MASSIMO DI PIERRO

PYTHON TUTORIAL (DRAFT NOTES)

EXPERTS4SOLUTIONS

Copyright 2008-2013 by Massimo Di Pierro. All rights reserved.

THE CONTENT OF THIS BOOK IS PROVIDED UNDER THE TERMS OF THE CREATIVE COMMONS PUBLIC LICENSE BY-NC-ND 3.0.

http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode

THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created ore extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For more information about appropriate use of this material contact:

Massimo Di Pierro
School of Computing
DePaul University
243 S Wabash Ave
Chicago, IL 60604 (USA)
Email: massimo.dipierro@gmail.com

Library of Congress Cataloging-in-Publication Data:

ISBN: ...

Build Date: May 13, 2013

Contents

1 Overview of the Python Language **9**

1.1 About Python 9

1.1.1 Python vs Java and C++ syntax 10

1.1.2 help, dir 10

1.2 Python Types 11

1.2.1 `decimal` 12

1.2.2 `str` 12

1.2.3 `list` and `array` 13

1.2.4 `tuple` 15

1.2.5 `dict` 16

1.2.6 `set` 18

1.3 Python control flow statements 19

1.3.1 `for...in` 20

1.3.2 `while` 22

1.3.3 `if...elif...else` 22

1.3.4 `try...except...else...finally` 23

1.3.5 `def...return` 25

1.3.6 `lambda` 28

1.4 Classes 29

1.4.1 Special methods and operator overloading 30

1.4.2 Financial Transaction 31

1.5 File input/output 33

1.6 `import` modules 34

1.6.1 `math` 34

1.6.2 os . . . 35
1.6.3 sys . . . 35
1.6.4 datetime . . . 36
1.6.5 time . . . 36
1.6.6 urllib and json . . . 37
1.6.7 cPickle . . . 38
1.6.8 sqlite database and persistence . . . 38
1.6.9 matplotlib . . . 41

2 Numerical and Financial Algorithms 47
2.1 Solving Systems of Linear Equations . . . 47
2.2 Modern Portfolio Theory . . . 48
2.3 Linear Least Squares and χ^2 . . . 51
2.4 Trading and technical analysis . . . 54

3 The database abstraction layer 57
3.1 Dependencies . . . 57
3.2 Connection strings . . . 59
3.2.1 Connection pooling . . . 61
3.2.2 Connection failures . . . 62
3.2.3 Replicated databases . . . 62
3.3 Reserved keywords . . . 62
3.4 DAL, Table, Field . . . 63
3.5 Record representation . . . 65
3.6 Migrations . . . 69
3.7 insert . . . 70
3.8 commit and rollback . . . 71
3.9 Query, Set, Rows . . . 72
3.10 select . . . 73
3.10.1 Shortcuts . . . 74
3.10.2 Fetching a Row . . . 75
3.10.3 Recursive selects . . . 76
3.10.4 Serializing Rows in views . . . 76
3.10.5 orderby, groupby, limitby, distinct, having . . . 78
3.10.6 Logical operators . . . 80

3.10.7 `count`, `isempty`, `delete`, `update` 81
3.10.8 Expressions 82
3.10.9 `case` 82
3.10.10 `update_record` 83
3.10.11 Inserting and updating from a dictionary 83
3.10.12 `first` and `last` 83
3.10.13 `as_dict` and `as_list` 84
3.10.14 Combining rows 84
3.10.15 `find`, `exclude`, `sort` 85
3.11 One to many relation 86
3.11.1 Inner joins 87
3.11.2 Left outer join 89
3.11.3 Grouping and counting 89
3.12 Many to many 90
3.13 Other Operators and Expressions 91
3.13.1 `like`, `regexp`, `startswith`, `contains`, `upper`, `lower` 91
3.13.2 `year`, `month`, `day`, `hour`, `minutes`, `seconds` 92
3.13.3 `belongs` 92
3.13.4 `sum`, `avg`, `min`, `max` and `len` 93
3.13.5 Substrings 94
3.13.6 Default values with `coalesce` and `coalesce_zero` . . . 94
3.13.7 CSV (one Table at a time) 95

Bibliography **97**

1

Overview of the Python Language

1.1 About Python

Python is a general-purpose high-level programming language. Its design philosophy emphasizes programmer productivity and code readability. It has a minimalist core syntax with very few basic commands and simple semantics. It also has a large and comprehensive standard library, including an Application Programming Interface (API) to many of the underlying operating system (OS) functions. Python provides built-in objects such as linked lists (`list`), tuples (`tuple`), hash tables (`dict`), arbitrarily long integers (`long`), complex numbers, and arbitrary precision decimal numbers.

Python supports multiple programming paradigms, including object-oriented (`class`), imperative (`def`), and functional (`lambda`) programming. Python has a dynamic type system and automatic memory management using reference counting (similar to Perl, Ruby, and Scheme).

Python was first released by Guido van Rossum in 1991[7]. The language has an open, community-based development model managed by the non-profit Python Software Foundation. There are many interpreters and compilers that implement the Python language, including one in Java (Jython), one built on .Net (IronPython) and one built in Python itself (PyPy). In this brief review, we refer to the reference C implementation created by Guido.

You can find many tutorials, the official documentation, and library references of the language on the official Python website. [1]

For additional Python references, we can recommend the books in ref. [7] and ref. [8].

You may skip this chapter if you are already familiar with the Python language.

1.1.1 Python vs Java and C++ syntax

	Java/C++	**Python**
assignment	$a = b$;	$a = b$
comparison	if $(a == b)$	if $a == b$:
loops	for$(a = 0; a < n; a++)$	for a in $range(0, n)$:
block	Braces {...}	indentation
function	$float f$(float a) {	def $f(a)$:
function call	$f(a)$	$f(a)$
arrays/lists	$a[i]$	$a[i]$
member	a.member	a.member
nothing	*null / void**	*None*

As in Java, variables which are primitive types (bool, int, float) are passed by copy but more complex types, unlike C++, are passed by reference.

1.1.2 help, dir

The Python language provides two commands to obtain documentation about objects defined in the current scope, whether the object is built-in or user-defined.

We can ask for `help` about an object, for example "1":

```
>>> help(1)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
```

```
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base.  It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
...
```

and, since "1" is an integer, we get a description about the `int` class and all its methods. Here the output has been truncated because it is very long and detailed.

Similarly, we can obtain a list of methods of the object "1" with the command `dir`:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
'__floordiv__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__',
'__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__',
'__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__',
'__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__str__', '__sub__', '__truediv__', '__xor__']
```

1.2 Python Types

Python is a dynamically typed language, meaning that variables do not have a type and therefore do not have to be declared. Variables may also change the type of value they hold through its life. Values, on the other hand, do have a type. You can query a variable for the type of value it contains:

```
>>> a = 3
>>> print type(a)
<type 'int'>
>>> a = 3.14
>>> print type(a)
```

```
<type 'float'>
>>> a = 'hello python'
>>> print type(a)
<type 'str'>
```

Python also includes, natively, data structures such as lists and dictionaries.

1.2.1 decimal

Python also has a module for decimal floating point arithmetic which allows decimal numbers to be represented exactly. The class Decimal incorporates a notion of significant places (unlike hardware based binary floating point, the decimal module has a user alterable precision):

```
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 28 # set precision
>>> Decimal(1) / Decimal(7)
Decimal('0.14285714285714285714285714291')
```

Decimal numbers can be used almost everywhere in place of floating point number arithmetics but are slower and should be used only where arbitrary precision arithmetics is required. It does not suffer from the overflow, underflow, and precision issues described above.

```
>>> from decimal import Decimal
>>> a = Decimal(10.0)**300
>>> a*a
Decimal('1.0000000000000000000000000000E+600')
```

1.2.2 str

Strings are delimited by `'...'`, `"..."`, `"'..."'`, or `"""..."""`. Triple quotes delimit multiline strings.

```
>>> a = 'this is an ASCII string'
```

It is also possible to write variables into strings in various ways:

```
>>> print 'number is ' + str(3)
number is 3
>>> print 'number is %s' % (3)
number is 3
>>> print 'number is %(number)s' % dict(number=3)
number is 3
```

The final notation is more explicit and less error prone, and is to be preferred.

Many Python objects, for example numbers, can be serialized into strings using `str` or `repr`. These two commands are very similar but produce slightly different output. For example:

```
>>> for i in [3, 'hello']:
...     print str(i), repr(i)
3 3
hello 'hello'
```

For user-defined classes, `str` and `repr` can be defined/redefined using the special operators `__str__` and `__repr__`. These are briefly described later in this chapter. For more information on the topic, refer to the official Python documentation [9].

Another important characteristic of a Python string is that it is an iterable object, similar to a list:

```
>>> for i in 'hello':
...     print i
h
e
l
l
o
```

1.2.3 `list` and `array`

A list is a native mutable Python object made of a sequence of arbitrary objects. Arrays are not native to Python but unlike lists each element must be of the same type but are computationally faster.

The main methods of a Python lists and arrays are append, insert, and delete. Other useful methods include count, index, reverse and sort. Note: arrays do not have a sort method.

```
>>> b = [1, 2, 3]
>>> print type(b)
<type 'list'>
>>> b.append(8)
>>> b.insert(2, 7) # insert 7 at index 2 (3rd element)
>>> del b[0]
>>> print b
[2, 7, 3, 8]
```

```
>>> print len(b)
4
>>> b.append(3)
>>> b.reverse()
print b," 3 appears ", b.count(3), " times.  The number 7 appears at index ", b.
    index(7)
[3, 8, 3, 7, 2] 3 appears 2 times.  The number 7 appears at index 3
```

Lists can be sliced:

```
>>> a= [2, 7, 3, 8]
>>> print a[:3]
[2, 7, 3]
>>> print a[1:]
[7, 3, 8]
>>> print a[-2:]
[3, 8]
```

and concatenated/joined:

```
>>> a = [2, 7, 3, 8]
>>> a = [2, 3]
>>> b = [5, 6]
>>> print a + b
[2, 3, 5, 6]
```

A list is iterable; you can loop over it:

```
>>> a = [1, 2, 3]
>>> for i in a:
...     print i
1
2
3
```

There is a very common situation for which a *list comprehension* can be used. Consider the following code:

```
>>> a = [1,2,3,4,5]
>>> b = []
>>> for x in a:
...     if x % 2 == 0:
...         b.append(x * 3)
>>> print b
[6, 12]
```

This code clearly processes a list of items, selects and modifies a subset of the input list, and creates a new result list. This code can be entirely replaced with the following list comprehension:

```
>>> a = [1,2,3,4,5]
```

```
>>> b = [x * 3 for x in a if x % 2 == 0]
>>> print b
[6, 12]
```

Python has a module called `array`. It provides an efficient array implementation. Unlike lists, array elements must all be of the same type and the type must be either a char, short, int, long, float or double. A type of char, short, int or long may be either signed or unsigned.

```
>>> from array import array
>>> a = array('d',[1,2,3,4,5])
array('d',[1.0, 2.0, 3.0, 4.0, 5.0])
```

An array object can be used in the same way as a list but its elements must all be of the same type, specified by the first argument of the constructor ("d" for double, "l" for signed long, "f" for float, and "c" for character). For a complete list of avilable options, refer to the official Python documentation.

Using "array" over "list" can be faster but, more importantly, the "array" storage is more compact for large arrays.

1.2.4 `tuple`

A tuple is similar to a list, but its size and elements are immutable. If a tuple element is an object, the object itself is mutable but the reference to the object is fixed. A tuple is defined by elements separated by a comma and optionally delimited by round brackets:

```
>>> a = 1, 2, 3
>>> a = (1, 2, 3)
```

The round brakets are required for a tuple of zero elements such as

```
>>> a = () # this is an empty tuple
```

A trailing comma is required for a one element tuple but not for two or more elements.

```
>>> a = (1)   # not a tuple
>>> a = (1,)  # this is a tuple of one element
>>> b = (1,2) # this is a tuple of two elements
```

Since lists are mutable, this works:

```
>>> a = [1, 2, 3]
>>> a[1] = 5
```

```
>>> print a
[1, 5, 3]
```

the element assignment does not work for a tuple:

```
>>> a = (1, 2, 3)
>>> print a[1]
2
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

A tuple, like a list, is an iterable object. Notice that a tuple consisting of a single element must include a trailing comma, as shown below:

```
>>> a = (1)
>>> print type(a)
<type 'int'>
>>> a = (1,)
>>> print type(a)
<type 'tuple'>
```

Tuples are very useful for efficient packing of objects because of their immutability. The brackets are often optional. You may easily get each element of a tuple by assigning multiple variables to a tuple at one time:

```
>>> a = (2, 3, 'hello')
>>> (x, y, z) = a
>>> print x
2
>>> print z
hello
>>> a = 'alpha', 35, 'sigma' # notice the rounded brackets are optional
>>> p, r, q = a
print r
35
```

1.2.5 dict

A Python `dict`-ionary is a hash table that maps a key object to a value object. For example:

```
>>> a = {'k':'v', 'k2':3}
>>> print a['k']
v
>>> print a['k2']
3
```

```
>>> 'k' in a
True
>>> 'v' in a
False
```

You will notice that the format to define a dictionary is the same as JavaScript Object Notation [JSON]. Dictionaries may be nested:

```
>>> a = {'x':3, 'y':54, 'z':{'a':1,'b':2}}
>>> print a['z']
{'a': 1, 'b': 2}
>>> print a['z']['a']
1
```

Keys can be of any hashable type (int, string, or any object whose class implements the `__hash__` method). Values can be of any type. Different keys and values in the same dictionary do not have to be of the same type. If the keys are alphanumeric characters, a dictionary can also be declared with the alternative syntax:

```
>>> a = dict(k='v', h2=3)
>>> print a['k']
v
>>> print a
{'h2': 3, 'k': 'v'}
```

Useful methods are `has_key`, `keys`, `values`, `items` and `update`:

```
>>> a = dict(k='v', k2=3)
>>> print a.keys()
['k2', 'k']
>>> print a.values()
[3, 'v']
>>> a.update({'n1':'new item'})     # adding a new item
>>> a.update(dict(n2='newer item')) # alternate method to add a new item
>>> a['n3'] = 'newest item'         # another method to add a new item
>>> print a.items()
[('k2', 3), ('k', 'v'), ('n3', 'newest item'), ('n2', 'newer item'), ('n1', 'new
    item')]
```

The `items` method produces a list of tuples, each containing a key and its associated value.

Dictionary elements and list elements can be deleted with the command `del`:

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> print a
[1, 3]
>>> a = dict(k='v', h2=3)
```

```
>>> del a['h2']
>>> print a
{'k': 'v'}
```

Internally, Python uses the `hash` operator to convert objects into integers, and uses that integer to determine where to store the value. Using a key that is not hashable will cause an `unhashable type` error

```
>>> hash("hello world")
-1500746465
>>> k = [1,2,3]
>>> a = {k:'4'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

1.2.6 `set`

A set is something between a list and dictionary. It represents a non-ordered list of unique elements. Elements in a set cannot be repeated. Internally it is implemented as a hash-table, similar to a set of keys in a dictionary. A set is created using the `set` constructor. Its argument can be a list, a tuple, or an iterator:

```
>>> s = set([1,2,3,4,5,5,5,5])  # notice duplicate elements are removed
>>> print s
set([1,2,3,4,5])
>>> s = set((1,2,3,4,5))
>>> print s
set([1,2,3,4,5])
>>> s = set(i for i in range(1,6))
>>> print s
set([1, 2, 3, 4, 5])
```

Since sets are non-ordered list, appending to the end is not applicable. Instead of `append`, add elements to a set using the `add` method:

```
>>> s = set()
>>> s.add(2)
>>> s.add(3)
>>> s.add(2)
>>> print s
set([2, 3])
```

Notice that the same element can not be added twice (2 in the example). The is no exception/error thrown when trying to add the same element more

than once.

Since sets are non-ordered, the order you add items is not necessarily the order they will be returned.

```
>>> s = set([6,'b','beta',-3.4,'a',3,5.3])
>>> print (s)
set(['a', 3, 6, 5.3, 'beta', 'b', -3.4])
```

The `set` object supports normal set operations like union, intersection, and difference:

```
>>> a = set([1,2,3])
>>> b = set([2,3,4])
>>> c = set([2,3])
>>> print a.union(b)
set([1, 2, 3, 4])
>>> print a.intersection(b)
set([2, 3])
>>> print a.difference(b)
set([1])
>>> if len(c) == len(a.intersection(c)):
...     print "c is a subset of a"
... else:
...     print "c is not a subset of a"
...
c is a subset of a
```

to check for membership:

```
>>> 2 in a
True
```

1.3 Python control flow statements

Python uses indentation to delimit blocks of code. A block starts with a line ending with colon and continues for all lines that have a similar or higher indentation as the next line. For example:

```
>>> i = 0
>>> while i < 3:
...    print i
...    i = i + 1
0
1
2
```

It is common to use four spaces for each level of indentation. It is a good policy not to mix tabs with spaces, which can result in (invisible) confusion.

1.3.1 `for...in`

In Python, you can loop over iterable objects:

```
>>> a = [0, 1, 'hello', 'python']
>>> for i in a:
...     print i
0
1
hello
python
```

In the example above you will notice that the loop index, 'i', takes on the values of each element in the list [0, 1, 'hello', 'python'] sequentially. Python `range` keyword creates a list of integers automatically, that may be used in a 'for' loop without manually creating a long list of numbers.

```
>>> a = range(0,5)
>>> print a
[0, 1, 2, 3, 4]
>>> for i in a:
...     print i
0
1
2
3
4
```

The parameters for `range(a,b,c)` are first parameter is the starting value of the list. The second parameter is next value if the list contained one more element. The third parameter is the increment value.

`range` can also be called with one parameter. It is matched to "b" above with the first paramater defaulting to 0 and the third to 1.

```
>>> print range(5)
[0, 1, 2, 3, 4]
>>> print range(53,57)
[53,54,55,56]
>>> print range(102,200,10)
[102, 112, 122, 132, 142, 152, 162, 172, 182, 192]
>>> print range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

`range` is very convenient for creating a list of numbers, however as the list grows in length, the memory required to store the list also grows. A more efficient option is to use the keyword is `xrange`. It generates an iterable range instead of generating the entire list of elements.

```
>>> for i in xrange(0, 4):
...     print i
0
1
2
3
```

This is equivalent to the C/C++/C#/Java syntax:

```
for(int i=0; i<4; i=i+1) { print i; }
```

Another useful command is `enumerate`, which counts while looping and returns a tuple consisting of (index, value):

```
>>> a = [0, 1, 'hello', 'python']
>>> for (i, j) in enumerate(a):   # the ( ) around i, j are optional
...     print i, j
0 0
1 1
2 hello
3 python
```

There is also a keyword `range(a, b, c)` that returns a list of integers starting with the value `a`, incrementing by `c`, and ending with the last value smaller than `b`, `a` defaults to `0` and `c` defaults to `1`. `xrange` is similar but does not actually generate the list, only an iterator over the list; thus it is better for looping.

You can jump out of a loop using `break`

```
>>> for i in [1, 2, 3]:
...      print i
...      break
1
```

You can jump to the next loop iteration without executing the entire code block with `continue`

```
>>> for i in [1, 2, 3]:
...      print i
...      continue
...      print 'test'
1
2
```

```
3
```

Python also supports list comprehensions and you can build lists using using the following syntax:

```
>>> a = [i*i for i in [0, 1, 2, 3]:
>>> print a
[0, 1, 4, 9]
```

Sometimes you may need a counter to "count" the elements of a list while looping:

```
>>> a = [e*(i+1) for (i,e) in ['a','b','c','d']]
>>> print a
['a', 'bb', 'ccc', 'dddd']
```

1.3.2 `while`

Comparison operators in Python follow the C/C++/Java operators of ==, !=, ... etc. However Python also accepts the <> operator as not equal to and is equivalent to !=. Logical operators are `and`, `or` and `not`.

The `while` loop in Python works much as it does in many other programming languages, by looping an indefinite number of times and testing a condition before each iteration. If the condition is `False`, the loop ends.

```
>>> i = 0
>>> while i < 10:
...     i = i + 1
>>> print i
10
```

The `for` loop was introduced earlier in this chapter.

There is no `loop...until` or `do...while` construct in Python.

1.3.3 `if...elif...else`

The use of conditionals in Python is intuitive:

```
>>> for i in range(3):
...     if i == 0:
...         print 'zero'
...     elif i == 1:
...         print 'one'
```

```
...     else:
...         print 'other'
zero
one
other
```

`elif` means "else if". Both `elif` and `else` clauses are optional. There can be more than one `elif` but only one `else` statement. Complex conditions can be created using the `not`, `and` and `or` logical operators.

```
>>> for i in range(3):
...     if i == 0 or (i == 1 and i + 1 == 2):
...         print '0 or 1'
```

1.3.4 `try...except...else...finally`

Python can throw - pardon, raise - Exceptions:

```
>>> try:
...     a = 1 / 0
... except Exception, e:
...     print 'oops: %s' % e
... else:
...     print 'no problem here'
... finally:
...     print 'done'
oops: integer division or modulo by zero
done
```

If an exception is raised, it is caught by the `except` clause and the `else` clause is not not executed. The `finally` clause is always executed.

There can be multiple `except` clauses for different possible exceptions:

```
>>> try:
...     raise SyntaxError
... except ValueError:
...     print 'value error'
... except SyntaxError:
...     print 'syntax error'
syntax error
```

The `finally` clause is guaranteed to be executed while the `except` and `else` are not. In the example below the function returns within a `try` block. This is bad practice, but it shows that the finally will execute regardless of the reason the `try` block is exited.

```
>>> def f(x):
...     try:
...         r = x*x
...         return r  # bad practice
...     except:
...         print "exception occurred %s" % e
...     else:
...         print "nothing else to do"
...     finally:
...         print "Finally we get here"
...
>>> y = f(3)
Finally we get here
>>> print "result is ", y
result is  9
```

For every `try` you must have either a `except` or `finally` while the `else` is optional

Here is a list of built-in Python exceptions

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- GeneratorExit
      +-- StopIteration
      +-- StandardError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
```

```
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |    |    +-- UnicodeError
      |    |         +-- UnicodeDecodeError
      |    |         +-- UnicodeEncodeError
      |    |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
```

For a detailed description of each of them, refer to the official Python documentation.

Any object can be raised as an exception, but it is good practice to raise objects that extend one of the built-in exception classes.

1.3.5 `def...return`

Functions are declared using `def`. Here is a typical Python function:

```
>>> def f(a, b):
...     return a + b
>>> print f(4, 2)
6
```

There is no need (or way) to specify the type of an argument(s) or the return value(s). In this example, a function `f` is defined that can take two arguments.

Functions are the first code syntax feature described in this chapter to introduce the concept of *scope*, or *namespace*. In the above example, the identifiers `a` and `b` are undefined outside of the scope of function `f`:

```
>>> def f(a):
...     return a + 1
>>> print f(1)
```

```
2
>>> print a
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print a
NameError: name 'a' is not defined
```

Identifiers defined outside of function scope are accessible within the function; observe how the identifier a is handled in the following code:

```
>>> a = 1
>>> def f(b):
...     return a + b
>>> print f(1)
2
>>> a = 2
>>> print f(1) # new value of a is used
3
>>> a = 1 # reset a
>>> def g(b):
...     a = 2 # creates a new local a
...     return a + b
>>> print g(2)
4
>>> print a # global a is unchanged
1
```

If a is modified, subsequent function calls will use the new value of the global a because the function definition binds the storage location of the identifier a, not the value of a itself at the time of function declaration; however, if a is assigned-to inside function g, the global a is unaffected because the new local a hides the global value. The external-scope reference can be used in the creation of *closures*:

```
>>> def f(x):
...     def g(y):
...         return x * y
...     return g
>>> doubler = f(2) # doubler is a new function
>>> tripler = f(3) # tripler is a new function
>>> quadrupler = f(4) # quadrupler is a new function
>>> print doubler(5)
10
>>> print tripler(5)
15
>>> print quadrupler(5)
20
```

Function `f` creates new functions; and note that the scope of the name `g` is entirely internal to `f`. Closures are extremely powerful.

Function arguments can have default values and can return multiple results as a tuple: (Notice the parentheses are optional and are omitted in the example.)

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(5)
>>> print x
7
>>> print y
3
```

Function arguments can be passed explicitly by name, therefore the order of arguments specified in the caller can be different than the order of arguments with which the function was defined:

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(b=5, a=2)
>>> print x
7
>>> print y
-3
```

Functions can also take a runtime-variable number of arguments. Parameters that start with $*$ and $**$ must be the last two parameters. If the $**$ parameter is used it must be last in the list. Extra values passed in will be placed in the `*identifier` parameter while named values will be placed into the ft **identifier. Notice that when passing values into the function the unnamed values must be before any and all named values.

```
>>> def f(a, b, *extra, **extraNamed):
...     print "a = ", a
...     print "b = ", b
...     print "extra = ", extra
...     print "extranamed = ", extraNamed
>>> f(1, 2, 5, 6, x=3, y=2, z=6)
a =  1
b =  2
extra =  (5, 6)
extranamed =  {'y': 2, 'x': 3, 'z': 6}
```

Here the first two paramters (1 and 2) are matched with the paramters `a` and `b` while the tuple 5, 6 is placed into `extra` and the remaining items (which are

in a dictionary format) are placed into `extraNamed`

In the opposite case, a list or tuple can be passed to a function that requires individual positional arguments by unpacking them:

```
>>> def f(a, b):
...     return a + b
>>> c = (1, 2)
>>> print f(*c)
3
```

and a dictionary can be unpacked to deliver keyword arguments:

```
>>> def f(a, b):
...     return a + b
>>> c = {'a':1, 'b':2}
>>> print f(**c)
3
```

1.3.6 `lambda`

`lambda` provides a way to define a short unnamed function:

```
>>> a = lambda b: b + 2
>>> print a(3)
5
```

The expression "`lambda` [a]:[b]" literally reads as "a function with arguments [a] that returns [b]". The `lambda` expression is itself unnamed, but the function acquires a name by being assigned to identifier `a`. The scoping rules for `def` apply to `lambda` equally, and in fact the code above, with respect to `a`, is identical to the function declaration using `def`:

```
>>> def a(b):
...     return b + 2
>>> print a(3)
5
```

The only benefit of `lambda` is brevity; however, brevity can be very convenient in certain situations. Consider a function called `map` that applies a function to all items in a list, creating a new list:

```
>>> a = [1, 7, 2, 5, 4, 8]
>>> map(lambda x: x + 2, a)
[3, 9, 4, 7, 6, 10]
```

This code would have doubled in size had `def` been used instead of `lambda`. The main drawback of `lambda` is that (in the Python implementation) the

syntax allows only for a single expression; however, for longer functions, `def` can be used and the extra cost of providing a function name decreases as the length of the function grows. Just like `def`, `lambda` can be used to *curry* functions: new functions can be created by wrapping existing functions such that the new function carries a different set of arguments:

```
>>> def f(a, b): return a + b
>>> g = lambda a: f(a, 3)
>>> g(2)
5
```

> Python functions, created with either `def` or `lambda` allow re-factoring of existing functions in terms of a different set of arguments.

1.4 Classes

Because Python is dynamically typed, Python classes and objects may seem odd. In fact, member variables (attributes) do not need to be specifically defined when declaring a class and different instances of the same class can have different attributes. Attributes are generally associated with the instance, not the class (except when declared as "class attributes", which is the same as "static member variables" in C++/Java).

Here is an example:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.myvariable = 3
>>> print myinstance.myvariable
3
```

Notice that `pass` is a do-nothing command. In this case it is used to define a class `MyClass` that contains nothing. `MyClass()` calls the constructor of the class (in this case the default constructor) and returns an object, an instance of the class. The `(object)` in the class definition indicates that our class extends the built-in `object` class. This is not required, but it is good practice.

Here is a more involved class with multiple methods:

```
>>> class Complex(object):
...     z = 2
...     def __init__(self, real=0.0, imag=0.0):
...         self.real, self.imag = real, imag
```

```
...     def magnitude(self):
...         return (self.real**2 + self.imag**2)**0.5
...     def __add__(self,other):
...         return Complex(self.real+other.real,self.imag+other.imag)
>>> a = Complex(1,3)
>>> b = Complex(2,1)
>>> c = a + b
>>> print c.magnitude()
5
```

Functions declared inside the class are methods. Some methods have special reserved names. For example, `__init__` is the constructor. In the example we created a class to store the `real` and the `imag` part of a complex number. The constructor takes these two variables and stores them into `self` (not a keyword but a variable that plays the same role as `this` in Java and `(*this)` in C++. (This syntax is necessary to avoid ambiguity when declaring nested classes, such as a class that is local to a method inside another class, something the Python allows but Java and C++ do not).

The `self` variable is defined by the first argument of each method. They all must have it but they can use another variable name. Even if we use another name, the first argument of a method always refers to the object calling the method. It plays the same role as the `this` keyword in the Java and the C++ languages.

`__add__` is also a special method (all special methods start and end in double underscore) and it overloads the `+` operator between `self` and `other`. In the example, `a+b` is equivalent to a call to `a.__add__(b)` and the `__add__` method receives `self=a` and `other=b`.

All variables are local variables of the method except variables declared outside methods which are called *class variables*, equivalent to C++ *static member variables* that hold the same value for all instances of the class.

1.4.1 Special methods and operator overloading

Class attributes, methods, and operators starting with a double underscore are usually intended to be private (*for example* to be used internally but not exposed outside the class) although this is a convention that is not enforced

by the interpreter.

Some of them are reserved keywords and have a special meaning.

For example:

- `__len__`
- `__getitem__`
- `__setitem__`

They can be used, for example, to create a container object that acts like a list:

```
>>> class MyList(object):
>>>     def __init__(self, *a): self.a = list(a)
>>>     def __len__(self): return len(self.a)
>>>     def __getitem__(self, key): return self.a[key]
>>>     def __setitem__(self, key, value): self.a[key] = value
>>> b = MyList(3, 4, 5)
>>> print b[1]
4
>>> b.a[1] = 7
>>> print b.a
[3, 7, 5]
```

Other special operators include `__getattr__` and `__setattr__`, which define the get and set methods (getters and setters) for the class, and `__add__`, `__sub__`, `__mul__`, `__div__` which overload arithmetic operators. For the use of these operators we refer the reader to the chapter on linear algebra where they will be used to implement algebra for matrices.

1.4.2 Financial Transaction

As one more example of a class, we will implement a class that represents a financial transaction. We can think of a simple transaction as a single money transfer of quantity `a` that occurs at a given time t. We adopt the convention that a positive `amount` represents money flowing in and a negative value represents money flowing out.

The Net Present Value (computed at time t_0) for a transaction occurring at time t days from now of amount A is defined as

$$\mathrm{NPV}(t, A) = Ae^{-tr} \tag{1.1}$$

where r is the daily risk free interest rate. If t is measured in days, r has to be the daily risk free return. Here we will assume it defaults to $r = 005/365$ (5% annually).

Here is a possible implementation of the Transaction as a class and CashFlow as a class.

```
from datetime import date
from math import exp
today = date.today()
r_free = 0.05/365.0

class FinancialTransaction(object):
    def __init__(self,t,a,description=''):
        self.t= t
        self.a = a
        self.description = description
    def npv(self, t0=today, r=r_free):
        return self.a*exp(r*(t0-self.t).days)
    def __str__(self):
        return '%.2f dollars in %i days (%s)' % \
            (self.a, self.t, self.description)

class CashFlow(object):
    def __init__(self):
        self.transactions = []
    def add(self,transaction):
        self.transactions.append(transaction)
    def npv(self, t0, r=r_free):
        return sum(x.npv(t0,r) for x in self.transactions)
    def __str__(self):
        return '\n'.join(str(x) for x in self.transactions)
```

Here we assume t and t_0 are `datetime.date` objects that store a date. The `date` constructor takes the year, the month, and the day separated by a comma. The expression `(t0-t).days` computes the distance in days between t_0 and t.

What is the net present value at the beginning of 2012 for a fixed rate bond that pays one coupon of $1000 the 20th of each month for the following 24 month)?

```
>>> fixed_rate_bond = CashFlow()
>>> today = date(2012,1,1)
>>> for year in range(2012,2014):
...     for month in range(1,13):
...         coupon = FinancialTransaction(date(year,month,20),1000)
...         fixed_rate_bond.add(coupon)
```

```
>>> print round(fixed_rate_bond.npv(today,r=0.05/365),0)
22826
```

This means the cost for this bond should be $22826.

1.5 File input/output

In Python you can open and write in a file with:

```
>>> file = open('myfile.txt', 'w')
>>> file.write('hello world')
>>> file.close()
```

Similarly, you can read back from the file with:

```
>>> file = open('myfile.txt', 'r')
>>> print file.read()
hello world
```

Alternatively, you can read in binary mode with "rb", write in binary mode with "wb", and open the file in append mode "a", using standard C notation.

The `read` command takes an optional argument, which is the number of bytes. You can also jump to any location in a file using `seek`.

You can read back from the file with `read`

```
>>> print file.seek(6)
>>> print file.read()
world
```

and you can close the file with:

```
>>> file.close()
```

> In the standard distribution of Python, which is known as CPython, variables are reference-counted, including those holding file handles, so CPython knows that when the reference count of an open file handle decreases to zero, the file may be closed and the variable disposed. However, in other implementations of Python such as PyPy, garbage collection is used instead of reference counting, and this means that it is possible that there may accumulate too many open file handles at one time, resulting in an error before the *gc* has a chance to close and dispose of them all. Therefore it is best to explicitly close file handles when they are no longer needed.

1.6 `import` modules

The real power of Python is in its library modules. They provide a large and consistent set of Application Programming Interfaces (APIs) to many system libraries (often in a way independent of the operating system).

For example, if you need to use a random number generator, you can do:

```
>>> import random
>>> print random.randint(0, 9)
5
```

This prints a random integer in the range of (0,9], 5 in the example. The function `randint` is defined in the module `random`. It is also possible to import an object from a module into the current namespace:

```
>>> from random import randint
>>> print randint(0, 9)
```

or import all objects from a module into the current namespace:

```
>>> from random import *
>>> print randint(0, 9)
```

or import everything in a newly defined namespace:

```
>>> import random as myrand
>>> print myrand.randint(0, 9)
```

In the rest of this book, we will mainly use objects defined in modules `math`, `os`, `sys`, `datetime`, `time` and `cPickle`. We will also use the `random` module but we will describe it in a later chapter.

In the following subsections we consider those modules that are most useful.

1.6.1 `math`

Here is a sampling of some of the methods availble in the `math` package.

- `math.isinf(x)` returns true if the floating point number ft x is positive or negative infinity
- `math.isnan(x)` returns true if the floating point number ft x is NaN. See Python documentation or IEEE 754 standards for more information.
- `math.exp(x)` returns ft e**x

- `math.log(x[, base]` returns the logarithm of ft x to the optional `base`. If ft base is not supplied ft e is assumed.
- `math.cos(x)`,`math.sin(x)`,`math.tan(x)` returns the cos, sin, tan of the value of `x`. `x` is in radians.
- `math.pi`, `math.e` the constants for ft pi and ft e to available precision

ft math.isinf(x) will return

1.6.2 `os`

This module provides an interface to the operating system API. For example:

```
>>> import os
>>> os.chdir('..')
>>> os.unlink('filename_to_be_deleted')
```

> Some of the `os` functions, such as `chdir`, are not thread-safe, *for example* they should not be used in a multi-threaded environment.

`os.path.join` is very useful; it allows the concatenation of paths in an OS-independent way:

```
>>> import os
>>> a = os.path.join('path', 'sub_path')
>>> print a
path/sub_path
```

System environment variables can be accessed via:

```
>>> print os.environ
```

which is a read-only dictionary.

1.6.3 `sys`

The `sys` module contains many variables and functions, but the used the most is `sys.path`. It contains a list of paths where Python searches for modules. When we try to import a module, Python searches the folders listed in `sys.path`. If you install additional modules in some location and want Python to find them, you need to append the path to that location to `sys.path`.

```
>>> import sys
>>> sys.path.append('path/to/my/modules')
```

1.6.4 datetime

The use of the datetime module is best illustrated by some examples:

```
>>> import datetime
>>> print datetime.datetime.today()
2008-07-04 14:03:90
>>> print datetime.date.today()
2008-07-04
```

Occasionally you may need to time-stamp data based on the UTC time as opposed to local time. In this case you can use the following function:

```
>>> import datetime
>>> print datetime.datetime.utcnow()
2008-07-04 14:03:90
```

The `datetime` module contains various classes: `date`, `datetime`, `time` and `timedelta`. The difference between two date or two datetime or two time objects is a `timedelta`:

```
>>> a = datetime.datetime(2008, 1, 1, 20, 30)
>>> b = datetime.datetime(2008, 1, 2, 20, 30)
>>> c = b - a
>>> print c.days
1
```

We can also parse dates and datetimes from strings for example:

```
>>> s = '2011-12-31'
>>> a = datetime.datetime.strptime(s,'%Y-%m-%d')  #modified
>>> print s.year, s.day, s.month
2011 31 12  #modified
```

Notice that "%Y" matches the 4-digits year, "%m" matches the month as a number (1-12), "%d" matches the day (1-31), "%H" matches the hour, "%M" matches the minute, and "%S" matches the second. Check the Python documentation for more options.

1.6.5 time

The time module differs from `date` and `datetime` because it represents time as seconds from the epoch (beginning of 1970).

```
>>> import time
>>> t = time.time()
1215138737.571
```

Refer to the Python documentation for conversion functions between time in seconds and time as a `datetime`.

1.6.6 `urllib` and `json`

The `urllib` is a module to download data or a web page from a URL.

```
>>> page = urllib.urlopen('http://www.google.com/')
>>> html = page.read()
```

Usually urllib is used to download data posted online. The challenge may be parsing the data (converting from the representation used to post it to a proper Python representation).

Below create a simple helper class that can download data from Yahoo Finance and convert each stock's historical data into a list of dictionaries. Each list element corresponds to a trading day of history of the stock and each dictionary stores the data relative to that trading day (date, open, close, volume, adjusted close, arithmetic_return, log_return, etc.):

Listing 1.1: in file: `nlib.py`

```
>>> from nlib import YStock
>>> google = YStock('GOOG')
>>> current = google.current()
>>> price = current['price']
>>> market_cap = current['market_cap']
>>> h = google.historical()
>>> last_adjusted_close = h[-1]['adjusted_close']
>>> last_log_return = h[-1]['log_return']
```

Many web services return data in JSON format. JSON is slowly replacing XML as favorite protocol for data stransfer on the web. It is lighter, simpler to use and more human readbale. JSON is a serialized Javascript. the JSON data can be converted to a Python object using a library called `simplejson`. Once you have installed **simplejson** you can use it to convert Python objects into JSON objects and vice-versa:

```
>>> import simplejson
>>> a = [1,2,3]
>>> b = simplejson.dumps(a)
>>> print type(b)
<type 'str'>
>>> c = simplejson.loads(b)
```

```
>>> a == c
True
```

`simplejson`'s `loads` and `dumps` method work very much as `cPickle`'s methods but they serialize the objects into a string using JSON instead of the pickle protocol.

1.6.7 `cPickle`

This is a very powerful module. It provides functions that can serialize almost any Python object, including self-referential objects. For example, let's build a weird object:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.x = 'something'
>>> a = [1 ,2, {'hello':'world'}, [3, 4, [myinstance]]]
```

and now:

```
>>> import cPickle
>>> b = cPickle.dumps(a)
>>> c = cPickle.loads(b)
```

In this example, `b` is a string representation of `a`, and `c` is a copy of `a` generated by de-serializing `b`. cPickle can also serialize to and de-serialize from a file:

```
>>> cPickle.dump(a, open('myfile.pickle', 'wb'))
>>> c = cPickle.load(open('myfile.pickle', 'rb'))
```

1.6.8 `sqlite` database and persistence

The Python dictionary type is very useful but it lacks persistence because it is stored in RAM (it is lost if a program ends) and cannot be shared by more than one process running concurrently. Moreover it is not transaction safe. This means that it is not possible to group operations together so that they succeed or fail as one.

Think for example of using the dictionary to store a bank account. The key is the account number and the value is a list of transactions. We want the dictionary to be safely stored on file. We want it to be accessible by multiple processes/applications. We want transaction safety: it should not

be possible for an application to fail during a money transfer resulting in the disappearance of money.

Python provides a module called `shelve` with the same interface as `dict` which is stored on disk instead of RAM. One problem with this module is that the file is not locked when accessed. If two processes try to access it concurrently, the data becomes corrupted. This module also does not provide transactional safety.

The proper alternative consists of using a database. There are two types of databases: relational databases (which normally use SQL syntax) and non relational databases (often referred to as NoSQL). Key-value persistent storage databases usually follow under the latter category. Relational databases excel at storing structured data (in the form of tables), establishing relations between rows of those tables, and searches involving multiple tables linked by references. NoSQL databases excel at storing and retrieving schema-less data and replication of data (redundancy for fail safety).

Python comes with an embedded SQL database called SQLite. All data in the database are stored in one single file. It supports the SQL query language and transactional safety. It is very fast and allows concurrent read (from multiple processes) although not concurrent write (the file is locked when a process is writing to the file until the transaction is committed).

Installing and using any of these database systems is beyond the scope of this book and not necessary for our purposes. In particular we are not concerned with relations, data replications, and speed.

Here we describe a `PersistentDictionary` which works like a dictionary but stores the key:value in in a sqlite database and also supports the "*" wildcard.

This code now allows us to do the following:

- Create a persistent dictonary:

```
>>> from nlib import PersistentDictionary
>>> p = PersistentDictionary(path='storage.sqlite',autocommit=False)
```

- Store data in it

```
>>> p['some/key'] = 'some value'
```

  where "some/key" must be a string and "some value" can be any Python

pickleable object.

- Generate a UUID to be used as the key:

```
>>> key = p.uuid()
>>> p[key] = 'some other value'
```

- Retrieve the data

```
>>> data = p['some/key']
```

- Loop over keys

```
>>> for key in p: print key, p[key]
```

- List all keys

```
>>> keys = p.keys()
```

- List all keys matching a pattern

```
>>> keys = p.keys('some/*')
```

- List all key-value pairs matching a pattern

```
>>> for key,value in p.items('some/*'): print key, value
```

- Delete keys matching a pattern:

```
>>> del p['some/*']
```

We will now use our persistence storage to download 2011 financial data from the SP100 stocks. This will allow us to later perform various analysis tasks on these stocks:

Listing 1.2: in file: `nlib.py`

```
>>> SP100 = ['AA', 'AAPL', 'ABT', 'AEP', 'ALL', 'AMGN', 'AMZN', 'AVP', 'AXP', 'BA',
      'BAC',
... 'BAX', 'BHI', 'BK', 'BMY', 'BRK.B', 'CAT', 'C', 'CL', 'CMCSA', 'COF', 'COP', '
     COST',
... 'CPB', 'CSCO', 'CVS', 'CVX', 'DD', 'DELL', 'DIS', 'DOW', 'DVN', 'EMC', 'ETR', '
     EXC',
... 'F', 'FCX', 'FDX', 'GD', 'GE', 'GILD', 'GOOG', 'GS', 'HAL', 'HD', 'HNZ', 'HON',
      'HPQ',
... 'IBM', 'INTC', 'JNJ', 'JPM', 'KFT', 'KO', 'LMT', 'LOW', 'MA', 'MCD', 'MDT', '
     MET',
... 'MMM', 'MO', 'MON', 'MRK', 'MS', 'MSFT', 'NKE', 'NOV', 'NSC', 'NWSA', 'NYX', '
     ORCL',
... 'OXY', 'PEP', 'PFE', 'PG', 'PM', 'QCOM', 'RF', 'RTN', 'S', 'SLB', 'SLE', 'SO',
     'T',
... 'TGT', 'TWX', 'TXN', 'UNH', 'UPS', 'USB', 'UTX', 'VZ', 'WAG', 'WFC', 'WMB', '
     WMT',
... 'WY', 'XOM', 'XRX']
```

```
>>> from datetime import date
>>> storage = PersistentDictionary('sp100.sqlite')
>>> for symbol in SP100:
...     key = symbol+'/2011'
...     if not key in storage:
...         storage[key] = YStock(symbol).historical(start=date(2011,1,1),
...                                                  stop=date(2011,12,31))
```

Notice that while storing one item may be slower than storing an individual item in its own files, accessing the file system becomes progressivly slower as the number of files increases. Storing data in database, long term, is a winning strategy as it scales better, is easier to search for and extract data than multiple flat files. Which type of database is most appropriate depends on the type of data and the the type of queries we need to perform on the data.

1.6.9 matplotlib

`matplotlib` is the *de facto* standard plotting library for Python. It is one of the best and most versatile plotting libraries available. It has a two modes of operation. One mode of operation, called `pylab`, follows a matlab-like syntax. The other mode follows a more Python-style syntax. Here we use this latter.

In matplotlib we need to distinguish the following objects:

- `Figure`: a blank grid which can contain pairs of XY axes
- `Axes`: a pair of XY axes that may contain multiple superimposed plots
- `FigureCanvas`: a binary representation of a figure with everything that it contains.
- `plot`: a representation for a dataset like a line plot or a scatter plot

In matplotlib canvas can be visualized in a window or serialized into an image file. Here we take the latter approach and create two helper functions that take data and configuration parameters and output PNG images.

Using matplotlib can be complex so we use a helper object called `Canvas`:

As an example we can plot the adjusted closing price for AAPL:

Listing 1.3: in file: `nlib.py`

```
>>> from nlib import PersistentDictionary, Canvas
>>> storage = PersistentDictionary('sp100.sqlite')
>>> appl = storage['AAPL/2011']
>>> points = [(x,y['adjusted_close']) for (x,y) in enumerate(appl)]
>>> Canvas(title='Apple Stock (2011)',xlab='trading day',ylab='adjusted close').
    plot(points,legend='AAPL').save('images/aapl2011.png')
```

Figure 1.1: Example of line plot. Adjusted closing price for the APPL stock in 2011 (source Yahoo Finance)

Here is an example here of a histogram of daily arithmetic returns for the AAPL stock in 2011:

Listing 1.4: in file: `nlib.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> appl = storage['AAPL/2011'][1:] # skip 1st day
>>> points = [day['arithmetic_return'] for day in appl]
>>> Canvas(title='Apple Stock (2011)',xlab='arithmetic return', ylab='frequency').
    hist(points).save('images/aapl2011hist.png')
```

Here is a scatterplot for random data points:

Listing 1.5: in file: `nlib.py`

```
>>> from random import gauss
```

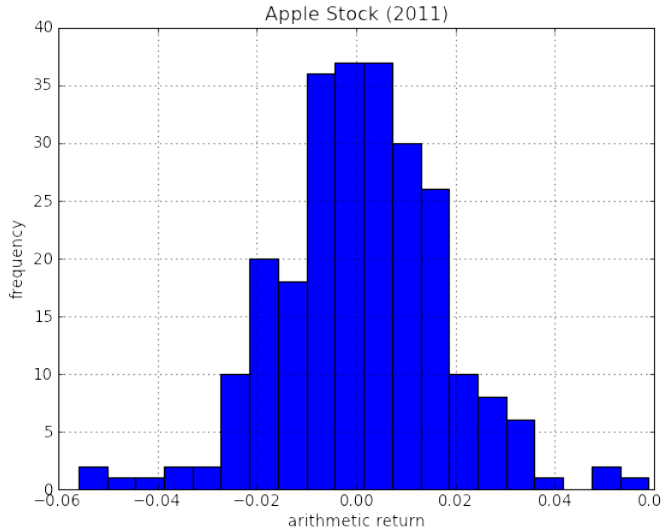

Figure 1.2: Example of histogram plot. Distribution of daily arithmetic returns for the APPL stock in 2011 (source Yahoo Finance)

```
>>> points = [(gauss(0,1),gauss(0,1),gauss(0,0.2),gauss(0,0.2)) for i in range(30)]
>>> Canvas(title='example scatter plot', xrange=(-2,2), yrange=(-2,2)).ellipses(
     points).save('images/scatter.png')
```

here is a scatter plot showing the return and variance of the S&P100 stocks:

Listing 1.6: in file: `nlib.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> points = []
>>> for key in storage.keys('*/2011'):
...     v = [day['log_return'] for day in storage[key][1:]]
...     ret = sum(v)/len(v)
...     var = sum(x**2 for x in v)/len(v) - ret**2
...     points.append((var*math.sqrt(len(v)),ret*len(v),0.0002,0.02))
>>> Canvas(title='S&P100 (2011)',xlab='risk',ylab='return',
...      xrange = (min(p[0] for p in points),max(p[0] for p in points)),
...      yrange = (min(p[1] for p in points),max(p[1] for p in points))
...      ).ellipses(points).save('images/sp100rr.png')
```

Notice the daily log returns have been multiplied by the number of days in one year to obtain the annual return. Similarly the daily volatility has multiplied by the square root of the number of days in one year to obtain the

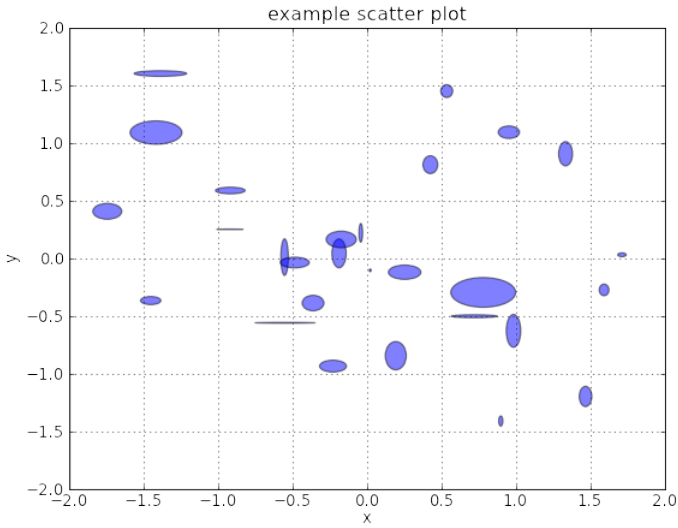

Figure 1.3: Example of scatter plot using some random points.

annual volatility (risk). The reason for this procedure will be explained in a later chapter.

Listing 1.7: in file: `nlib.py`

```
>>> def f(x,y): return (x-1)**2+(y-2)**2
>>> points = [[f(0.1*i-3,0.1*j-3) for i in range(61)] for j in range(61)]
>>> Canvas(title='example 2d function').imshow(points).save('images/color2d.png')
```

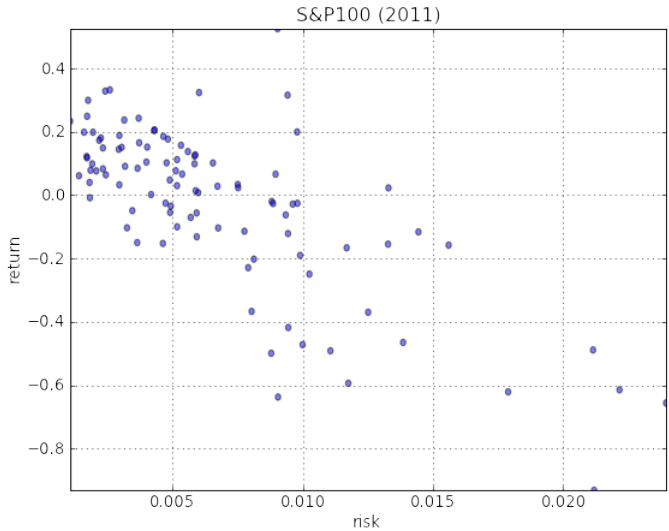

Figure 1.4: Example of scatter plot. Risk-return plot for the S&P100 stocks in 2011 (source Yahoo Finance)

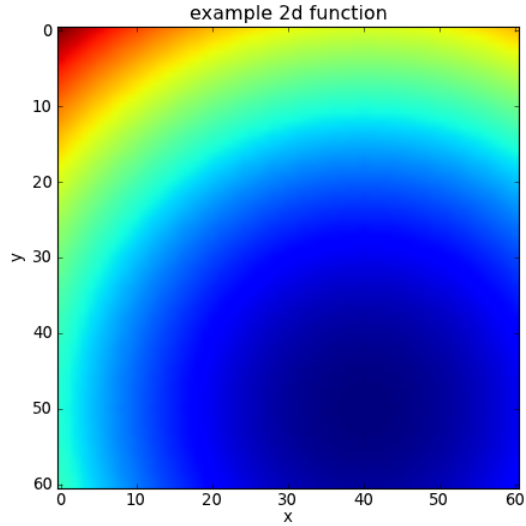

Figure 1.5: Example of 2d color plot using for $f(x, y) = (x-1)^2 + (y-2)^2$

2

Numerical and Financial Algorithms

2.1 Solving Systems of Linear Equations

Linear algebra is also fundamental for solving systems of linear equations such as the following:

$$x_0 + 2x_1 + 2x_2 = 3 \tag{2.1}$$
$$4x_0 + 4x_1 + 2x_2 = 6 \tag{2.2}$$
$$4x_0 + 6x_1 + 4x_2 = 10 \tag{2.3}$$

This can be rewritten using the equivalent linear algebra notation:

$$Ax = b \tag{2.4}$$

where

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{pmatrix} \quad \textit{and} \quad b = \begin{pmatrix} 3 \\ 6 \\ 10 \end{pmatrix} \tag{2.5}$$

The solution of the equaton can now be written as:

$$x = A^{-1}b \tag{2.6}$$

We can easily solve the system with our Python library:

Listing 2.1: in file: `nlib.py`

```
>>> A = Matrix([[1,2,2],[4,4,2],[4,6,4]])
>>> b = Matrix([[3],[6],[10]])
>>> x = (1/A)*b
>>> print x
[[-1.0], [3.0], [-1.0]]
```

Notice that b is a column vector and therefore

```
>>> b = Matrix([[3],[6],[10]])
```

but not

```
>>> b = Matrix([[3,6,10]]) # wrong
```

We can also obtain a column vector by performing a transposition of a row vector:

```
>>> b = Matrix([[3,6,10]]).t
```

2.2 Modern Portfolio Theory

Modern portfolio theory [25] is an investment approach that tries to maximize return given a fixed risk. Many different metrics have been proposed. One of them is the *Sharpe ratio*.

For a stock or a portfolio with an average return r and risk σ the Sharpe ratio is defined as

$$\text{Sharpe}(r, \sigma) \equiv (r - \bar{r})/\sigma \tag{2.7}$$

Here $\bar{r}$ is the current risk free investement rate. Usually the risk is measured as the standard deviation of its daily (or monthly or yearly) return.

Consider the stock price p_{it} of stock i at time t and its arithmetic daily return $r_{it} = (p_{i,t+1} - p_{it})/p_{it}$ given a risk free interest equal to $\bar{r}$.

For each stock we can compute the average return and average risk (variance of daily returns) and display it in a risk-return plot as we did in chapter 2.

We can try to building arbitrary portfolios by investing in multiple stocks at the same time. Modern portfolio theory states that there is a maximum Sharpe ratio we can achieve and there is only one portfolio that can achieve it. It is called the tangency portfolio.

A portfolio is identified by fractions of $1 invested in each stock in the portfolio. Our goal is to determine the tangent portfolio.

If we assume that daily returns for the stocks are Gaussian, then the solving algorithm is simple.

All we need is to compute the average return for each stock defined as:

$$r_i = 1/T \sum_t r_{it} \tag{2.8}$$

and the covarance matrix:

$$A_{ij} = \frac{1}{T} \sum_t (r_{it} - r_i)(r_{jt} - r_j) \tag{2.9}$$

Modern Portfolio Theory tells use that the tangent portfolio is given by:

$$\mathbf{x} = A^{-1}(\mathbf{r} - \bar{r}\mathbf{1}) \tag{2.10}$$

Here is the algorithm:

Listing 2.2: in file: `nlib.py`

```
def Markowitz(mu, A, r_free):
    """Assess Markowitz risk/return.
    Example:
    >>> cov = Matrix([[0.04, 0.006,0.02],
    ...                       [0.006,0.09, 0.06],
    ...                       [0.02, 0.06, 0.16]])
    >>> mu = Matrix([[0.10],[0.12],[0.15]])
    >>> r_free = 0.05
    >>> x, ret, risk = Markowitz(mu, cov, r_free)
    >>> print x
    [0.556634..., 0.275080..., 0.1682847...]
    >>> print ret, risk
    0.113915... 0.186747...
    """
    x = Matrix(A.rows, 1)
    x = (1/A)*(mu - r_free)
    x = x/sum(x[r,0] for r in range(x.rows))
    portfolio = [x[r,0] for r in range(x.rows)]
    portfolio_return = mu*x
    portfolio_risk = sqrt(x*(A*x))
    return portfolio, portfolio_return, portfolio_risk
```

Here is an example. We consider three assets (0,1,2) with the following covariance matrix,

```
>>> cov = Matrix([[0.04, 0.006,0.02],
...                        [0.006,0.09, 0.06],
...                        [0.02, 0.06, 0.16]])
```

and the following expected returns (arithmetic returns, not log returns, because the former are additive while the latter are not):

```
>>> mu = Matrix([[0.10],[0.12],[0.15]])
```

Given the following risk free interest rate:

```
>>> r_free = 0.05
```

We compute the tangent portfolio (highest Sharpe ratio), its return and its risk with one function call:

```
>>> x, ret, risk = Markowitz(mu, cov, r_free)
>>> print x
[0.5566343042071198, 0.27508090614886727, 0.16828478964401297]
>>> print ret, risk
0.113915857605 0.186747095412
>>> print (ret-r_free).risk
0.34225891152
>>> for r in range(3): print (mu[r,0]-r_free)/sqrt(cov[r,r])
0.25
0.233333333333
0.25
```

Investing 55% in asset 0, 27% in asset 1, and 16% in asset 2 the resulting portfolio has an expected return of 11.39% and a risk of 18.67% which corresponds to a Sharpe ration of 0.34, much higher than 0.25, 0.23, and 0.23 for the individual assets.

Notice that in general the tangent portfolio is not the one we want to invest in. In fact given a fixed amount of money to invest we can invest a faction α in the tangent portfolio and leave the remaining $1-\alpha$ fraction in the bank at the risk free rate $\bar{r}$. This choice will give an overall return equal to:

$$\alpha \mathbf{x} \cdot \mathbf{r} + (1-\alpha)\bar{r} \tag{2.11}$$

and an overall risk:

$$\sqrt{\alpha \mathbf{x}^t A \mathbf{x}} \tag{2.12}$$

which we can adjust to our subjective preferences by choosing α. Whatever

α we choose this strategy gives us the same Sharpe ratio as the tangent portfolio.

(2.13)

2.3 Linear Least Squares and χ^2

Consider a set of data points $(x_J, y_j) = (t_j, o_j \pm do_j)$. We want to fit them with a linear combination of linear independent functions f_i, so that

$$\begin{aligned} c_0 f_0(t_0) + c_1 f_1(t_0) + c_2 f_2(t_0) + ... &= e_0 \simeq o_0 && (2.14)\\ c_0 f_0(t_1) + c_1 f_1(t_1) + c_2 f_2(t_1) + ... &= e_1 \simeq o_1 && (2.15)\\ c_0 f_0(t_2) + c_1 f_1(t_2) + c_2 f_2(t_2) + ... &= e_2 \simeq o_2 && (2.16)\\ ... &= ... && (2.17) \end{aligned}$$

we want to find the $\{c_i\}$ which minimizes the sum of the squared distances between the actual "observed" data o_j and the predicted "expected" data e_j, in units of do_j. This metric is called χ^2 in general and *least squares* when the $do_j = 1$.

$$\chi^2 = \sum_j \left| \frac{e_j - o_j}{do_j} \right|^2 \tag{2.18}$$

If we define the matrix A and B as

$$A = \begin{pmatrix} \frac{f_0(t_0)}{do_0} & \frac{f_1(t_0)}{do_0} & \frac{f_2(t_0)}{do_0} & ... \\ \frac{f_0(t_1)}{do_1} & \frac{f_1(t_1)}{do_1} & \frac{f_2(t_1)}{do_1} & ... \\ \frac{f_0(t_2)}{do_2} & \frac{f_1(t_2)}{do_2} & \frac{f_2(t_2)}{do_2} & ... \\ ... & ... & ... & ... \end{pmatrix} \qquad b = \begin{pmatrix} \frac{o_0}{do_0} \\ \frac{o_1}{do_1} \\ \frac{o_2}{do_2} \\ ... \end{pmatrix} \tag{2.19}$$

Then the problem is reduced to

$$\chi^2 = \min_{\mathbf{c}} |A\mathbf{c} - \mathbf{b}|^2 \tag{2.20}$$
$$= \min_{\mathbf{c}} (A\mathbf{c} - \mathbf{b})^t (A\mathbf{c} - \mathbf{b}) \tag{2.21}$$
$$= \min_{\mathbf{c}} (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{b}^t A\mathbf{c} + \mathbf{b}^t\mathbf{b}) \tag{2.22}$$

This is the same as solving the following equation:

$$\nabla_c (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{c}^t A^t\mathbf{b} + \mathbf{b}^t\mathbf{b}) = 0 \tag{2.23}$$
$$A^t A\mathbf{c} - A^t\mathbf{b} = 0 \tag{2.24}$$

Its solution is:

$$\mathbf{c} = (A^t A)^{-1}(A^t\mathbf{b}) \tag{2.25}$$

The algorithm below implements a fitting function based on the above procedure. It takes as input a list of functions f_i and a list of points $p_j = (t_j, o_j, do_j)$ and returns three objects: a list with the c coefficients, the value of χ^2 for the best fit and the fitting function.

Listing 2.3: in file: `nlib.py`

```
def fit_least_squares(points, f):
    """
    Computes c_j for best linear fit of y[i] \pm dy[i] = fitting_f(x[i])
    where fitting_f(x[i]) is \sum_j c_j f[j](x[i])

    parameters:
    - a list of fitting functions
    - a list with points (x,y,dy)

    returns:
    - column vector with fitting coefficients
    - the chi2 for the fit
    - the fitting function as a lambda x: ....
    """
    def eval_fitting_function(f,c,x):
        if len(f)==1: return c*f[0](x)
        else: return sum(func(x)*c[i,0] for i,func in enumerate(f))
```

```
    A = Matrix(len(points),len(f))
    b = Matrix(len(points))
    for i in range(A.rows):
        weight = 1.0/points[i][2] if len(points[i])>2 else 1.0
        b[i,0] = weight*float(points[i][1])
        for j in range(A.cols):
            A[i,j] = weight*f[j](float(points[i][0]))
    c = (1.0/(A.t*A))*(A.t*b)
    chi = A*c-b
    chi2 = norm(chi,2)**2
    fitting_f = lambda x, c=c, f=f, q=eval_fitting_function: q(f,c,x)
    return c.values, chi2, fitting_f

# examples of fitting functions
def POLYNOMIAL(n):
    return [(lambda x, p=p: x**p) for p in range(n+1)]
CONSTANT  = POLYNOMIAL(0)
LINEAR    = POLYNOMIAL(1)
QUADRATIC = POLYNOMIAL(2)
CUBIC     = POLYNOMIAL(3)
QUARTIC   = POLYNOMIAL(4)
```

As an example, we can use it to perform a polynomial fit:

Given a set of points we want to the find the coefficients of a polynomial that best approximates those points.

In other words, we want to find the c_i such that, given t_j and o_j,

$$c_0 + c_1 t_0^1 + c_2 t_0^2 + ... = e_0 \simeq o_0 \pm do_0 \quad (2.26)$$

$$c_0 + c_1 t_1^1 + c_2 t_1^2 + ... = e_1 \simeq o_1 \pm do_1 \quad (2.27)$$

$$c_0 + c_1 t_2^1 + c_2 t_2^2 + ... = e_2 \simeq o_2 \pm do_2 \quad (2.28)$$

$$... \quad ... \quad (2.29)$$

$$(2.30)$$

Here is how we can generate some random points and solve the problem for a polynomial of degree 2 (or quadratic fit):

Listing 2.4: in file: `nlib.py`

```
>>> points = [(k,5+0.8*k+0.3*k*k+math.sin(k),2) for k in range(100)]
>>> a,chi2,fitting_f = fit_least_squares(points,QUADRATIC)
>>> for p in points[-10:]:
...     print p[0], round(p[1],2), round(fitting_f(p[0]),2)
90 2507.89 2506.98
```

```
91 2562.21 2562.08
92 2617.02 2617.78
93 2673.15 2674.08
94 2730.75 2730.98
95 2789.18 2788.48
96 2847.58 2846.58
97 2905.68 2905.28
98 2964.03 2964.58
99 3023.5 3024.48
>>> Canvas(title='polynomial fit',xlab='t',ylab='e(t),o(t)'
...       ).errorbar(points[:10],legend='o(t)'
...       ).plot([(p[0],fitting_f(p[0])) for p in points[:10]],legend='e(t)'
...       ).save('images/polynomialfit.png')
```

And here is a plot of the first 10 points compared with the best fit:

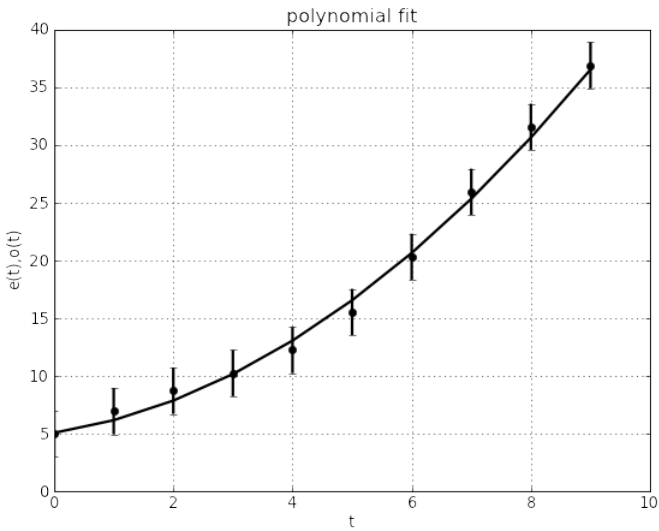

Figure 2.1: Random data with their error bars and the polynomial best fit.

2.4 Trading and technical analysis

In finance, *technical analysis* is an empirical discipline which consists of forecasting the direction of prices through the study of patterns in historical data (in particular price and volume). As an example we implement a simple

strategy that consists of the following steps:

- We fit the adjusted closing price for the previous seven days and use our fitting function to predict the adjusted close for the next day.
- If we have cash and predict the price will go up, we buy the stock.
- If we hold the stock and predict the price will go down, we sell the stock.

Listing 2.5: in file: `nlib.py`

```
class MyTrader(Trader):
    def model(self,window):
        ``the forecasting model''
        # we fit last few days quadratically
        points = [(x,y['adjusted_close']) for (x,y) in enumerate(window)]
        a,chi2,fitting_f = fit_least_squares(points,QUADRATIC)
        # and we extrapolate tomorrow's price
        tomorrow_prediction = fitting_f(len(points))
        return tomorrow_prediction

    def strategy(self, history, ndays=7):
        ``the trading strategy''
        if len(history)<ndays:
            return
        else:
            today_close = history[-1]['adjusted_close']
            tomorrow_prediction = self.model(history[-ndays:])
            return 'buy' if tomorrow_prediction>today_close else 'sell'
```

Now we back test the strategy using financial data for AAPL for the year 2011:

Listing 2.6: in file: `nlib.py`

```
>>> from datetime import date
>>> data = YStock('aapl').historical(
...         start=date(2011,1,1),stop=date(2011,12,31))
>>> print MyTrader().simulate(data,cash=1000.0)
1133...
>>> print 1000.0*math.exp(0.03)
1030...
>>> print 1000.0*data[-1]['adjusted_close']/data[0]['adjusted_close']
1228...
```

Our strategy did considerably better than the risk free return of 3%, but not as well as investing and holding AAPL shares over the same period.

Of course we can always engineer a strategy based on historical data that will outperform holding the stock, but *that past performance is never a guarantee of future performance*.

According to the definition from investopedia.com: "Technical analysts believe that the historical performance of stocks and markets are indications of future performance."

The efficacy of both technical and fundamental analysis is disputed by the efficient-market hypothesis which states that stock market prices are essentially unpredictable [15].

It is easy to extend the previous class to implement other strategies and backtest them.

3
The database abstraction layer

Extract from the web2py Complete Reference Manual

3.1 Dependencies

web2py comes with a Database Abstraction Layer (DAL), an API that maps Python objects into database objects such as queries, tables, and records. The DAL dynamically generates the SQL in real time using the specified dialect for the database back end, so that you do not have to write SQL code or learn different SQL dialects (the term SQL is used generically), and the application will be portable among different types of databases. A partial list of supported databases is show in the table below. Please check on the web2py web site and mailing list for more recent adapters. Google NoSQL is treated as a particular case in Chapter 13.

The Windows binary distribution works out of the box with SQLite and MySQL. The Mac binary distribution works out of the box with SQLite. To use any other database back-end, run from the source distribution and install the appropriate driver for the required back end.

Once the proper driver is installed, start web2py from source, and it will find the driver. Here is a list of drivers:

database	drivers (source)
SQLite	sqlite3 or pysqlite2 or zxJDBC [**?**] (on Jython)
PostgreSQL	psycopg2 [**?**] or pg8000 [**?**] or zxJDBC [**?**] (on Jython)
MySQL	pymysql [**?**] or MySQLdb [**?**]
Oracle	cx_Oracle [**?**]
MSSQL	pyodbc [**?**]
FireBird	kinterbasdb [**?**] or fdb or pyodbc
DB2	pyodbc [**?**]
Informix	informixdb [**?**]
Ingres	ingresdbi [**?**]
Cubrid	cubriddb [**?**] [**?**]
Sybase	Sybase [**?**]
Teradata	pyodbc [**?**]
SAPDB	sapdb [**?**]
MongoDB	pymongo [**?**]
IMAP	imaplib [**?**]

`sqlite3`, `pymysql`, `pg8000`, and `imaplib` ship with web2py. Support of MongoDB is experimental. The IMAP option allows to use DAL to access IMAP. web2py defines the following classes that make up the DAL:

The **DAL** object represents a database connection. For example:

```
from dal import DAL, Field
db = DAL('sqlite://storage.db')
```

Table represents a database table. You do not directly instantiate Table; instead, `DAL.define_table` instantiates it.

```
db.define_table('mytable', Field('myfield'))
```

The most important methods of a Table are: `.insert`, `.truncate`, `.drop`, and `.import_from_csv_file`.

Field represents a database field. It can be instantiated and passed as an argument to `DAL.define_table`.

DAL Rows is the object returned by a database select. It can be thought of as a list of `Row` rows:

```
rows = db(db.mytable.myfield!=None).select()
```

Row contains field values.

```
for row in rows:
    print row.myfield
```

Query is an object that represents a SQL "where" clause:

```
myquery = (db.mytable.myfield != None) | (db.mytable.myfield > 'A')
```

Set is an object that represents a set of records. Its most important methods are `count`, `select`, `update`, and `delete`. For example:

```
myset = db(myquery)
rows = myset.select()
myset.update(myfield='somevalue')
myset.delete()
```

Expression is something like an `orderby` or `groupby` expression. The Field class is derived from the Expression. Here is an example.

```
myorder = db.mytable.myfield.upper() | db.mytable.id
db().select(db.table.ALL, orderby=myorder)
```

3.2 Connection strings

A connection with the database is established by creating an instance of the DAL object:

```
>>> db = DAL('sqlite://storage.db', pool_size=0)
```

`db` is not a keyword; it is a local variable that stores the connection object `DAL`. You are free to give it a different name. The constructor of `DAL` requires a single argument, the connection string. The connection string is the only web2py code that depends on a specific back-end database. Here are examples of connection strings for specific types of supported back-end databases (in all cases, we assume the database is running from localhost on its default port and is named "test"):

SQLite	`sqlite://storage.db`
MySQL	`mysql://username:password@localhost/test`
PostgreSQL	`postgres://username:password@localhost/test`
MSSQL	`mssql://username:password@localhost/test`
FireBird	`firebird://username:password@localhost/test`
Oracle	`oracle://username/password@test`
DB2	`db2://username:password@test`
Ingres	`ingres://username:password@localhost/test`
Sybase	`sybase://username:password@localhost/test`
Informix	`informix://username:password@test`
Teradata	`teradata://DSN=dsn;UID=user;PWD=pass;DATABASE=name`
Cubrid	`cubrid://username:password@localhost/test`
SAPDB	`sapdb://username:password@localhost/test`
IMAP	`imap://user:password@server:port`
MongoDB	`mongodb://username:password@localhost/test`
Google/SQL	`google:sql`
Google/NoSQL	`google:datastore`

Notice that in SQLite the database consists of a single file. If it does not exist, it is created. This file is locked every time it is accessed. In the case of MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Ingres and Informix the database "test" must be created outside web2py. Once the connection is established, web2py will create, alter, and drop tables appropriately.

It is also possible to set the connection string to `None`. In this case DAL will not connect to any back-end database, but the API can still be accessed for testing. Examples of this will be discussed in Chapter 7.

Some times you may need to generate SQL as if you had a connection but without actually connecting to the database. This can be done with

```
db = DAL('...', do_connect=False)
```

In this case you will be able to call `_select`, `_insert`, `_update`, and `_delete` to generate SQL but not call `select`, `insert`, `update`, and `delete`. In most of the cases you can use `do_connect=False` even without having the required database drivers.

Notice that by default web2py uses utf8 character encoding for databases. If you work with existing databases that behave differently, you have to change it with the optional parameter `db_codec` like

```
db = DAL('...', db_codec='latin1')
```

otherwise you'll get UnicodeDecodeErrors tickets.

3.2.1 Connection pooling

The second argument of the DAL constructor is the `pool_size`; it defaults to zero.

As it is rather slow to establish a new database connection for each request, web2py implements a mechanism for connection pooling. Once a connection is established and the page has been served and the transaction completed, the connection is not closed but goes into a pool. When the next http request arrives, web2py tries to recycle a connection from the pool and use that for the new transaction. If there are no available connections in the pool, a new connection is established.

When web2py starts, the pool is always empty. The pool grows up to the minimum between the value of `pool_size` and the max number of concurrent requests. This means that if `pool_size=10` but our server never receives more than 5 concurrent requests, then the actual pool size will only grow to 5. If `pool_size=0` then connection pooling is not used.

Connections in the pools are shared sequentially among threads, in the sense that they may be used by two different but not simultaneous threads. There is only one pool for each web2py process.

The `pool_size` parameter is ignored by SQLite and Google App Engine. Connection pooling is ignored for SQLite, since it would not yield any benefit.

3.2.2 Connection failures

If web2py fails to connect to the database it waits 1 seconds and tries again up to 5 times before declaring a failure. In case of connection pooling it is possible that a pooled connection that stays open but unused for some time is closed by the database end. Thanks to the retry feature web2py tries to re-establish these dropped connections.

3.2.3 Replicated databases

The first argument of `DAL(...)` can be a list of URIs. In this case web2py tries to connect to each of them. The main purpose for this is to deal with multiple database servers and distribute the workload among them). Here is a typical use case:

```
db = DAL(['mysql://...1','mysql://...2','mysql://...3'])
```

In this case the DAL tries to connect to the first and, on failure, it will try the second and the third. This can also be used to distribute load in a database master-slave configuration. We will talk more about this in Chapter 13 in the context of scalability.

3.3 Reserved keywords

`check_reserved` is yet another argument that can be passed to the DAL constructor. It tells it to check table names and column names against reserved SQL keywords in target back-end databases.

This argument is `check_reserved` and it defaults to None.

This is a list of strings that contain the database back-end adapter names.

The adapter name is the same as used in the DAL connection string. So if you want to check against PostgreSQL and MSSQL then your connection string would look as follows:

```
db = DAL('sqlite://storage.db',
         check_reserved=['postgres', 'mssql'])
```

The DAL will scan the keywords in the same order as of the list.

There are two extra options "all" and "common". If you specify all, it will check against all known SQL keywords. If you specify common, it will only check against common SQL keywords such as `SELECT`, `INSERT`, `UPDATE`, etc.

For supported back-ends you may also specify if you would like to check against the non-reserved SQL keywords as well. In this case you would append `_nonreserved` to the name. For example:

```
check_reserved=['postgres', 'postgres_nonreserved']
```

The following database backends support reserved words checking.

PostgreSQL	`postgres(_nonreserved)`
MySQL	`mysql`
FireBird	`firebird(_nonreserved)`
MSSQL	`mssql`
Oracle	`oracle`

3.4 `DAL`, `Table`, `Field`

You can experiment with the DAL API using the web2py shell.

Start by creating a connection. For the sake of example, you can use SQLite. Nothing in this discussion changes when you change the back-end engine.

```
>>> db = DAL('sqlite://storage.db')
```

The database is now connected and the connection is stored in the global variable `db`.

At any time you can retrieve the connection string.

```
>>> print db._uri
sqlite://storage.db
```

and the database name

```
>>> print db._dbname
sqlite
```

The connection string is called a `_uri` because it is an instance of a Uniform Resource Identifier.

The DAL allows multiple connections with the same database or with different databases, even databases of different types. For now, we will assume the presence of a single database since this is the most common situation.

The most important method of a DAL is `define_table`:

```
>>> db.define_table('person', Field('name'))
```

It defines, stores and returns a `Table` object called "person" containing a field (column) "name". This object can also be accessed via `db.person`, so you do not need to catch the return value.

Do not declare a field called "id", because one is created by web2py anyway. Every table has a field called "id" by default. It is an auto-increment integer field (starting at 1) used for cross-reference and for making every record unique, so "id" is a primary key. (Note: the id's starting at 1 is back-end specific. For example, this does not apply to the Google App Engine NoSQL.)

Optionally you can define a field of `type='id'` and web2py will use this field as auto-increment id field. This is not recommended except when accessing legacy database tables. With some limitation, you can also use different primary keys and this is discussed in the section on "Legacy databases and keyed tables".

Tables can be defined only once but you can force web2py to redefine an existing table:

```
db.define_table('person', Field('name'))
db.define_table('person', Field('name'), redefine=True)
```

The redefinition may trigger a migration if field content is different.

> Because usually in web2py models are executed before controllers, it is possible that some table are defined even if not needed. It is therefore necessary to speed up the code by making table definitions lazy. This is done by setting the `DAL(...,lazy_tables=True)` attributes. Tables will be actually created only when accessed.

3.5 Record representation

It is optional but recommended to specify a format representation for records:

```
>>> db.define_table('person', Field('name'), format='%(name)s')
```

or

```
>>> db.define_table('person', Field('name'), format='%(name)s %(id)s')
```

or even more complex ones using a function:

```
>>> db.define_table('person', Field('name'),
       format=lambda r: r.name or 'anonymous')
```

The format attribute will be used for two purposes:

- To represent referenced records in select/option drop-downs.
- To set the `db.othertable.person.represent` attribute for all fields referencing this table. This means that SQLTABLE will not show references by id but will use the format preferred representation instead.

These are the default values of a Field constructor:

```
Field(name, 'string', length=None, default=None,
      required=False, requires='<default>',
      ondelete='CASCADE', notnull=False, unique=False,
      ...)
```

Not all of them are relevant for every field. "length" is relevant only for fields of type "string". "uploadfield" and "authorize" are relevant only for fields of type "upload". "ondelete" is relevant only for fields of type "reference" and "upload".

- `length` sets the maximum length of a "string", "password" or "upload" field. If `length` is not specified a default value is used but the default value is not guaranteed to be backward compatible. *To avoid unwanted migrations on upgrades, we recommend that you always specify the length for string, password and upload fields.*
- `default` sets the default value for the field. The default value is used when performing an insert if a value is not explicitly specified. It is also used to pre-populate forms built from the table using SQLFORM. Note, rather than being a fixed value, the default can instead be a function (including a lambda function) that returns a value of the appropriate type for the field.

In that case, the function is called once for each record inserted, even when multiple records are inserted in a single transaction.

- `required` tells the DAL that no insert should be allowed on this table if a value for this field is not explicitly specified.
- `requires` is a validator or a list of validators. This is not used by the DAL, but it is used by SQLFORM. The default validators for the given types are shown in the following table:

field type	default field validators
`string`	`IS_LENGTH(length)` default length is 512
`text`	`IS_LENGTH(65536)`
`blob`	`None`
`boolean`	`None`
`integer`	`IS_INT_IN_RANGE(-1e100, 1e100)`
`double`	`IS_FLOAT_IN_RANGE(-1e100, 1e100)`
`decimal(n,m)`	`IS_DECIMAL_IN_RANGE(-1e100, 1e100)`
`date`	`IS_DATE()`
`time`	`IS_TIME()`
`datetime`	`IS_DATETIME()`
`password`	`None`
`upload`	`None`
`reference <table>`	`IS_IN_DB(db,table.field,format)`
`list:string`	`None`
`list:integer`	`None`
`list:reference <table>`	`IS_IN_DB(db,table.field,format,multiple=True)`
`json`	`IS_JSON()`
`bigint`	`None`
`big-id`	`None`
`big-reference`	`None`

Decimal requires and returns values as `Decimal` objects, as defined in the Python `decimal` module. SQLite does not handle the `decimal` type so internally we treat it as a `double`. The (n,m) are the number of digits in total and the number of digits after the decimal point respectively.

The `big-id` and, `big-reference` are only supported by some of the database engines and are experimental. They are not normally used as field types unless for legacy tables, however, the DAL constructor has a `bigint_id` argument that when set to `True` makes the `id` fields and `reference` fields `big-id` and `big-reference` respectively.

The `list:<type>` fields are special because they are designed to take advantage of certain denormalization features on NoSQL (in the case of Google App Engine NoSQL, the field types `ListProperty` and `StringListProperty`) and back-port them all the other supported relational databases. On relational databases lists are stored as a `text` field. The items are separated by a `|` and each `|` in string item is escaped as a `||`. They are discussed in their own section.

The `json` field type is pretty much explanatory. It can store any json serializable object. It is designed to work specifically for MongoDB and backported to the other database adapters for portability.

> Notice that `requires=...` is enforced at the level of forms, `required=True` is enforced at the level of the DAL (insert), while `notnull`, `unique` and `ondelete` are enforced at the level of the database. While they sometimes may seem redundant, it is important to maintain the distinction when programming with the DAL.

- `ondelete` translates into the "ON DELETE" SQL statement. By default it is set to "CASCADE". This tells the database that when it deletes a record, it should also delete all records that refer to it. To disable this feature, set `ondelete` to "NO ACTION" or "SET NULL".
- `notnull=True` translates into the "NOT NULL" SQL statement. It prevents the database from inserting null values for the field.
- `unique=True` translates into the "UNIQUE" SQL statement and it makes sure that values of this field are unique within the table. It is enforced at the database level.

Most attributes of fields and tables can be modified after they are defined:

```
db.define_table('person',Field('name',default=''),format='%(name)s')
db.person._format = '%(name)s/%(id)s'
db.person.name.default = 'anonymous'
```

(notice that attributes of tables are usually prefixed by an underscore to avoid conflict with possible field names).

You can list the tables that have been defined for a given database connection:

```
>>> print db.tables
['person']
```

You can also list the fields that have been defined for a given table:

```
>>> print db.person.fields
['id', 'name']
```

You can query for the type of a table:

```
>>> print type(db.person)
<class 'gluon.sql.Table'>
```

and you can access a table from the DAL connection using:

```
>>> print type(db['person'])
<class 'gluon.sql.Table'>
```

Similarly you can access fields from their name in multiple equivalent ways:

```
>>> print type(db.person.name)
<class 'gluon.sql.Field'>
>>> print type(db.person['name'])
<class 'gluon.sql.Field'>
>>> print type(db['person']['name'])
<class 'gluon.sql.Field'>
```

Given a field, you can access the attributes set in its definition:

```
>>> print db.person.name.type
string
>>> print db.person.name.unique
False
>>> print db.person.name.notnull
False
>>> print db.person.name.length
32
```

including its parent table, tablename, and parent connection:

```
>>> db.person.name._table == db.person
True
>>> db.person.name._tablename == 'person'
True
>>> db.person.name._db == db
True
```

A field also has methods. Some of them are used to build queries and we will see them later. A special method of the field object is `validate` and it

calls the validators for the field.

```
print db.person.name.validate('John')
```

which returns a tuple `(value, error)`. `error` is `None` if the input passes validation.

3.6 Migrations

`define_table` checks whether or not the corresponding table exists. If it does not, it generates the SQL to create it and executes the SQL. If the table does exist but differs from the one being defined, it generates the SQL to alter the table and executes it. If a field has changed type but not name, it will try to convert the data (If you do not want this, you need to redefine the table twice, the first time, letting web2py drop the field by removing it, and the second time adding the newly defined field so that web2py can create it.). If the table exists and matches the current definition, it will leave it alone. In all cases it will create the `db.person` object that represents the table.

We refer to this behavior as a "migration". web2py logs all migrations and migration attempts in the file "databases/sql.log".

The first argument of `define_table` is always the table name. The other unnamed arguments are the fields (Field). The function also takes an optional last argument called "migrate" which must be referred to explicitly by name as in:

```
>>> db.define_table('person', Field('name'), migrate='person.table')
```

The value of migrate is the filename (in the "databases" folder for the application) where web2py stores internal migration information for this table. These files are very important and should never be removed while the corresponding tables exist. In cases where a table has been dropped and the corresponding file still exist, it can be removed manually. By default, migrate is set to True. This causes web2py to generate the filename from a hash of the connection string. If migrate is set to False, the migration is not performed, and web2py assumes that the table exists in the datastore and it contains (at least) the fields listed in `define_table`. The best practice is to give an explicit name to the migrate table.

There may not be two tables in the same application with the same migrate filename.

The DAL class also takes a "migrate" argument, which determines the default value of migrate for calls to `define_table`. For example,

```
>>> db = DAL('sqlite://storage.db', migrate=False)
```

will set the default value of migrate to False whenever `db.define_table` is called without a migrate argument.

> Notice that web2py only migrates new columns, removed columns, and changes in column type (except in sqlite). web2py does not migrate changes in attributes such as changes in the values of `default`, `unique`, `notnull`, and `ondelete`.

Migrations can be disabled for all tables at once:

```
db = DAL(...,migrate_enabled=False)
```

This is the recommended behavior when two apps share the same database. Only one of the two apps should perform migrations, the other should disabled them.

3.7 `insert`

Given a table, you can insert records

```
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
```

Insert returns the unique "id" value of each record inserted.

You can truncate the table, i.e., delete all records and reset the counter of the id.

```
>>> db.person.truncate()
```

Now, if you insert a record again, the counter starts again at 1 (this is back-end specific and does not apply to Google NoSQL):

```
>>> db.person.insert(name="Alex")
1
```

Notice you can pass parameters to `truncate`, for example you can tell SQLITE to restart the id counter.

```
db.person.truncate('RESTART IDENTITY CASCADE')
```

The argument is in raw SQL and therefore engine specific.

web2py also provides a bulk_insert method

```
>>> db.person.bulk_insert([{'name':'Alex'}, {'name':'John'}, {'name':'Tim'}])
[3,4,5]
```

It takes a list of dictionaries of fields to be inserted and performs multiple inserts at once. It returns the IDs of the inserted records. On the supported relational databases there is no advantage in using this function as opposed to looping and performing individual inserts but on Google App Engine NoSQL, there is a major speed advantage.

3.8 `commit` and `rollback`

No create, drop, insert, truncate, delete, or update operation is actually committed until you issue the commit command

```
>>> db.commit()
```

To check it let's insert a new record:

```
>>> db.person.insert(name="Bob")
2
```

and roll back, i.e., ignore all operations since the last commit:

```
>>> db.rollback()
```

If you now insert again, the counter will again be set to 2, since the previous insert was rolled back.

```
>>> db.person.insert(name="Bob")
2
```

Code in models, views and controllers is enclosed in web2py code that looks like this:

```
try:
    execute models, controller function and view
except:
    rollback all connections
    log the traceback
```

```
    send a ticket to the visitor
else:
    commit all connections
    save cookies, sessions and return the page
```

There is no need to ever call `commit` or `rollback` explicitly in web2py unless one needs more granular control.

3.9 `Query`, `Set`, `Rows`

Let's consider again the table defined (and dropped) previously and insert three records:

```
>>> db.define_table('person', Field('name'))
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
>>> db.person.insert(name="Carl")
3
```

You can store the table in a variable. For example, with variable `person`, you could do:

```
>>> person = db.person
```

You can also store a field in a variable such as `name`. For example, you could also do:

```
>>> name = person.name
```

You can even build a query (using operators like ==, !=, <, >, <=, >=, like, belongs) and store the query in a variable `q` such as in:

```
>>> q = name=='Alex'
```

When you call `db` with a query, you define a set of records. You can store it in a variable `s` and write:

```
>>> s = db(q)
```

Notice that no database query has been performed so far. DAL + Query simply define a set of records in this db that match the query. web2py determines from the query which table (or tables) are involved and, in fact, there is no need to specify that.

3.10 select

Given a Set, s, you can fetch the records with the command select:

```
>>> rows = s.select()
```

It returns an iterable object of class gluon.sql.Rows whose elements are Row objects. gluon.sql.Row objects act like dictionaries, but their elements can also be accessed as attributes, like gluon.storage.Storage.The former differ from the latter because its values are read-only.

The Rows object allows looping over the result of the select and printing the selected field values for each row:

```
>>> for row in rows:
        print row.id, row.name
1 Alex
```

You can do all the steps in one statement:

```
>>> for row in db(db.person.name=='Alex').select():
        print row.name
Alex
```

The select command can take arguments. All unnamed arguments are interpreted as the names of the fields that you want to fetch. For example, you can be explicit on fetching field "id" and field "name":

```
>>> for row in db().select(db.person.id, db.person.name):
        print row.name
Alex
Bob
Carl
```

The table attribute ALL allows you to specify all fields:

```
>>> for row in db().select(db.person.ALL):
        print row.name
Alex
Bob
Carl
```

Notice that there is no query string passed to db. web2py understands that if you want all fields of the table person without additional information then you want all records of the table person.

An equivalent alternative syntax is the following:

```
>>> for row in db(db.person.id > 0).select():
```

```
        print row.name
Alex
Bob
Carl
```

and web2py understands that if you ask for all records of the table person (id > 0) without additional information, then you want all the fields of table person.

Given one row

```
row = rows[0]
```

you can extract its values using multiple equivalent expressions:

```
>>> row.name
Alex
>>> row['name']
Alex
>>> row('person.name')
Alex
```

The latter syntax is particularly handy when selecting en expression instead of a column. We will show this later.

You can also do

```
rows.compact = False
```

to disable the notation

```
row[i].name
```

and enable, instead, the less compact notation:

```
row[i].person.name
```

Yes this is unusual and rarely needed.

3.10.1 Shortcuts

The DAL supports various code-simplifying shortcuts. In particular:

```
myrecord = db.mytable[id]
```

returns the record with the given `id` if it exists. If the `id` does not exist, it returns `None`. The above statement is equivalent to

```
myrecord = db(db.mytable.id==id).select().first()
```

You can delete records by id:

```
del db.mytable[id]
```

and this is equivalent to

```
db(db.mytable.id==id).delete()
```

and deletes the record with the given `id`, if it exists.

You can insert records:

```
db.mytable[0] = dict(myfield='somevalue')
```

It is equivalent to

```
db.mytable.insert(myfield='somevalue')
```

and it creates a new record with field values specified by the dictionary on the right hand side.

You can update records:

```
db.mytable[id] = dict(myfield='somevalue')
```

which is equivalent to

```
db(db.mytable.id==id).update(myfield='somevalue')
```

and it updates an existing record with field values specified by the dictionary on the right hand side.

3.10.2 Fetching a `Row`

Yet another convenient syntax is the following:

```
record = db.mytable(id)
record = db.mytable(db.mytable.id==id)
record = db.mytable(id,myfield='somevalue')
```

Apparently similar to `db.mytable[id]` the above syntax is more flexible and safer. First of all it checks whether `id` is an int (or `str(id)` is an int) and returns `None` if not (it never raises an exception). It also allows to specify multiple conditions that the record must meet. If they are not met, it also returns `None`.

3.10.3 Recursive `select`s

Consider the previous table person and a new table "thing" referencing a "person":

```
>>> db.define_table('thing',
        Field('name'),
        Field('owner_id','reference person'))
```

and a simple select from this table:

```
>>> things = db(db.thing).select()
```

which is equivalent to

```
>>> things = db(db.thing._id>0).select()
```

where `._id` is a reference to the primary key of the table. Normally `db.thing._id` is the same as `db.thing.id` and we will assume that in most of this book.

For each Row of things it is possible to fetch not just fields from the selected table (thing) but also from linked tables (recursively):

```
>>> for thing in things: print thing.name, thing.owner_id.name
```

Here `thing.owner_id.name` requires one database select for each thing in things and it is therefore inefficient. We suggest using joins whenever possible instead of recursive selects, nevertheless this is convenient and practical when accessing individual records.

You can also do it backwards, by selecting the things referenced by a person:

```
person =  db.person(id)
for thing in person.thing.select(orderby=db.thing.name):
    print person.name, 'owns', thing.name
```

In this last expressions `person.thing` is a shortcut for

```
db(db.thing.owner_id==person.id)
```

i.e. the Set of `thing`s referenced by the current `person`. This syntax breaks down if the referencing table has multiple references to the referenced table. In this case one needs to be more explicit and use a full Query.

3.10.4 Serializing `Rows` in views

Given the following action containing a query

```
def index()
    return dict(rows = db(query).select())
```

The result of a select can be displayed in a view with the following syntax:

```
{{extend 'layout.html'}}
<h1>Records</h1>
{{=rows}}
```

Which is equivalent to:

```
{{extend 'layout.html'}}
<h1>Records</h1>
{{=SQLTABLE(rows)}}
```

`SQLTABLE` converts the rows into an HTML table with a header containing the column names and one row per record. The rows are marked as alternating class "even" and class "odd". Under the hood, Rows is first converted into a SQLTABLE object (not to be confused with Table) and then serialized. The values extracted from the database are also formatted by the validators associated to the field and then escaped.

Yet it is possible and sometimes convenient to call SQLTABLE explicitly.

The SQLTABLE constructor takes the following optional arguments:

- `linkto` the URL or an action to be used to link reference fields (default to None)
- `upload` the URL or the download action to allow downloading of uploaded files (default to None)
- `headers` a dictionary mapping field names to their labels to be used as headers (default to {}). It can also be an instruction. Currently we support `headers='fieldname:capitalize'`.
- `truncate` the number of characters for truncating long values in the table (default is 16)
- `columns` the list of fieldnames to be shown as columns (in tablename.fieldname format).

Those not listed are not displayed (defaults to all).

- `**attributes` generic helper attributes to be passed to the most external TABLE object.

Here is an example:

```
{{extend 'layout.html'}}
<h1>Records</h1>
{{=SQLTABLE(rows,
     headers='fieldname:capitalize',
     truncate=100,
     upload=URL('download'))
}}
```

> `SQLTABLE` is useful but there are times when one needs more. `SQLFORM.grid` is an extension of SQLTABLE that creates a table with search features and pagination, as well as ability to open detailed records, create, edit and delete records. `SQLFORM.smartgrid` is a further generalization that allows all of the above but also creates buttons to access referencing records.

Here is an example of usage of `SQLFORM.grid`:

```
def index():
    return dict(grid=SQLFORM.grid(query))
```

and the corresponding view:

```
{{extend 'layout.html'}}
{{=grid}}
```

`SQLFORM.grid` and `SQLFORM.smartgrid` should be preferred to `SQLTABLE` because they are more powerful although higher level and therefore more constraining. They will be explained in more detail in chapter 8.

3.10.5 `orderby`, `groupby`, `limitby`, `distinct`, `having`

The `select` command takes five optional arguments: orderby, groupby, limitby, left and cache. Here we discuss the first three.

You can fetch the records sorted by name:

```
>>> for row in db().select(
        db.person.ALL, orderby=db.person.name):
        print row.name
Alex
Bob
Carl
```

You can fetch the records sorted by name in reverse order (notice the tilde):

```
>>> for row in db().select(
        db.person.ALL, orderby=~db.person.name):
        print row.name
Carl
Bob
Alex
```

You can have the fetched records appear in random order:

```
>>> for row in db().select(
        db.person.ALL, orderby='<random>'):
        print row.name
Carl
Alex
Bob
```

The use of `orderby='<random>'` is not supported on Google NoSQL. However, in this situation and likewise in many others where built-ins are insufficient, imports can be used:

```
import random
rows=db(...).select().sort(lambda row: random.random())
```

You can sort the records according to multiple fields by concatenating them with a "|":

```
>>> for row in db().select(
        db.person.ALL, orderby=db.person.name|db.person.id):
        print row.name
Carl
Bob
Alex
```

Using `groupby` together with `orderby`, you can group records with the same value for the specified field (this is back-end specific, and is not on the Google NoSQL):

```
>>> for row in db().select(
        db.person.ALL,
        orderby=db.person.name, groupby=db.person.name):
        print row.name
Alex
Bob
Carl
```

You can use `having` in conjunction with `groupby` to group conditionally (only those `having` the condition are grouped.

```
>>> print db(query1).select(db.person.ALL, groupby=db.person.name, having=query2)
```

Notice that query1 filters records to be displayed, query2 filters records to be

grouped.

With the argument `distinct=True`, you can specify that you only want to select distinct records. This has the same effect as grouping using all specified fields except that it does not require sorting. When using distinct it is important not to select ALL fields, and in particular not to select the "id" field, else all records will always be distinct.

Here is an example:

```
>>> for row in db().select(db.person.name, distinct=True):
        print row.name
Alex
Bob
Carl
```

Notice that `distinct` can also be an expression for example:

```
>>> for row in db().select(db.person.name,distinct=db.person.name):
        print row.name
Alex
Bob
Carl
```

With limitby=(min, max), you can select a subset of the records from offset=min to but not including offset=max (in this case, the first two starting at zero):

```
>>> for row in db().select(db.person.ALL, limitby=(0, 2)):
        print row.name
Alex
Bob
```

3.10.6 Logical operators

Queries can be combined using the binary AND operator "&":

```
>>> rows = db((db.person.name=='Alex') & (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
4 Alex
```

and the binary OR operator "|":

```
>>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
1 Alex
```

You can negate a query (or sub-query) with the "`!=`" binary operator:

```
>>> rows = db((db.person.name!='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

or by explicit negation with the " ~ " unary operator:

```
>>> rows = db(~(db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

> Due to Python restrictions in overloading "`and`" and "`or`" operators, these cannot be used in forming queries. The binary operators "`&`" and "`|`" must be used instead. Note that these operators (unlike "`and`" and "`or`") have higher precedence than comparison operators, so the "extra" parentheses in the above examples are mandatory. Similarly, the unary operator " ~ " has higher precedence than comparison operators, so ~-negated comparisons must also be parenthesized.

It is also possible to build queries using in-place logical operators:

```
>>> query = db.person.name!='Alex'
>>> query &= db.person.id>3
>>> query |= db.person.name=='John'
```

3.10.7 `count`, `isempty`, `delete`, `update`

You can count records in a set:

```
>>> print db(db.person.id > 0).count()
3
```

Notice that `count` takes an optional `distinct` argument which defaults to False, and it works very much like the same argument for `select`. `count` has also a `cache` argument that works very much like the equivalent argument of the `select` method.

Sometimes you may need to check if a table is empty. A more efficient way than counting is using the `isempty` method:

```
>>> print db(db.person.id > 0).isempty()
False
```

or equivalently:

```
>>> print db(db.person).isempty()
False
```

You can delete records in a set:

```
>>> db(db.person.id > 3).delete()
```

And you can update all records in a set by passing named arguments corresponding to the fields that need to be updated:

```
>>> db(db.person.id > 3).update(name='Ken')
```

3.10.8 Expressions

The value assigned an update statement can be an expression. For example consider this model

```
>>> db.define_table('person',
        Field('name'),
        Field('visits', 'integer', default=0))
>>> db(db.person.name == 'Massimo').update(
        visits = db.person.visits + 1)
```

The values used in queries can also be expressions

```
>>> db.define_table('person',
        Field('name'),
        Field('visits', 'integer', default=0),
        Field('clicks', 'integer', default=0))
>>> db(db.person.visits == db.person.clicks + 1).delete()
```

3.10.9 `case`

An expression can contain a case clause for example:

```
>>> db.define_table('person',Field('name'))
>>> condition = db.person.name.startswith('M')
>>> yes_or_no = condition.case('Yes','No')
>>> for row in db().select(db.person.name, yes_or_no):
...     print row.person.name,  row(yes_or_no)
Max Yes
John No
```

3.10.10 update_record

web2py also allows updating a single record that is already in memory using `update_record`

```
>>> row = db(db.person.id==2).select().first()
>>> row.update_record(name='Curt')
```

`update_record` should not be confused with

```
>>> row.update(name='Curt')
```

because for a single row, the method `update` updates the row object but not the database record, as in the case of `update_record`.

It is also possible to change the attributes of a row (one at a time) and then call `update_record()` without arguments to save the changes:

```
>>> row = db(db.person.id > 2).select().first()
>>> row.name = 'Curt'
>>> row.update_record() # saves above change
```

The `update_record` method is available only if the table's `id` field is included in the select, and `cacheable` is not set to `True`.

3.10.11 Inserting and updating from a dictionary

A common issue consists of needing to insert or update records in a table where the name of the table, the field to be updated, and the value for the field are all stored in variables. For example: `tablename`, `fieldname`, and `value`.

The insert can be done using the following syntax:

```
db[tablename].insert(**{fieldname:value})
```

The update of record with given id can be done with:

```
db(db[tablename]._id==id).update(**{fieldname:value})
```

Notice we used `table._id` instead of `table.id`. In this way the query works even for tables with a field of type "id" which has a name other than "id".

3.10.12 first and last

Given a Rows object containing records:

```
>>> rows = db(query).select()
>>> first_row = rows.first()
>>> last_row = rows.last()
```

are equivalent to

```
>>> first_row = rows[0] if len(rows)>0 else None
>>> last_row = rows[-1] if len(rows)>0 else None
```

3.10.13 `as_dict` and `as_list`

A Row object can be serialized into a regular dictionary using the `as_dict()` method and a Rows object can be serialized into a list of dictionaries using the `as_list()` method. Here are some examples:

```
>>> rows = db(query).select()
>>> rows_list = rows.as_list()
>>> first_row_dict = rows.first().as_dict()
```

These methods are convenient for passing Rows to generic views and or to store Rows in sessions (since Rows objects themselves cannot be serialized since contain a reference to an open DB connection):

```
>>> rows = db(query).select()
>>> session.rows = rows # not allowed!
>>> session.rows = rows.as_list() # allowed!
```

3.10.14 Combining rows

Row objects can be combined at the Python level. Here we assume:

```
>>> print rows1
person.name
Max
Tim
>>> print rows2
person.name
John
Tim
```

You can do a union of the records in two set of rows:

```
>>> rows3 = rows1 & rows2
>>> print rows3
name
Max
```

```
Tim
John
Tim
```

You can do a union of the records removing duplicates:

```
>>> rows3 = rows1 | rows2
>>> print rows3
name
Max
Tim
John
```

3.10.15 `find`, `exclude`, `sort`

Some times you to perform two selects and one contains a subset of a previous select. In this case it is pointless to access the database again. The `find`, `exclude` and `sort` objects allow you to manipulate a Rows objects and generate another one without accessing the database. More specifically:

- `find` returns a new set of Rows filtered by a condition and leaves the original unchanged.
- `exclude` returns a new set of Rows filtered by a condition and removes them from the original Rows.
- `sort` returns a new set of Rows sorted by a condition and leaves the original unchanged.

All these methods take a single argument, a function that acts on each individual row.

Here is an example of usage:

```
>>> db.define_table('person',Field('name'))
>>> db.person.insert(name='John')
>>> db.person.insert(name='Max')
>>> db.person.insert(name='Alex')
>>> rows = db(db.person).select()
>>> for row in rows.find(lambda row: row.name[0]=='M'):
        print row.name
Max
>>> print len(rows)
3
>>> for row in rows.exclude(lambda row: row.name[0]=='M'):
```

```
        print row.name
Max
>>> print len(rows)
2
>>> for row in rows.sort(lambda row: row.name):
        print row.name
Alex
John
```

They can be combined:

```
>>> rows = db(db.person).select()
>>> rows = rows.find(
        lambda row: 'x' in row.name).sort(
            lambda row: row.name)
>>> for row in rows:
        print row.name
Alex
Max
```

Sort takes an optional argument `reverse=True` with the obvious meaning.

The `find` method as an optional limitby argument with the same syntax and functionality as the Set select `method`.

3.11 One to many relation

To illustrate how to implement one to many relations with the web2py DAL, define another table "thing" that refers to the table "person" which we redefine here:

```
>>> db.define_table('person',
                    Field('name'),
                    format='%(name)s')
>>> db.define_table('thing',
                    Field('name'),
                    Field('owner_id', 'reference person'),
                    format='%(name)s')
```

Table "thing" has two fields, the name of the thing and the owner of the thing. The "owner_id" field id a reference field. A reference type can be specified in two equivalent ways:

```
Field('owner_id', 'reference person')
Field('owner_id', db.person)
```

The latter is always converted to the former. They are equivalent except in the case of lazy tables, self references or other types of cyclic references where the former notation is the only allowed notation.

When a field type is another table, it is intended that the field reference the other table by its id. In fact, you can print the actual type value and get:

```
>>> print db.thing.owner_id.type
reference person
```

Now, insert three things, two owned by Alex and one by Bob:

```
>>> db.thing.insert(name='Boat', owner_id=1)
1
>>> db.thing.insert(name='Chair', owner_id=1)
2
>>> db.thing.insert(name='Shoes', owner_id=2)
3
```

You can select as you did for any other table:

```
>>> for row in db(db.thing.owner_id==1).select():
        print row.name
Boat
Chair
```

Because a thing has a reference to a person, a person can have many things, so a record of table person now acquires a new attribute thing, which is a Set, that defines the things of that person. This allows looping over all persons and fetching their things easily:

```
>>> for person in db().select(db.person.ALL):
        print person.name
        for thing in person.thing.select():
            print '    ', thing.name
Alex
     Boat
     Chair
Bob
     Shoes
Carl
```

3.11.1 Inner joins

Another way to achieve a similar result is by using a join, specifically an INNER JOIN. web2py performs joins automatically and transparently when the query links two or more tables as in the following example:

```
>>> rows = db(db.person.id==db.thing.owner_id).select()
>>> for row in rows:
        print row.person.name, 'has', row.thing.name
Alex has Boat
Alex has Chair
Bob has Shoes
```

Observe that web2py did a join, so the rows now contain two records, one from each table, linked together. Because the two records may have fields with conflicting names, you need to specify the table when extracting a field value from a row. This means that while before you could do:

```
row.name
```

and it was obvious whether this was the name of a person or a thing, in the result of a join you have to be more explicit and say:

```
row.person.name
```

or:

```
row.thing.name
```

There is an alternative syntax for INNER JOINS:

```
>>> rows = db(db.person).select(join=db.thing.on(db.person.id==db.thing.owner_id))
>>> for row in rows:
    print row.person.name, 'has', row.thing.name
Alex has Boat
Alex has Chair
Bob has Shoes
```

While the output is the same, the generated SQL in the two cases can be different. The latter syntax removes possible ambiguities when the same table is joined twice and aliased:

```
>>> db.define_table('thing',
        Field('name'),
        Field('owner_id1','reference person'),
        Field('owner_id2','reference person'))
>>> rows = db(db.person).select(
    join=[db.person.with_alias('owner_id1').on(db.person.id==db.thing.owner_id1).
          db.person.with_alias('owner_id2').on(db.person.id==db.thing.owner_id2)])
```

The value of `join` can be list of `db.table.on(...)` to join.

3.11.2 Left outer join

Notice that Carl did not appear in the list above because he has no things. If you intend to select on persons (whether they have things or not) and their things (if they have any), then you need to perform a LEFT OUTER JOIN. This is done using the argument "left" of the select command. Here is an example:

```
>>> rows=db().select(
        db.person.ALL, db.thing.ALL,
        left=db.thing.on(db.person.id==db.thing.owner_id))
>>> for row in rows:
        print row.person.name, 'has', row.thing.name
Alex has Boat
Alex has Chair
Bob has Shoes
Carl has None
```

where:

```
left = db.thing.on(...)
```

does the left join query. Here the argument of `db.thing.on` is the condition required for the join (the same used above for the inner join). In the case of a left join, it is necessary to be explicit about which fields to select.

Multiple left joins can be combined by passing a list or tuple of `db.mytable.on(...)` to the `left` attribute.

3.11.3 Grouping and counting

When doing joins, sometimes you want to group rows according to certain criteria and count them. For example, count the number of things owned by every person. web2py allows this as well. First, you need a count operator. Second, you want to join the person table with the thing table by owner. Third, you want to select all rows (person + thing), group them by person, and count them while grouping:

```
>>> count = db.person.id.count()
>>> for row in db(db.person.id==db.thing.owner_id).select(
        db.person.name, count, groupby=db.person.name):
        print row.person.name, row[count]
Alex 2
```

```
Bob 1
```

Notice the count operator (which is built-in) is used as a field. The only issue here is in how to retrieve the information. Each row clearly contains a person and the count, but the count is not a field of a person nor is it a table. So where does it go? It goes into the storage object representing the record with a key equal to the query expression itself. The count method of the Field object has an optional `distinct` argument. When set to `True` it specifies that only distinct values of the field in question are to be counted.

3.12 Many to many

In the previous examples, we allowed a thing to have one owner but one person could have many things. What if Boat was owned by Alex and Curt? This requires a many-to-many relation, and it is realized via an intermediate table that links a person to a thing via an ownership relation.

Here is how to do it:

```
>>> db.define_table('person',
                    Field('name'))
>>> db.define_table('thing',
                    Field('name'))
>>> db.define_table('ownership',
                    Field('person', 'reference person'),
                    Field('thing', 'reference thing'))
```

the existing ownership relationship can now be rewritten as:

```
>>> db.ownership.insert(person=1, thing=1) # Alex owns Boat
>>> db.ownership.insert(person=1, thing=2) # Alex owns Chair
>>> db.ownership.insert(person=2, thing=3) # Bob owns Shoes
```

Now you can add the new relation that Curt co-owns Boat:

```
>>> db.ownership.insert(person=3, thing=1) # Curt owns Boat too
```

Because you now have a three-way relation between tables, it may be convenient to define a new set on which to perform operations:

```
>>> persons_and_things = db(
        (db.person.id==db.ownership.person) \
        & (db.thing.id==db.ownership.thing))
```

Now it is easy to select all persons and their things from the new Set:

```
>>> for row in persons_and_things.select():
        print row.person.name, row.thing.name
Alex Boat
Alex Chair
Bob Shoes
Curt Boat
```

Similarly, you can search for all things owned by Alex:

```
>>> for row in persons_and_things(db.person.name=='Alex').select():
        print row.thing.name
Boat
Chair
```

and all owners of Boat:

```
>>> for row in persons_and_things(db.thing.name=='Boat').select():
        print row.person.name
Alex
Curt
```

A lighter alternative to Many 2 Many relations is tagging. Tagging is discussed in the context of the `IS_IN_DB` validator. Tagging works even on database backends that do not support JOINs like the Google App Engine NoSQL.

3.13 Other Operators and Expressions

3.13.1 `like`, `regexp`, `startswith`, `contains`, `upper`, `lower`

Fields have a like operator that you can use to match strings:

```
>>> for row in db(db.log.event.like('port%')).select():
        print row.event
port scan
```

Here "port%" indicates a string starting with "port". The percent sign character, "%", is a wild-card character that means "any sequence of characters".

The like operator is case-insensitive but it can be made case-sensitive with

```
db.mytable.myfield.like('value',case_sensitive=True)
```

web2py also provides some shortcuts:

```
db.mytable.myfield.startswith('value')
```

```
db.mytable.myfield.contains('value')
```

which are equivalent respectively to

```
db.mytable.myfield.like('value%')
db.mytable.myfield.like('%value%')
```

Notice that `contains` has a special meaning for `list:<type>` fields and it was discussed in a previous section.

The `contains` method can also be passed a list of values and an optional boolean argument `all` to search for records that contain all values:

```
db.mytable.myfield.contains(['value1','value2'], all=True)
```

or any value from the list

```
db.mytable.myfield.contains(['value1','value2'], all=false)
```

There is a also a `regexp` method that works like the `like` method but allows regular expression syntax for the look-up expression. It is only supported by PostgreSQL and SQLite.

The `upper` and `lower` methods allow you to convert the value of the field to upper or lower case, and you can also combine them with the like operator:

```
>>> for row in db(db.log.event.upper().like('PORT%')).select():
        print row.event
port scan
```

3.13.2 `year`, `month`, `day`, `hour`, `minutes`, `seconds`

The date and datetime fields have day, month and year methods. The datetime and time fields have hour, minutes and seconds methods. Here is an example:

```
>>> for row in db(db.log.event_time.year()==2009).select():
        print row.event
port scan
xss injection
unauthorized login
```

3.13.3 `belongs`

The SQL IN operator is realized via the belongs method which returns true when the field value belongs to the specified set (list of tuples):

```
>>> for row in db(db.log.severity.belongs((1, 2))).select():
        print row.event
port scan
xss injection
```

The DAL also allows a nested select as the argument of the belongs operator. The only caveat is that the nested select has to be a `_select`, not a `select`, and only one field has to be selected explicitly, the one that defines the set.

```
>>> bad_days = db(db.log.severity==3)._select(db.log.event_time)
>>> for row in db(db.log.event_time.belongs(bad_days)).select():
        print row.event
port scan
xss injection
unauthorized login
```

In those cases where a nested select is required and the look-up field is a reference we can also use a query as argument. For example:

```
db.define_table('person',Field('name'))
db.define_table('thing,Field('name'), Field('owner_id','reference thing'))
db(db.thing.owner_id.belongs(db.person.name=='Jonathan')).select()
```

In this case it is obvious that the next select only needs the field referenced by the `db.thing.owner_id` field so we do not need the more verbose `_select` notation.

A nested select can also be used as insert/update value but in this case the syntax is different:

```
lazy = db(db.person.name=='Jonathan').nested_select(db.person.id)
db(db.thing.id==1).update(owner_id = lazy)
```

In this case `lazy` is a nested expression that computes the `id` of person "Jonathan". The two lines result in one single SQL query.

3.13.4 `sum`, `avg`, `min`, `max` and `len`

Previously, you have used the count operator to count records. Similarly, you can use the sum operator to add (sum) the values of a specific field from a group of records. As in the case of count, the result of a sum is retrieved via the store object:

```
>>> sum = db.log.severity.sum()
>>> print db().select(sum).first()[sum]
6
```

You can also use `avg`, `min`, and `max` to the average, minimum, and maximum value respectively for the selected records. For example:

```
>>> max = db.log.severity.max()
>>> print db().select(max).first()[max]
3
```

`.len()` computes the length of a string, text or boolean fields.

Expressions can be combined to form more complex expressions. For example here we are computing the sum of the length of all the severity strings in the logs, increased of one:

```
>>> sum = (db.log.severity.len()+1).sum()
>>> print db().select(sum).first()[sum]
```

3.13.5 Substrings

One can build an expression to refer to a substring. For example, we can group things whose name starts with the same three characters and select only one from each group:

```
db(db.thing).select(distinct = db.thing.name[:3])
```

3.13.6 Default values with `coalesce` and `coalesce_zero`

There are times when you need to pull a value from database but also need a default values if the value for a record is set to NULL. In SQL there is a keyword, `COALESCE`, for this. web2py has an equivalent `coalesce` method:

```
>>> db.define_table('sysuser',Field('username'),Field('fullname'))
>>> db.sysuser.insert(username='max',fullname='Max Power')
>>> db.sysuser.insert(username='tim',fullname=None)
print db(db.sysuser).select(db.sysuser.fullname.coalesce(db.sysuser.username))
"COALESCE(sysuser.fullname,sysuser.username)"
Max Power
tim
```

Other times you need to compute a mathematical expression but some fields have a value set to None while it should be zero. `coalesce_zero` comes to the rescue by defaulting None to zero in the query:

```
>>> db.define_table('sysuser',Field('username'),Field('points'))
>>> db.sysuser.insert(username='max',points=10)
```

```
>>> db.sysuser.insert(username='tim',points=None)
>>> print db(db.sysuser).select(db.sysuser.points.coalesce_zero().sum())
"SUM(COALESCE(sysuser.points,0))"
10
```

3.13.7 CSV (one Table at a time)

When a Rows object is converted to a string it is automatically serialized in CSV:

```
>>> rows = db(db.person.id==db.thing.owner_id).select()
>>> print rows
person.id,person.name,thing.id,thing.name,thing.owner_id
1,Alex,1,Boat,1
1,Alex,2,Chair,1
2,Bob,3,Shoes,2
```

You can serialize a single table in CSV and store it in a file "test.csv":

```
>>> open('test.csv', 'wb').write(str(db(db.person.id).select()))
```

This is equivalent to

```
>>> rows = db(db.person.id).select()
>>> rows.export_to_csv_file(open('test.csv', 'wb'))
```

You can read the CSV file back with:

```
>>> db.person.import_from_csv_file(open('test.csv', 'r'))
```

When importing, web2py looks for the field names in the CSV header. In this example, it finds two columns: "person.id" and "person.name". It ignores the "person." prefix, and it ignores the "id" fields. Then all records are appended and assigned new ids. Both of these operations can be performed via the appadmin web interface.

Bibliography

[1] http://www.python.org

[2] http://www.sqlite.org/

[3] http://numpy.scipy.org/

[4] http://www.scipy.org/

[5] http://matplotlib.sourceforge.net/

[6] http://www.python.org/dev/peps/pep-0008/

[7] http://www.network-theory.co.uk/docs/pytut/

[8] http://oreilly.com/catalog/9780596158071

[9] http://www.python.org/doc/

[10] M. Farach-Colton *et al.*, "Mathematical Support for Molecular Biology", DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (1999) Volume 47 ISBN-10: 0-8218-0826-5

[11] B. Korber *et al.* Timing the Ancestor of the HIV-1 Pandemic Strains, Science (9 Jun 2000) Vol. 288 no. 5472.

[12] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (1970)

[13] H. M. Markowitz "Portfolio Selection". The Journal of Finance 7 (1952)

[14] http://remembersaurus.com/mincemeatpy/

[15] Andrew Lo and Jasmina Hasanhodzic, The Evolution of Technical Analysis: Financial Prediction from Babylonian Tablets to Bloomberg Terminals. Bloomberg Press (2010). ISBN 1576603490

[16] Lindholm, Erik, et al. "NVIDIA Tesla: A unified graphics and computing architecture." Micro, IEEE 28.2 (2008): 39-55.

[17] Munshi, Aaftab. "OpenCL: Parallel Computing on the GPU and CPU." SIGGRAPH, Tutorial (2008).

[18] Klöckner, Andreas, et al. "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation." Parallel Computing 38.3 (2012): 157-174.

[19] Oliphant, Travis E. A Guide to NumPy. Vol. 1. USA: Trelgol Publishing, 2006.

[20] https://github.com/mdipierro/ocl

[21] https://github.com/mdipierro/canvas

[22] http://srossross.github.com/Meta/html/index.html

[23] Behnel, Stefan, et al. "Cython: The best of both worlds." Computing in Science & Engineering 13.2 (2011): 31-39.

[24] http://srossross.github.com/Clyther/

[25] Markowitz, Harry M. "Foundations of portfolio theory." The Journal of Finance 46.2 (2012): 469-477.