conventions for solving similar problems. Programming Python is usually idiomatic. For example, when converting a sequence of data, it is encouraged to use a list comprehension:

```
line = '1 2 3'
# it is concise and clear if you know what's a list comprehension.
values = [float(tok) for tok in line.split()]
```

rather than a loop:

```
line = '1 2 3'
# it works, but is not idiomatic to Python, i.e., not "Pythonic".
values = []
for tok in line.split():
    values.append(float(tok))
```

But it doesn't mean using list comprehensions is always preferred. Consider a list of lines:

```
lines = ['1 2 3\n', '4 5 6\n']
# nested list comprehensions are not easy to understand.
values = [float(tok) for line in lines for tok in line.split()]
# so a loop now looks more concise.
values = []
for line in lines:
    values.extend(float(tok) for tok in line.split())
```

Python has a good balance between freedom and discipline in coding. The idiomatic style is a powerful weapon to create maintainable code.

CHAPTER

THREE

CONTENTS

This project is intended to provide introductory information about Python for technical computing. It includes a set of documents and the corresponding code snippets. The code is hosted at https://bitbucket.org/yungyuc/pyengr and you can find the up-to-date documentation built at http://pyengr.readthedocs.org/en/latest/. The project is licensed under GNU GPLv2.

3.1 Basic Python Programming

This is a course for basic Python programming. The audience is those who want to understand the way in which an experienced Python programmer thinks, or those who want to be a Python expert.

In this course, you will be introduced to the most essential elements in the Python programming language. You will be given many examples to familiarize yourself to the practice of "one obvious way to do it", and start to understand the rationale behind the formality. This course will lead your way to "import this".

3.1.1 Start Running Python: Execution and Importation

This class is one hour long. You will first be introduced with the fundamental concepts about writing Python programs, and then the main running mode of the runtime.

Pythonicity and PEP8

The Zen of Python (Python Enhancement Proposal (PEP) 20):

```
$ python -c 'import this'
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

When writing a Python program, it is important to write it "pythonically". "Pythonic" is quite a vaguely defined adjective, and it roughly means "writing a program in the way that an experienced Python programmer feels comfortable". Therefore, pythonicity is not something can be taught. Instead, writing pythonic programs needs deliberate reading and mimicking good code, which in fact is the sure way to learn programming. You will gradually understand the Zen of Python mentioned above, and gain the productivity enabled by pythonic programming.

Python programming is centered around "the one way to do it". Python encourages using a best practice to solve a problem in code. Although one can always find many approaches to program, using more than one way to do it is not considered a friendly coding style for code readers. The one-way spirit indeed takes away a certain amount of the diversity of our code, but give us in return readability, maintainability, and the eventual productivity.

Perhaps using Python-specific constructs could be the easiest way to demonstrate pythonicity. If you are familiar with C and come to learn Python, you tend to write code like:

```
lst = [1, 3, 5, 2]
literals = []
i = 0
while i < len(lst):
    literals.append(str(lst[i]))
    i += 1
```

Because you know `for` has a different semantic in Python than in C, you chose to use `while`. Perfectly valid but not pythonic. You can then improve it by using `for` with the sequence `lst`:

```
lst = [1, 3, 5, 2]
literals = []
for it in lst:
    literals.append(str(it))
```

A bit better but still unpythonic. This can change by using a list comprehension:

```
lst = [1, 3, 5, 2]
literals = [str(it) for it in literals]
```

Now the five lines of code at the beginning becomes a one-liner. Although you might not know what's a list comprehension, you could still guess from the expression that the resulting `literals` is a list and the code involves something about looping (because of the `for`). It is now pythonic.

But note, not all code using Python-specific constructs is pythonic. Although the following version is even shorter than the previous one, it's not really more readable than the longer one:

```
literals = [str(it) for it in [1, 3, 5, 2]]
```

Things can be trickier if there're nested list comprehensions:

```
literals = [str(it) for it in [val+10 for val in [1, 3, 5, 2]]]
```

It's OK, but split it into two lines isn't harmful either. Remember, pythonicity is vaguely defined. Finding a quick way to "a good Python coding style" is usually an effort of vain. Relying on the Zen of Python and constant practicing is more rewarding.

Running Python

On Debian/Ubuntu.

Interactive Interpreter

Invoke and use the interactive environment for simple tasks.

Python Script

Write Python code in a file.

Use shebang and set the executable bit.

3.1.2 Package Installation

Python Modules and PYTHONPATH

Python Packages and Import Rules

Absolute and relative imports.

`virtualenv`, `pip`, and `distribute`

The easy way to install new packages. Use docutils and django as examples.

Manual Installation

Use NumPy as an example.

3.1.3 Input, Output, and String Processing

Read and Write Files

Stream I/O and Files

String Formatting

String Tokenization, Concatenation, and Other Processing

Stripping and testing.

String Templating

Regular Expression Interface

3.1.4 Execution Control

Functions

Yield?

Positional and Keyword Parameters

Conditional Statements

Boolean comparison and testing for singleton.

Looping

3.1.5 Containers

Sequence: `list` and `tuple`

`[]` or `list()` constructs a list for you:

```
>>> la = []
>>> lb = list()
>>> print(la, lb)
([], [])
```

Some built-ins return a list:

```
>>> a = range(10)
>>> print(type(a), a)
(<type 'list'>, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

A `tuple` can also hold anything, but cannot be changed once constructed. It can be created with `()` or `tuple()`:

```
>>> ta = (1)
>>> print(type(ta), ta)
(<type 'int'>, 1)
>>> ta = (1,)
>>> print(type(ta), ta)
(<type 'tuple'>, (1,))
>>> ta[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
  \end{lstlisting}
```

Slicing

List Comprehension

List comprehension is a very useful technique to construct a list from another iterable:

```
>>> values = [10.0, 20.0, 30.0, 15.0]
>>> print([it/10 for it in values])
[1.0, 2.0, 3.0, 1.5]
```

List comprehension can even be nested:

```
>>> values = [[10.0, 1.0], [20.0, 2.0], [30.0, 3.0], [15.0, 1.5]]
>>> print([jt for it in values for jt in it])
[10.0, 1.0, 20.0, 2.0, 30.0, 3.0, 15.0, 1.5]
```

Iterator

Use `reversed()` and `sorted()` as examples.

Simple sort:

```
>>> a = [87, 82, 38, 56, 84]
>>> b = sorted(a) # b is a new list.
>>> print(b)
[38, 56, 82, 84, 87]
>>> a.sort() # this method does in-place sort.
>>> print(a)
[38, 56, 82, 84, 87]
```

Not-so-simple sort:

```
>>> a = [('a', 0), ('b', 2), ('c', 1)]
>>> print(sorted(a)) # sorted with the first value.
[('a', 0), ('b', 2), ('c', 1)]
>>> print(sorted(a, key=lambda k: k[1])) # use the second.
[('a', 0), ('c', 1), ('b', 2)]
```

Built-in calculation functions for iterables:

```
>>> values = [10.0, 20.0, 30.0, 15.0]
>>> min(values), max(it for it in values)
(10.0, 30.0)
>>> sum(values)
75.0
>>> sum(values)/len(values)
18.75
```

Set

A `set` holds any hashable element, and its elements are distinct:

```
>>> sa = {1, 2, 3}
>>> print(type(sa), sa)
(<type 'set'>, set([1, 2, 3]))
>>> print({1, 2, 2, 3}) # no duplication is possible.
set([1, 2, 3])
>>> len({1, 2, 2, 3})
3
```

It's unordered:

```
>>> [it for it in {3, 2, 1}]
[1, 2, 3]
>>> [it for it in {3, 'q', 1}]
['q', 1, 3]
>>> 'q' < 1
False
```

Add elements after construction of the set:

```
>>> sa = {1, 2, 3}
>>> sa.add(1)
>>> sa
set([1, 2, 3])
>>> sa.add(10)
>>> sa
set([1, 2, 3, 10])
```

Remove elements:

```
>>> sa = {1, 2, 3, 10}
>>> sa.remove(5) # err with non-existing element
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
>>> sa.discard(2) # really discard an element
>>> sa
set([1, 10, 3])
```

Subset or superset:

```
>>> {1, 2, 3} < {2, 3, 4, 5} # not a subset
False
>>> {2, 3} < {2, 3, 4, 5} # subset
True
>>> {2, 3, 4, 5} > {2, 3} # superset
True
```

Union and intersection:

```
>>> {1, 2, 3} | {2, 3, 4, 5} # union
set([1, 2, 3, 4, 5])
>>> {1, 2, 3} & {2, 3, 4, 5} # intersection
set([2, 3])
>>> {1, 2, 3} - {2, 3, 4, 5} # difference
set([1])
```

A `set` can be used with a sequence to quickly calculate unique elements:

```
>>> data = [1, 2.0, 0, 'b', 1, 2.0, 3.2]
>>> sorted(set(data))
[0, 1, 2.0, 3.2, 'b']
```

But there's a problem: It doesn't support unhashable objects:

```
>>> data = [dict(a=200), 1, 2.0, 0, 'b', 1, 2.0, 3.2]
>>> set(data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Read the Python Cookbook for a solution :-)

Dictionary

A `dict` stores any number of key-value pairs. It is the most used Python container since it's everywhere for Python namespace.

```
>>> {'a': 10, 'b': 20} == dict(a=10, b=20)
True
>>> da = {1: 10, 2: 20} # any hashable can be a key
>>> da[1] + da[2]
30
>>> class SomeClass(object):
...     pass
...
>>> print(type(SomeClass().__dict__))
<type 'dict'>
```

To test whether something is in a dictionary or not:

```
>>> da = {1: 10, 2: 20}
>>> 3 in da
False
```

Access a key-value pair:

```
>>> da[3] # it fails for 3 is not in the dictionary
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> print(da[3] if 3 in da else 30) # works but wordy
30
>>> da.get(3, 30) # it's the way to go
30
>>> da # indeed we don't have 3 as a key
{1: 10, 2: 20}
>>> da.setdefault(3, 30) # how about this?
30
>>> da # we added 3 into the dictionary!
{1: 10, 2: 20, 3: 30}
```

Iterating a `dict` automatically gives you its keys:

```
>>> da = {1: 10, 2: 20}
>>> ','.join('%s'%key for key in da)
'1,2'
>>> ','.join('%d'%da[key] for key in da)
'10,20'
```

`items()` and `iteritems()` give you both key and value at once:

```
>>> da.items() # returns a list
[(1, 10), (2, 20)]
>>> type(da.iteritems()) # returns an iterator
<type 'dictionary-itemiterator'>
>>> ','.join('%s:%s'%(key, value) for key, value in da.iteritems())
'1:10,2:20'
```

A dictionary view changes with the dictionary:

```
>>> da = {1: 10, 2: 20}
>>> daiit = da.iteritems() # an iterator
```

```
>>> type(daiit)
<type 'dictionary-itemiterator'>
>>> davit = da.viewitems() # a view object
>>> davit
dict_items([(1, 10), (2, 20)])
>>> da[3] = 30 # change the dictionary
>>> ','.join('%s:%s'%(key, value) for key, value in daiit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <genexpr>
RuntimeError: dictionary changed size during iteration
>>> ','.join('%s:%s'%(key, value) for key, value in davit)
'1:10,2:20,3:30'
```

Dictionary for Switch-Case

Make Your Own Data Structures: Collection ABCs

3.1.6 Object-Oriented Programming

Organize Data with Functions

Encapsulation

`@property`

Exploit Existing Types

3.2 Multi-Language Programming

3.2.1 Build System

If you want to use Python with other programming languages, a build system is usually needed. A build system is used to automate the processes of compiling, linking, packaging, and deploying software. This chapter will focus on a tool called SCons, which is implement with pure Python. Building scripts of SCons can be highly modularized and reused, and cross-platform as well.

Using a build system involves writing building scripts. Building scripts of SCons can have three parts:

- Front-end script (`SConstruct`),
- Rule script (`Sconscript`), and
- Tools (`site_scons/site_tools/*`).

Below is the `SConstruct` and the `SConscript` files of an example project. The `SConstruct` file is:

```
import os, sys

AddOption('--cc', dest='cc', type=str, action='store', default='gcc',
    help='C compiler (SCons tool): gcc, intelc; default is %default')
AddOption('--optimize', dest='optimize', type=int, action='store', default=2,
    help='Optimization level; default is %default.')
AddOption('--no-separation', dest='separate', action='store_false',
    default=True,
```

```
    help='Do not separate build and source directories.')

env = Environment(ENV=os.environ, LIBDIR='lib', BINDIR='bin')
env.Tool('mingw' if sys.platform.startswith('win') else 'default')
env.Tool(GetOption('cc'))
env.Tool('cython')
env.Tool('pyext')

env.Append(CFLAGS='-O%s'%GetOption('optimize'))
env.Append(CPPPATH='include/')
env.Append(LIBPATH='lib/')
env['BUILDPREFIX'] = 'build/' if GetOption('separate') else ''

everything = list()
Export('env', 'everything')
SConscript(['SConscript'])
Default(everything)

# vim: set ft=python ff=unix fenc=utf8 ai et sw=4 ts=4 tw=79:
```

The `SConscript` file is:

```
import os

Import('env', 'everything')

if env['BUILDPREFIX']:
    for path in Glob('src/*'):
        env.VariantDir(env['BUILDPREFIX']+str(path), str(path))
    env.VariantDir(env['BUILDPREFIX']+'cython', 'cython')

everything.append(env.StaticLibrary(
    '%s/inter_build'%env['LIBDIR'],
    [fname for fname in Glob(env['BUILDPREFIX']+'src/*/*.c')
        if 'src/bin/' not in str(fname)]
))

for cfn in Glob(env['BUILDPREFIX']+'cython/*.pyx'):
    cfn = str(cfn)
    modname = os.path.splitext(os.path.basename(cfn))[0]
    cython = env.PythonObject(env.Cython(cfn))
    pymod = env.SharedLibrary('%s/core'%env['LIBDIR'], cython, LIBPREFIX='')
    everything.append(pymod)

cenv = env.Clone()
cenv.Prepend(LIBS='inter_build')
everything.append(cenv.Program(
    '%s/inter_build'%env['BINDIR'],
    Glob(env['BUILDPREFIX']+'src/bin/*.c')
))

# vim: set ft=python ff=unix fenc=utf8 ai et sw=4 ts=4 tw=79:
```

SCons tools provide a means to reuse the building code. For example, we can use the SCons tools provided by the Cython team to build your cython code, by copying the files `cython.py` and `pyext.py` into the directory `site_scons/site_tools` inside your project.

3.2.2 Foreign Function Interface

3.2.3 Wrapping Other Languages with Helpers

3.2.4 Developing Python Extension Modules

3.3 Managing a Python Software Project

3.3.1 Basic Version Control

For code development, the history is of the same importance as the end results. As such we need a version control system (VCS) to help tracking the history. There are many VCS available, and here we will introduce one of the most powerful systems: Mercurial (hg, which is also used for the development of Python).

In this session, you will learn the basic of managing source code with the VCS tool Mercurial. We will cover the following topics:

1. *Initialization*
2. *Basic Concepts*
3. *Commit*
4. *Ignorance*
5. *Publish to Bitbucket*
6. *Mercurial Queue*

When coming to this course, please prepare yourself a laptop with Internet connection, preferably running Ubuntu/Debian. If you are using Windows or Mac, you are on your own for installing required software.

Initialization

Mercurial is categorized as a decentralised VCS (DVCS). "Decentralised" means everyone in a collaborative team can maintain standalone development history, and synchronize it when necessary. The separation of tracking and synchronization makes the applications of the system broader than those of conventional centralised VCS.

Install

On a Debian/Ubuntu, the following command installs Mercurial for you:

```
$ sudo apt-get install mercurial
```

The command line `hg` should be available for you to use:

```
$ hg version
Mercurial Distributed SCM (version 2.2.2)
(see http://mercurial.selenic.com for more information)

Copyright (C) 2005-2012 Matt Mackall and others
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Note: Because the command line is named `hg`, often we use it to refer to Mercurial.

Configure

By default, Mercurial reads `~/.hgrc` for configuration. Before any action, we need to at least add the following setting into the configuration file:

```
[ui]
username = Your Name <your@email.address>
```

Mercurial has to be told who is working on repositories, so that it can record correct information. Note the uesrname here is arbitrary. It doesn't need to be the same as any of your local or online credential, but it's good to set to a consistent value in all your environments.

In this course we also add the following setting:

```
[diff]
git = True
```

to use the diff format that's compatible to another popular VCS Git.

Initialize a New Repository

To this point we are ready to initialize our first Mercurial repository:

```
$ hg init proj; ls -al
total 12
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 ./
drwxrwxr-x 7 yungyuc yungyuc 4096 Jun  5 06:06 ../
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 proj/
```

Repository File Layout

A *repository* is the database that Mercurial stores history to. In the project we just created, the repository is in the subdirectory `.hg/` of `proj/`:

```
$ ls -al proj/
total 12
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 ./
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:34 ../
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 .hg/
$ ls -al proj/.hg/
total 20
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 ./
drwxrwxr-x 3 yungyuc yungyuc 4096 Jun  5 06:06 ../
-rw-rw-r-- 1 yungyuc yungyuc   57 Jun  5 06:06 00changelog.i
-rw-rw-r-- 1 yungyuc yungyuc   33 Jun  5 06:06 requires
drwxrwxr-x 2 yungyuc yungyuc 4096 Jun  5 06:06 store/
```

As you can see, a Mercurial repository is nothing more than a directory named `.hg/` containing some data. Tracking (or managing) a software project with Mercurial pretty much is changing the `.hg/` directory, and we don't do it by hands, but by the convenient tools of Mercurial, specifically, the `hg` command line.

Basic Concepts

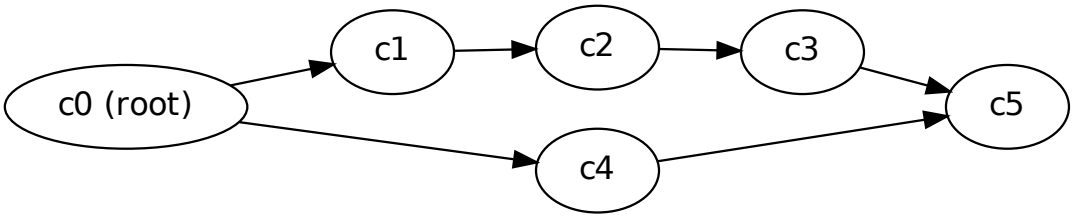
There are some fundamental concepts we need to remember before using Mercurial:

- Working copy: it's basically the working directory of everything are you tracking in the project.
- Changeset: the difference between two tracked revision of the working copy (directory).
- Repository: where we store the changesets.

Graph of Changes

The following figure shows the graphical representation (directed acyclic graph, DAG) of a Mercurial repository:

Figure 3.1: Changesets in a repository



In the figure each node represents a changeset, and **c0** is the root. Every repository can have one and only one root. Because the root is the first "change" in the repository, the repository we just initialized has no root:

```
$ hg log
$ hg id
000000000000 tip
```

Using the Help System of Mercurial

As you can see, there's nothing after `hg log`, and the "tip" id (the latest changeset in a repository) is null. You can find more information about the command by using `hg help`:

```
$ hg help log
hg log [OPTION]... [FILE]

aliases: history

show revision history of entire repository or files

    Print the revision history of the specified files or the entire project.

    If no revision range is specified, the default is "tip:0" unless --follow
    is set, in which case the working directory parent is used as the starting
    revision.

    File history is shown without following rename or copy history of files.
    Use -f/--follow with a filename to follow history across renames and
    copies. --follow without a filename will only show ancestors or
    descendants of the starting revision.
```

```
18
19      By default this command prints revision number and changeset id, tags,
20      non-trivial parents, user, date and time, and a summary for each commit.
21      When the -v/--verbose switch is used, the list of changed files and full
22      commit message are shown.
23
24      Note:
25         log -p/--patch may generate unexpected diff output for merge
26         changesets, as it will only compare the merge changeset against its
27         first parent. Also, only files different from BOTH parents will appear
28         in files:.
29
30      Note:
31         for performance reasons, log FILE may omit duplicate changes made on
32         branches and will not show deletions. To see all changes including
33         duplicates and deletions, use the --removed switch.
34
35      See "hg help dates" for a list of formats valid for -d/--date.
36
37      See "hg help revisions" and "hg help revsets" for more about specifying
38      revisions.
39
40      See "hg help templates" for more about pre-packaged styles and specifying
41      custom templates.
42
43      Returns 0 on success.
44
45  options:
46
47   -f --follow              follow changeset history, or file history across
48                            copies and renames
49   -d --date DATE           show revisions matching date spec
50   -C --copies              show copied files
51   -k --keyword TEXT [+]    do case-insensitive search for a given text
52   -r --rev REV [+]         show the specified revision or range
53      --removed             include revisions where files were removed
54   -u --user USER [+]       revisions committed by user
55   -b --branch BRANCH [+]   show changesets within the given named branch
56   -P --prune REV [+]       do not display revision or any of its ancestors
57   -p --patch               show patch
58   -g --git                 use git extended diff format
59   -l --limit NUM           limit number of changes displayed
60   -M --no-merges           do not show merges
61      --stat                output diffstat-style summary of changes
62      --style STYLE         display using template map file
63      --template TEMPLATE   display with template
64   -I --include PATTERN [+] include names matching the given patterns
65   -X --exclude PATTERN [+] exclude names matching the given patterns
66      --mq                  operate on patch repository
67   -G --graph               show the revision DAG
68
69  [+] marked option can be specified multiple times
70
71  use "hg -v help log" to show more info
```

Commit

Let's make the first commit:

```
$ touch file_a
$ hg add file_a
$ hg ci -m "Initial commit."
$ hg log
changeset:   0:2fee2d78ec72
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sat Jun 15 20:52:23 2013 +0800
summary:     Initial commit.
```

Mercurial command-line is very smart and knows how to shorthand commands. `hg ci` is equivalent to `hg commit`. "Commit" means to "take the difference between the current revision and the working copy and store the difference in the repository as a new changeset". Therefore after the commit you have a new changeset. If you want to see what files are in each of the changesets, use `hg log --stat`:

```
$ hg log --stat
changeset:   0:2fee2d78ec72
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sat Jun 15 20:52:23 2013 +0800
summary:     Initial commit.

 file_a |  0
 1 files changed, 0 insertions(+), 0 deletions(-)
```

Adding New Files

When we make new files in the working copy, by default Mercurial doesn't track them. For example, let's make several empty files:

```
$ touch file_b file_c file_d
$ hg ci -m "This commit won't work."
nothing changed
$ ls
file_a  file_b  file_c  file_d
```

See? `hg ci` doesn't allow us to commit a changeset because it thinks "nothing changed", but indeed there are three new files `file_b`, `file_c`, and `file_d`. It becomes clear that Mercurial doesn't "know" these new files when we use the `hg st` (status) command:

```
$ hg st
? file_b
? file_c
? file_d
```

The question marks (`?`) indicate those files are not tracked by Mercurial. We need to `hg add` them:

```
$ hg add file_b file_c file_d
yungyuc@hayate:~/work/writing/pyengr/tmp/proj
$ hg st
A file_b
A file_c
A file_d
```

```
yungyuc@hayate:~/work/writing/pyengr/tmp/proj
$ hg ci -m "Add three more files."
$ hg log --stat
changeset:   1:7fb98d36f680
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:04:51 2013 +0800
summary:     Add three more files.

 file_b |  0
 file_c |  0
 file_d |  0
 3 files changed, 0 insertions(+), 0 deletions(-)

changeset:   0:2fee2d78ec72
user:        yungyuc <yyc@solvcon.net>
date:        Sat Jun 15 20:52:23 2013 +0800
summary:     Initial commit.

 file_a |  0
 1 files changed, 0 insertions(+), 0 deletions(-)
```

Modification of Files

Mercurial will detect the changed contents of tracked files. Let's try it with some change:

```
$ echo "Some texts." >> file_a
```

`hg st` knows `file_a` is changed (see the `M` in front of `file_a`):

```
$ hg st
M file_a
```

And you can check the difference by `hg diff`:

```
$ hg diff
diff --git a/file_a b/file_a
--- a/file_a
+++ b/file_a
@@ -0,0 +1,1 @@
+Some texts.
```

Finally we can commit:

```
$ hg ci -m "Change file_a."
$ hg log
changeset:   2:35f496a1ff0b
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:27:45 2013 +0800
summary:     Change file_a.

changeset:   1:7fb98d36f680
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:04:51 2013 +0800
summary:     Add three more files.

changeset:   0:2fee2d78ec72
```

```
user:        yungyuc <yyc@solvcon.net>
date:        Sat Jun 15 20:52:23 2013 +0800
summary:     Initial commit.
```

The Simplest Work Flow

After learning to commit files, you basically can use Mercurial to track anything. The general procedure is:

1. Initialize a repository by `hg init name` to start a project.
2. Create some blank files, `hg add file1 file2 ...`, and `hg ci -m "Commit log message."`
3. Edit the files and `hg ci -m "Some meaningful commit logs."` the changeset.
4. Continue with steps 1–3.

Mercurial discourages editing history, so even with some history-changing functionalities (like MQ), you cannot easily change what you've committed. Your repository is a pretty safe strongbox for your work.

Ignorance

When adding a bunch of files to a repository, sometimes we are lazy and do something like this:

```
$ touch file_1 file_2 file_3 file_4 generated
$ hg add
adding file_1
adding file_2
adding file_3
adding file_4
adding generated
$ hg st
A file_1
A file_2
A file_3
A file_4
A generated
```

Assume `generated` is a file generated form a script. We don't want to track it since it changes every time when we run the script. One way to do it is to be explicit when adding:

```
$ hg revert .
forgetting file_1
forgetting file_2
forgetting file_3
forgetting file_4
forgetting generated
$ hg add file_[1-4]
yungyuc@hayate:~/work/writing/pyengr/tmp/proj
$ hg st
A file_1
A file_2
A file_3
A file_4
? generated
```

It resolves the issue, but with two drawbacks:

1. Now we can't be lazy any more.

2. `hg st` says it doesn't know about `generated`, about which we don't care.

Mercurial provides an ignore file to better solve this problem. Let's add `.hgignore` into the repository:

```
$ echo "syntax: glob
> generated" > .hgignore
$ hg st
A file_1
A file_2
A file_3
A file_4
? .hgignore
$ hg add .hgignore
$ hg st
A .hgignore
A file_1
A file_2
A file_3
A file_4
$ hg ci -m "Add ignorance." .hgignore
$ hg ci -m "Add 4 empty files."
$ hg log --stat -l 2
changeset:   4:06dacab043bf
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:18 2013 +0800
summary:     Add 4 empty files.

 file_1 |  0
 file_2 |  0
 file_3 |  0
 file_4 |  0
 4 files changed, 0 insertions(+), 0 deletions(-)

changeset:   3:871d0c94b01e
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:02 2013 +0800
summary:     Add ignorance.

 .hgignore |  2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

A real example of `.hgignore` can be found at https://bitbucket.org/yungyuc/pyengr/src/tip/.hgignore.

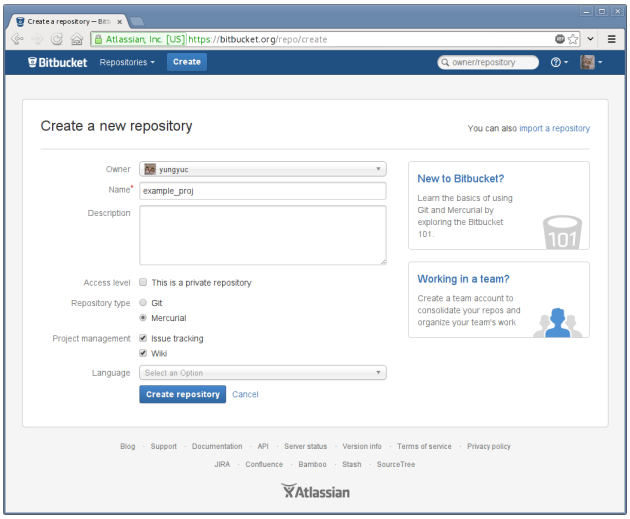
Publish to Bitbucket

Bitbucket is a online hosting service for Mercurial (and Git, which I ignore here). We can push our local repository to Bitbucket (or BB in short) to make it available to the world (a public BB repository) or a selected group of people (a private BB repository).

To proceed, you need an account at Bitbucket. It's free. After having the account, you can create a repository:

Click the "Create repository" button and we are ready to go. If you have added your SSH key to BB, you can push your local changes to BB with it:

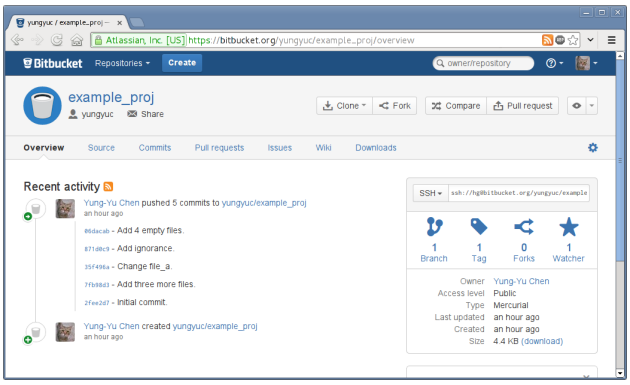
```
$ hg push ssh://hg@bitbucket.org/yungyuc/example_proj
pushing to ssh://hg@bitbucket.org/yungyuc/example_proj
searching for changes
remote: adding changesets
```




```
remote: adding manifests
remote: adding file changes
remote: added 5 changesets with 10 changes to 9 files
```

Note: Of course you need to replace `ssh://hg@bitbucket.org/yungyuc/example_proj` with the repository you created. And if you haven't set a SSH key at BB, you will need to use the HTTP protocol to communicate with your BB repository: `https://username@bitbucket.org/username/example_proj` (replace `username` with your BB user name).

After pushing the changes, you should see the front page of your BB repository like:



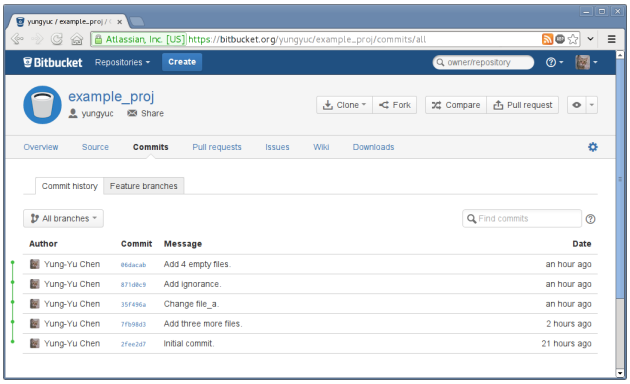
Clicking "Commits" will bring us to a page to view a graphical history of the commits:

Since we've made the BB repository public, everyone in the world can collaborate on it.

Mercurial Queue

Mercurial Queue is often called "mq". mq is an important feature of Mercurial, but it is implemented as an "extension". To enable it, edit your `~/.hgrc` and add the following lines:

```
[extensions]
hgext.mq=
```



Note that if there is already a section named `[extensions]`, don't repeat it and just add the second line `hgext.mq=` to your setting file `~/.hgrc`.

Mercurial queue is a tool for us to manage "patches". The extension was inspired by quilt and seamlessly integrated into Mercurial. Because Mercurial discourage modification of history, mq is the answer for history-editing actions. Mercurial queue allows us to systematically change what has been committed into a repository, and we fully understand we are changing the history, because mq uses a different set of commands.

After enable the extension, you will have a bunch of new commands: `qnew`, `qref`, `qpush`, `qpop`, `qfin`, and several others.

Create a Patch

Use `hg qnew` to create a new patch:

```
$ hg qnew test -m "Patch for testing."
$ hg log -l 3
changeset:   5:860f045d5a1a
tag:         qbase
tag:         qtip
tag:         test
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:00:30 2013 +0800
summary:     Patch for testing.

changeset:   4:06dacab043bf
tag:         qparent
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:18 2013 +0800
summary:     Add 4 empty files.

changeset:   3:871d0c94b01e
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:02 2013 +0800
summary:     Add ignorance.
```

The first argument after `hg qnew` command is the patch name. In this example we created a patch named "test". As we saw in the output of `hg log`, a mq patch is nothing more than a regular changeset! But since it's a "patch", there must be something distinguish it from a regular changeset, isn't it?

```
$ cat .hg/patches/test
# HG changeset patch
```

```
# Parent 06dacab043bf1beb5d01f20c5d127341d980c4b8
Patch for testing.
```

Here's the difference: mq maintains a directory `.hg/patches` for all patches belonging to a "Mercurial queue". Each patch is a file in the directory with the file name set to the patch name.

When creating a new patch without any change in the working copy, you will get an empty patch like the "test" patch we made. If we `qnew` a patch with existing modification in the working copy, the modification will be incorporated into the patch:

```
$ echo "some text" >> file_1
yungyuc@hayate:~/work/writing/pyengr/tmp/proj
$ hg qnew modify -m "Create a patch with some modification in working copy."
yungyuc@hayate:~/work/writing/pyengr/tmp/proj
$ hg qdiff
diff --git a/file_1 b/file_1
--- a/file_1
+++ b/file_1
@@ -0,0 +1,1 @@
+some text
$ hg log -l 4
changeset:   6:efbbac003006
tag:         modify
tag:         qtip
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:17:01 2013 +0800
summary:     Create a patch with some modification in working copy.

changeset:   5:860f045d5a1a
tag:         qbase
tag:         test
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:00:30 2013 +0800
summary:     Patch for testing.

changeset:   4:06dacab043bf
tag:         qparent
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:18 2013 +0800
summary:     Add 4 empty files.

changeset:   3:871d0c94b01e
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:02 2013 +0800
summary:     Add ignorance.
```

Incremental Change

mq allows us to slowly cook a changeset, i.e., a patch. We can modify the working copy bit by bit, and save the changes into the patch. At the beginning only `file_1` was changed:

```
$ hg qdiff --stat
 file_1 |  1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Let's make more change:

```
$ echo "some other code" > file_3
$ hg diff --stat
 file_3 |  1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Use `hg qref` to "refresh" the patch. After the refreshment, the modification is moved from the working copy to the patch:

```
$ hg qref
$ hg diff --stat
$ hg qdiff --stat
 file_1 |  1 +
 file_3 |  1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
```

Popping and Pushing Patches

A committed changeset can't be easily changed. In fact, it's nearly impossible to do it without the mq extension in Mercurial. The "obvious" way to change history in Mercurial is mq.

Right now we have two patches applied in our repository:

```
$ hg qapp
test
modify
```

The first applied patch is "test", while the second is "modify". Since they are patches, we can unapply and reapply them. And we do that with `hg qpop` and `hg qpush` commands, respectively.

Although it is Mercurial "queue", it actually operates like a stack, and we can pop and push patches from and to a mq. Let's pop the last patch for demonstration:

```
$ hg qpop
popping modify
now at: test
$ hg qapp
test
$ hg log -l 2
changeset:   5:860f045d5a1a
tag:         qbase
tag:         qtip
tag:         test
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:00:30 2013 +0800
summary:     Patch for testing.

changeset:   4:06dacab043bf
tag:         qparent
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:18 2013 +0800
summary:     Add 4 empty files.
```

And then push it back:

```
$ hg qpush
applying modify
now at: modify
```

We can also pop or push everything at once:

```
$ hg qpop -a
popping modify
popping test
patch queue now empty
$ hg qpush -a
applying test
patch test is empty
applying modify
now at: modify
$ hg qapp
test
modify
```

Finalization

After a series of hack, we will turn the patches in a mq back into regular changesets. We will do it by using `hg qfin` command:

```
$ hg qfin
abort: no revisions specified
```

One common mistake in using the command is forgetting to specify the patch to finish. By default `hg qfin` doesn't finish all patches, so that we can selectively finish one:

```
$ hg qser
test
modify
$ hg qfin test
$ hg qser
modify
```

Alternatively, we can also finish all patches at once:

```
$ hg qfin -a
$ hg qser
$ hg log -l 3
changeset:   6:be3db2f671d5
tag:         tip
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:33:01 2013 +0800
summary:     Create a patch with some modification in working copy.

changeset:   5:4e435afd759f
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 23 18:25:29 2013 +0800
summary:     Patch for testing.

changeset:   4:06dacab043bf
user:        yungyuc <yyc@solvcon.net>
date:        Sun Jun 16 16:47:18 2013 +0800
summary:     Add 4 empty files.
```

Note that when a patch is applied in a repository, Mercurial won't let you push, until now:

```
$ hg push ssh://hg@bitbucket.org/yungyuc/example_proj
pushing to ssh://hg@bitbucket.org/yungyuc/example_proj
```

```
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 2 changesets with 2 changes to 2 files
```

Other Topics

This is a basic tutorial to version control and Mercurial. There are several important topics that we haven't touched:

1. Clone and pull.
2. Branch (multiple heads) and merge.
3. Tag.
4. Multiple mq and mq repository.

We will visit them another time.

3.3.2 Unit Tests

3.3.3 Documentation

3.3.4 Management of Runtime and Dependencies

3.3.5 Packaging and Distribution

3.4 Numerical Analysis

3.4.1 Basic Array Operations

3.4.2 Linear Algebra

3.4.3 Fourier Analysis

Fourier Transform and Discrete Fourier Transform

Consider the Fourier transform pair [1]:

$$F(\omega) = \mathcal{F}\left\{f(t)\right\} \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(x) e^{-i2\pi\omega x} dx \tag{3.1}$$

$$f(x) = \mathcal{F}^{-1}\left\{F(\omega)\right\} \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} F(\omega) e^{i2\pi\omega x} d\omega \tag{3.2}$$

x denotes the temporal or spatial coordinate, and ω denotes the frequency coordinate. Equation (**??**) defines the forward Fourier transform from $f(x)$ to $F(\omega)$. Equation (**??**) defines the backward (inverse) Fourier transform from $F(\omega)$ to $f(x)$.

[1] William L. Briggs and Van Emden Henson, *The DFT: An Owners' Manual for the Discrete Fourier Transform*, SIAM, 1987. http://www.amazon.com/gp/product/0898713420/

Suppose the function $f(x)$ can be sampled in an interval $[0, A]$ with N discrete points of the same sub-interval $\Delta x = A/N$ as:

$$f_n \stackrel{\text{def}}{=} f(x_n) = f(n\Delta x), \quad n = 0, \ldots, N-1$$

The forward discrete Fourier transform (DFT) can be defined to be:

$$\tilde{F}(\frac{k}{A}) \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} f(n\Delta x) e^{-i2\pi \frac{nk}{N}} \tag{3.3}$$

There is a relationship between $F(\omega)$ (in Eq. (**??**)) and $\tilde{F}(k/A)$ (in Eq. (**??**)), which will be derived in what follows.

Assume $f(x) = 0$ for $x < 0, x > A$. Equation (**??**) can then be rewritten as:

$$F(\omega) = \int_0^A f(x) e^{-i2\pi\omega x} dx \tag{3.4}$$

To facilitate the derivation, the integrand in Eq. (**??**) be defined as:

$$g(x) \stackrel{\text{def}}{=} f(x) e^{-i2\pi\omega x} \tag{3.5}$$

Aided by the trapezoid rule and Eq. (**??**), the integration of Eq. (**??**) can be approximated as:

$$F(\omega) \cong \frac{\Delta x}{2} \left[g(0) + 2 \sum_{n=1}^{N-2} g(x_n) + g(A) \right] \tag{3.6}$$

Assume

$$g(0) = g(A)$$

then Eq. (**??**) can be written as:

$$F(\omega) \cong \Delta x \sum_{n=0}^{N-1} g(x_n) = \frac{A}{N} \sum_{n=0}^{N-1} f(x_n) e^{-i2\pi\omega x_n} \tag{3.7}$$

Because the longest wave length that the sampling interval allows is A, the frequency of the fundamental mode is

$$\Delta\omega = \frac{1}{A} \tag{3.8}$$

which is the spacing of the frequency-domain (ω) grid that covers the frequency interval $[-\Omega/2, \Omega/2]$ with N points. Aided by using Eq. (**??**), it can be obtained that

$$\Omega = N\Delta\omega = \frac{N}{A}$$

and thus

$$A\Omega = N \tag{3.9}$$

Because

$$\Delta x = \frac{A}{N}, \quad \Delta\omega = \frac{1}{A}$$

it can be shown that

$$\Delta x \Delta\omega = \frac{1}{N} \tag{3.10}$$

Equations (**??**) and (**??**) are the *reciprocity relations*.

To proceed, write

$$x_n \omega_k = (n\Delta x)(k\Delta\omega) = \frac{nA}{N}\frac{k}{A} = \frac{nk}{N}$$

Equation (**??**) becomes

$$F(\frac{k}{A}) \cong \frac{A}{N}\sum_{n=0}^{N-1} f(n\Delta x)e^{-i2\pi\frac{nk}{N}}$$

Substituting Eq. (**??**) into the previous equation gives:

$$F(\frac{k}{A}) \cong \frac{A}{N}\tilde{F}(\frac{k}{A}) \tag{3.11}$$

which defines the scaling relation between the Fourier transform (Eq. (**??**)) and the discrete Fourier transform (Eq. (**??**)).

Example Code

```
class Transform(object):
    def __init__(self, ngrid, extent, average=False):
        from numpy import arange, empty
        from fftw3 import Plan
        self.ngrid = ngrid
        self.extent = extent
        self.interval = interval = extent[1] - extent[0]
        # calculate xgrid.
        self.xgrid = xgrid = arange(ngrid, dtype='float64')
        xgrid /= ngrid-1
        xgrid *= interval
        xgrid += extent[0]
        self.dx = dx = xgrid[1] - xgrid[0]
        # calculate bandwidth, kgrid, and kscale.
        self.bw = bw = 1.0 / dx
        self.kgrid = kgrid = arange(ngrid, dtype='float64')
        kgrid /= ngrid
        kgrid *= bw
        kgrid -= bw/2
        self.kscale = 1.0 if average else interval/2
        self.kscale /= ngrid/2
        # make x-/k-arrays.
        self.xarrw = empty(ngrid, dtype='complex128')
        self.karr = empty(ngrid, dtype='complex128')
        self.karrw = empty(ngrid, dtype='complex128')
        # make fftw plans.
        self.wforward = Plan(self.xarrw, self.karrw,
            direction='forward', flags=['estimate'])
        self.wbackward = Plan(self.karrw, self.xarrw,
            direction='backward', flags=['estimate'])

    def forward(self):
        from numpy.fft import fft, fftshift
        self.karr[:] = fftshift(fft(self.xarrw))
        self.wforward()
        self.karrw[:] = fftshift(self.karrw)
```

```
37          self.karr *= self.kscale
38          self.karrw *= self.kscale
39
40      def report(self):
41          import sys
42          sys.stdout.write('ngrid: %d; ' % self.ngrid)
43          sys.stdout.write('extent: %g, %g; ' % tuple(self.extent))
44          sys.stdout.write('interval: %g; ' % self.interval)
45          sys.stdout.write('dx: %g; ' % self.dx)
46          sys.stdout.write('bandwidth: %g; ' % self.bw)
47          sys.stdout.write('krange: %g, %g ' % (self.kgrid[0], self.kgrid[-1]))
48          sys.stdout.write('\n')
49
50  class SineTransform(Transform):
51      def __init__(self, ngrid, extent, freq, **kw):
52          from numpy import sin, pi
53          super(SineTransform, self).__init__(ngrid, extent, **kw)
54          # remember the frequency.
55          self.freq = freq
56          # initialize x/t data.
57          self.xarrw[:] = sin(2*pi * freq * self.xgrid)
58          # for plotting.
59          self.fig = None
60          self.xax = None
61          self.kax = None
62
63      def plot(self, figsize=(12, 6)):
64          from numpy import absolute
65          from matplotlib import pyplot as plt
66          # create the figure.
67          self.fig = fig = plt.figure(figsize=figsize)
68          # plot in t/x-space.
69          self.xax = xax = fig.add_subplot(1, 2, 1)
70          xax.plot(self.xgrid, self.xarrw.real)
71          xax.set_title('$N$ = %d' % self.ngrid)
72          xax.set_xlim(self.xgrid[0], self.xgrid[-1])
73          xax.set_ylim(-1.1, 1.1)
74          xax.set_xlabel('$t$/$x$ (s/m)')
75          xax.grid()
76          # plot in f/k-space.
77          self.kax = kax = fig.add_subplot(1, 2, 2)
78          kax.plot(self.kgrid, absolute(self.karr), label='numpy.fft.fft')
79          kax.plot(self.kgrid, absolute(self.karrw), label='fftw3.plan')
80          kax.set_xlim(self.kgrid[0], self.kgrid[-1])
81          kax.set_xlabel('$f$/$k$ (Hz/$\\frac{1}{\mathrm{m}}$')
82          kax.grid()
83          kax.legend()
84
85  class RectTransform(Transform):
86      def __init__(self, ngrid, extent, **kw):
87          from numpy import absolute, sinc
88          super(RectTransform, self).__init__(ngrid, extent, **kw)
89          # initialize x/t data.
90          self.xarrw.fill(0)
91          self.xarrw[absolute(self.xgrid) < 0.5] = 1
92          self.kana = sinc(self.kgrid)
93          # for plotting.
94          self.fig = None
```

```
95          self.xax = None
96          self.kax = None
97
98      def plot(self, figsize=(12, 6)):
99          from numpy import absolute
100         from matplotlib import pyplot as plt
101         # create the figure.
102         self.fig = fig = plt.figure(figsize=figsize)
103         # plot in t/x-space.
104         self.xax = xax = fig.add_subplot(1, 2, 1)
105         xax.plot(self.xgrid, self.xarrw.real)
106         xax.set_title('$N$ = %d' % self.ngrid)
107         xax.set_xlim(self.xgrid[0], self.xgrid[-1])
108         xax.set_ylim(-0.1, 1.1)
109         xax.set_xlabel('$t$/$x$ (s/m)')
110         xax.grid()
111         # plot in f/k-space.
112         self.kax = kax = fig.add_subplot(1, 2, 2)
113         kax.plot(self.kgrid, absolute(self.karr), label='numpy.fft.fft')
114         kax.plot(self.kgrid, absolute(self.karrw), label='fftw3.Plan')
115         kax.plot(self.kgrid, absolute(self.kana), label='analytical')
116         kax.set_xlim(self.kgrid[0], self.kgrid[-1])
117         kax.set_xlabel('$f$/$k$ (Hz/$\\frac{1}{\mathrm{m}}$')
118         kax.grid()
119         kax.legend()
120
121 def main():
122     from matplotlib import pyplot as plt
123
124     stfm = SineTransform(2**7, (-1.5, 1.5), 1.0, average=True)
125     stfm.report()
126     stfm.forward()
127     stfm.plot()
128
129     rtfm1 = RectTransform(2**5, (-1., 1.), average=True)
130     rtfm1.report()
131     rtfm1.forward()
132     rtfm1.plot()
133     rtfm2 = RectTransform(100, (-5., 5.))
134     rtfm2.report()
135     rtfm2.forward()
136     rtfm2.plot()
137
138     plt.show()
139
140 if __name__ == '__main__':
141     main()
```

class `pyengr.fourier.`**`Fourier`**(*ngrid*, *extent*, *average=False*)

Fourier transform pair that supports both `numpy.fft` and `fftw3.Plan`.

3.4.4 Visualization

3.5 High-Performance and High-Throughput Computing

3.5.1 Multi-Threaded Programming

3.5.2 Distributed Computing

3.6 Recipes

3.6.1 Solving Partial Differential Equations

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p