

Python 102

Overview

Course: Python 102 tutorial, PyCon 2010, Atlanta

Presenter: Stuart Williams (stuart@swilliams.ca)

Intended audience: Programmers who want a fast introduction to intermediate and some advanced features of Python.

Tutorial format: Frequently alternating presentation of concepts and exercise sets. Each pair of concepts and exercises ranges in length from 5 minutes on simple topics, to 20 minutes on more involved topics.

Requirements: A laptop computer with Python 2.6 (or 3.1) installed.

License: This PyCon 2010 *Python 102* Tutorial by Stuart Williams is licensed under a Creative Commons Attribution-Share Alike 2.5 Canada License (<http://creativecommons.org/licenses/by-sa/2.5/ca/>).

Strategy

You'll learn by seeing and doing. The interactive Python interpreter is used very heavily.

We'll use Python 2.6 but with two "import `__future__`" features from 3.0.

Early exercises are exploration via the interpreter. Later sections will have more traditional "do this" exercises.

In order to be suitable for a wide range of levels of experience and ability there are some examples and exercises that some or most of you won't figure out in the time we have today. Don't be discouraged. If you follow 50% you're doing great and you can try the harder ones after the course. Very few will follow 100%.

For each topic I'll demonstrate with a lot of of examples.

Expect some examples to fail. This is intentional to help you learn how Python handles errors and to learn some of its boundaries.

I am not providing the text of these exercises online because by typing them yourselves you will learn more.

Feel free to interrupt with questions.

Be patient if I delay my answer to a question based on my planned outline.

Numbers, etc.

Start the interpreter

```
>>> 0 0
>>> 1 + 2 1
>>> 1 * 2 2
>>> 1 - 2 3
>>> 1 / 2 4
>>> 1 / 2.0 5
>>> 1 // 2.0 6
>>> 1 / 2 7
>>> from __future__ import division 8
>>> 1 / 2 9

>>> int(4.3) 10
>>> int('4') 11
>>> float('4') 12
>>> long(4) 13
>>> 4 == 4L 14
>>> 4 is 4L 15

>>> 1 + 2 16
>>> 1 + 2.0 17
>>> 1 + 2L 18
>>> 1.0 + 2L 19

>>> coerce(1, 2) 20
>>> coerce(1, 2.0) 21
>>> coerce(1, 2L) 22
>>> coerce(1.0, 2L) 23

>>> 1 == 2 24
>>> 1 == 1 25
>>> 1 == 1 == 1 26
>>> True == 1 27
>>> 1 != 2 28
>>> 1 > 2 29
>>> 1 < 2 30
>>> 2 < 1 31
>>> cmp(1, 2) 32
>>> cmp(2, 1) 33
>>> cmp(1, 1) 34

>>> a = 0 35
>>> a.__cmp__(0) 36
>>> a.__cmp__(1) 37
>>> int(0).__cmp__(0) 38
>>> int(0).__cmp__(1) 39
>>> int(1).__cmp__(0) 40
>>> cmp(1, 0) 41
```

>>> 1 < 2	42
>>> 1 < 2 < 3	43
>>> 1 < 2 < 1	44
>>> 0 != 1	45
>>> a = 1	46
>>> a	47
>>> a++	48
>>> a	49
>>> a += 1	50
>>> a	51
>>> x, y, z = 0, 0, 0	52
>>> x	53
>>> y	54
>>> z	55
>>> x, y, z	56
>>> x = y = z = 1	57
>>> x, y, z	58
>>> 2 + 3j	59
>>> 1 / 3.0	60
>>> from decimal import Decimal, getcontext	61
>>> Decimal(1) / Decimal(3)	62
>>> getcontext().prec = 2	63
>>> Decimal(1) / Decimal(3)	64
>>> abs(-4)	65
>>> round(1 / 3.0, 2)	66
>>> round(1234, -2)	67
>>> 2 ** 32	68
>>> ord('a')	69
>>> ord('z') - ord('a')	70
>>> ord('a')	71
>>> chr(97)	72
>>> chr(ord('a'))	73
>>> print 3	74
>>> print(3)	75
>>> from __future__ import print_function	76
>>> print 3	77
>>> print(3)	78
>>> import operator	79
>>> operator.add(7, 3)	80
>>> print(operator.__doc__)	81
>>> help(operator)	82

Exercises: Numbers

>>> 14 / 12	83
>>> 14 % 12	84
>>> divmod(14, 12)	85
>>> print(divmod.__doc__)	86
>>> hex(16)	87
>>> oct(16)	88
>>> 0xff	89
>>> 0o377	90
>>> 0b11111111	91
>>> 2 + 3j + 4 + 3j	92
>>> print(abs.__doc__)	93
>>> print(hex.__doc__)	94
>>> print(oct.__doc__)	95
>>> print(pow.__doc__)	96
>>> print(round.__doc__)	97
>>> import math	98
>>> help(math)	99
>>> dir(math)	100
>>> print(' '.join(dir(math)))	101
>>> print('\n'.join(dir(math)))	102
>>> import operator	103
>>> dir(operator)	104
>>> print(operator.pow.__doc__)	105
>>> operator.pow	106
>>> operator.pow(2)	107
>>> operator.pow(2, 8)	108
>>> operator.__pow__ == operator.pow	109
>>> dir(0)	110
>>> int(0).__class__	111
>>> str(0)	112
>>> str(0).__class__	113
>>> int(0).__str__()	114
>>> 0.__str__()	115
>>> (0).__str__()	116
>>> int.__div__	117
>>> int.__div__ == int.__truediv__	118
>>> int.__div__ = int.__truediv__	119

String Literals, Operators, Immutability

>>> 'hello'	120
-------------	-----

>>> "hello"	121
>>> "today's the day"	122
>>> '''today's the "day"'''	123
>>> 10 * ' '	124
>>> 'with' + 'hold'	125
>>> 'with' 'hold'	126
>>> 'o' in 'hello'	127
>>> s = 'jello'	128
>>> s[0] = 'h'	129
>>> s	130
>>> s = 'h' + s[1:]	131
>>> s	132
>>> print(id.__doc__)	133
>>> s = 'hello'	134
>>> t = 'hello'	135
>>> id(s) == id(t)	136

Exercises: String Literals, Operators, Immutability

>>> u'hello'	137
>>> u'hello' == 'hello'	138
>>> u'hello' is 'hello'	139
>>> help('is')	140
>>> import operator	141
>>> print(operator.is.__doc__)	142
>>> type(u'hello')	143
>>> print(u'hello'.__doc__)	144
>>> u = u'hello'	145
>>> u.__class__	146
>>> u.__class__.__bases__	147
>>> u.__class__.__bases__	148
>>> u.__class__.__bases__[0].__bases__	149
>>> object.__base__	150
>>> ('This is a very long'	151
... ' string, is it not?')	
>>> '\a'	152
>>> len('\a')	153
>>> '\a' == '\x07'	154
>>> len('\b')	155
>>> len('\c')	156
>>> len('\d')	157
>>> '\c'	158

```

>>> u'h' 159
>>> u2 = u'$' 160
>>> u2 161
>>> u3 = u'\u0024' 162
>>> u3 163
>>> u4 = u'\N{dollar sign}' 164
>>> u2 == u2 == u3 165
>>> u2.encode() 166
>>> u2.encode() == '$' 167

>>> u5 = u'Pound sign:\N{pound sign}:' 168
>>> u5.encode() 169
>>> u5.encode('ascii') 170
>>> u5.encode('ascii', 'ignore') 171
>>> u5.encode('ascii', 'replace') 172

```

String Methods

```

>>> len('hello') 173
>>> 'hello'.startswith('h') 174
>>> 'hello'.endswith('o') 175

>>> 'hello world'.capitalize() 176
>>> 'hello world'.title() 177
>>> 'hello world'.title().swapcase() 178

>>> ' hello '.strip() 179
>>> ' hello '.rstrip() 180
>>> ' hello '.lstrip() 181

>>> 'hello'.count('h') 182
>>> 'hello'.count('l') 183

>>> 'hello'.index('el') 184
>>> 'hello'.index('z') 185
>>> 'hello'.find('el') 186
>>> 'hello'.find('z') 187
>>> 'hello'.find('h') 188
>>> 'hello'.find('e') 189
>>> 'hello'.find('l') 190

>>> 'hello'.isalnum() 191
>>> 'hello there'.isalnum() 192
>>> 'hello'.center(20) 193
>>> 'hello'.center(20, '-') 194

>>> 'hello'.replace('h', 'z') 195
>>> 'steep'.replace('e', '3') 196
>>> 'steep'.replace('e', '3', 1) 197

```

Exercises: String Methods

>>> dir('hello')	198
>>> dir(str)	199
>>> dir(str) == dir('hello')	200
>>> help(str)	201
>>> help(str.isalpha)	202
>>> help(str.isdigit)	203
>>> help(str.islower)	204
>>> help(str.isspace)	205
>>> help(str.istitle)	206
>>> help(str.isupper)	207
>>> 'hello'.find('e')	208
>>> 'hello'.find('l')	209
>>> 'hello'.rfind('l')	210
>>> 'hello'.rindex('l')	211
>>> 'yellow is mellow'.find('ow')	212
>>> 'yellow is mellow'.find('ow', 6)	213
>>> 'hello'.ljust(20, '*')	214
>>> 'hello'.rjust(20, '+')	215

What does the following expression check for?

```
s.index('e', 0, s.index(' '))
```

Hint: See `help(str.index)` and try substituting 'hello world' for `s` in the expression.

Write an expression to check for more spaces than non-spaces in a string.

>>> 'Wait...'.rstrip('.')	216
>>> list('hello')	217

String Formatting

>>> locals()	218
>>> who = 'Professor Plum'	219
>>> where = 'kitchen'	220
>>> what = 'knife'	221
>>> locals()	222
>>> '{what}'.format(what='knife')	223
>>> '{what} in {where}'\n... .format(what='knife',\n... where='kitchen')	224

```

>>> 'with {what}'.format(locals()) 225
>>> 'with {0[what]}'.format(locals()) 226
>>> 'with {0[what]}'.format( 227
...     dict(what='knife',
...     where='kitchen'))

>>> 'arg is {0}'.format( 228
...     dict(what='knife',
...     where='kitchen'))

>>> 'with {what}'.format(locals()) 229
>>> 'with {what}'.format(**locals()) 230

>>> import sys 231
>>> print(sys.version_info) 232
>>> 'major {0[0]}, minor {0[1]}'.format( 233
...     sys.version_info)

>>> sys.byteorder 234
>>> 'Byte order: {0.byteorder}' \ 235
...     .format(sys)

```

Exercises: String Formatting

```

>>> s2 = 'with the {0} ' 236
>>> s2.format('knife') 237
>>> s3 = 'in the {1}' 238
>>> s3.format('kitchen') 239
>>> s = s2 + s3 240
>>> s 241
>>> s.format('knife') 242
>>> s.format('knife', 'kitchen') 243
>>> s2 244
>>> s3 245
>>> s2.format('knife', 'kitchen') 246
>>> s3.format('knife', 'kitchen') 247

>>> s2 = 'with the {what} ' 248
>>> s2.format('knife') 249
>>> s2.format(what='knife') 250
>>> '{0} {what}'.\ 251
...     format('kitchen', what='knife')
... s3 = 'in the {where}'
... s = s2 + s3
... s
... s.format(what='knife', where='kitchen')
... s.format(where='kitchen', what='knife')
... '{what} {where} {what}'.format(
...     where='kitchen', what='knife')

```

Use format to print the bytearray and maxint from the sys module. Then add the 4th element of sys.version_info to the string.

Introspection

```
>>> type('hello') 252
>>> 'hello'.__class__ 253
>>> 'hello'.__doc__ 254
>>> print(str.__doc__) 255

>>> print(dir.__doc__) 256
>>> dir(str) 257
>>> dir(str.strip) 258
>>> callable(str.strip) 259
>>> callable(str) 260

>>> str.__dict__ 261
>>> list(str.__dict__) 262
>>> print(vars.__doc__) 263
>>> vars(str) 264
>>> list(vars(str)) 265
>>> list(vars(str)) == list(str.__dict__) 266

>>> len(dir(str)) 267
>>> len(list(str.__dict__)) 268
>>> set([1, 2, 3, 3]) 269
>>> set([1, 2, 3]) - set([1, 2]) 270
>>> set(dir(str)) - set(list(str.__dict__)) 271
>>> print(dir.__doc__) 272

>>> import __builtin__ 273
>>> dir(__builtin__) 274

>>> str('hello') 275
>>> repr('hello') 276
>>> eval(str('hello')) 277
>>> eval(repr('hello')) 278

>>> 'hello'.__class__ 279
>>> type('hello') 280
>>> u'hello' 281
>>> u'hello'.__class__ 282
>>> type(u'hello') 283
>>> print('hello'.__doc__) 284
```

Note class unicode(basestring) which documents the inheritance.

```

>>> u = u'hello' 285
>>> u.__class__ 286
>>> u.__class__.__bases__ 287
>>> u.__class__.__bases__[0].__bases__ 288

```

unicode -> basestring -> object -> None

```

>>> basestring('hello') 289

```

Exercises: Introspection

```

>>> x = 4.3 290
>>> str(x) 291
>>> x.__str__() 292
>>> repr(x) 293
>>> x.__repr__() 294

>>> s = 'hello' 295
>>> s.__class__ 296
>>> s.__class__.__bases__ 297

```

Where is `KeyError`, what is it, and what are its parent classes?

Strings have what predicate methods? (e.g. `isspace` and `iscapital`). Produce a list of them.

Many of the predicate methods have a corresponding conversion method, e.g. `'Hello'.lower()` returns a string for which `islower` is true (i.e. `'Hello'.lower().islower() == True`). Use introspection to generate a list of these pairings, i.e. the methods whose names start with `'is'` paired with the conversion method without the `'is'` prefix.

Tuples and Lists

```

>>> m = [1, 2, 3] 298
>>> m.count(3) 299
>>> t = (1, 2, 3) 300
>>> t.count(3) 301
>>> u = (1, 2, 3) 302
>>> t == u 303
>>> t is u 304
>>> m.count(5) 305
>>> m.index(5) 306
>>> m.insert(0, 6) 307
>>> m 308
>>> m.index(6) 309
>>> m.index(6, 5) 310
>>> m 311

```

>>> m.insert(4, 'e')	312
>>> m	313
>>> m.remove('e')	314
>>> m	315
>>> m.remove('e')	316
>>> m = range(6)	317
>>> m.reverse()	318
>>> m	319
>>> m.sorted()	320
>>> m	321
>>> sorted(m)	322
>>> m	323
>>> sorted(m, reverse=True)	324
>>> m	325
>>> m.sort()	326
>>> m	327
>>> m.sort(reverse=True)	328
>>> m	329

Exercises: Tuples and Lists

>>> m = range(4)	330
>>> m	331
>>> m.append(5)	332
>>> m	333
>>> m.extend([6, 7, 8])	334
>>> m	335
>>> r = m.pop()	336
>>> r, m	337
>>> m.pop(), m	338
>>> m.pop(), m	339
>>> m.pop(), m	340
>>> m	341
>>> m.extend([4, 5, 6])	342
>>> m	343
>>> m.pop(0), m	344
>>> m.pop(0), m	345
>>> m	346
>>> m.append([7, 8, 9])	347
>>> m	348
>>> m.append(('a', 'b'))	349
>>> m	350

How can we access a `list` as a LIFO (Last In, First Out) stack? How can we access a `list` as a FIFO (First In, First Out) queue?

More String Methods

```
>>> ','.join(['one', 'two', 'three']) 351
>>> ', '.join(['one', 'two', 'three']) 352
>>> 'www.test.com'.partition('.') 353
>>> 'hello there'.partition('.') 354
>>> 'www.test.com'.rpartition('.') 355
>>> 'hello there'.split() 356
>>> 'hello there world'.split() 357
>>> 'www.python.org'.split('.') 358
>>> 'www.python.org'.split('.', 1) 359
>>> 'www.python.org'.rsplit('.', 1) 360
>>> lines = 'hello\nthere\nworld' 361
>>> lines.split() 362
>>> lines = lines + '\n' 363
>>> lines 364
>>> lines.split() 365
>>> lines.split('\n') 366
>>> lines.splitlines() 367
>>> lines.splitlines(True) 368
>>> lines.splitlines() 369
```

Exercises: More String Methods

```
>>> ('one two three').split() 370

>>> ', '.join('one two three'.split()) 371

>>> ('one two three'.replace(' ', ', ')) 372

>>> m = range(20) 373
>>> m 374
>>> n = m[:] 375
>>> n == m 376
>>> n is m 377
>>> n[0] = 'first' 378
>>> n == m 379
>>> m 380
>>> m[0:-1:2] 381
>>> m[::2] 382
>>> m[::-2] 383
>>> m[::-1] 384

>>> del m[0:-1:2] 385
>>> m 386
>>> m = range(16) 387
>>> m 388
```

```

>>> ['low even'] * 5 389
>>> m[0:10:2] = ['low even'] * 5 390
>>> m 391
>>> del m[0:10:2] 392
>>> m.index(10) 393
>>> m[0:m.index(10)] 394
>>> m[0:m.index(10)] = [] 395
>>> m 396
>>> s1 = slice(2, 4) 397
>>> m[s1] 398
>>> s1 = slice(0, 4, 2) 399
>>> m[s1] 400

```

Nested Sequences

```

>>> m = [ ['one', 'two', 'three'], 401
...       ['One', 'Two', 'Three']]

>>> m 402
>>> m[0] 403
>>> m[1] 404
>>> m[0][0] 405

>>> month_days = [['Jan', 31], 406
...               ['Feb', 28], ['Mar', 31]]
... month_days
... month_days = (('Jan', 31),
...               ('Feb', 28), ('Mar', 31))
... month_days

```

Decorate, Sort, Undecorate (DSU) and Alternatives

```

>>> months = [('Jan', 1, 31), 407
...           ('Feb', 2, 28),
...           ('Mar', 3, 31),
...           ('Apr', 4, 30)]

>>> dsu = [(days, (name, order, days)) 408
...         for (name, order, days) in months]

>>> dsu 409
>>> dsu.sort() 410
>>> dsu 411
>>> [ b for (a, b) in dsu ] 412

```

```

>>> print(min.__doc__) 413
>>> from operator import itemgetter 414
>>> print(itemgetter.__doc__) 415
>>> days = itemgetter(2) 416
>>> min(months, key=days) 417
>>> max(months, key=days) 418

```

Exercises: DSU Alternative

Sort months alphabetically with the `sorted` function (or the `list.sort` method).

Sort months by the number of days in each.

List Comprehensions and Functional Programming

```

>>> range(8) 419
>>> [element for element in range(8)] 420
>>> [2 * element for element in range(8)] 421
>>> [2 + element for element in range(8)] 422

>>> [e for e in range(8) if e % 3 == 0] 423

>>> [(c, j) 424
...     for c in ['a', 'b']]
...     for j in range(4)]

>>> [(i, j) 425
...     for i in range(10)
...     if i % 2 == 0
...     for j in range(i)
...     if j % 2 != 0]

>>> it = (2 * i for i in range(10)) 426
>>> it 427
>>> list(it) 428

```

Exercises: List Comprehensions and Functional Programming

```

>>> def iseven(n): 429
...     return n % 2 == 0

```

```

>>> [e for e in range(10) if iseven(e)] 430
>>> filter(iseven, range(10)) 431
>>> print(filter.__doc__) 432
>>> map(iseven, range(10)) 433
>>> print(map.__doc__) 434

>>> import operator 435
>>> from functools import reduce 436
>>> reduce(operator.add, range(10)) 437
>>> reduce(operator.mul, range(10)) 438
>>> reduce(operator.mul, range(1, 10)) 439
>>> print(reduce.__doc__) 440

```

Write a list comprehension or generator expression with three loops.

Objects

Everything in Python is an object and has:

- a single *id*,
- a single *value*,
- some number of *attributes* (part of its value),
- a single *type*,
- (zero or) one or more *names* (in one or more namespaces),
- and usually (and indirectly), one or more *base classes*.

```

>>> dir() 441
>>> s = 'xyz' 442
>>> dir() 443
>>> id(s) 444
>>> type(s) 445
>>> s 446

>>> t = 'xyz' 447
>>> dir() 448
>>> id(s) == id(t) 449

>>> u = t 450
>>> dir() 451
>>> id(u) == id(t) 452

>>> m = [1, 2, 3] 453
>>> m 454
>>> dir() 455
>>> m[1] = 20 456

```

```

>>> m 457
>>> m[1] = s 458
>>> m 459
>>> id(s) == id(m[1]) 460
>>> m[1] = 'xyz' 461
>>> m 462
>>> id(s) == id(m[1]) 463
>>> n = m 464
>>> dir() 465
>>> id(n) == id(m) 466
>>> m 467
>>> m[1] = 'abc' 468
>>> m 469
>>> n 470
>>> m == n 471

```

Exercises: Objects

If `m` is just a name then how is the `list` object it refers to mutated?

```

>>> m = range(10) 472
>>> m 473
>>> m[0] 474
>>> m.__getitem__(0) 475
>>> m[1] = 'one' 476
>>> m 477
>>> m.__setitem__(2, 'two') 478
>>> m 479
>>> m[0:10:2] 480
>>> m.__getitem__(slice(0, 10, 2)) 481
>>> m[3] = 'three' 482
>>> m 483
>>> m.__setitem__(slice(4, 6), 484
...     ['four', 'five'])
>>> m 485

```

Dictionaries (1)

```

>>> d = dict([(k, v) for v, k in 486
...     enumerate(
...     'zero one two'.split())])
>>> d 487
>>> s = 'zero one two' 488

```

```

>>> m = s.split() 489
>>> m 490
>>> enumerate(m) 491
>>> list(enumerate(m)) 492
>>> n = list(enumerate(m)) 493
>>> n 494
>>> [(k, v) for v, k in n] 495
>>> dict([(k, v) for v, k in n]) 496

>>> d 497
>>> d.items() 498
>>> list(d.iteritems()) 499

>>> d = { 500
...     'zero': 0,
...     'one': 1,
...     'two': 2,
...     }

>>> d 501
>>> from operator import itemgetter 502
>>> sorted(d.iteritems(), 503
...        key=itemgetter(1))

>>> {'Jan': 1, 'Feb': 2, 'Mar': 3} 504
>>> dict([('Jan', 1), 505
...      ('Feb', 2),
...      ('Mar', 3)])

>>> dict(Jan=1, Feb=2, Mar=3) 506

>>> 'Jan Feb Mar'.split() 507
>>> range(1, 4) 508
>>> zip('Jan Feb Mar'.split(), 509
...     range(1, 4))

>>> dict(zip('Jan Feb Mar'.split(), 510
...          range(1, 4)))

>>> list(enumerate('Jan Feb Mar'.split())) 511
>>> [(k, v+1) for v, k in enumerate( 512
...     'Jan Feb Mar'.split())]

>>> d = dict([(k, v+1) for v, k in 513
...     enumerate('Jan Feb Mar'.split())])

```

Exercises: Dictionaries (1)

```
>>> dict(Jan=1, Feb=2) 514
>>> dict(Jan=Janvier, Feb=Fevrier) 515
>>> dict(Jan='Janvier', Feb='Fevrier') 516
>>> dict(1=Jan, 2=Feb) 517
```

```
>>> months = 'january february march april' 518
```

From months, create months_to_int of the form [None, 'Jan', 'Feb', 'Mar', ...]

From months, create int_to_months of the form {'Jan': 1, 'Feb': 2, ... }

Dictionaries (2)

```
>>> {1: 'one', 2: 'two'} 519
>>> {(1,): 'one', (2,): 'two'} 520
>>> {[1]: 'one', [2]: 'two'} 521
```

```
>>> d = dict([(k, v) for v, k in 522
...     enumerate('zero one two three '
...     'four five six'.split())])
```

```
>>> d 523
>>> [item for item in d] 524
>>> list(iter(d)) 525
```

```
>>> [t for t in d.items()] 526
>>> [t for t in d.iteritems()] 527
```

```
>>> d.keys() 528
>>> d.values() 529
```

```
>>> import collections 530
>>> dd = collections.defaultdict(int) 531
>>> int() 532
>>> for c in 'mississippi': 533
...     dd[c] += 1
```

```
>>> dd.items() 534
>>> dd 535
```

Exercises: Dictionaries (2)

```
>>> d = dict.fromkeys(['one', 'two'], 99) 536
>>> d2 = dict(three=3, four=4) 537
>>> d 538
>>> d2 539
>>> d.update(d2) 540
>>> d 541
>>> d2.clear() 542
>>> d2 543
>>> d['one'] 544
>>> d['nine'] 545
>>> d.get('one') 546
>>> d.get('nine') 547
>>> d.get('nine', 0) 548
>>> d 549
>>> d.setdefault('nine', 9) 550
>>> d 551
>>> d.pop('nine') 552
>>> d 553
>>> d.pop('ten') 554
>>> d.pop('ten', 10) 555
>>> d 556
>>> d.popitem() 557
>>> d 558
```

Files

```
>>> f = open('eg.txt') 559
>>> f 560
>>> f.read() 561
>>> f 562
>>> len(f.read()) 563
>>> f 564
>>> f = open('eg.txt') 565
>>> f 566
>>> len(f.read()) 567
>>> f = open('eg.txt') 568
>>> dir(f) 569
>>> hasattr(f, '__iter__') 570
>>> hasattr(f, 'next') 571

>>> f.next() 572
>>> f.next() 573
>>> f.next() 574
>>> f.next() 575
>>> f.next() 576
>>> for line in open('eg.txt'): 577
...     print(line, end='')
```

```

>>> f = open('eg.txt')
>>> for line in f:
...     print(line, end='')

>>> f.close()
>>> del f
>>> from __future__ import with_statement
>>> with open('eg.txt') as f:
...     for line in f:
...         print(line, end='')

>>> f

```

Other file operations

- `open('output.txt', 'w')`
- `open('output.txt', 'wb')`
- `f.write()`
- `f.readline()`
- `f.readlines()`

Exercises: Files (and Dictionaries)

Write code which reads a file and uses `collections.defaultdict` to produce a histogram of the frequency of each unique line in the file.

Namespaces, global

A *namespace* is a mapping from names to objects.

A *scope* is a section of Python code where a namespace is directly accessible. The namespace search order is (from the python.org tutorial):

- the innermost scope, which is searched first, contains the local names;
- the namespaces of any enclosing functions, which are searched starting with the nearest enclosing scope; (or the module if outside any function)
- the middle scope, searched next, contains the current module's global names;
- and the outermost scope (searched last) is the namespace containing built-in names.

All namespaces *changes* (assignment, `import`, function definition, `del`) happen in the local scope.

```

>>> x = 1 585
>>> def test1(): 586
...     print(x)

>>> test1() 587

>>> def test2(): 588
...     x = 2
...     print x

>>> x 589
>>> test2() 590
>>> x 591

>>> def test3(): 592
...     print(x)
...     x = 3

>>> x 593
>>> test3() 594
>>> x 595

>>> def test4(): 596
...     global x
...     print(x)
...     x = 4
...     print(x)

>>> x 597
>>> test4() 598
>>> x 599

```

“If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).” [Python tutorial section 9.2 at <http://docs.python.org/tutorial>]

A Simple Class

Remember, everything in Python is an object and has:

- a single *id*,
- a single *value*,
- some number of *attributes* (part of its value),

- a single *type*,
- (zero or) one or more *names* (in one or more namespaces),
- and usually (indirectly), one or more *base classes*.

Most objects are instances of classes. The type of an object is its class.

Classes are instances of *metaclasses*. The type of a class is a metaclass, i.e. `type(type(anObject))` is a metaclass.

Are classes and metaclasses objects?

1. The `class` statement creates a new namespace and all its name assignments (e.g. function definitions) are bound to the class object.
2. Instances are created by calling the class: `ClassName()` or `ClassName(parameters)`.

`ClassName.__init__(<new object>, ...)` is called automatically.

3. Accessing an attribute `method_name` on a class instance creates a *method object* if `method_name` is a method (in `ClassName` or its superclasses). A method object binds the object (the class instance) as the first parameter.

point1.py:

```
class Point(object):
    """Example point class"""
    def __init__(self, x=0, y=0):
        # Note that self exists by now
        self.x, self.y = x, y

    def __repr__(self):
        return 'Point({0}, {1})'.format(self.x, self.y)

    __str__ = __repr__

    def translate(self,
                  deltax=None, deltay=None):
        """Translate the point"""
        if deltax:
            self.x += deltax
        if deltay:
            self.y += deltay

    def __sub__(self, other):
        from math import sqrt
        return sqrt(
            (self.x - other.x) ** 2 +
            (self.y - other.y) ** 2)
```

```
>>> from point1 import Point
>>> p1 = Point()
>>> p1
>>> p1.translate(2, 4)
>>> p1
```

600
601
602
603
604

```

>>> p1.translate(-3.5, 4.9) 605
>>> p1 606
>>> p2 = Point(1, 2) 607
>>> p2 608

>>> dir(p1) 609
>>> p1.__dict__ 610

>>> points = [p1, p2] 611
>>> points 612
>>> from operator import attrgetter 613
>>> sorted([p1, p2], 614
...     key=attrgetter('x'),
...     reverse=True)

>>> max(points, 615
...     key=attrgetter('x'))

```

Exercises: A Simple Class

Design an `Employee` class with first name, last name, and age. Include a constructor to which one may supply initial values. Override methods to customize the result of `str` and `repr` on instances of the class. Consider what tests you would write to test your class. Use paper if it's faster.

More of a Simple Class

point2.py:

```

import math

class Point(object):
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y

    def __repr__(self):
        return 'Point({0}, {1})'.format(self.x, self.y)

    __str__ = __repr__

    def translate(self, deltax=None, deltay=None):
        if deltax:
            self.x += deltax
        if deltay:
            self.y += deltay

```

```

def __sub__(self, other):
    return math.sqrt(
        (self.x - other.x) ** 2 +
        (self.y - other.y) ** 2)

>>> from point2 import Point
>>> p1 = Point(1, 1)
>>> p1
>>> p2 = Point(1, 2)
>>> p2
>>> p1 - p2
>>> Point(1, 1) - Point(2, 2)
>>> Point(1, 1) - Point(1, 1)
>>> Point(1, 1) - Point(2, 1)
>>> Point(1, 1) - 2

>>> points = [p1, p2]
>>> from operator import methodcaller
>>> distance_from_zero = methodcaller(
...     '__sub__', Point(0, 0))

>>> distance_from_zero(p1)
>>> distance_from_zero(p2)
>>> points
>>> max(points, key=distance_from_zero)

```

616
617
618
619
620
621
622
623
624
625

626
627
628

629
630
631
632

Exercises: More of a Simple Class

Fix class `Point` by either giving meaning to `Point - int` or by raising a more appropriate exception when `other` is not an instance of `Point`. Either way you'll want to figure out how to use the `isinstance` function.

Think about `map` versus `methodcaller`. Use them together to generate a list of distances from zero from a list of points.

Standard Methods

- `__new__`, `__init__`, `__del__`, `__repr__`, `__str__`, `__format__`
- `__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__call__`, `__dir__`
- `__len__`, `__getitem__`, `__missing__`, `__setitem__`, `__delitem__`, `__contains__`, `__iter__`
- `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`, `__cmp__`, `__nonzero__`, `__hash__`

- `__add__`, `__sub__`, `__mul__`, `__div__`, `__floordiv__`, `__mod__`, `__divmod__`, `__pow__`, `__and__`, `__xor__`, `__or__`, `__lshift__`, `__rshift__`, `__neg__`, `__pos__`, `__abs__`, `__invert__`, `__iadd__`, `__isub__`, `__imul__`, `__idiv__`, `__itruediv__`, `__ifloordiv__`, `__imod__`, `__ipow__`, `__iand__`, `__ixor__`, `__ior__`, `__ilshift__`, `__irshift__`
- `__int__`, `__long__`, `__float__`, `__complex__`, `__oct__`, `__hex__`, `__coerce__`
- `__radd__`, `__rsub__`, `__rmul__`, `__rdiv__`, etc.
- `__enter__`, `__exit__`, `__next__`

Subclassing

`cpoint1.py`:

```
from point2 import Point
class ColorPoint(Point):
    """Example color point class"""
    def __init__(self, x=0, y=0, color=None):
        super(ColorPoint, self).__init__(x, y)
        self.color = color

    def __str__(self):
        return 'ColorPoint(%f, %f, %r)' \
            % (self.x, self.y, self.color)
```

```
>>> from cpoint1 import ColorPoint
>>> c1 = ColorPoint()
>>> c1
>>> c1.translate(2, 4)
>>> c1
>>> c2 = ColorPoint(9, 11, 'blue')
>>> c2
>>> c1 - c2
>>> str(c2)
>>> repr(c2)
```

633
634
635
636
637
638
639
640
641
642

Exercises: Subclassing

Why did `str(c2)` above print `Point(9.000000, 11.000000)` instead of `ColorPoint(9.000000, 11.000000, 'blue')`? What's the one-line fix to `ColorPoint`? What's the more robust fix to `Point` that will fix all subclasses of `Point`?

Design a subclass `Assistant` of your `Employee` class where multiple employees can be assigned one assistant and of course one assistant can be assigned to multiple employees.

Design a class which subclasses `dict` overriding its `keys` method to always return the list of keys in sorted order.

Note the solution to the previous exercise is not the same as an ordered dictionary which returns keys in insertion order, unlike a `dict` which does not specify an order. Think about how you would design an ordered dictionary subclass of `dict`.

Dynamic Classes (advanced)

```
>>> class MyClass(object):                                     643
...     pass

>>> dir(MyClass)                                             644
>>> list(MyClass.__dict__)                                   645

>>> myc1 = MyClass()                                       646
>>> myc1                                                    647
>>> type(myc1)                                             648
>>> myc1.__dict__                                          649

>>> def myhello(self):                                       650
...     print('Called hello()')

>>> myc2 = MyClass()                                       651
>>> myc2.hello = myhello                                    652
>>> myc2.hello()                                           653
>>> MyClass.hello = myhello                                654
>>> myc2.hello()                                           655
>>> myc2.__dict__                                          656
>>> myc2.hello                                             657
>>> del myc2.hello                                         658
>>> myc2.__dict__                                          659
>>> myc2.hello                                             660
>>> myc2.hello()                                           661
>>> myc2.hello.im_class                                    662
>>> myc2.hello.__func__                                    663
>>> myc2.hello.__self__                                    664
>>> myc2                                                    665

>>> def myinit1(self):                                       666
...     print('Called myinit1()')
...     self.field = 'a value'

>>> MyClass.__init__                                       667
>>> MyClass.__init__ = myinit1                             668
>>> MyClass.__init__                                       669
>>> myc3 = MyClass()                                       670
>>> myc3.field                                             671
```

```

>>> def myinit2(self, ival):
...     print('Called myinit2()')
...     self.field = ival
672

>>> MyClass.__init__
673
>>> MyClass.__init__ = myinit2
674
>>> MyClass.__init__
675
>>> myc4 = MyClass('one')
676
>>> myc4.__dict__
677
>>> myc4.field
678
>>> myc5 = MyClass()
679

>>> class MyClass(object):
...     def __init__(self, ival):
...         print('Called myinit()')
...         self.field = ival
...     #
...     def hello(self):
...         print('Called hello()')
680

```

Dynamic Classes with type (advanced)

```

>>> print(type.__doc__)
681
>>> MyClass2 = type(
682
>>>     'MyClass2',
683
>>>     (object,),
684
>>>     { '__init__': myinit2,
685
...       'hello': myhello})

>>> myc2 = MyClass2('two')
686
>>> type(myc2)
687
>>> myc2.__class__
688
>>> myc2.__dict__
689
>>> myc2.field
690
>>> myc2.hello()
691

```

The dynamic call to type is what the class statement actually invokes... almost.

If there is a `__metaclass__` method available in the bases, it is called instead.

Reality: 99.9% of all class definitions are like this:

```

class AClass(object):
    def __init__(self, initial_value):
        self.ifieldd = initial_value

    def method1(self):
        print('do something')

```

Not covered:

- class and static methods
- decorators
- property and descriptors
- metaclasses

Iterators

- A for loop evaluates an expression to get an iterable and then calls `iter()` to get an iterator.
- The iterator's `next()` method is called repeatedly until `StopIteration` is raised.
- `iter(foo)`
 - checks for `foo.__iter__()` and calls it if it exists
 - else checks for `foo.__getitem__()`, calls it starting at zero, handles `IndexError` by raising `StopIteration`.
- Note `iter(callable, sentinel)` does something else.

```
>>> class MyIt(object):                                     692
...     pass

>>> myit = MyIt()                                       693
>>> iter(myit)                                          694
>>> def mygetitem(self, n):                             695
...     print('Called mygetitem', n)
...     return [0, 1, 2][n]

>>> MyIt.__getitem__ = mygetitem                       696
>>> iter(myit)                                         697
>>> list(iter(myit))                                   698
>>> 1 in myit                                          699
>>> x, y, z = myit                                     700

>>> myit2 = iter([1, 2, 3])                             701
>>> 2 in myit2                                         702
>>> 2 in myit2                                         703

>>> class ListOfThree(object):                          704
...     def __iter__(self):
...         self.count = 0
...         return self
...     #
...     def next(self):
```

```

...         if self.count < 3:
...             self.count += 1
...             return self.count
...         raise StopIteration

>>> m3 = ListOfThree()
>>> m3it = iter(m3)
>>> m3it.next()
>>> m3it.next()
>>> m3it.next()
>>> m3it.next()
>>> m3it.next()

>>> list(m3it)
>>> list(m3it)

```

705
706
707
708
709
710

711
712

Exercises: Iterators

Consider how you would write a subclass of `dict` whose iterator would return its keys, as does `dict`, but in sorted order, and without using `yield`.

Design a class `reversed` to mimic Python's built in `reverse` function. Assume an indexable sequence as parameter.

Generators

```

>>> it = (2 * i for i in range(5))
>>> it
>>> hasattr(it, 'next')
>>> dir(it)

```

713
714
715
716

A generator's `send` and `throw` methods provide a communication path back into generator, i.e. they can be used as cogenerators.

```

>>> for i in (2 * i for i in range(5)):
...     print(i)

>>> def list123():
...     yield 1
...     yield 2
...     yield 3

>>> list123
>>> list123()
>>> it = list123()
>>> it.next()

```

717

718

719
720
721
722

```

>>> it.next() 723
>>> it.next() 724
>>> it.next() 725

>>> def even(limit): 726
...     for i in range(0, limit, 2):
...         print('Yielding', i)
...         yield i
...     print('done loop, falling out')

>>> it = iter(even(3)) 727
>>> it 728
>>> it.next() 729
>>> it.next() 730
>>> it.next() 731

>>> for i in even(3): 732
...     print(i)

>>> list(even(10)) 733
>>> def paragraphs(lines): 734
...     result = ''
...     for line in lines:
...         if line.strip() == '':
...             yield result
...             result = ''
...         else:
...             result += line
...     yield result

>>> list(paragraphs(open('eg.txt'))) 735
>>> len(list(paragraphs(open('eg.txt')))) 736

```

Exercises: Generators

Write a generator `sdouble(str)` that takes a string and returns that string “doubled” 5 times. E.g. `sdouble('s')` would yield these values: `['s', 'ss', 'sss', 'ssss', 'sssssss', 'ssssssssssssss']`.

Re-solve the last (iterator subclass of `dict`) exercise to use `yield` in the `next` method.

Write a generator that returns sentences out of a paragraph. Make some simple assumptions about how sentences start and/or end.

Re-solve the histogram exercise for words, not lines, i.e. write code which reads a file and produces a histogram of the frequency of all *words* in the file.

Explore the `itertools` module.

Loops: else, continue, break

```
>>> for i in range(5): 737
...     if i in [2, 4]:
...         print('continue', i)
...         continue
...     print(i)
... else:
...     print('Else')
```

The `else` clause is always executed last unless `break` is used to exit the `for` loop or `while` loop.

```
>>> for i in range(10): 738
...     if i == 4:
...         print('break', i)
...         break
...     print(i)
... else:
...     print('Else')
```

```
>>> i = 1 739
>>> while i <= 5: 740
...     print(i)
...     i += 1
... else:
...     print('while: else' )
```

```
>>> t = 10 741
>>> if t < 20: 742
...     desc = 'cool'
... else:
...     desc = 'warm'
```

```
>>> desc 743
>>> desc = ('cool' if t < 20 else 'warm') 744
>>> desc 745
```

Functions

```
>>> def f3(arg1, arg2, kwarg1=0, kwarg2=0): 746
...     print('arg1: {0}, arg2: {1}, '
...           'kwarg1: {2}, kwarg2: {3}'
...           .format(arg1, arg2, kwarg1, kwarg2))
```

```
>>> f3(1, 2) 747
>>> f3(1, 2, 3) 748
>>> f3(1, 2, kwarg2=4) 749
>>> f3(1, kwarg1=3) 750
```

```

>>> def f4(arg1, arg2, kwarg1=0, kwarg2=0,
...         *args, **kwargs):
...     print('arg1: {0}, arg2: {1}, '
...           'kwarg1: {2}, kwarg2: {3}'
...           .format(arg1, arg2, kwarg1, kwarg2))
...     if args:
...         print('args:', str(args))
...     if kwargs:
...         print('kwargs:', kwargs)

```

751

```

>>> f4(1, 2)
>>> f4(arg1=1, arg2=2)
>>> f4(arg2=1, arg1=2)
>>> f4(1, 2, 3)
>>> f4(1, 2, kwarg2=4)
>>> f4(1, kwarg1=3)
>>> f4(1, 2, 3, 4, 5, 6)
>>> f4(1, 2, 3, 4, keya=7, keyb=8)
>>> f4(1, 2, 3, 4, 5, 6, keya=7, keyb=8)

```

752
753
754
755
756
757
758
759
760

```

>>> tuple12 = 1, 2
>>> tuple12
>>> tuple56 = 5, 6
>>> tuple56
>>> dict78 = dict(keya=7, keyb=8)
>>> dict78
>>> f4(*tuple12)
>>> f4(**dict78)
>>> f4(1, 2, **dict78)
>>> f4(1, 2, *tuple56, **dict78)
>>> f4(1, 2, 3, 4, *tuple56, **dict78)

```

761
762
763
764
765
766
767
768
769
770
771

Exercises: Functions

Write a better `__init__` method for your `Employee` class which can handle these calls:

- `Employee()`
- `Employee('Jane', 'Doe', 48)`
- `Employee('Jane', 'Doe')`
- `Employee(last='Doe', first='Jane')`
- `Employee('Jane Doe')`
- `Employee('Doe, Jane')`
- `Employee('Jane Doe', assistant=Employee('John Doe'))`

Dictionaries Example

```
>>> # Here's an example of functions as first 772
>>> # class objects to create a simple calculator. 773
>>> 7+3 774
>>> import operator 775
>>> operator.add(7, 3) 776

>>> expr = '7+3' 777
>>> lhs, op, rhs = expr 778
>>> lhs, op, rhs 779
>>> lhs, rhs = int(lhs), int(rhs) 780
>>> lhs, op, rhs 781
>>> op, lhs, rhs 782
>>> operator.add(lhs, rhs) 783

>>> ops = { 784
...     '+': operator.add,
...     '-': operator.sub,
...     }

>>> ops[op] (lhs, rhs) 785

>>> def calc(expr): 786
...     lhs, op, rhs = expr
...     lhs, rhs = int(lhs), int(rhs)
...     return ops[op] (lhs, rhs)

>>> calc('7+3') 787
>>> calc('9-5') 788
>>> calc('8/2') 789
>>> ops['/'] = operator.div 790
>>> calc('8/2') 791
```

Exceptions: try/except/else/finally

```
>>> try: 792
...     int('four')
... except ValueError as e:
...     print('Caught error:' , e)
... else:
...     print('Else: (fell off try)')
... finally:
...     print('Finally always happens')

>>> try: 793
...     int('4')
```

```
... except ValueError as e:
...     print('Caught error:', e)
... else:
...     print('Else: (fell off try)')
... finally:
...     print('Finally is now')
```

Python 3

Changes

Major breakages (<http://www.python.org/doc/essays/ppt/pycon2008/Py3kAndYou.pdf>):

- Print function: `print(a, b, file=sys.stderr)`
- Distinguish sharply between text and data
 - `b"..."` for bytes literals
 - `"..."` for (Unicode) str literals
- Dict `keys()` returns a set view [`+items()/values()`]
- No default `<`, `<=`, `>`, `>=` implementation
- `1/2` returns 0.5
- Library cleanup

Suggestions

- `from __future__ import division`
- `from __future__ import print_function`
- `from __future__ import unicode_literals`
- 2to3 tool
- `python -3`
- `dict.iterkeys()` and similar

Standard Library Tour

- data types: calendar collections datetime decimal math random sets
- file formats: bz2 csv pickle struct zip zlib
- text: pprint repr textwrap
- operating system: os os.path shutil
- internet: email ftplib hashlib html web
- programming: difflib doctest filecmp fileinput functools gettext glob itertools logging
optparse pdb re timeit

Next Steps

- Enjoy the conference (if you're attending)
- Practice
- Make a habit of learning
 - tutorials
 - books
 - videos
 - newsgroups
- Have fun!