learning-python Documentation

Release 0.9

Huang, Hao

Jan 24, 2018

Contents

Contents:

CHAPTER 1

Introduction

1.1 Q&A

What's the frist programming language you learned ?

How many programming languages you are using for daily work ?

What's your favorite programming language ?

What is Python ? A interpreted, interactive, OO programming language.

Why do people use Python ?

Why is it called Python ? Monty Python's Flying Circus

What is Python good for ? high-level, a large standard library, a wide variety of 3rd-party packages

What can I do with Python ?

Which Python should I choose ?

- Version: 2.7 or 3.4
- Alternate Implementations: cpython, pypy, Jython, IronPython, stackless

Notes: this tutorial restrict on 2.7 and latest 3.x of cpython

1.2 Features

- Open source
- Dynamic scripting language
- OO and functional
- Free
- Portable

- Powerful
- Easy to learn and to use

1.3 Design and History

- History
- indentation
- explicit *self*
- no *case* statement
- *lambda* only allows expressions
- The Zen of Python(PEP 20)
- Guido

1.4 Resources

- docs
- pypi
- pyCon/weekly

1.5 Run programs

Try interpreter

```
>>> 1 + 1
2
>>> ^D or exit() or quit()

>>> dir()

>>> l = [1, 2, 3]
>>> dir(l)

>>> help(l)
>>> help(l.extend)
```

$ python hello.py

.pyc

- compiled .py
- After 3.2: *__pycache__*

Built-in document

$ pydoc with

$ pydoc -p 8080 # then open http://localhost:8080

1.6 Python's Arsenal

numeric, bool, sequence, dict, text, set, binary sequence, exception, object

~250 libs:

CHAPTER 2

Demos

- C extensions, ctypes, cython, swig, sip, pyrex
- i18n & l11n, babel, pytz, dateutil, lunar, datetime
- Tkinter, Wxpython, PyQT, PyGTK, PyWin32, IronPython, Jython, JPype
- Numpy, Scipy, Matplotlib, sympy, networkX, panda
- pygame, cgkit, pyglet, PySoy, Panda3D
- Xapian
- NLTK
- PIL, Pillow, PyOpenGL, Blender, Maya
- PyMedia, ID3
- scikit-learn, PyBrain, Milk
- GAE
- OpenStack
- Crypto, cryptography, pycrypto, pyOpenSSL, m2crypto, paramiko
- PySerial
- PyRo
- Raspberry Pi, Arduino
- PyXLL, DataNitro
- PyCLIPS, Pyke, Pyrolog, pyDatalog
- zenoss
- PythonCAD, PythonOCC, FreeCAD
- ReportLab, Sphinx, Cheetah, PyPDF
- Mayavi, matplotlib, VTK, VPython

- lxml, xmlrpclib
- json, csv
- Orange, Pattern bundle, Scrapy
- PySolFC
- Django, TruboGears, web2py, Pylons, Zope, WebWare
- IronPython, pyjs(a.k.a pyjamas)

CHAPTER 3

Core Types

Built-in type	Example
None	None
Booleans	True,False
Numbers	123,3.14,3+4j,0xef,Decimal,Fraction
Strings	‘spam’,”Bob’s”,b’ax01c’,u’spxc4m’
Lists	[1,[2,’Three’],4.5], list(range(10))
Tuples	(1,’spam’,4,’U’),tuple(‘spam’),namedtuple
Dicts	{‘food’:’spam’,’taste’:’yum’},dict(hours=10)
Sets	{1,’a’,’bc’},set(‘abc’)
Files	open(‘eggs.txt’)
functions	def,lambda
modules	import,‘__module__‘
classes	object,types,metaclasses

3.1 None

A special object serves as an empty placeholder (much like a NULL pointer in C). it is returned from functions that don’t explicitly return anything.

```
>>> None, type(None)
(None, <type 'NoneType'>)

>>> def foo(): pass
...
>>> print foo()
None
```

3.2 Numeric types

A complete inventory of Python's numeric toolbox includes:

- Integer and float-point
- Complex number
- Decimal: fixed-precision
- Fraction: rational number
- Sets: collections with numeric operations
- Booleans: true and false
- Built-in functions and modules: round, math, random, etc.
- Expressions; unlimited integer precision; bitwise operations; hex, otctal, and binary formats
- Third-party extensions: vectors, libraries, visulization, plotting, etc.

Literals:

```
1234, -24, 0, 9999999999999999
1.23, 1., 3.14e-10, 4E210, 4.0e+210
0o177, 0x9ff, 0b10101011, 0177
3+4j, 3.0+4.0j, 3j
set('spam'), {1, 2, 3, 4}
Decimal('1.0'), Decimal('-Inf'), Fraction(1, 3)
bool(x), True, False
```

Example:

```
>>> 3 ** 2 / (4 + 1)
1
>>> 3 ^ 9
10
>> 3 | 9
11
>>> 1 << 3
8
>>> 0xf - 0b1000 + 010
15
>>> (2 + 1j) * -1j
(1-2j)
>>> ({1, 2, 3} & { 3, 5, 7} | {3, 4}) - {Decimal('3')}
set([4])
>>> Decimal('3') == 3
True

>>> True = False     # 2.x
>>> True == False
True
```

Operator precedence

```
+, -, *, /, //, >>, <<, **, &, |, ^, %, ~
<, >, !=, ==, <=, >=, in, not in, not, and, or
```

Built-in functions

abs, bin, bool, divmod, float, hex, int, oct, pow, round

```
>>> bool(3) and True or False
True
>>> ' '.join([bin(13),  hex(13), oct(13)])
'0b1101 0xd 015'
>>> divmod(7, 3)
(2, 1)
>>> abs(-3)
3
>>> pow(2, 8) == 2 ** 8
True
>>> round(3.14)
3.0
>>> int('3') + float('.5')
5.5
>>> int('10', base=16) - int('10') - int('10', base=8) - int('10', base=2)
-4
```

Built-in modules

numbers, math, cmath, decimal, fractions, random, statistics

```
>>> type(3)         # 2.x
<type 'int'>
>>> type(2**100)
<type 'long'>
>>> 2**100
1267650600228229401496703205376L
>>> type(3L)
<type 'long'>

>>> from numbers import Number, Complex, Real, Rational, Integral
>>> issubclass(Integral, Complex)
True
>>> isinstance(1, Complex)
True

>>> math.factorial(3) + math.log(math.e) + math.sqrt(9) + math.sin(math.pi/2) + math.
↪ceil(0.1)  # 6+1+3+1+1
12.0
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> cmath.sqrt(-1)
1j

>>> from random import *
>>> random()
0.06091254441752425
>>> sample(range(10), 3)
[0, 1, 4]
>>> choice(range(10))
5
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 7, 0, 1, 2, 3, 9, 6, 4, 8]
```

```
>>> gauss(0, 1)
-0.8042047260239109

>>> from decimal import *
>>> .1 * 3 - .3
5.551115123125783e-17
>>> Decimal('.1') * Decimal('3') - Decimal('.3')
Decimal('0.0')
>>> 1.20 * 1.30
1.56
>>> Decimal('1.20') * Decimal('1.30')
Decimal('1.5600')
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')

>>> from fractions import Fraction
>>> (6/5) * (7/3) - 2.8
4.440892098500626e-16
>>> Fraction(6, 5) * Fraction(7, 3) - Fraction('2.8')
Fraction(0, 1)
>>> gcd(15, 6)
>>> 3

>>> from statistics import *
>>> mean([1, 2, 3, 4, 4])
>>> 2.8
>>> median([1, 3, 5])
>>> 3
>>> mod([1, 1, 2, 3, 3, 3, 3, 4])
>>> 3
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

New in 2.6

- PEP 3141: A Type Hierarchy for Numbers

New in 3.0

- PEP 0237: Essentially, long renamed to int. That is, there is only one built-in integral type, named int; but it behaves mostly like the old long type.
- PEP 0238: An expression like 1/2 returns a float. Use 1//2 to get the truncating behavior. (The latter syntax has existed for years, at least since Python 2.2.)
- The sys.maxint constant was removed, since there is no longer a limit to the value of integers. However, sys.maxsize can be used as an integer larger than any practical list or string index. It conforms to the implementation's "natural" integer size and is typically the same as sys.maxint in previous releases on the same platform (assuming the same build options).
- The repr() of a long integer doesn't include the trailing L anymore, so code that unconditionally strips that character will chop off the last digit instead. (Use str() instead.)

- Octal literals are no longer of the form 0720; use 0o720 instead.
- PEP 3141 – A Type Hierarchy for Numbers
- Ordering Comparisions: The ordering comparison operators (<, <=, >=, >) raise a TypeError exception when the operands don't have a meaningful natural ordering. Thus, expressions like 1 < '', 0 > None or len <= len are no longer valid, and e.g. None < None raises TypeError instead of returning False. A corollary is that sorting a heterogeneous list no longer makes sense – all the elements must be comparable to each other. Note that this does not apply to the == and != operators: objects of different incomparable types always compare unequal to each other.
- Changed Syntax: True, False, and None are reserved words. (2.6 partially enforced the restrictions on None already.)

3.3 Strings

Literals

- Single quotes: 'spa"m'
- Double quotes: "spa'm"
- Triple quotes: ''' ... spam ... ''', """ ... spam ... """
- Escape sequences: "stpna0m"
- Raw strings: r"C:newtest.spm"
- Bytes literals in 3.x and 2.6+: b'spx01am'
- Unicode literals in 2.x and 3.3+: u'eggsu0020spam'

Single- and double-quoted strings are the same

Implicit concatenation:

```
>>> title = "Meaning " 'of' " Life"
>>> title
'Meaning of Life'
```

Escape characters

Escape	Meaning
\newline	Ignored(continuation line)
\\	Backslash (stores one \)
\'	Single quote(stores ‘)
\"	Double quote(stores “)
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline(linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xhh	Character with hex value hh(exactly 2 digits)
\ooo	Character with octal value ooo(up to 3 digits)
\0	Null: binary 0 character(doesn’t end string)
\N{id}	Unicode database ID
\uhhhh	Unicode character with 16bit hex value
\Uhhhhhhhh	Unicode character with 32bit hex value
\other	Not an escape(keeps both \ and other)

Raw strings suppress escapes:

```
>>> path = r'C:\new\text.dat'
>>> path               # Show as Python code
'C:\\new\\text.dat'
>>> print(path)        # User-friendly format
C:\new\text.dat
>>> len(path)          # String length
15
```

Triple quotes code multiline block strings:

```
>>> mantra = """Always look
...   on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
>>> print(mantra)
Always look
  on the bright
side of life.
```

Basic operations:

```
>>> len('abc')
3
>>> 'abc' + 'def'
'abcdef'
>>> 'Ni!' * 4
'Ni!Ni!Ni!Ni!'

>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')
...
```

```
h a c k e r
>>> "k" in myjob
True
>>> "z" in myjob
False
>>> 'spam' in 'abcspamdef'
True
```

Indexing and slicing:

```
>>> S = 'spam'
>>> S[0], S[2]
('s', 'a')
>>> S[1:3], S[1:], S[:1]
('pa', 'pam', 'spa')

>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'
>>> S = 'hello'
>>> S[::1]            # Reversing items
'olleh'
>>> S = 'abcedfg'
>>> S[5:1:1]
'fdec'

>>> 'spam'[1:3]
'pa'
>>> 'spam'[slice(1, 3)]
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, 1)]
'maps'
```

String conversion:

```
>>> int("42"), str(42)
(42, '42')
>>> repr(42)
'42'
>>> str('spam'), repr('spam')
('spam', "'spam'")

>>> str(3.1415), float("1.5")
('3.1415', 1.5)
>>> text = "1.234E-10"
>>> float(text)
1.234e-10

>>> ord('s')
115
>>> chr(115)
's'
```

Changing string:

```
>>> S = 'spam'       # Immutable objects
>>> S[0] = 'x'       # Raises an error!
TypeError: 'str' object does not support item assignment

>>> S = S + 'SPAM!'  # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[1]
>>> S
'spamBurger!'

>>> S = 'splot'
>>> id(S)
18598192
>>> S = S.replace('pl', 'pamal')
>>> id(S)
18598096
>>> S
'spamalot'
>>> id('spam')
18597136
>>> id('spamalot')
18597760

>>> 'That is %d %s bird!' % (1, 'dead')
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead')
'That is 1 dead bird!'
```

str, the *bytes* type is immutable. There is a separate mutable type to hold buffered binary data, *bytearray*.

String methods in 3.4

- str.capitalize
- str.casefold
- str.center
- str.count
- **str.encode(encoding="utf-8",-errors="strict")**
- **str.endswith(suffix[,-start[,-end]])**

```
>>> [name for name in os.listdir('/etc/') if name.endswith('.conf')][:5]
['asl.conf', 'autofs.conf', 'dnsextd.conf', 'ftpd.conf', 'ip6addrctl.conf']
```

- str.expendtabs
- **str.find(sub[,-start[,-end]])**

```
>>> 'abcd'.find('a')
0
>>> 'abcd'.find('1')
-1
>>> 'abcd'.find('d', 2)
3
>>> 'abcd'.find('d')
3
```

- **str.format(*args, **kwargs)**
- str.format_map
- **str.index(sub[, start[, end]])**

```
>>> 'abcd'.find('e')
-1
>>> 'abcd'.index('e')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

- str.isalnum
- **str.isalpha()**

```
>>> 'abd'.isalpha()
True
>>> 'abd1'.isalpha()
False
>>> '1234'.isdigit()
True
>>> '123a'.isdigit()
False
>>> '12ab'.isalnum()
True
>>> '\n\t '.isspace()
True
```

- str.isdecimal
- str.isdigit
- str.isidentifier
- str.islower
- str.isnumeric
- str.isprintable
- str.isspace
- str.istitle
- str.isupper
- **str.join(iterable)**

```
>>> ','.join(['ab', 'c', 'd'])
'ab,c,d'
>>> ','.join('abcd')
'a,b,c,d'
```

- str.ljust
- **str.lower()**

```
>>> 'PyTHon'.lower()
'python'
>>> 'PyTHon'.upper()
'PYTHON'
```

- str.lstrip
- str.maketrans
- str.partition
- **str.replace(old, new[, count])**

```
>>> 'PyTHon'.replace('TH', 'C')
'PyCon'
```

- str.rfind
- str.rindex
- str.rjust
- str.rpartition
- str.rsplit
- str.rstrip
- **str.split(sep=None, maxsplit=-1)**

```
>>> 'a b  \t\t c\nd'.split()
['a', 'b', 'c', 'd']
>>> 'a,b,c,d'.split(',')
['a', 'b', 'c', 'd']
>>> 'a b  \t\t c\nd'.split(None, 2)
['a', 'b', 'c\nd']
```

- **str.splitlines([keepends])**
- **str.startswith(prefix[, start[, end]])**
- **str.strip([chars])**

```
>>> '   line\n'.strip()
'line'
>>> '   line\n'.lstrip()
'line\n'
>>> '   line\n'.rstrip()
'   line'
```

- str.swapcase
- str.title
- str.translate
- **str.upper()**
- str.zfill

printf-style String Formatting

%s, %d, %f, %g, %x

Text vs. data instead of unicode vs. 8-bit

In 2.x:

```
>>> type('hello'), repr('hello')
(<type 'str'>, "'hello'")
>>> type(u''), repr(u'')
(<type 'unicode'>, "u'\\u4f60\\u597d'")
>>> type(''), type(u'hello')
(<type 'str'>, <type 'unicode'>)

>>> issubclass(str, basestring)
True
>>> issubclass(unicode, basestring)
True

>>> u'hello' + ' world'
u'hello world'
```

- *str* is 8-bit, it represents ascii string and binary data.
- *unicode* represents text.
- unicode.encode => str
- str.decode => unicode
- Keep text in unicode inside your system. Encode and decode at the bournday(incoming/outgoing) of your system.
- open().read() returns *str*

In 3.x:

```
>>> type('hello'), type(u'hello'), type(b'hello')
(<class 'str'>, <class 'str'>, <class 'bytes'>)

>>> type(''), type(u'')
(<class 'str'>, <class 'str'>)
>>> type(b'')
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> type(''.encode()), repr(''.encode())
(<class 'bytes'>, "b'\\xe4\\xbd\\xa0\\xe5\\xa5\\xbd'")

>>> 'hello' + b' world'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly

>>> type(open('name.txt').read())
<class 'str'>
>>> type(open('name.txt', 'br').read())
<class 'bytes'>
>>> type(os.listdir()[0])
<class 'str'>
>>> type(sys.argv[0])
<class 'str'>
```

- All text are unicode. The type used to hold text is *str*.
- Encoded unicode is represented as binary data. The type used to hold binary data is *bytes*.
- Mixing text and binary data raises TypeError.

- *basestring* was removed. *str* and *bytes* don't share a base class.
- open().read() returns *str*; open(, 'b').read() returns *bytes*.
- sys.stdin, sys.stdout and sys.stderr are unicode-only text files.
- Filenames are passed to and returned from APIs as (Unicode) strings.

See Unicode HOWTO

3.4 Lists

- Ordered collections of arbitrary objects
- Accessed by offset
- Variable-length, heterogeneous, and arbitrarily nestable
- Of the category "mutable sequence"
- Arrays of object references

Operation	Interpretation
L = []	An empty list
L = [123, 'abc', 1.23, {}]	Four items: indexes 0..3
L = ['Bob', 40.0, ['dev', 'mgr']]	Nested sublists
L = list('spam')	List of an iterable's items, list of successive integers
L = list(range(-4, 4))	
L[i]	Index, index of index, slice, length
L[i][j]	
L[i:j]	
len(L)	
L1 + L2	Concatenate, repeat
L* 3	
for x in L: print(x)	Iteration, membership
3 in L	
L.append(4)	Methods: growing
L.extend([5,6,7])	
L.insert(i, X)	
L.index(X)	Methods: searching
L.count(X)	
L.sort()	Methods: sorting, reversing,
L.reverse()	
L.copy()	copying (3.3+), clearing (3.3+)
L.clear()	
L.pop(i)	Methods, statements: shrinking
L.remove(X)	
del L[i]	
del L[i:j]	
L[i:j] = []	Index assignment, slice assignment
L[i] = 3	
L[i:j] = [4,5,6]	
L = [x**2 for x in range(5)]	List comprehensions and maps
list(map(ord, 'spam'))	

Built-in functions range() and xrange():

```
>>> range(5)           # 2.x
[0, 1, 2, 3, 4]
>>> xrange(5)
xrange(5)
>>> type(range(5)), type(xrange(5))
(<type 'list'>, <type 'xrange'>)

>>> range(5)           # 3.x
range(0, 5)
>>> type(range(5))
<class 'range'>
```

Change in 3.0: range() now behaves like xrange() used to behave, except it works with values of arbitrary size. The latter no longer exists.

3.5 Dictionaries

- Accessed by key, not offset position
- Unordered collections of arbitrary objects
- Variable-length, heterogeneous, and arbitrarily nestable
- Of the category "mutable mapping"
- Tables of object references (hash tables)

Operation	Interpretation
D = {}	Empty dict
D = {'name': 'Bob', 'age': 40}	Two items
E = {'cto': {'name': 'Bob', 'age': 40}}	Nesting
D = dict(name='Bob', age=40)	Alternative construction techniques:
D = dict([('name', 'Bob'), ('age', 40)])	keywords, key/value pairs, zipped key/value pairs, key lists
D = dict(zip(keyslist, valueslist))	
D = dict.fromkeys(['name', 'age'])	
D['name']	Indexing by key
E['cto']['age']	
'age' in D	Membership: key present test
D.keys()	Methods: all keys,
D.values()	all values,
D.items()	all key+value tuples,
D.copy()	copy (top-level),
D.clear()	clear (remove all items),
D.update(D2)	merge by keys,
D.get(key, default?)	fetch by key, if absent default (or None),
D.pop(key, default?)	remove by key, if absent default (or error)
D.setdefault(key, default?)	fetch by key, if absent set default (or None),
D.popitem()	remove/return any (key, value) pair; etc.
len(D)	Length: number of stored entries
D[key] = 42	Adding/changing keys
del D[key]	Deleting entries by key
list(D.keys())	Dictionary views (Python 3.X)
D1.keys() & D2.keys()	
D.viewkeys(), D.viewvalues()	Dictionary views (Python 2.7)
D = {x: x*2 for x in range(10)}	Dictionary comprehensions (Python 3.X, 2.7)

Built-in function zip():

```
>>> zip(range(5), 'abc')
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Change in 3.0: zip() now returns an iterator.

3.6 Tuples

- Ordered collections of arbitrary objects
- Accessed by offset
- Of the category "immutable sequence"
- Fixed-length, heterogeneous, and arbitrarily nestable
- Arrays of object references

Operation	Interpretation
()	An empty tuple
T = (0,)	A one-item tuple (not an expression)
T = (0, 'Ni', 1.2, 3)	A four-item tuple
T = 0, 'Ni', 1.2, 3	Another four-item tuple (same as prior line)
T = ('Bob', ('dev', 'mgr'))	Nested tuples
T = tuple('spam')	Tuple of items in an iterable
T[i]	Index, index of index, slice, length
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	Concatenate, repeat
T* 3	
for x in T: print(x)	Iteration, membership
'spam' in T	
[x ** 2 for x in T]	
T.index('Ni')	Methods in 2.6, 2.7, and 3.X: search, count
T.count('Ni')	
namedtuple('Emp', ['name', 'jobs'])	Named tuple extension type

Named tuple Immutable records

3.7 Sets

- Unordered collections of arbitrary objects
- Accessed by iteration, membership test, not offset position
- Variable-length, heterogeneous, and arbitrarily nestable
- Of the category "mutable mapping"
- Collections of object references

Notes: largely because of their implementation, sets can only contain immutable (a.k.a. "hashable", __hash__) object types. Hence, lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values.

```
>>> x = set('abcde')
>>> y = set('bdxyz')

>>> x
set(['a', 'c', 'b', 'e', 'd'])

>>> x  y                                          # Difference
set(['a', 'c', 'e'])

>>> x | y                                          # Uninon
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                                          # Intersection
set(['b', 'd'])

>>> x ^ y                                          # Symmetric difference (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])
```

```
>>> x > y, x < y                                # Superset, subset
(False, False)

>>> 'e' in x                                    # Membership
True

>>> z = x.intersection(y)                       # Same as x & y
>>> z
set(['b', 'd'])

>>> z.add('SPAM')                               # Insert one item
>>> z
set(['b', 'd', 'SPAM'])

>>> z.update(set(['X', 'Y']))                   # Merge: in-place union
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])

>>> z.remove('b')                               # Delete one item
>>> z
set(['Y', 'X', 'd', 'SPAM'])

>>> for item in set('abc'): print(item * 3)     # Iterable, unordered
aaa
ccc
bbb

>>> {i for i in 'abc'}                          # Set compression
set(['a', 'c', 'b'])
```

fronzenset The frozenset type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Immutables numbers, strings, tuples, frozensets

Mutables lists, dicts, sets, bytearray

See Scala's mutable and immutable collections

3.8 The meaning of True and False in Python

True and false are intrinsic properties of every object in Python, each object is either ture of false, as follows:

- Numbers are false if zero, and true otherwise
- Other objects are false if empty, and true otherwise

  False None, '', [], {}, 0, 0.0, (), set([])

  True "something", [1, 2], {'eggs': 'spam'}, 1, 0.1, (3, 4), {5, 6}

```
d = {1: 2}
if d:
  print "it goes here"
else:
  print "not here"
```

3.9 Files

Operation	Interpretation
output = open(r'C:spam', 'w')	Create output file ('w' means write)
input = open('data', 'r')	Create input file ('r' means read)
input = open('data')	Same as prior line ('r' is the default)
aString = input.read()	Read entire file into a single string
aString = input.read(N)	Read up to next N characters (or bytes) into a string
aString = input.readline()	Read next line (including n newline) into a string
aList = input.readlines()	Read entire file into list of line strings (with n)
output.write(aString)	Write a string of characters (or bytes) into file
output.writelines(aList)	Write all line strings in a list into file
output.close()	Manual close (done for you when file is collected)
output.flush()	Flush output buffer to disk without closing
anyFile.seek(N)	Change file position to offset N for next operation
for line in open('data'): use line	File iterators read line by line
open('f.txt', encoding='latin-1')	Python 3.X Unicode text files (str strings)
open('f.bin', 'rb')	Python 3.X bytes files (bytes strings)
codecs.open('f.txt', encoding='utf8')	Python 2.X Unicode text files (unicode strings)
open('f.bin', 'rb')	Python 2.X bytes files (str strings)

Storing Native Python Objects: pickle

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                    # Pickle any object to file
>>> F.close()

>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)                   # Load any object from file
>>> E
{'a': 1, 'b': 2}

>>> open('datafile.pkl', 'rb').read()    # Format is prone to change!
b'\x80\x03}q\x00(X\x01\x00\x00\x00bq\x01K\x02X\x01\x00\x00\x00aq\x02K\x01u.'
```

Storing Python Objects in JSON Format

```
>>> name = dict(first='Bob', last='Smith')
>>> rec = dict(name=name, job=['dev', 'mgr'], age=40.5)
>>> rec
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}

>>> import json
>>> S = json.dumps(rec)
>>> S
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'

>>> O = json.loads(S)
>>> O
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
>>> O == rec
True
```

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
    "job": [
        "dev",
        "mgr" ],
    "name": {
        "last": "Smith",
        "first": "Bob"
    },
    "age": 40.5
}
>>> P = json.load(open('testjson.txt'))
>>> P
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

Storing Packed Binary Data: struct

Format characters

```
>>> F = open('data.bin', 'wb')                      # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, b'spam', 8)     # Make packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                                   # Write byte string
>>> F.close()

>>> F = open('data.bin', 'rb')                      # Get packed binary data
>>> data = F.read()
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data)          # Convert to Python objects
>>> values
(7, b'spam', 8)
```

File Context Managers

```
with open(r'C:\code\data.txt') as myfile:
  for line in myfile:
    ...use line here...

=>

myfile = open(r'C:\code\data.txt')
try:
  for line in myfile:
    ...use line here...
finally:
  myfile.close()
```

3.10 The standard type hierarchy

None This type has a single value.

NotImplemented This type has a single value. raise NotImplemented

```
class CarInterface:

  def drive(self):
    raise NotImplemented
```

Ellipsis This type has a single value. literal **...** or the built-in name **Ellipsis**.

```
>>> ...
Ellipsis
>>> bool(...)
True
>>> def foo():
...   ...
...
>>> foo
<function foo at 0x10606a840>

>>> a = [1]
>>> a.append(a)
>>> a
[1, [...]]

>>> from numpy import array
>>> a = array([[1,2,3], [4,5,6], [7,8,9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[...,1]
array([2, 5, 8])
>>> a[1,...]
array([4, 5, 6])
```

numbers.Number

- numbers.Integral: Integers(int), Booleans(bool)
- numbers.Real(float)
- numbers.Complex(complex)

Sequences

- Immutable: Strings, Tuples, Bytes
- Mutable: Lists, ByteArrays

Set types

- Mutable: Sets
- Immutable: ForzenSets

Mappings

Mutable: Dictionaries

Callable types

- User-defined functions
- Instance methods

- Generators
- Built-in functions
- Built-in methods
- Classes: __new__, __init__
- Class instances: __call__

Modules

I/O objects(Also known as file objects)

Internal types

- Code objects
- Frame objects
- Traceback objecs
- Slice objects
- Static methods objects
- Class methods objects

3.11 Type objects

The largest point to notice here is that everything in a Python system is an **object** type. In fact, even types themselves are an object type in Python: the type of an object is an object of type **type**.

```
>>> class Foo: pass
...
>>> type(Foo())
<class '__main__.Foo'>
>>> type(Foo)
<class 'type'>
>>> issubclass(Foo, object)
True
>>> isinstance(Foo, type)
True

>>> type(object)
<class 'type'>
>>> type(type)
<class 'type'>

>>> type(1)
<class 'int'>
>>> type(int)
<class 'type'>
>>> issubclass(int, object)
True

>>> def foo(): pass
...
>>> import types
>>> types.FunctionType
<class 'function'>
```

```
>>> type(foo) == types.FunctionType
True
>>> type(types.FunctionType)
<class 'type'>
```

- See types — Dynamic type creation and names for built-in types
- See PEP 3115 – Metaclasses in Python 3000

CHAPTER 4

Syntax

Python program structure

1. *Programs* are composed of modules.
2. *Modules* contain statements.
3. *Statements* contain expressions.
4. *Expressions* create and process objects.

4.1 Lexical analysis

Line structure A Python program is divided into a number of logical lines.

Logical lines The end of a logical line is represented by the token NEWLINE.

Physical lines A physical line is a sequence of characters terminated by an end-of-line sequence(CR/LF).

Comments A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line.

Encoding declarations If a comment in the first or second line of the Python script matches the regular expression **coding[=:]s*([-w.]+)**.

The recommended forms of this expression are:

```
# -*- coding: <encoding-name> -*-
which is recognized also by GNU Emacs, and

# vim:fileencoding=<encoding-name>
which is recognized by Bram Moolenaar's VIM.
```

If no encoding declaration is found, the default encoding is UTF-8.

Explicit line joining Two or more physical lines may be joined into logical lines using backslash characters

```
if 1900 < year < 2100 and 1 <= month <= 12 \
  and 1 <= day <= 31 and 0 <= hour < 24 \
  and 0 <= minute < 60 and 0 <= second < 60:    # Looks like a valid date
    return 1
```

Implicit line joining Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes.

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Blank lines A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored.

Indentation Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Identifiers

Keywords

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Reserved classes of identifiers Certain classes of identifiers (besides keywords) have special meanings.

_* Not imported by from module import *.

The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the builtins module.

When not in interactive mode, _ has no special meaning and is not defined.

__*__ System-defined names. See Special method names.

__* Class-private names.

Literals string, bytes, numeric(interger, float, imaginary)

Operators:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

Delimiters:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

4.2 Expressions

Identifiers(names), Literals, lists, dicts, sets, Attributes, subscriptions, slices, calls, arithmetic and bitwise operations, shifting operations, comparisions, boolean operations, Expression lists

Condition expressions:

```
x if C else y
```

Lambda expressions:

```
lambda (x, y): x + y
=>
def <lambda>(x, y):
  return x + y
```

Generator expressions:

```
(x*y for x in range(10) for y in bar(x))
```

Yield expressions:

```
def foo(n):
  for i in range(n):
    yield i
```

4.3 Assignment

```
i = 1
i += 2

a, b = 2, 3.14
a, b = b, a      # swap

first, second, _ = (1, 2, 3)           # pattern match
a, (b, c), d = [1, [2, 3], 4]
first, second, *others = range(10)     # py3.
```

Operation Interpretation	
spam = 'Spam'	Basic form
spam, ham = 'yum', 'YUM'	Tuple assignment (positional)
[spam, ham] = ['yum', 'YUM']	List assignment (positional)
a, b, c, d = 'spam'	Sequence assignment, generalized
a, *b = 'spam'	Extended sequence unpacking (Python 3.X)
spam = ham = 'lunch'	Multiple-target assignment
spams += 42	Augmented assignment (equivalent to spams = spams + 42)

Assignments create references not copies:

```
>>> class Foo: pass
...
>>> a = Foo()
>>> b = a, c = copy.copy(a)
```

```
>>> id(a), id(b), id(c)
(4378338248, 4378338248, 4378329040)
```

Quiz: What happens when copy built-in types such as *int* ?

PEP 3132 - Extended Iterable Unpacking. The specification for the *target feature.

4.4 Pass

when it is excuted, nothing happens. It's useful as a placeholder

4.5 If

```
if x > 0
  print 'Positive'
elif x < 0:
  print 'Nagtive'
else:
  print 'Zero'
```

4.6 Loop

for, while, break, continue

```
x = 7
while x > 0:
  print x * 2
  x -= 1
else:
  print 'End'

for i in range(10):
  print i
  i = 5
```

Notes: *break* terminates the nearest enclosing loop, skipping the optional *else* clause if the loop has one.

4.7 Try/Raise

```
def foo():
  if random.random() < .1:
    raise SomeException("BOOM!")

try:
  foo()
except SomeException as err1:
  print err1
except AnotherException as err2:
  raise
```

```
except (AException, BException) as err3:
  pass
else:
  print 'No exception occurs'
finally:
  print "This block is always evaluated"
```

4.8 With

```
# this file will be closed automatically
# even exception is raised within this block

with open('somefile', 'w') as writer:
  write_content_to(writer)
```

Context Manager __enter__() __exit__()

```
# py3.1, 2.7
with A() as a, B() as b:
  do some thing
=>
# py2.6
with A() as a:
  with B() as b:
    do some thing
```

4.9 Yield

```
def start_from(n):
  while True:
    yield n
    n += 1
```

4.10 Return

```
def foo(n):
  return 'Even' if n % 2 == 0 else 'Odd'

# If no explicit return value is given,
# return value is None

def foo(n):
  pass
```

4.11 Import

```
import sys
import os.path

from random import *
from os.path import (join, exist)
from math import pi

import numpy as np
from pyquery import PyQuery as pq
```

4.12 Global, local and nonlocal

```
a = 1

def foo():
  global a
  a = 2

def bar():
  a = 2   # local

print(a)  # => 1
bar()
print(a)  # => 1
foo()
print(a)  # => 2
```

built-in functions: locals(), globals()

```
def create_account(initial):
  balance = initial

  def query():
    return balance

  def dec(n):
    nonlocal balance
    if balance < n:
      raise ValueError('Not enough money')
    balance -= n

  return {
    'query': query,
    'dec': dec,
    }

a1 = make_account(100)
a2 = make_account(100)
a1['dec'](50)
print(a1['query'])    # => 50
print(a2['query'])    # => 100
```

PEP 3104 - Access to Names in Outer Scopes. The specification for the nonlocal statement.

4.13 Assert

```
def factorial(n):
  assert n >= 0 and isinstance(n, numbers.Integral), \
    "Factorial of negative and non-integral is undefined"

assert expression1 [, expression2]
=>
if __debug__:
  if not expression1:
    raise AssertError(expression2):
```

In the current implementation, the built-in variable __debug__ is True under normal circumstances, False when optimization is requested (command line option -O). Assignments to __debug__ are illegal.

4.14 Del

```
>>> a = 1
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> class Foo:
...   def __init__(self, a):
...     self.a = a
...
>>> foo = Foo(3)
>>> foo.a
3
>>> del foo.a
>>> foo.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Foo instance has no attribute 'a'
```

4.15 Print

Change in 3.0:

> Print is a Function PEP 3105 – Make print a function

4.16 Exec

Change in 3.0:

> Removed keyword: exec() is no longer a keyword; it remains as a function. (Fortunately the function syntax was also accepted in 2.x.) Also note that exec() no longer takes a stream argument; instead of exec(f) you can use exec(f.read()).

4.17 Iterations

In a sense, iterable objects include both physical sequences and virtual sequences computed on demand.

The full iteration protocol: iter, next and StopIteration

- The *iterable* object you request iteration for, whose *__iter__* is run by *iter*
- The *iterator* object returned by the iterable that actually produces values during the iteration, whose *__next__* is run by next and raises *StopIteration* when finished producing results

PEP 3114: the standard next() method has been renamed to __next__().

```
>>> l = [1,2,3]
>>> next(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>> iter(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not iterable

>>> i, j = iter(l), iter(l)
>>> i.__next__()
1
>>> next(i)
2
>>> next(j)
1
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> f = open('tt.py')
>>> iter(f) is f
True
>>> f.__next__()
'x = 7\n'
>>> next(f)
'while x > 0:\n'
```

Techically, when the for loop begins, it first obtains an iterator from the iterable object by passing it to the iter built-in function; the object returned by iter in turn has the required next method.

A generator is also an iterator

```
>>> def foo():
...     yield 1
...     yield 2
...
>>> i = foo()
>>> next(i)
1
>>>
>>> next(i)
```

```
2
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> type(foo)
    <type 'function'>
>>> type(i)
<type 'generator'>
```

4.18 Comprehensions

Syntactically, its syntax is derived from a construct in set theory notation that applies an operation to each item in a set.

List comprehensions:

```
>>> l = [1, 2, 3, 4, 5]
>>> res = []
>>> for x in l:
...   res.append(x + 10)
...
>>> res
[11, 12, 13, 14, 15]

>>> [x + 10 for x in l]
[11, 12, 13, 14, 15]
```

Notes: list comprehensions might run much faster than manual for loop statements (often roughly twice as fast) because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code. Especially for larger data sets, there is often a major performance advantage to using this expression.

Filter clauses: if

```
>>> [x + 10 for x in l if x % 2 == 0]
[12, 14]
```

Nested loops: for

```
>>> [x + y for x in 'abc' for y in '123']
['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

Dict comprehensions:

```
>>> {x:ord(x)-ord('a') for x in 'abc'}
{'a': 0, 'c': 2, 'b': 1}
```

Set comprehensions:

```
>>> {x for x in 'abc'}
{'a', 'c', 'b'}
```

Generator expressions:

```
>>> (x for x in 'abc')
<generator object <genexpr> at 0x104f3ca20>
```

CHAPTER 5

Functions

5.1 def

Format:

```
def name(arg1, arg2, ..., argN):
  ...
  [return value]
```

Executes at runtime:

```
if test:
  def func():
    print 'Define func this way'
else:
  def func():
    print 'Or else this way'
...
func()     # Call the func defined
```

5.2 Scopes

- If a variable is assigned inside a def, it is *local* to that function.
- If a variable is assigned in an enclosing def, it is *nonlocal* to nested functions.
- If a variable is assigned outside all defs, it is *global* to the entire file.

```
x = 99           # Global(module)

def func():
  x = 88         # Local(func): a different variable
```

```
  def inner():
    print(x)     # Nonlocal(inner)
```

Name Resolution: The LEGB Rule

- Name assignments create or change local names by default.
- Name references search at most four scopes: local(L), then enclosing(E) functions (if any), then global(G), then built-in(B).
- Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes, respectively.

The built-in scope:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
↪'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
↪'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
↪'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError',
↪'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError',
↪'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit',
↪'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
↪'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
↪'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError',
↪'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
↪'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
↪'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
↪'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
↪'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
↪'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
↪ 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
↪ '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
↪'__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
↪ 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
↪'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
↪'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
↪'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
↪'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object',
↪'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed
↪', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
↪'super', 'tuple', 'type', 'vars', 'zip']
>>> zip
<class 'zip'>
>>> zip is builtins.zip
True
```

Use *global* and *nonlocal* for changes:

```
>>> x = 99
>>> def func():
...   global x
...   x = 88
...
>>> print(x)
99
>>> func()
>>> print(x)
```

```
88

>>> def func():
...     x = 88
...     def inner():
...         nonlocal x
...         x = 77
...     print(x)
...     inner()
...     print(x)
...
>>> func()
88
77

>>> x = 99
>>> def func():
...     nonlocal x
...     x = 88
...
  File "<stdin>", line 2
SyntaxError: no binding for nonlocal 'x' found
```

See PEP 3104: nonlocal statement. Using nonlocal x you can now assign directly to a variable in an outer (but non-global) scope. nonlocal is a new reserved word

5.3 Arguments

Argument Matching Basics

- Positionals: matched from left to right
- Keywords: matched by argument name
- Defaults: specify values for optional arguments that aren't passed
- Varargs collecting: collect arbitrarily many positional or keyword arguments
- Varargs unpacking: pass arbitrarily many positional or keyword arguments
- Keyword-only arguments: arguments that must be passed by name

Syntax	Interpretation
func(value)	Normal argument: matched by position
func(name=value)	Keyword argument: matched by name
func(*iterable)	Pass all objects in iterable as individual positional arguments
func(**dict)	Pass all key/value pairs in dict as individual keyword arguments
def func(name)	Normal argument: matches any passed value by position or name
def func(name=value)	Default argument value, if not passed in the call
def func(*name)	Matches and collects remaining positional arguments in a tuple
def func(**name)	Matches and collects remaining keyword arguments in a dictionary
def func(*other, name)	Arguments that must be passed by keyword only in calls (3.X)
def func(*, name=value)	Arguments that must be passed by keyword only in calls (3.X)

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}

>>> def func(a, b, c, d): print(a, b, c, d)
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)      # Same as func(1, 2, 3, 4)
1 2 3 4

>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)     # Same as func(a=1, b=2, c=3, d=4)
1 2 3 4

>>> func(*(1, 2), **{'d': 4, 'c': 3})    # Same as func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *(2, 3), **{'d': 4})         # Same as func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4})      # Same as func(1, 2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4)                # Same as func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, *(2,), c=3, **{'d':4})       # Same as func(1, 2, c=3, d=4)
1 2 3 4
```

Quiz: Write a function max accepts any number of arguments and returns the bigest of them.

3.x keyword-only arguments:

```
>>> def kwonly(a, *b, c, **d): print(a, b, c, d)
>>> kwonly(1, 2, c=3)
1 (2,) 3 {}
>>> kwonly(a=1, c=3)
1 () 3 {}
>>> kwonly(1, 2, 3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, c=3, d=4, e=5)
1 (2,) 3 {'d':4, 'e': 5}
```

Keyword-only arguments must be specified after a single star, not two. Named arguments cannot appear after the **args arbitrary keywords form, and a ** can't appear by itself in the arguments list.

```
>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax
```

Why keyword-only arguments ?

```
def process(*args, notify=False): ...

process(X, Y, Z)             # Use flag's default
process(X, Y, notify=True)   # Override flag default
```

Without keyword-only arguments we have to use both *args and **args and manually inspect the keywords, but with keyword-only arguments less code is required.

Quiz: try to implement the same feature above without using keyword-only arguments.

5.4 Function design principles

- use arguments for inputs and return for outputs.
- use global variables only when truly necessary.
- don't change mutable arguments unless the caller expects it.
- each function should have a single, unified purpose.
- each function should be relatively small.
- avoid changing variables in another module file directly.

5.5 "First Class" Objects

Python functions are full-blown objects:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
func(arg)
```

5.6 Function Introspection

```
>>> def mul(a, b):
...     """Multiple a by b times"""
...     return a * b
...
>>> mul('spam', 8)
'spamspamspamspamspamspamspamspam'

>>> mul.__name__
'mul'
>>> mul.__doc__
'Multiple a by b times'

>>> mul.__code__
<code object func at 0x104f24c90, file "<stdin>", line 1>
>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
2
```

5.7 Function Annotations in 3.x

Annotations are completely optional, and when present are simply attached to the function object's __annotations__ attribute for use by other tools.

```
>>> def foo(a: 'x', b: 5 + 6, c: list) -> max(2, 9):
...     ...
...
```

```
>>> foo.__annotations__
{'a': 'x', 'return': 9, 'c': <class 'list'>, 'b': 11}
```

See PEP 3107: Function argument and return value annotations.

5.8 Anonymous Functions: lambda

lambda argument1, argument2,... argumentN : expression using arguments

- lambda is an expression, not a statement.
- lambda's body is a single expression, not a block of statements.
- annotations are not supported in lambda

5.9 Functional programming tools

map, filter, functools.reduce

5.10 Generator functions

yield vs. return:

```
>>> def gensquares(N):
...   for i in range(N):
...     yield i ** 2
...
>>> for i in gensquares(5): # Resume the function
...   print(i, end=' : ')
...
0 : 1 : 4 : 9 : 16 :

>>> x = gensquares(2)
>>> x
<generator object gensquares at 0x000000000292CA68>
>>> next(x)
0
>>> next(x)
1
>>> next(x)
Traceback (most recent call last):
File "<stdin>", line 1, in <module> StopIteration

>>> y = gensquares(5)
>>> iter(y) is y
True
```

Why using generators ?

send vs. next:

```
>>> def gen():
...   for i in range(10):
...     x = yield i
...     print('x=', x)
...
>>> g = gen()
>>> next(g)
0
>>> g.send(77)
x= 77
1
>>> g.send(88)
x= 88
2
>>> next(g)
x= None
3
```

See PEP 342 – Coroutines via Enhanced Generators

yield from allows a generator to delegate part of its operations to another generator.

For simple iterators, yield from iterable is essentially just a shortened form of *for item in iterable: yield item*:

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

However, unlike an ordinary loop, yield from allows subgenerators to receive sent and thrown values directly from the calling scope, and return a final value to the outer generator:

```
>>> def accumulate():
...     tally = 0
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc) # Ensure the accumulator is ready to accept values
>>> for i in range(4):
...     acc.send(i)
...
>>> acc.send(None) # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None) # Finish the second tally
```

```
>>> tallies
[6, 10]
```

See PEP 380: Syntax for Delegating to a Subgenerator

itertools: Functions creating iterators for efficient looping

```
>>> from itertools import *
>>> def take(n, iterable):
...     "Return first n items of the iterable as a list"
...     return list(islice(iterable, n))
...
>>>

>>> take(10, count(2))
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> take(10, cycle('abcd'))
['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b']
>>> take(5, repeat(6))
[6, 6, 6, 6, 6]

>>> list(accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]
>>> list(chain('abc', 'ABC'))
['a', 'b', 'c', 'A', 'B', 'C']
>>> list(takewhile(lambda x: x<5, [1,4,6,4,1]))
[1, 4]

>>> list(permutations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'C'), ('B', 'D'), ('C', 'A'), (
↪'C', 'B'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D', 'C')]
>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

5.11 Function Decorators

Decorator is just a function returning another function. It is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
@f1(arg)
@f2
def func(): pass

def func(): pass
func = f1(arg)(f2(func))
```

Common examples for decorators are classmethod() and staticmethod():

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

functools

```
>>> def bar(func):
...   def inner():
...     print('New function')
...     return func()
...   return inner
...
>>> @bar
... def foo():
...   print('I am foo')
...
>>> foo()
New function
I am foo
>>> foo.__name__     # It's bad!
'inner'

>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

CHAPTER 6

Modules and Packages

Modules are processed with two statemets and one important function:

import Lets a client fetch a module as a whole

from Allows clients to fetch particular names from a module

imp.reload (reload in 2.x) Provides a way to reload a module's code without stopping Python

At a base level, a Python program consists of text files containing Python statements, with one main top-level file, and zero or more supplemental files known as modules.

```
def spam(text):             # File b.py
  print(text, 'spam')

import b                    # File a.py
b.spam('gumby')             # Prints 'gumby spam'
```

Import statements are executed at runtime, their net effect is to assign module names—simple variables like b—to loaded module objects. Objects defined by a module are also created at runtime, as the import is executing: every name assigned at the top-level of the file becomes an attribute of the module, accessible to importers.

6.1 How imports work

1. Find the module's file.
2. Compile it to byte code (if needed).
3. Run the module's code to build the objects it defines.

Module search path:

1. the home directory of the program
2. PYTHONPATH (if set)
3. Standard lib directories

4. The content of any .pth files (if present)
5. The site-packages home of third-party extensions

Ultimately, the concatenation of these four components becomes sys.path, a mutable list of directory name.

Byte code files: .pyc, __pycache__(3.2+)

Because module names become variable names inside a Python program (without the .py), they should also follow the normal variable name rules.

Imports happen only once

```
print('Hello')     # simple.py
spam = 1

>>> import simple
Hello
>>> import simple
>>> simple.spam
1
>>> simple.spam = 2
>>> import simple
>>> simple.spam
2
```

sys.modules This is a dictionary that maps module names to modules which have already been loaded.

```
>>> 'simple' in sys.modules.keys()
True
>>> sys.modules['simple']
<module 'simple' from '/Users/huanghao/workspace/learning-python/chapter06/simple.py'>
```

Import and from are assignments

> Just like def, import and from are executable statements, not compile-time declarations. They may be nested in if tests, to select among options; appear in function defs, to be loaded only on calls (subject to the preceding note); be used in try statements, to pro- vide defaults; and so on. They are not resolved or run until Python reaches them while executing your program. In other words, imported modules and names are not available until their associated import or from statements run.

Reloading modules

```
>>> simple.spam
2
>>> import imp
>>> imp.reload(simple)
Hello
<module 'simple' from '/Users/huanghao/workspace/learning-python/chapter06/simple.py'>
>>> simple.spam
1
```

6.2 Module usage

The import statement:

```
>>> import module1                    # Get module as a whole (one or more)
>>> module1.printer('Hello world!')   # Qualify to get names
```

```
Hello world!

>>> import module1 as module2
```

The from statement:

```
>>> from module1 import printer       # Copy out a variable (one or more)
>>> printer('Hello world!')           # No need to qualify name
Hello world!

>>> from module1 import printer as display
```

The from * statement:

```
>>> from module1 import *             # Copy out _all_ variables
>>> printer('Hello world!')
Hello world!
```

6.3 Packages

A directory of Python code is said to be a package, so such imports are known as package imports.

```
import dir1.dir2.mod
```

```
dir0 $ tree dir1
dir1
|-- __init__.py
|-- __pycache__
|   `-- __init__.cpython-34.pyc
`-- dir2
    |-- __init__.py
    |-- __pycache__
    |   |-- __init__.cpython-34.pyc
    |   `-- mod.cpython-34.pyc
    `-- mod.py

3 directories, 6 files
```

Package __init__.py file

dir0/dir1/dir2/mod.py

- dir1 and dir2 both must contain an __init__.py file.
- dir0, the container, does not require an __init__.py file; this file will simply be ignored if present.
- dir0, not dir0/dir1, must be listed on the module search path sys.path.

Package initialization file roles

- Package initialization
- Module usability declarations
- Module namespace initialization
- from * statmenet behavior

```
print('dir1 init')     # dir1/__init__.py
x = 1

print('dir2 init')     # dir1/dir2/__init__.py
y = 2

print('in mod.py')     # dir1/dir2/mod.py
z = 3
```

```
>>> import dir1.dir2.mod
dir1 init
dir2 init
in mod
>>> import dir1.dir2.mod

>>> from imp import reload
>>> reload(dir1)
dir1 init
<module 'dir1' from './dir1/__init__.py'>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from './dir1/dir2/__init__.py'>
>>> reload(dir1.dir2.mod)
in mod
<module 'dir1.dir2.mod' from './dir1/dir2/mod.py'>

>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

```
>>> dir2.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'dir2' is not defined
>>> from dir1 import dir2
>>> dir2.y
2
```

6.4 Relative imports

- Imports with dots: both 3.x and 2.x, it's relative-only and will not look sys.path
- Imports without dots: in 2.x, relative-then-absolute; in 3.x, it's absolute-only

```
from . import spam                          # Relative to this package
from .spam import name

from .string import name1, name2            # Imports names from mypkg.string
from . import string                        # Imports mypkg.string
from .. import string                       # Imports string sibling of mypkg

from __future__ import absolute_import      # Use 3.X relative import model in 2.X
```

See PEP 0328 - Imports: Multi-Line and Absolute/Relative

6.5 Python3.3 namespace packages

Allows packages to span multiple directories, and requires no initialization file.

Specification

> Regular packages will continue to have an __init__.py and will reside in a single directory.
>
> Namespace packages cannot contain an __init__.py. As a consequence, pkgutil.extend_path and pkg_resources.declare_namespace become obsolete for purposes of namespace package creation. There will be no marker file or directory for specifying a namespace package.
>
> During import processing, the import machinery will continue to iterate over each directory in the parent path as it does in Python 3.2. While looking for a module or package named "foo", for each directory in the parent path:
>
> - If <directory>/foo/__init__.py is found, a regular package is imported and returned.
> - If not, but <directory>/foo.{py,pyc,so,pyd} is found, a module is imported and returned. The exact list of extension varies by platform and whether the -O flag is specified. The list here is representative.
> - If not, but <directory>/foo is found and is a directory, it is recorded and the scan continues with the next directory in the parent path.
> - Otherwise the scan continues with the next directory in the parent path.
>
> If the scan completes without returning a module or package, and at least one directory was recorded, then a namespace package is created. The new namespace package:
>
> - Has a __path__ attribute set to an iterable of the path strings that were found and recorded during the scan.
> - Does not have a __file__ attribute.
>
> Note that if "import foo" is executed and "foo" is found as a namespace package (using the above rules), then "foo" is immediately created as a package. The creation of the namespace package is not deferred until a sub-level import occurs.
>
> A namespace package is not fundamentally different from a regular package. It is just a different way of creating packages. Once a namespace package is created, there is no functional difference between it and a regular package.

```
$ tree ns
ns
|-- dir1
|   `-- sub
|       `-- mod1.py
`-- dir2
    `-- sub
        `-- mod2.py

$ cat ns/dir1/sub/mod1.py
print('dir1/sub.mod1')

$ cat ns/dir2/sub/mod2.py
print('dir2/sub.mod2')
```

```
$ PYTHONPATH=./ns/dir1:./ns/dir2 python3
```

```
>>> import sub
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
_NamespacePath(['./ns/dir1/sub', './ns/dir2/sub'])

>>> from sub import mod1
dir1/sub.mod1
>>> import sub.mod2
dir2/sub.mod2

>>> mod1
<module 'sub.mod1' from './ns/dir1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from './ns/dir2/sub/mod2.py'>
```

Namespace package nesting

```
$ mkdir ns/dir2/sub/lower
$ echo "print('dir2/sub.lower.mod3')" > ns/dir2/sub/lower/mod3.py
```

```
>>> import sub.lower.mod3
dir2/sub.lower.mod3
>>> sub.lower
<module 'sub.lower' (namespace)>
>>> sub.lower.__path__
_NamespacePath(['./ns/dir2/sub/lower'])
```

Mix regular packages with namespace packages

```
$ tree ns
ns
|-- dir1
|   |-- empty
|   `-- sub
|       `-- mod1.py
`-- dir2
    `-- sub
        |-- lower
        |   `-- mod3.py
        |-- mod2.py
        `-- pkg
            `-- __init__.py
```

```
>>> import empty
>>> empty
<module 'empty' (namespace)>
>>> empty.__path__
_NamespacePath(['./ns/dir1/empty'])

>>> import sub.pkg
dir2/sub2.pkg.__init__
>>> sub.pkg.__file__
'./ns/dir2/sub/pkg/__init__.py'
```

See PEP 420: Implicit Namespace Packages

6.6 Data hiding in modules

_X

```
$ cat unders.py
a, _b, c, _d = 1, 2, 3, 4
```

```
>>> from unders import *
>>> a, c
(1, 3)
>>> _b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_b' is not defined

>>> from unders import _b, _d
>>> _b, _d
(2, 4)
```

__all__

```
$ cat alls.py
__all__ = ['a', '_c']       # __all__ has precedence over _X
a, b, _c, _d = 1, 2, 3, 4
```

```
>>> from alls import *
>>> a, _c
(1, 3)
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined

>>> from alls import b, _d
>>> b, _d
(2, 4)
```

6.7 Enable future language features: __future__

```
from __future__ import featurename
```

When used in a script, this statement must appear as the first executable statement in the file (possibly following a docstring or comment), because it enables special compilation of code on a per-module basis.

```
>>> import __future__
>>> dir(__future__)
['CO_FUTURE_ABSOLUTE_IMPORT', 'CO_FUTURE_BARRY_AS_BDFL', 'CO_FUTURE_DIVISION', 'CO_
↪FUTURE_PRINT_FUNCTION', 'CO_FUTURE_UNICODE_LITERALS', 'CO_FUTURE_WITH_STATEMENT',
↪'CO_GENERATOR_ALLOWED', 'CO_NESTED', '_Feature', '__all__', '__builtins__', '__
↪cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__
↪', 'absolute_import', 'all_feature_names', 'barry_as_FLUFL', 'division', 'generators
↪', 'nested_scopes', 'print_function', 'unicode_literals', 'with_statement']
```

6.8 __name__ and __main__

- If the file is being run as a top-level program file, __name__ is set to the string "__main__" when it starts.
- If the file is being imported instead, __name__ is set to the module's name as known by its clients.

```
def test():
  print(":)")

if __name__ == '__main__':
  test()
```

```
$ python runme.py
:)
$ python -m runme
:)
```

6.9 Docstrings

Just like docstring of function, the first string in a python file works as docstring of that module.

```
>>> import socket
>>> help(socket)
Help on module socket:

NAME
    socket

MODULE REFERENCE
    http://docs.python.org/3.4/library/socket

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.
...
```

CHAPTER 7

Classes and OOP

Everything in python are objects, even classes and codes.

Object identity

Every object has an identify, a type and a value. An object's identify never changes once it has been created. You may think of it as the object's address in memory. The *is* operator compares the identity of two objects. The *id()* functions returns an integer representing its identity.

CPython implementation detail: For CPython, id(x) is the memory address where x is stored.

An object's type determines the operators that the object supports and also defines the possible values for objects of that type. The *type()* function returns an object's type (which is an object itself).

The value of some objects can change. Objects whose value can change are said to be mutable; objects whose value is unchangeable once they are created are called immutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.

7.1 Basic usage

```
class Person:
    """
    Class represents a person
    """

    def __init__(self, name, job=None, pay=0):
        """
        Constructor method
        """
        self.name = name
        self.job = job
        self.pay = pay

    def __del__(self):
```

```
        """
        Deconstructor: it will be called when gc recycle this object, or *del* called.
        """

    def last_name(self):
        """
        The first argument of instance methods is *self*.
        It's the reference to current instance, just like *this* in C++ and Java.
        """
        return self.name.split()[-1]

    def give_raise(self, percent):
        self.pay = int(self.pay * (1 + percent))


if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', 'engineer', 8000)

    print(bob.last_name(), bob.pay)      # ('Smith', 0)
    sue.give_raise(.1)
    print(sue.last_name(), sue.pay)      # ('Jones', 8800)
```

Provding print displays:

```
>>> print(bob)
<__main__.Person instance at 0x1031634d0>

class Person:
  ...

  def __str__(self):
    return self.name

  def __repr__(self):
    return '[%s: %s, %s]' % (self.__class__.__name__, self.name, self.pay)

>>> bob
[Person: Bob Smith, 0]
>>> print(bob)
Bob Smith

>>> str(sue)
'Sue Jones'
>>> repr(sue)
'[Person: Sue Jones, 8800]'
```

Subclasses:

```
class Manager(Person):

  def __init__(self, name, pay):
    super().__init__(name, 'manager', pay)

  def give_raise(percent, bouns=.1)
    Person.give_raise(self, percent + bouns)

>>> tom = Manager('Tom Jones', 'manager', 5000)
```

```
>>> tom.give_raise(.1)
>>> repr(tom)
[Manager: Tom Jones, 6000]
```

Special class attributes:

```
>>> tom.__class__
<class 'person.Manager'>
>>> tom.__class__.__bases__
(<class 'person.Person'>,)
>>> tom.__dict__
{'job': 'manager', 'name': 'Tom Jones', 'pay': 6000}
```

Class methods and static methods

Properties:

```
class Person:
  def __init__(self, name):
    self._name = name

  def getName(self):
    print('fetch...')
    return self._name

  def setName(self, value):
    print('change...')
    self._name = value

  def delName(self):
    print('remove...')
    del self._name

  name = property(getName, setName, delName, "name property docs")

bob = Person('Bob Smith')
print(bob.name)              # getName
bob.name = 'Robert Smith'    # setName
print(bob.name)
```

Class decorator Similar as function decorator. It's a callable object which accepts a class and return a class.

7.2 Special attributes

object.__dict__ A dictionary or other mapping object used to store an object's (writable) attributes.

instance.__class__ The class to which a class instance belongs.

class.__bases__ The tuple of base classes of a class object.

class.__name__ The name of the class or type.

class.__qualname__ The qualified name of the class or type.

New in version 3.3.

See PEP 3155 - Qualified name for classes and functions

class.__mro__ This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

class.mro() This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in __mro__.

class.__subclasses__() Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

7.3 Operator overloadding

Emulating numeric types

```
class Number:

  def __eq__(self, right):
    ...

  def __add__(self, right):
    ...

  def __sub__(self, right):
    ...

  def __mul__(self, right):
    ...

ten = two * five
six - one = ten - five
```

Full methods list for numeric types

Method	Operator
__add__	+
__sub__	-
__mul__	*
__truediv__	/
__floordiv__	//
__mod__	%
__divmod__	divmod
__pow__	**, pwer
__lshift__	<<
__rshift__	>>
__and__	&
__xor__	^
__or__	|
__radd__	+
__iadd__	+=
__neg__	-
__pos__	+
__abs__	abs
__invert__	~
__complex__	complex
__int__	int
__float__	float
__round__	round
__index__	operator.index()

Comparisons

Method	Operator
__lt__	<
__le__	<=
__eq__	==
__ne__	!=
__gt__	>
__ge__	>=

To automatically generate ordering operations from a single root operation, see functools.total_ordering().

String related

Method	Operator
__str__	str
__repr__	repr
__bytes__	bytes
__format__	format

Emulating callable objects

Method	Operator
__call__	()

```
>>> class Foo:
...   def __call__(self):
...     print("Callable")
...
>>> foo = Foo()
>>> foo()
Callable
```

Emulating container types

Method	Operator
__len__	len
__length_hit__	operator.length_hint()
__getitem__	v = obj[key]
__setitem__	obj[key] = v
__delitem__	del obj[key]
__iter__	for _ in obj, Iteration
__reversed__	reversed()
__contains__	key in obj

With statment context manager

Method	Operator
__enter__	with
__exit__	with

```
class cd:

    def __init__(self, path):
        self.path = path
        self.old = os.getcwd()

    def __enter__(self):
        os.chdir(self.path)

    def __exit__(self, exc_type, exc_value, traceback):
        os.chdir(self.old)

  with cd('/some/path'):
      ...
  # cd back to old path even exception occurs
```

See PEP 0343 - The "with" statement

Instance and subclass checks

Method	Operator
__instancecheck__	isinstance
__subclasscheck__	issubclass

See PEP 3119 - Introducing Abstract Base Classes

Misc.

Method	Operator	Comments
__hash__	hash	members of hashable collections including set, forzenset, dict.
__bool__	bool	if a class defines neither __bool__ and __len__, all its instances considered true

7.4 Customize attribute access

Method	Operator
__getattr__	o.attr, getattr(o, 'attr')
__getattribute__	o.attr, getattr(o, 'attr')
__setattr__	o.attr = val
__delattr__	del o.attr
__dir__	dir()

```
class Proxy:

  def __init__(self, wrapped):
    self.__dict__['_wrapped'] = wrapped

  def __getattr__(self, name):
    return getattr(self._wrapped, name)

  def __setattr__(self, name, value):
    setattr(self._wrapped, name, value)
```

```
>>> d = {}
>>> p = Proxy(d)
>>> p['a'] = 1
>>> p.b = 2
>>> p.keys()
dict_keys(['a'])
>>> p.__dict__
{'b': 2, '_wrapped': {'a': 1}}
```

Comparison between __getattr__ and __getattribute__

- Both methods should return the (computed) attribute value or raise an AttributeError exception
- __getattr__ is called when an attribute lookup has not found; however __getattribute__ is called unconditionally.
- If AttributeError was raised in __getattribute__ then __getattr__ will be called.
- In order to avoid infinite recursion in __getattribute__, its implementation should always call object.__getattribute__(self, name) to get attributes it needs.
- Similarly, always call object.__setattr__(self, name, value) in __setattr__.

Descriptor

- __get__
- __set__
- __delete__

See Descriptor HowTo Guide

Slots

__slots__

See Saving 9GB of ram with Python's __slots__

7.5 Customize class creation

Method	Operator
__new__	C() before __init__
__init__	C()
__del__	del o (gc)
__prepare__	

__new__ Called to create a new instance of class cls.

- If __new__() returns an instance of cls, then the new instance's __init__() method will be invoked like __init__(self, ...), where self is the new instance and the remaining arguments are the same as were passed to __new__().
- If __new__() does not return an instance of cls, then the new instance's __init__() method will not be invoked.

__new__() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It's also commonly overridden in custom metaclasses in order to customize class creation.

```
class LoginForm(forms.Form):

  email = forms.EmailField()
  password = forms.PasswordField()

form = LoginForm(request.POST)     # {'username': 'abcd', 'password': 'abcd'}
if not form.is_valid()
  return HttpResponse(form.error_as_string())
```

```
class Form(six.with_metaclass(DeclarativeFieldsMetaclass, BaseForm)):
  ...

class DeclarativeFieldsMetaclass:

  def __new__(mcs, name, bases, attrs):
    current_fields = []
      for key, value in list(attrs.items()):
        if isinstance(value, Field):
          current_fields.append((key, value))
          attrs.pop(key)
          attrs['declared_fields'] = OrderedDict(current_fields)
    ...
    new_class = (super(DeclarativeFieldsMetaclass, mcs)
      .__new__(mcs, name, bases, attrs))
    ...

    new_class.base_fields = declared_fields
    new_class.declared_fields = declared_fields

    return new_class

class BaseForm(object):
  def __init__(self, ...):
```

```
    ...
    self.fields = copy.deepcopy(self.base_fields)

  def __getitem__(self, name):
    "Returns a BoundField with the given name."
    try:
      field = self.fields[name]
    except KeyError:
      raise KeyError(
        "Key %r not found in '%s'" % (name, self.__class__.__name__))
    return BoundField(self, field, name)
```

By default, classes are construted using type(name, bases, dict). In the following exmaple, both MyClass and MySubclass are instances of Meta:

```
class Meta(type):
  pass

class MyClass(metaclass=Meta):
  pass

class MySubclass(MyClass):
  pass
```

When a class definition is executed, the following steps occur:

1. the appropriate metaclass is determined
2. the class namespace is prepared
3. the class body is executed
4. the class object is created

Determining the appropriate metaclass

- if no bases and no explicit metaclass are given, then type() is used
- if an explicit metaclass is given and it is not an instance of type(), then it is used directly as the metaclass
- if an instance of type() is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used

Preparing the class namespace

- namespace = metaclass.__prepare__(name, bases, **kwds)
- otherwise, an empty dict() instance

kwds come from the class definition.

Executing the class body

    exec(body, globals(), namespace)

Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling

    metaclass(name, bases, namespace, **kwds)

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

```
class OrderedClass(type):

     @classmethod
     def __prepare__(metacls, name, bases, **kwds):
        return collections.OrderedDict()

     def __new__(cls, name, bases, namespace, **kwds):
        result = type.__new__(cls, name, bases, dict(namespace))
        result.members = tuple(namespace)
        return result

class A(metaclass=OrderedClass):
    def one(self): pass
    def two(self): pass
    def three(self): pass
    def four(self): pass
```

```
>>> A.members
('__module__', 'one', 'two', 'three', 'four')
```

See PEP 3115 - Metaclasses in Python 3000 Introduced the __prepare__ namespace hook

See PEP 3135 - New super Describes the implicit __class__ closure reference

See Special method names for the full list of special method names

7.6 Advanced topics

The "New style" class model From 2.2, python introduced a new flavor of classes, known as new-style classes. classes following the original and traditional model became known as classic classes. In 3.x only the new style remained.

For 2.x, classes must explicitly inherit from object to be considered "new style", otherwise they are "classic":

```
class Foo:          # classic
  pass

class Bar(object):  # new style
  pass
```

See Old and New classes

MRO and super #TODO

CHAPTER 8

Exceptions

- try/expect/else/finally
- raise Exc/raise
- assert
- with/as, contextmanager
- Built-in exceptions
- traceback

8.1 Basics

Default exception handler:

```
>>> def fetcher(obj, index):
...   return obj[index]
...
>>> x = 'spam'
>>> fetcher(x, 3)
m

>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Catching exception:

```
>>> try:
...   fetcher(x, 4)
... except IndexError:
```

```
...   print('got exception')
... print('continue')
...
got exception
continue
```

Raising exceptions:

```
>>> raise IndexError('cross the line')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: cross the line

>>> assert False, 'Nobody like it'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nobody like it
```

User-defined exceptions:

```
>>> class AlreadyGotOne(Exception): pass
...
>>> def grail():
...   raise AlreadyGotOne()
...
>>> try:
...   grail()
... except AlreadyGotOne:
...   print('got exception')
...
got exception
```

Termination actions:

```
>>> try:
...   fetcher(x, 3)
... finally:
...   print('after fetch')
...
'm'
after fetch

>>> def after():
...   try:
...     fetcher(x, 4)
...   finally:
...     print('after fetch')
...   print('after try?')
...
>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Context manager:

```
with threading.Lock() as lock:
  do_something()

# is equivalent to

lock = threading.Lock()
lock.acquire()
try:
  do_something()
finally:
  lock.release()

# If we don't use context manager or finally clause

lock.acquire()
some_something()     # if exception happens here
lock.release()       # then this line won't be called
```

8.2 Exception coding detail

try/except/else/finally:

```
try:
  some_actions()
excpet Exception1:
  handler1
except Exception2:
  handler2
...
except:
  handler all exceptions
else:
  no_exception_occurs
finally:
  termination
```

The raise statement

To trigger exceptions explicitly, you can use the following three forms of raise statements:

- raise instance

  It's the most common way to raise an instance of some exception.

- raise class

  If we pass a class instead, python calls constructor without arguments, to create an instance to raise.

- raise

  This form reraises the most recently raised exception. It's commonly used in exception handlers to propagate exceptions that have been caught.

```
>>> try:
      ..   1/0
... except ZeroDivisionError:
...   print('oops')
...   raise
```

```
...
oops
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Scopes and try except variables

```
>>> try:      # py3
...   1/0
... except Exception as x:
...   print(x)
...
division by zero
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

```
>>> try:      # py2
...   1/0
... except Exception as x:
...   print x
...
integer division or modulo by zero
>>> x
ZeroDivisionError('integer division or modulo by zero',)

>>> try:      # the old py2 way to assign exception variable
...   1/0    # which already abandoned in py3
... except Exception, x:
...   print x
...
integer division or modulo by zero
```

See PEP 3110: Catching exceptions.

Catching multiple exceptions in single except:

```
>>> import random
>>> def random_error():
...   if random.random() < 0.5:
...     1/0
...   else:
...     [][1]
...
>>> random_error()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in random_error
IndexError: list index out of range
>>> random_error()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in random_error
ZeroDivisionError: integer division or modulo by zero

>>> def run():
```

```
...   try:
...     random_error()
...   except (IndexError, ZeroDivisionError):
...     print('got')
...
>>> run()
got
```

Careful in py2: the () is essential:

```
>>> try:
...   random_error()
... except IndexError, ZeroDivisionError:   # valid syntax in py2
...   print 'got'
...
got
>>> ZeroDivisionError
IndexError('list index out of range',)
```

The 3.x exception chaining: raise from

Exceptions can sometimes be triggered in resonpse to other exceptions - both delibrately and by new program errors. To support full disclosure in such cases, 3.x support a new raise from syntax:

raise newexception from otherexception

- If the form clause is used explicitly, the other exception will be attached to __cause__ attribute of the new exception being raised. If the raised exception is not caught, python prints out the whole exception chain.

```
>>> try:
...   try:
...     1/0
...   except Exception as e:
...     raise TypeError('Bad') from e
... except Exception as e:
...   raise ValueError('Worse') from e
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: Bad

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
ValueError: Worse
```

- When an exception is raised implicitly by a program error inside an exception handler, a similar procedure is followed automatically: the previous exception is attached to the new exception's __context__ attribute and is again displayed if uncaught.

```
>>> try:
...   1/0
```

```
... except:
...   badname
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'badname' is not defined
```

See PEP 3134: Exception chaining.

Suppressing exception context

3.3 introduces a new syntax to disable display of chained exception context. No debugging capability is lost, as the original exception context remains available if needed.

```
>>> try:
...   1/0
... except ZeroDivisionError:
...   raise ValueError("zero can't be used as demoninator")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: zero can't be used as demoninator
```

```
>>> def test():
...   try:
...     1/0
...   except ZeroDivisionError:
...     raise ValueError("zero can't be used as demoinator") from None
...
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in test
ValueError: zero can't be used as demoinator

>>> try:
...   test()
... except Exception as e:
...   print(e.__context__)
...
division by zero
```

See PEP 409: Suppressing exception context

The assert statement

Just for somewhat debugging and testing purposes.

assert test, msg # msg is optional

=>

```
if __debug__:
  if not test:
    raise AssertError(msg)
```

8.3 Built-in exceptions

In Python, all exceptions must be instances of a class that derives from *BaseException*.

Programmers are encouraged to derive new exceptions from the *Exception* class or one of its subclass, and not from *BaseException*.

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
```

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

BaseException The base class for all built-in exceptions.

args: The tuple of arguments given to the exception constructor. If str() is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

with_traceback(tb):

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

Exception All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

SystemExit This exception is raised by the sys.exit() function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is an integer, it specifies the system exit status (passed to C's exit() function); if it is None, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

```
>>> try:
...   sys.exit(5)
... except:
...   print(sys.exc_info())
...   print("system didn't exit")
...
(<class 'SystemExit'>, SystemExit(5,), <traceback object at 0x103d67648>)
system didn't exit
```

KeyboardInterrupt Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. The exception inherits from BaseException so as to not be accidentally caught by code that catches Exception and thus prevent the interpreter from exiting.

GeneratorExit Raised when a generator's close() method is called. It directly inherits from BaseException instead of Exception since it is technically not an error.

See Built-in Exceptions

CHAPTER 9

A tour of std libs

Text processing string, re, difflib, textwrap, unicodedata, stringprep, readline, rlcompleter

Binary data services struct, codecs

Data types **datetime**, calendar, **collections**, collections.abc, **heapq**, **bisect**, **array**, **weakref**, **types**, **copy**, pprint, reprlib, enum

Numeric and Math numbers, math, cmath, decimal, fractions, random, statistics

Functional programming itertools, functools, operator

File and directory access pathlib, **os.path**, fileinput, stat, filecmp, **tempfile**, **glob**, fnmatch, linecache, **shutil**, macpath

Data peristence pickle, copyreg, shelve, marshal, dbm, sqlite3

Data compression zlib, gzip, bz2, lzma, zipfile, tarfile

File formats csv, configparser, **netrc**, xdrlib, plistlib

Cryptographic **hashlib**, hmac

OS **os**, **io**, **time**, **argparse**, getopt, **logging**, getpass, curses, **platform**, **errno**, ctypes

Concurrent threading, multiprocessing, concurrent.futures, subprocess, sched, queue, dummy_threading

Networking socket, ssl, select, selectors, asyncio, asyncore, asynchat, signal, mmap

Internet email, json, mailcap, mailbox, mimetypes, base64, binhex, binascii, quopri, uu

Structured markup html, xml.etree, xml.dom, xml.sax

Internet protocols webbrowser, cgi, wsgiref, urllib.request, urllib.response, urllib.parse, urllib.error, urllib.robotparser, http, fptlib, poplib, imaplib, nntplib, smtplib, smtpd, telnetlib, uuid, socketserver, http.server, http.cookies, http.cookiejar, xmlrpc, ipaddress

Multimedia audioop, aifc, sunau, wave, chunk, colorsys, imghdr, sndhdr, ossaudiodev

I18n gettext, locale

Porgram frameworks turtle, cmd, shlex

GUI tkinter

Development tools pydoc, doctest, unittest, unittest.mock, test

Debugging and profiling bdb, faulthandler, pdb, profile, timeit, trace, tracemalloc

Packaing and distribution distutils, ensurepip, venv

Runtime **sys**, sysconfig, **builtins**, __main__, warnings, **contextlib**, **abc**, **atexit**, **traceback**, __future__, gc, inspect, site, fpectl

Custom python interpreters code, codeop

Importing zipimport, pkgutil, modulefinder, runpy, **importlib**

Language parser, ast, symtable, symbol, token, keyword, tokenize, tabnanny, pyclbr, py_compile, compileall, dis, pickletools

Misc formatter

MS msilib, msvcrt, winreg, winsound

Unix posix, pwd, spwd, grp, crypt, termios, tty, pty, fcntl, pipes, resource, nis, syslog

Superseded optparse, imp

Where you can find those standard libs ?

sys.path

/usr/lib/python2.7 /usr/lib/python3.4

9.1 datetime - Basic date and time types

There are two kinds of date and time objects: “naive” and “aware”.

An aware object has sufficient knownledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate oitself relative to other aware objects.

Availabe types

- datetime.date
- datetime.time
- datetime.dateteime
- datetime.timedelta
- datetime.tzinfo: An abstract base class for time zone information objects.
- datetime.timezone: A class that implements the tzinfo abc as a fixed offset from the UTC.

Objects of these types are immutable.

Objects of the date type are always naive.

Naive:

```
>>> import datetime
>>> now = datetime.datetime.now()                    # Date and Time
>>> now
datetime.datetime(2014, 10, 21, 7, 32, 44, 964045)
>>> now.date()
```

```
datetime.date(2014, 10, 21)
>>> now.time()
datetime.time(7, 32, 44, 964045)

>>> day = datetime.date(2012, 12, 12)                # Date
>>> day.year, day.month, day.day, day.isoweekday(), day.isoformat()
(2012, 12, 12, 3, '2012-12-12')
>>> tm = datetime.time(19, 30, 00)                   # Time
>>> tm.hour, tm.minute, tm.second, tm.isoformat()
(19, 30, 0, '19:30:00')

>>> past = dateteime.datetime.now() - now            # Time delta
>>> past, str(past), past.total_seconds()
(datetime.timedelta(0, 615, 431954), '0:10:15.431954', 615.431954)

>>> day.strftime('%b %d %Y %a')                      # Format
'Dec 12 2012 Wed'
>>> timestr = '[Sun Oct 19 08:10:01 2014]'           # Parse
>>> datetime.datetime.strptime(timestr, '[%a %b %d %H:%M:%S %Y]')
datetime.datetime(2014, 10, 19, 8, 10, 1)
```

Aware:

```
>>> beijing = timezone(timedelta(hours=8), 'Asia/Shanghai')
>>> finland = timezone(timedelta(hours=2), 'Europe/Helsinki')

>>> t1 = datetime.datetime(2014, 10, 6, 15, 0, 0, tzinfo=beijing)
>>> str(t1), t1.tzname()
('2014-10-06 15:00:00+08:00', 'Asia/Shanghai')

>>> t2 = t1.astimezone(finland)
>>> str(t2), t2.tzname()
('2014-10-06 09:00:00+02:00', 'Europe/Helsinki')
```

9.2 collections - Container datetypes

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that class a factory function to supply missing values

namedtuple():

```
>>> from collections import namedtuple
>>> Person = namedtuple('Person', ['name', 'age', 'gender'])
>>> bob = Person('Bob', 30, 'male')
>>> jane = Person(name='Jane', gender='female', age=29)
>>> bob, bob[2]
(Persion(name='Bob', age=30, gender='male'), 'male')
>>> type(jane), jane.age
(<class '__main__.Persion'>, 29)
```

```
>>> bob._asdict()
OrderedDict([('name', 'Bob'), ('age', 30), ('gender', 'male')])
>>> bob._replace(name='Tom', age=52)
Persion(name='Tom', age=52, gender='male')
```

```
>>> class Person(namedtuple('Person', ['name', 'age', 'gender'])):
...   __slots__ = ()
...   @property
...   def lastname(self):
...     return self.name.split()[-1]
...
>>> john = Person('John Lennon', 75, 'male')
>>> john.lastname
'Lennon'
```

9.3 deque: double-ended queue

Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Operation	list	Big O	deque	Big O
Add in the head	l.insert(0)	O(n)	d.appendleft	O(1)
Add in the tail	l.append()	O(1)	d.append	O(1)
Del in the head	l.pop(0)	O(n)	d.popleft	O(1)
Del in the tail	l.pop()	O(1)	d.pop	O(1)

```
def timing(initial, setup, testing, times=3):
    print('Testing the following code for {} times ...\n{}'.format(times, testing.
↪strip()))
    namespace = {}
    exec(initial, namespace)

    av = 0
    for i in range(times):
        exec(setup, namespace)

        begin = time.time()
        exec(testing, namespace)
        cost = time.time() - begin

        print('{}: {}'.format(i + 1, cost))
        av += cost
    print('av: {}\n'.format(av / times))
```

```
>>> timing('data = list(range(10**5))', 'l = []', '''
... for i in data:
...   l.insert(0, i)    # O(n)
... ''')
Testing the following code for 3 times ...
for i in data:
  l.insert(0, i)
1: 3.9300358295440674
2: 4.109051704406738
```

```
3: 4.1024134159088135
av: 4.04716698328654
```

```
$ python timing.py
Testing the following code for 3 times ...
for i in data:
  l.insert(0, i)        # O(N)
av: 4.171613295873006

Testing the following code for 3 times ...
for i in data:
  l.append(i)           # O(1)
av: 0.012801011403401693

Testing the following code for 3 times ...
for i in data:
  d.appendleft(i)       # O(1)
av: 0.014629840850830078

Testing the following code for 3 times ...
for i in data:
  d.append(i)           # O(1)
av: 0.014315048853556315

Testing the following code for 3 times ...
for _ in data:
  l.pop(0)              # O(n)
av: 1.6093259652455647

Testing the following code for 3 times ...
for _ in data:
  l.pop()               # O(1)
av: 0.014542102813720703

Testing the following code for 3 times ...
for _ in data:
  d.popleft()           # O(1)
av: 0.011040687561035156

Testing the following code for 3 times ...
for _ in data:
  d.pop()               # O(1)
av: 0.011482477188110352
```

See Time complexity

Chainmap

A ChainMap class is provided for quickly linking a number of mappings so they can be treated as a single unit.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Counter

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> from collections import Counter
>>> words = ['red', 'blue', 'red', 'green', 'blue', 'blue']
>>> cnt = Counter(words)
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
>>> cnt.most_common(2)
[('blue', 3), ('red', 2)]
```

OrderedDict

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> assert list(o.keys()) == sorted(d.keys())
```

defaultdict

Dictionary with default value:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

This technique is simpler and faster than an equivalent technique using dict.setdefault():

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

9.4 heapq - Heap queue algorithm

```
>>> def heapsort(iterable):
...     'Equivalent to sorted(iterable)'
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This module provides an implementation of the min heap queue algorithm, also known as the priority queue algorithm.

9.5 bisect - Array bisection algorithm

```
import bisect
import random

# Use a constant see to ensure that we see
# the same pseudo-random numbers each time
# we run the loop.
random.seed(1)

# Generate 20 random numbers and
# insert them into a list in sorted
# order.
l = []
for i in range(1, 20):
    r = random.randint(1, 100)
    position = bisect.bisect(l, r)
    bisect.insort(l, r)
    print('{:=2} {:=2} {}'.format(r, position, l))
```

9.6 array - Efficient arrays of numeric values

array.array is useful when you need a homogeneous C array of data for reasons other than doing math.

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

See also: bytearray vs array

9.7 weakref - Weak references

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some file objects, generators, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Several built-in types such as list and dict do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

Other built-in types such as tuple and int do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see Weak Reference Support.

weakref.ref

```
import weakref

class BigObject:
    def __del__(self):
        print('Deleting {}'.format(self))

o = BigObject()
r = weakref.ref(o)

print('obj: {}'.format(o))
print('ref: {}'.format(r))
print('r(): {}'.format(r()))

del o
print('r(): {}'.format(r()))
```

```
obj: <__main__.BigObject instance at 0x1036f43f8>
ref: <weakref at 0x1036e5c58; to 'instance' at 0x1036f43f8>
r(): <__main__.BigObject instance at 0x1036f43f8>
Deleting <__main__.BigObject instance at 0x1036f43f8>
r(): None
```

```
...
def callback(ref):
    print('Callback {}'.format(ref))
...
r = weakref.ref(o, callback)
...
```

```
obj: <__main__.BigObject instance at 0x10237a4d0>
ref: <weakref at 0x10236bc58; to 'instance' at 0x10237a4d0>
r(): <__main__.BigObject instance at 0x10237a4d0>
Callback <weakref at 0x10236bc58; dead>
Deleting <__main__.BigObject instance at 0x10237a4d0>
r(): None
```

weakref.proxy

```
p = weakref.proxy(o)
try:
  p.attr
except ReferenceError:
  ...
```

weakref.WeakValueDictionary #TODO

9.8 types - Dynamic type creation and names for built-in types

```
>>> import types
>>> type(lambda : ...) is types.FunctionType
True
```

9.9 copy - Shallow and deep copy operations

```
>>> import copy
>>> class Object: pass
...
>>> l1 = [1, [2, Object()]]
>>> l2 = l1
>>> l3 = copy.copy(l1)
>>> l4 = copy.deepcopy(l1)

>>> l3[0] = 3
>>> l3[1][0] = 4

>>> l1
[1, [4, <__main__.Object object at 0x107d2a278>]]
>>> l2
[1, [4, <__main__.Object object at 0x107d2a278>]]
>>> l3
[3, [4, <__main__.Object object at 0x107d2a278>]]
>>> l4
[1, [2, <__main__.Object object at 0x107d2a978>]]
```

9.10 os.path - Common pathname manipulations

```
>>> import os.path
>>> os.path.sep, os.path.extsep, os.path.pardir, os.path.curdir
('/', '.', '..', '.')

>>> os.path.dirname('/one/two/three'), os.path.basename('/one/two/three')
('/one/two', 'three')
>>> os.path.join('one', 'two', 'three')
'one/two/three'
>>> os.path.splitext('/path/file.ext')
('/path/file', '.ext')

>>> os.path.expanduser('~/file.txt')
'/Users/huanghao/file.txt'            # Mac

>>> os.getcwd()
'/tmp'
>>> os.path.abspath('file.txt')
'/tmp/file.txt'
>>> os.path.realpath('file.txt')
'/tmp/file.txt'
```

```
>>> os.path.isdir('/tmp'), os.path.isfile('/etc/hosts'), os.path.islink('/var'), os.
↪path.exists('/dev'), os.path.ismount('/dev')
(True, True, True, True, True)
```

9.11 tempfile - Generate temporary files and directories

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

9.12 glob - Unix style pathname pattern expansion

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']

>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

9.13 shutil - High-level file operations

- copyfileobj
- copyfile
- copymode
- copystat
- copy
- copy2: Identical to copy() except that copy2() also attempts to preserve all file metadata.
- copytree
- rmtree
- move
- disk_usage
- chown
- which
- make_archive
- unpack_archive
- get_terminal_size

```
>>> shutil.disk_usage(os.path.expanduser('~'))
usage(total=120473067520, used=51554127872, free=68656795648)

>>> shutil.get_terminal_size()
os.terminal_size(columns=130, lines=34)

>>> shutil.which('python3')
'/usr/local/bin/python3'

>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> shutil.make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

9.14 netrc - netrc file processing

:: $ cat ~/.netrc default login huanghao password 123456 machine company.com login hh password xxx

```
>>> import netrc
>>> import os
>>> rc = netrc.netrc(os.path.expanduser('~/.netrc'))

>>> rc.hosts
{'default': ('huanghao', None, '123456'), 'company.com': ('hh', None, 'xxx')}

>>> rc.authenticators('company.com')
('hh', None, 'xxx')

>>> rc.authenticators('home.me')
('huanghao', None, '123456')
```

See also Manual netrc

9.15 hashlib - Secure hashes and message digests

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64

>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

9.16 os - Miscellaneous operating system interfaces

Environments

- name
- uname
- umask
- environ

Process parameters

- getpid: current process id
- getppid: parent's process id
- getpgrp: current process group id
- getpgid(pid): process group id with process id pid

- getuid: real user id of current process
- getgid: real group id of current process
- geteuid: effective user id of current process
- getegid: effective group id of current process
- getgroups: list of supplemental group ids associated with the current process
- getresuid: real, effective, saved
- getresgid:
- getsid: process session id
- getlogin: name of the user logged in
- set*

File descriptor operations

- open
- close
- lseek
- read
- write
- sendfile
- dup
- dup2
- fchmod
- fchown
- fstat
- fsync
- ftruncate
- lockf
- isatty
- openpty
- pipe
- pipe2

Files and directories

- access
- chdir
- chflags
- chmod
- chown
- chroot

- getcwd
- link
- listdir
- mkdir
- makedirs
- mkfifo
- makedev
- major
- minor
- readlink
- remove
- removedirs
- rename
- rmdir
- stat
- symlink
- sync
- truncate
- unlink
- utime
- walk

Process management

- abort
- exec*
- _exit
- forkpty
- kill
- nice
- popen
- spawn*
- system
- times
- wait
- waitpid
- wait3
- wait4

Misc system information

- sep
- linesep
- pathsep
- devnull

9.17 io - Core tools for working with streams

Text and Binary I/O

```
f = io.StringIO("some initial text data")

f = io.BytesIO(b"some initial binary data: \x00\x01")
```

9.18 time - Time access and conversions

```
>>> import time
>>> time.time()
1413910801.16108
>>> time.ctime()
'Wed Oct 22 01:00:03 2014'

>>> time.sleep(.1)

>>> time.clock()
0.194521
```

9.19 argparse - Parser for command-line options, arguments and sub-commands

```
parser = argparse.ArgumentParser()
parser.add_argument('pattern')
parser.add_argument('files', nargs='*')
parser.add_argument('-n', '--line-numerber', action='store_true')
...
namespace = parser.parse_args()
```

9.20 logging - logging — Logging facility for Python

The standard API learned from log4j.

```
import logging

logfile = 'log.out'
```

```
logging.basicConfig(filename=logfile, level=logging.DEBUG)

logging.debug("this message should go to the log file")

logger = logging.getLogger(__name__)
logger.info("this message too")
```

```
$ cat log.out
DEBUG:root:this message should go to the log file
INFO:__main__:this message too
```

9.21 platform - Access to underlying platform's identifying data

```
>>> import platform
>>> platform.python_version_tuple()
('3', '4', '1')
>>> platform.platform()
'Darwin-13.2.0-x86_64-i386-64bit'
>>> platform.uname()
uname_result(system='Darwin', node='huanghao-mpa', release='13.2.0', version='Darwin␣
↪Kernel Version 13.2.0: Thu Apr 17 23:03:13 PDT 2014; root:xnu-2422.100.13~1/RELEASE_
↪X86_64', machine='x86_64', processor='i386')
```

9.22 errno - Standard errno system symbols

```
>>> os.mkdir('/tmp')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: '/tmp'

>>> errno.EEXIST
17

>>> try:
...     os.mkdir('/tmp')
... except OSError as err:
...     if err.errno == errno.EEXIST:
...         print('File exists')
...     else:
...         raise
...
File exists
```

CHAPTER 10

Debugging and Profiling

10.1 Timing

Does list comprehension run faster than map ?

```
$ python -m timeit 'list(map(str, range(10)))'
100000 loops, best of 3: 3.36 usec per loop

$ python -m timeit '[str(i) for i in range(10)]'
100000 loops, best of 3: 4.76 usec per loop
```

See also: Python List Comprehension Vs. Map

Equivalent to:

```
def timeit(setup, statements, repeat=3, number=1000000):
  exec(setup)
  for _ in range(repeat):
    start = time.time()
    for _ in range(number):
      exec(statements)
    cost = time.time() - start
```

Advanced control:

```
# -s: setup statement, run once
# -n: how many times to execute 'statement'
# -r: how many times to repeat the timer(default 3)
# multiple statements

$ python -m timeit -r 5 -n 1000 -s 'data=range(50)' 'l=[]' 'for i in data: l.append(i)
↪'
1000 loops, best of 5: 6.76 usec per loop
```

```
$ python -m timeit -r 5 -n 1000 -s 'data=range(50)' 'l=[None]*50' 'for i in data:␣
↪l[i]=i'
1000 loops, best of 5: 3.71 usec per loop
```

10.2 Profiling

```
$ python -m cProfile chapter10/grep.py pattern .
...
       706259 function calls (673963 primitive calls) in 1.597 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:1000(__init_
↪_)
...
      1    0.000    0.000    1.597    1.597 grep.py:1(<module>)
   3409    0.192    0.000    0.644    0.000 grep.py:15(search_file)
    702    0.001    0.000    0.002    0.000 grep.py:23(colored)
    702    0.005    0.000    0.040    0.000 grep.py:27(print_matches)
   6708    0.147    0.000    0.388    0.000 grep.py:34(is_binary)
      1    0.058    0.058    1.594    1.594 grep.py:45(main)
      1    0.000    0.000    0.003    0.003 grep.py:7(parse_args)
...
 397099    0.128    0.000    0.128    0.000 {method 'search' of '_sre.SRE_Pattern'␣
↪objects}
```

ncalls for the number of calls,

tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall is the quotient of tottime divided by ncalls

cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall is the quotient of cumtime divided by primitive calls

10.3 Tracing

```
$ python -m trace --count grep.py pattern .
...
$ cat grep.cover
  1: import os
  1: import re
...
202:         for i, line in enumerate(file.readlines()):
199:             m = pattern.search(line)
199:             if m:
 13:                 print_matches(name, i+1, line, m)
```

CHAPTER 11

Text processing

11.1 string — Common string operations

3.4

2.7

Constants:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
>>> string.octdigits
'01234567'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?
↪@[\\]^_`{|}~ \t\n\r\x0b\x0c'
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

11.2 Safely eval

```
def convert_string_into_dict(dictstring):
  return eval(dictstring)      # bad idea
```

ast.literal_eval(node_or_string) Safely evaluate an expression node or a string containing a Python expression. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and None.

This can be used for safely evaluating strings containing Python expressions from untrusted sources without the need to parse the values oneself.

11.3 Regular expression

Baisc usage:

```
>>> import re
>>> string = 'abc1234def'
>>> pattern = r'[0-9]+'
>>> re.search(pattern, string)
<_sre.SRE_Match object; span=(3, 7), match='1234'>
```

Syntax

Quiz: explain these meta characters: `.  ^ $ * + ? { } [ ] \ | ( )`

Flags

Raw string

Regex methods

- search
- match: ^ + search
- fullmatch: ^ + search + $
- split: general form of str.split
- findall: returns groups
- finditer: returns iteration of match objects
- sub: general form of str.replace
- escape: match meta chars

Match object

- expand
- group: 0(the whole)
- groups: groups from 1
- groupdict
- start
- end
- span

Example: router.py

See also Regular Expression HOWTO

11.4 Structed text

JSON & YAML & XML

11.4.1 JSON: JavaScript Object Notation

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
Pretty printing:

>>>
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Decoding JSON:

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Specializing JSON object decoding:

```
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Extending JSONEncoder:

```
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Using json.tool from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -mjson.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -mjson.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

See json.org

11.4.2 YAML: YAML Ain't Markup Language

PyYAML

```
>>> import yaml

>>> print yaml.load("""
... name: Vorlin Laruknuzum
... sex: Male
... class: Priest
... title: Acolyte
... hp: [32, 71]
... sp: [1, 13]
... gold: 423
... inventory:
... - a Holy Book of Prayers (Words of Wisdom)
... - an Azure Potion of Cure Light Wounds
... - a Silver Wand of Wonder
```

```
... """)

{'name': 'Vorlin Laruknuzum', 'gold': 423, 'title': 'Acolyte', 'hp': [32, 71],
'sp': [1, 13], 'sex': 'Male', 'inventory': ['a Holy Book of Prayers (Words of Wisdom)
↪',
'an Azure Potion of Cure Light Wounds', 'a Siver Wand of Wonder'], 'class': 'Priest'}

>>> print yaml.dump({'name': "The Cloak 'Colluin'", 'depth': 5, 'rarity': 45,
... 'weight': 10, 'cost': 50000, 'flags': ['INT', 'WIS', 'SPEED', 'STEALTH']})

name: The Cloak 'Colluin'
rarity: 45
flags: [INT, WIS, SPEED, STEALTH]
weight: 10
cost: 50000
depth: 5
```

11.4.3 XML

xml.etree.ElementTree: xmltest.py

XPath syntax:

Syntax	Meaning
tag	Selects all child elements with the given tag. For example: spam, spam/egg
*	Selects all child elements. For example, */egg
.	Selects the current node.
//	Selects all subelements, on all levels beneath the current element. For example, .//egg
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute.
[@attrib='value']	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
[tag]	Selects all elements that have a child named tag. Only immediate children are supported.
[position]	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression last() (for the last position), or a position relative to the last position (e.g. last()-1).

```
# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

Building xml documents:

```
>>> data = ET.Element('data')
>>> jm = ET.SubElement(data, 'artist')
>>> jm.attrib['name'] = 'John Mayer'
>>> j5 = ET.SubElement(data, 'artist')
>>> j5.attrib['name'] = 'John 5'
>>> rock = ET.SubElement(j5, 'genre')
>>> rock.text = 'Instrumental Rock'
>>> ET.dump(data)
<data><artist name="John Mayer" /><artist name="John 5"><genre>Instrumental Rock</
↪genre></artist></data>
```

Also see: lxml

11.5 HTML text

See html - HyperText Markup Language support

PyQuery - a jquery-like lib for python

```
>>> from pyquery import PyQuery as pq
>>> from lxml import etree
>>> d = pq("<html></html>")
>>> d = pq(etree.fromstring("<html></html>"))
>>> d = pq(url='http://google.com/')
>>> d = pq(filename=path_to_html_file)
```

Now d is like the $ in jquery:

```
>>> d("#hello")
[<p#hello.hello>]
>>> p = d("#hello")
>>> print(p.html())
Hello world !
>>> p.html("you know <a href='http://python.org/'>Python</a> rocks")
[<p#hello.hello>]
>>> print(p.html())
you know <a href="http://python.org/">Python</a> rocks
>>> print(p.text())
you know Python rocks
>>> d('p:first')
[<p#hello.hello>]
```

Also see: BeautifulSoup

11.6 Template system

11.6.1 Format

TODO

11.6.2 Jinja2

```
>>> from jinja2 import Template
>>> template = Template('Hello {{ name }}!')
>>> template.render(name='John Doe')
'Hello John Doe!'
```

Variables:

```
{{ foo.bar }}
{{ foo['bar'] }}
```

Filters:

```
{{ name|striptags|title }}
{{ list|join(', ') }}
```

Loop:

```
<ul>
{% for item in seq %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

Tests:

```
{% if loop.index is divisibleby 3 %}
{% if loop.index is divisibleby(3) %}
```

Comments:

```
{# note: disabled template because we no longer use this
  {% for user in users %}
      ...
  {% endfor %}
#}
```

Template Inheritance:

```
# Base template

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
```

```
</body>

# Child template

{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Include:

```
{% include 'header.html' %}
  Body
{% include 'footer.html' %}
```

See Template Designer Documentation

web.py

Also see: Mako

11.7 Lexical and syntax parser

Python Lex-Yacc

navie_lisp.py

CHAPTER 12

Auto-Testing

12.1 doctest

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
```

```
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

```
$ python example.py -v
$

$ python -m doctest example.py
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
...
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
```

12.2 unittest

Important concepts:

test fixture A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case A test case is the individual unit of testing.

test suite A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Example:

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = list(range(10))

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, list(range(10)))

        # should raise an exception for an immutable sequence
        self.assertRaises(TypeError, random.shuffle, (1,2,3))

    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
        with self.assertRaises(ValueError):
            random.sample(self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertTrue(element in self.seq)

if __name__ == '__main__':
    unittest.main()
```

```
$ python example.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK

$ python -v example.py
test_choice (__main__.TestSequenceFunctions) ... ok
test_sample (__main__.TestSequenceFunctions) ... ok
```

```
test_shuffle (__main__.TestSequenceFunctions) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.110s

OK
```

12.2.1 Mocking

record-replay pattern

Mock

- attribures
- methods
- side_effects

MagicMock

NonCallableMock, NonCallableMagicMock

patches

- patch
- with
- patch.object
- patch.dict
- path.multi
- start and stop

sentinel

12.3 pytest

See also: Nose

12.4 Tox

12.5 Selenium

12.5.1 WebDriver

12.5.2 PhantomJS

12.6 Coverage

python-coverage

12.6.1 coversall.io

CHAPTER 13

Tools and Environment

13.1 Development

ipython, Flake8(pep8,pyflakes,mcCabe), pylint, virtualenv(wrapper), autoenv, landscape.io

13.2 Documents

docstring, pydoc, Sphinx, readthedocs

13.3 Auto testing

13.4 Code maintaining and reviewing

github, Code review: Gerrit, github(Pull Request)

13.5 Packaging

setuptools

13.6 Deployment

Ansible, SaltStack, Puppet, fabric

13.7 Release

PyPi, pip

13.8 Monitoring

Psutil, Nagios, Ganglia

13.9 CI

Travis-CI, Jenkins/buildbot

13.10 Agile & DevOp

CHAPTER 14

Concurrenct execution

14.1 threading

14.1.1 Creating threads

- start()
- join()
- ident
- is_alive()
- daemon # TODO

mthreads.py

gui.py

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing or concurrent.futures.ProcessPoolExecutor. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

14.1.2 Thread-local data

localdata.py

```
$ python localdata.py
Child(Thread-1): x =  [0] mydata.x =  33
Child(Thread-1): x =  [33] mydata.x =  33
Main: x =  [33] mydata.x =  0

Child(Thread-2): x =  [33] mydata.x =  88
```

```
Child(Thread-2): x =  [88] mydata.x =  88
Main: x =  [88] mydata.x =  0

Child(Thread-3): x =  [88] mydata.x =  91
Child(Thread-3): x =  [91] mydata.x =  91
Main: x =  [91] mydata.x =  0
```

14.1.3 Locks

Primitive Lock: lock.py

```
$ python lock.py
-------------------- mess
abcdefghijkabclmnodefgpqrhijklstuvwmnoxyzApqrabcdefstuBCDvabcdwxyzAefghiEFGHIJKLMNOPQRSTUVWXYZ
gBChijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghDEFijklmnopqrstuvwxyzGHjklmnopqrstuvIJKLMNABCwxyzDEFGHABCDOPQRIJKLSTUVWEFGHMNOPXYZ
QRSTUVWXYZ
IJKLMNOPQRSTUVWXYZ
-------------------- neat
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Dead lock: deadlock.py

```
$ python deadlock.py
thread1 got the lock1. It will try to get lock2 one second later.
thread2 got the lock2. It will try to get lock1 one second later.
2 children are dead locked
^C
```

Reentrant Lock: reentrant_lock.py

```
$ python reentrant_lock.py
-------------------- Lock
23:41:24 Got a lock for the first time.
23:41:24 It will block self if acquire the same lock again.
23:41:27 False
23:41:27 Fail to acquire the same lock in 3 seconds, so thisline will appear 3␣
↪seconds later.
-------------------- RLock
23:41:27 Got a lock for the first time.
23:41:27 Got the same lock for the second time.
```

14.1.4 Condition

TODO

14.1.5 Semaphore

P and V: acquire and release

producers_and_customers.py

```
$ python producers_and_customers.py
producer1 + : 01 .
producer1 + : 02 ..
producer1 + : 03 ...
producer1 + : 04 ....
producer1 + : 05 .....
producer1 + : 06 ......
producer7 + : 07 .......
producer8 + : 08 ........
producer2 + : 09 .........
customer1 - : 08 ........
customer1 - : 07 .......
customer1 - : 06 ......
customer1 - : 05 .....
customer1 - : 04 ....
producer5 + : 05 .....
producer5 + : 06 ......
producer6 + : 07 .......
customer5 - : 06 ......
customer5 - : 05 .....
```

14.1.6 Event

wait4parent.py

```
$ python wait4parent.py
10:17:48 Child: wait for the event
10:17:48 Parent: wait a moment
10:17:51 Parent: now child can go on
10:17:51 Child: start my job
```

14.1.7 Timer

prompt.py

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start() # after 30 seconds, "hello, world" will be printed
```

14.1.8 Barrier

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)
```

```
def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

14.1.9 Queue

- Queue: FIFO
- LifoQueue: LIFO
- PriorityQueue: The lowest valued entries are retrieved first

worker.py: a simpler solution of producers and customers question

14.2 multiprocessing

Global Interpreter Lock

Process(target, args)

- start()
- run()
- join()
- name
- is_alive()
- daemon
- pid
- exitcode
- terminate()

Programming guidelines

14.2.1 Contexts and start methods

- spawn
- fork
- forkserver

14.2.2 Exchanging objects

Queue

Pipe

SimpleQueue

Connection

Listeners and Clients

Authentication keys

14.2.3 Synchronization

Lock

Barrier

BoundedSemaphore

Condition

Event

Semaphore

14.2.4 Sharing states

Shared memory

Server process

Shared ctypes Objects

Managers

Proxy

Cleanup

14.2.5 Pool of workers

14.3 subprocess

This module intends to replace several older modules and functions:

os.system os.spawn*

Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=())

- poll()
- wait()
- communicate()
- send_signal()
- terminate()
- kill()
- args
- stdin

- stdout
- stderr
- pid
- returncode
- call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
- check_call
- check_output

Replace older functions with subprocess

shlex

14.4 concurrent

ThreadPoolExecutor

ProcessPoolExecutor

Future

future/promise

14.5 sched

14.6 python-daemon

Correct daemon behaviour

According to Stevens in *[stevens]* §2.6, a program should perform the following steps to become a Unix daemon process.

- Close all open file descriptors.
- Change current working directory.
- Reset the file access creation mask.
- Run in the background.
- Disassociate from process group.
- Ignore terminal I/O signals.
- Disassociate from control terminal.
- Don't reacquire a control terminal.
- Correctly handle the following circumstances:
- Started by System V init process.
- Daemon termination by SIGTERM signal.
- Children generate SIGCLD signal.

See: PEP 3143 - Standard daemon process library

14.7 supervisor

http://supervisord.org/

CHAPTER 15

Networking

15.1 ipaddress - IPv4/IPv6 manipulation library

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')

>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')

>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True

>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]

>>> for addr in IPv4Network('192.0.2.0/28'):
...   addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
```

```
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')

>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

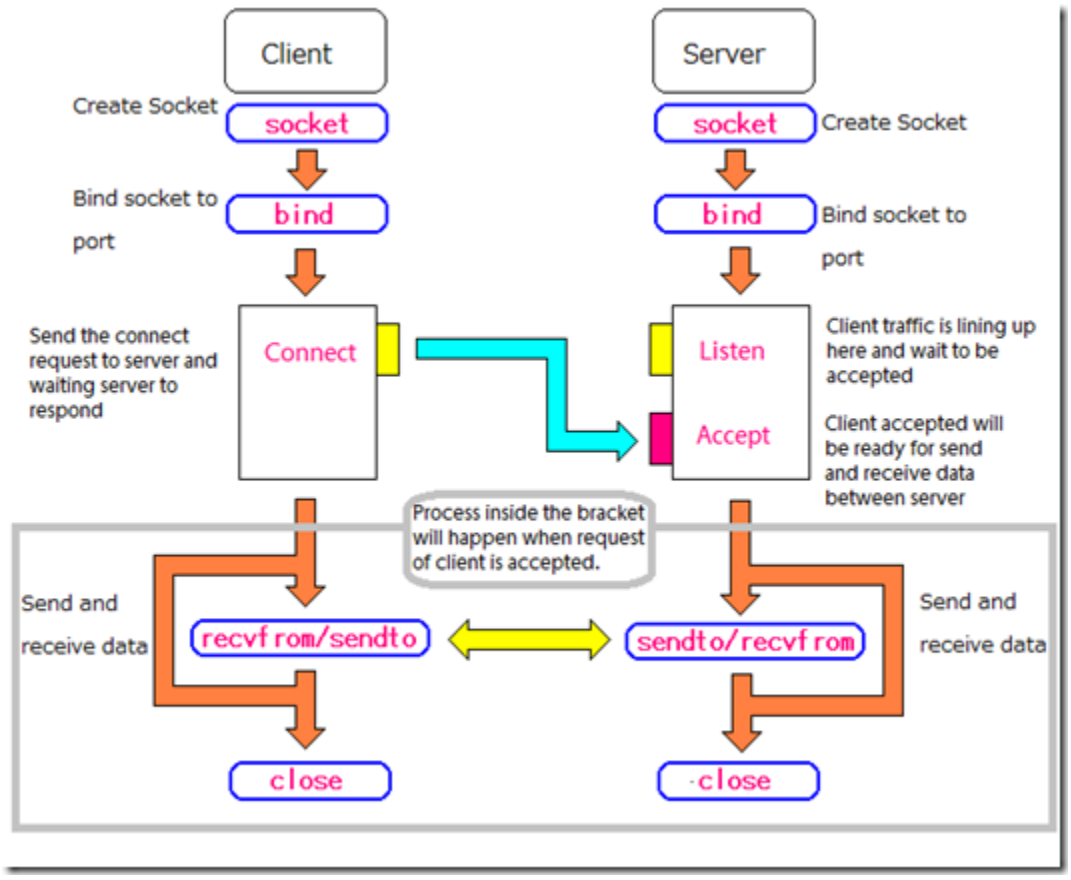
15.2 socket

15.2.1 APIs

TCP http/https, ftp/sftp, smtp, imap, pop3, ssh

UDP dns, nfs, dhcp, rip

- socket
- bind
- connect
- listen
- accept
- recv
- send
- close



server1.py and (client1.py or browser)

ab - Apache HTTP server benchmarking tool:

```
$ ab -n 5 -c 5 http://localhost:8000/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient).....done


Server Software:
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        58 bytes

Concurrency Level:      5
Time taken for tests:   15.007 seconds
Complete requests:      5
Failed requests:        0
Write errors:           0
Total transferred:      510 bytes
HTML transferred:       290 bytes
Requests per second:    0.33 [#/sec] (mean)
Time per request:       15006.968 [ms] (mean)
Time per request:       3001.394 [ms] (mean, across all concurrent requests)
```

```
Transfer rate:          0.03 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       1
Processing:  3001 9003 4745.5  10504   15006
Waiting:        1 6002 4745.2   7503   12005
Total:       3001 9004 4745.6  10505   15007

Percentage of the requests served within a certain time (ms)
  50%   9004
  66%  12005
  75%  12005
  80%  15007
  90%  15007
  95%  15007
  98%  15007
  99%  15007
 100%  15007 (longest request)
```

15.2.2 Multithreading or processing

Apache

server2.py:

```
ab -n 5 -c 5 http://localhost:8000/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient).....done


Server Software:
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        58 bytes

Concurrency Level:      5
Time taken for tests:   3.004 seconds
Complete requests:      5
Failed requests:        0
Write errors:           0
Total transferred:      510 bytes
HTML transferred:       290 bytes
Requests per second:    1.66 [#/sec] (mean)
Time per request:       3004.066 [ms] (mean)
Time per request:       600.813 [ms] (mean, across all concurrent requests)
Transfer rate:          0.17 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:  3002 3003   0.5   3003    3004
```

```
Waiting:        1    2   0.7      2       3
Total:       3003 3003   0.4   3003    3004

Percentage of the requests served within a certain time (ms)
  50%   3003
  66%   3003
  75%   3003
  80%   3004
  90%   3004
  95%   3004
  98%   3004
  99%   3004
 100%   3004 (longest request)
```

15.2.3 Multiplexing

server3.py

- select
- poll
- epoll
- kqueue

Blocking & Non-Blocking

Nginx, NodeJS, Twisted/Tornado

See: The C10K problem

15.3 http

server side:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

httpd = socketserver.TCPServer(("", PORT), Handler)

print("serving at port", PORT)
httpd.serve_forever()
```

```
$ python -m http.server 8000

$ python -m http.server 8000 --bind 127.0.0.1
```

client side:

http.client:

```
>>> import http.client
>>> conn = http.client.HTTPConnection("www.python.org")
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
```

requests & requests-cache:

```
>>> import requests
>>> res = requests.get('http://www.python.org')
>>> res.status_code, res.reason
(200, 'OK')
>>> res.content[:100]
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9"> ␣
↪ <![endif]-->\n<!-'

>>> import requests_cache
>>> requests.install_cache()
>>> import requests
>>> requests.get('http://www.python.org')    # a bit slow
<Response [200]>
>>> requests.get('http://www.python.org')    # very quick
<Response [200]>
```

```
$ ls cache.sqlite
cache.sqlite
```

15.4 xmlrpc

server side:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
server = SimpleXMLRPCServer(("localhost", 8000),
                            requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x,y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
```

```
# published as XML-RPC methods (in this case, just 'mul').
class MyFuncs:
    def mul(self, x, y):
        return x * y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

client side:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

Celery

15.5 Twisted

- gevent
- python-eventlet
- tulip(asyncio)
- Scrapy

CHAPTER 16

Databases

16.1 SQL

16.1.1 101

creation

CREATE TABLE table_name(field type, ..)

types

int, float, text, blob, varchar, char, datetime, date, time

autoincrement

querying

SELECT field, .. FROM table WHERE field = value, .. ORDER BY field GROUP BY field HAVING field ..

WHERE: filter by row ORDER BY: fields DESC, ASC GROUP BY: fields HAVING: filter by group

modification

INSERT INTO table(field, ..) VALUES(value, ..), ..

UPDATE table SET field = value, .. WHERE field = value and . . . or . . .

REPLACE table

DELETE FROM table WHERE field = value

transaction

BEGIN, COMMIT, ROLLBACK

autocommit

keys

primary key, condidate key, foreign key

index, unique, combined indexes(keys)

relationship

1:1, 1:n, n:n

16.1.2 sqlite

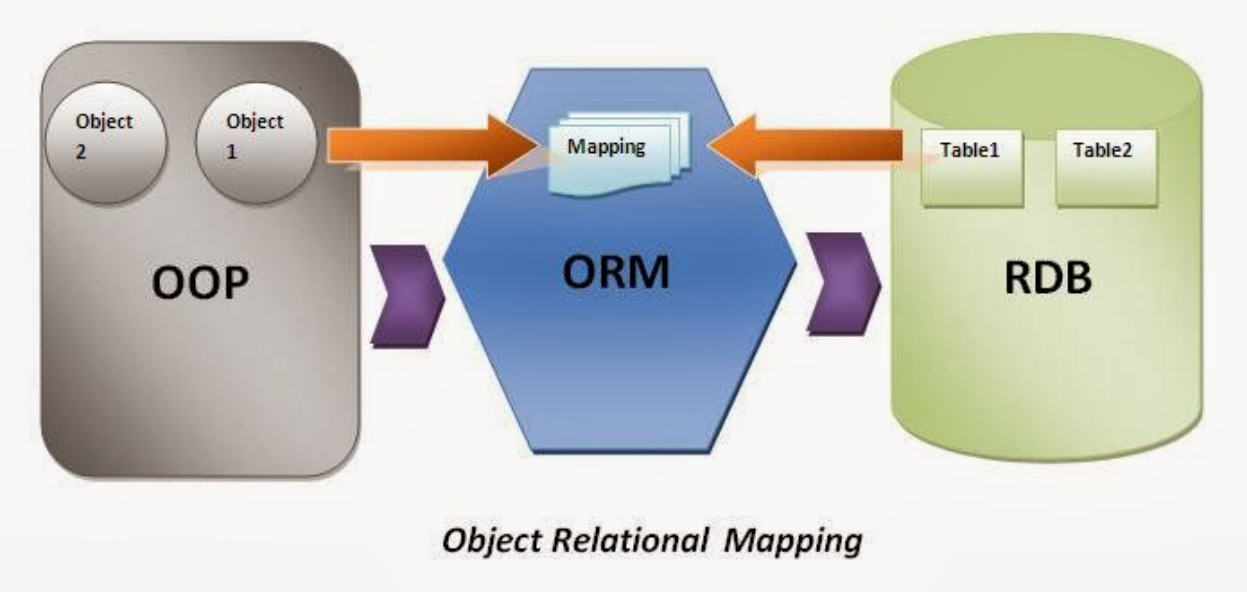
```
>>> import sqlite3
>>> conn = sqlite3.connect('cache.sqlite')
>>> c = conn.cursor()
>>> c.execute('select key from urls')
<sqlite3.Cursor object at 0x103093f80>
>>> c.fetchone()
('89b1b81005c639109c2248db8161bb0b903ad117561e28a162c3e55b7e5d6ca8',)

>>> for row in c.execute('select key from urls'): print(row)
...
('89b1b81005c639109c2248db8161bb0b903ad117561e28a162c3e55b7e5d6ca8',)
('c84998697121613e70ae1e68a5ba515718cb78c82b711cd337a1669baf8d1c66',)
```

```
# console 1
>>> conn = sqlite3.connect('test')
>>> c = conn.cursor()
>>> c.execute('create table test(key int, value varchar(32))')
<sqlite3.Cursor object at 0x102828f80>
>>> c.execute("insert into test values(1, 'a'), (2, 'b')")
<sqlite3.Cursor object at 0x102828f80>

# console 2
$ sqlite3 test
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
test
sqlite> select * from test;

# console 1
>>> conn.commit()

# console 2
sqlite> select * from test;
1|a
2|b
```

16.1.3 MySQLdb

16.1.4 PostgreSQL

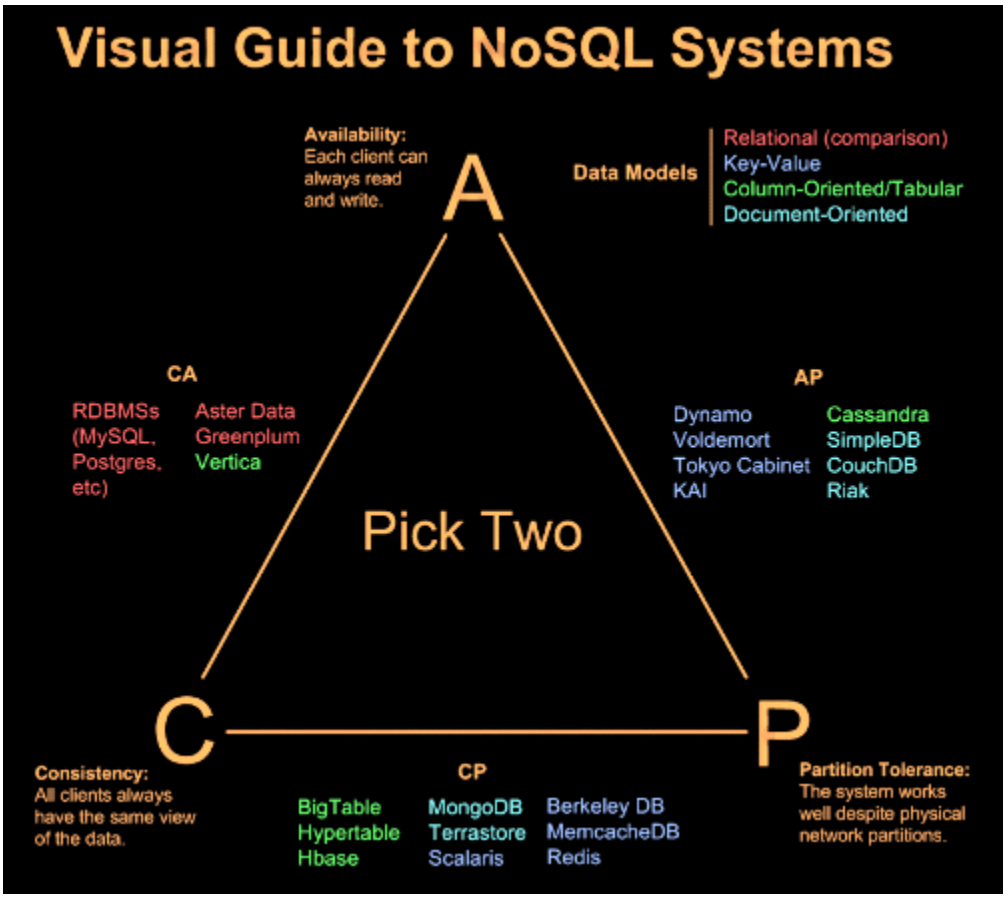
16.2 ORM



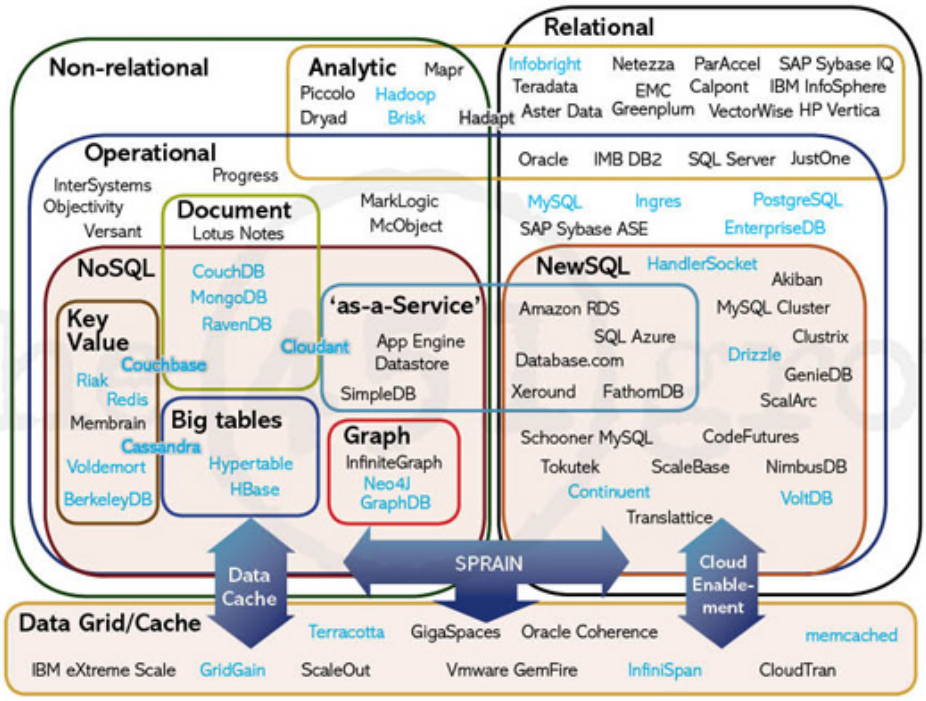
16.2.1 SQLAlchemy

16.2.2 Django ORM

16.3 Relation DB vs. NoSQL



See CAP theorem



16.3.1 Mongo

16.3.2 Redis

16.3.3 Memcache

CHAPTER 17

Message Queues

17.1 RabbitMQ

pika, amqplib

17.2 Redis

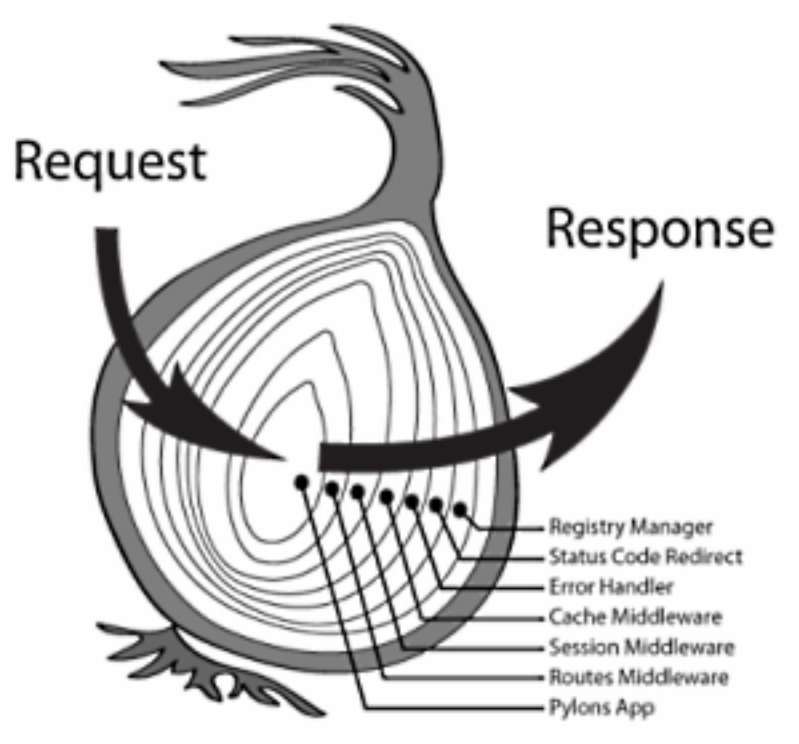
17.3 ZeroMQ

17.4 Celery

CHAPTER 18

Web

18.1 WSGI



Also see: paste

18.2 Django

18.3 Flask

18.4 Tornado

Also see: bottle/webpy/cherrypy

CHAPTER 19

Migrating to Python3

six 2to3

CHAPTER 20

Resources

Official

- Docs
- Tutorial
- Library Reference
- What's new ?

Articles

- The Hitchhiker's Guide to Python!
- PyMOTW - Python Module of the Week

PEP

- http://legacy.python.org/dev/peps/
- http://legacy.python.org/dev/peps/pep-0008/
- http://legacy.python.org/dev/peps/pep-0020/
- http://legacy.python.org/dev/peps/pep-3000/

pyCon

- http://www.pycon.org/
- http://pyvideo.org/

Groups & BBS

News

Books

- Learning Python
- Text Processing in Python
- The Python Standard Library by Example

- Expert Python Programming
- Python

CHAPTER 21

Indices and tables

- genindex
- modindex
- search

Bibliography

[stevens] Unix Network Programming, W. Richard Stevens, 1994 Prentice Hall.