

Python 102

Overview

Course: Python 102 tutorial, PyCon 2009, Chicago

Presenter: Stuart Williams (stuart of 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.

Requirements: A laptop computer with Python 2.6 installed.

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.



The PyCon 2009 Python 102 Tutorial by Stuart Williams is licensed under a [Creative Commons Attribution-NonCommercial-Share Alike 2.5 Canada License](https://creativecommons.org/licenses/by-nc-sa/2.5/ca/)

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
>>> 1 < 2 and 2 < 3	45
>>> any([1 < 2, 2 < 3])	46
>>> all([1 < 2, 2 < 3])	47
>>> all([1 < 2, 2 < 1])	48
>>> x = 5	49
>>> 1 < x < 10	50
>>> 1 < x != 7	51
>>> 1 < x != 5	52
>>> 1 < x and x != 7	53
>>> 1 < x and x != 5	54
>>> 0 != 1	55
>>> a = 1	56
>>> a	57
>>> a++	58
>>> a	59
>>> a += 1	60
>>> a	61
>>> x, y, z = 0, 0, 0	62
>>> x	63
>>> y	64
>>> z	65
>>> x, y, z	66
>>> x = y = z = 1	67
>>> x, y, z	68
>>> 2 + 3j	69
>>> 1 / 3.0	70
>>> from decimal import Decimal, getcontext	71
>>> Decimal(1) / Decimal(3)	72
>>> getcontext().prec = 2	73
>>> Decimal(1) / Decimal(3)	74
>>> abs(-4)	75
>>> round(1 / 3.0, 2)	76
>>> round(1234, -2)	77
>>> 2 ** 32	78
>>> ord('a')	79
>>> ord('z') - ord('a')	80
>>> ord('a')	81
>>> chr(97)	82
>>> chr(ord('a'))	83
>>> print 3	84
>>> print(3)	85

>>> from __future__ import print_function	86
>>> print 3	87
>>> print(3)	88
>>> import operator	89
>>> operator.add(7, 3)	90
>>> print(operator.__doc__)	91
>>> help(operator)	92

Exercise: Numbers

>>> 2 + 3j + 4 + 3j	93
>>> hex(16)	94
>>> oct(16)	95
>>> 0xff	96
>>> 0o377	97
>>> 0b11111111	98
>>> 14 / 12	99
>>> 14 % 12	100
>>> divmod(14, 12)	101
>>> print(abs.__doc__)	102
>>> print(divmod.__doc__)	103
>>> print(hex.__doc__)	104
>>> print(oct.__doc__)	105
>>> print(pow.__doc__)	106
>>> print(round.__doc__)	107
>>> import math	108
>>> help(math)	109
>>> dir(math)	110
>>> print(' '.join(dir(math)))	111
>>> print('\n'.join(dir(math)))	112
>>> x = 5	113
>>> 1 < x < 10 != 6	114
>>> 1 < x < 10 == 6	115
>>> x = 6	116
>>> 1 < x < 10 != 6	117
>>> import operator	118
>>> dir(operator)	119
>>> print(operator.pow.__doc__)	120
>>> operator.pow	121
>>> operator.pow(2)	122

>>> operator.pow(2, 8)	123
>>> operator.__pow__ == operator.pow	124
>>> dir(0)	125
>>> int(0).__class__	126
>>> str(0)	127
>>> str(0).__class__	128
>>> int(0).__str__()	129
>>> 0.__str__()	130
>>> (0).__str__()	131
>>> int.__div__	132
>>> int.__div__ == int.__truediv__	133
>>> int.__div__ = int.__truediv__	134

String Literals, Operators, Immutability

>>> 'hello'	135
>>> "hello"	136
>>> "bob's your uncle"	137
>>> '''bob's your "uncle"'''	138
>>> 10 * ' '	139
>>> 'with' + 'hold'	140
>>> 'with' 'hold'	141
>>> 'o' in 'hello'	142
>>> s = 'hello'	143
>>> s[0] = 'j'	144
>>> s = s[0] + s[1:]	145
>>> s	146
>>> print(id.__doc__)	147
>>> id(s)	148
>>> hello_id = id(s)	149
>>> s = 'there'	150
>>> s	151
>>> id(s)	152
>>> there_id = id(s)	153
>>> t = 'hello'	154
>>> id(t)	155
>>> id(t) == hello_id	156
>>> id('one')	157
>>> id('two')	158
>>> one = 'one'	159
>>> id(one)	160
>>> two = 'two'	161
>>> id(two)	162

Exercise: String Literals, Operators, Immutability

```
>>> u'hello' 163
>>> u'hello' == 'hello' 164
>>> u'hello' is 'hello' 165
>>> help('is') 166
>>> import operator 167
>>> print(operator.is.__doc__) 168
>>> type(u'hello') 169
>>> print(u'hello'.__doc__) 170

>>> u = u'hello' 171
>>> u.__class__ 172
>>> u.__class__.__bases__ 173
>>> u.__class__.__bases__ 174
>>> u.__class__.__bases__[0].__bases__ 175
>>> object.__base__ 176

>>> ('This is a very long' 177
...     ' string, is it not?')

>>> '\a' 178
>>> len('\a') 179
>>> '\a' == '\x07' 180
>>> len('\b') 181
>>> len('\c') 182
>>> len('\d') 183
>>> '\c' 184

>>> u'h' 185
>>> u1 = u'$' 186
>>> u1 187
>>> u2 = u'\u0024' 188
>>> u2 189
>>> u3 = u'\N{dollar sign}' 190
>>> u1 == u2 == u3 191
>>> u1.encode() 192
>>> u1.encode() == '$' 193

>>> u4 = u'Pound sign:\N{pound sign}:' 194
>>> u4.encode() 195
>>> u4.encode('ascii') 196
>>> u4.encode('ascii', 'ignore') 197
>>> u4.encode('ascii', 'replace') 198
```

String Methods

```
>>> len('hello') 199
```

>>> 'hello'.startswith('h')	200
>>> 'hello'.endswith('o')	201
>>> 'hello world'.capitalize()	202
>>> 'hello world'.title()	203
>>> 'hello world'.title().swapcase()	204
>>> ' hello '.strip()	205
>>> ' hello '.rstrip()	206
>>> ' hello '.lstrip()	207
>>> 'hello'.count('h')	208
>>> 'hello'.count('l')	209
>>> 'hello'.index('el')	210
>>> 'hello'.index('z')	211
>>> 'hello'.find('el')	212
>>> 'hello'.find('z')	213
>>> 'hello'.find('h')	214
>>> 'hello'.find('e')	215
>>> 'hello'.find('l')	216
>>> 'hello'.isalnum()	217
>>> 'hello there'.isalnum()	218
>>> 'hello'.center(20)	219
>>> 'hello'.center(20, '-')	220
>>> 'hello'.replace('h', 'z')	221
>>> 'steep'.replace('e', '3')	222
>>> 'steep'.replace('e', '3', 1)	223

Exercises: String Methods

>>> dir('hello')	224
>>> dir(str)	225
>>> dir(str) == dir('hello')	226
>>> help(str)	227
>>> help(str.isalpha)	228
>>> help(str.isdigit)	229
>>> help(str.islower)	230
>>> help(str.isspace)	231
>>> help(str.istitle)	232
>>> help(str.isupper)	233
>>> 'hello'.find('e')	234
>>> 'hello'.find('l')	235
>>> 'hello'.rfind('l')	236
>>> 'hello'.rindex('l')	237

```

>>> 'yellow is mellow'.find('ow')                238
>>> 'yellow is mellow'.find('ow', 6)             239

>>> 'hello'.ljust(20, '*')                       240
>>> 'hello'.rjust(20, '+')                       241

```

What does the following expression check for?

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

Hint: See `help(str.index)` and try setting `s == 'hello world'` and `s == 'world hello'`.

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

```

>>> 'Wait...'.rstrip('.')                       242
>>> list('hello')                              243

```

String Templates and Formatting

```

>>> from string import Template                 244
>>> templ = Template('I suspect $who, '       245
...     'in the $where, with the $what')

```

```

>>> templ.substitute({                        246
...     'who': 'Professor Plum',
...     'where': 'kitchen',
...     'what': 'knife'})

```

```

>>> locals()                                 247
>>> who = 'Professor Plum'                   248
>>> where = 'kitchen'                       249
>>> what = 'knife'                          250
>>> locals()                                 251
>>> templ.substitute(locals())               252

```

```

>>> '%(what)s' % dict(what='knife')         253
>>> '%(what)s' % {'what': 'knife'}         254
>>> '%(what)s in %(where)s' % locals()      255

```

```

>>> '{what}'.format(what='knife')           256
>>> '{what} in {where}'\                    257
...     .format(what='knife',
...     where='kitchen')

```

```

>>> 'with {what}'.format(locals())          258
>>> 'with {0[what]}'.format(locals())       259
>>> 'with {0[what]}'.format(               260
...     dict(what='knife',
...     where='kitchen'))

```

```

>>> 'arg is {0}'.format(
...     dict(what='knife',
...         where='kitchen'))
261

>>> import sys
262
>>> print(sys.version_info)
263
>>> 'major {0[0]}, minor {0[1]}'.format(
...     sys.version_info)
264

>>> sys.byteorder
265
>>> 'Byte order: {0.byteorder}' \
...     .format(sys)
266

```

Exercise: String Formatting

```

>>> s1 = 'with the %s '
267
>>> s1 % 'knife'
268
>>> s2 = 'in the %s'
269
>>> s2 % 'kitchen'
270
>>> s1 + s2 % 'knife', 'kitchen'
271
>>> (s1 + s2) % 'knife', 'kitchen'
272
>>> s1 + s2 % ('knife', 'kitchen')
273
>>> (s1 + s2) % ('knife', 'kitchen')
274
>>> s1
275
>>> s1 = s1.replace('%s', '{0}')
276
>>> s1
277
>>> s1.format('knife')
278
>>> s2 = s2.replace('%s', '{1}')
279
>>> s2
280
>>> s = s1 + s2
281
>>> s
282
>>> s.format('knife')
283
>>> s.format('knife', 'kitchen')
284
>>> s1
285
>>> s2
286
>>> s1.format('knife', 'kitchen')
287
>>> s2.format('knife', 'kitchen')
288

>>> s1 = 'with the {what} '
289
>>> s1.format('knife')
290
>>> s1.format(what='knife')
291
>>> '{0} {what}'.\
...     format('kitchen', what='knife')
292
... s2 = 'in the {where}'
... s = s1 + s2
... 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 `byteorder` and `maxint` from the `sys` module. Then add the 4th element of `sys.version_info` to the string.

Introspection

```
>>> type('hello') 293
>>> 'hello'.__class__ 294
>>> 'hello'.__doc__ 295
>>> print(str.__doc__) 296

>>> print(dir.__doc__) 297
>>> dir(str) 298
>>> dir(str.strip) 299
>>> callable(str.strip) 300
>>> callable(str) 301

>>> str.__dict__ 302
>>> list(str.__dict__) 303
>>> print(vars.__doc__) 304
>>> vars(str) 305
>>> list(vars(str)) 306
>>> list(vars(str)) == list(str.__dict__) 307

>>> len(dir(str)) 308
>>> len(list(str.__dict__)) 309
>>> set([1, 2, 3, 3]) 310
>>> set([1, 2, 3]) - set([1, 2]) 311
>>> set(dir(str)) - set(list(str.__dict__)) 312
>>> print(dir.__doc__) 313

>>> import __builtin__ 314
>>> dir(__builtin__) 315

>>> str('hello') 316
>>> repr('hello') 317
>>> eval(str('hello')) 318
>>> eval(repr('hello')) 319

>>> 'hello'.__class__ 320
>>> type('hello') 321
>>> u'hello' 322
>>> u'hello'.__class__ 323
>>> type(u'hello') 324
>>> print('hello'.__doc__) 325
```

Note class `unicode(basestring)` which documents the inheritance.

```
>>> u = u'hello' 326
>>> u.__class__ 327
>>> u.__class__.__bases__ 328
>>> u.__class__.__bases__[0].__bases__ 329
```

`unicode -> basestring -> object -> None`

```
>>> basestring('hello') 330
```

Exercise: Introspection

```
>>> x = 4.3 331
>>> str(x) 332
>>> x.__str__() 333
>>> repr(x) 334
>>> x.__repr__() 335
```

```
>>> s = 'hello' 336
>>> s.__class__ 337
>>> s.__class__.__bases__ 338
```

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] 339
>>> m.count(3) 340
>>> t = (1, 2, 3) 341
>>> t.count(3) 342
>>> u = (1, 2, 3) 343
>>> t == u 344
>>> t is u 345
>>> m.count(5) 346
>>> m.index(5) 347
>>> m.insert(0, 6) 348
>>> m 349
>>> m.index(6) 350
```

>>> m.index(6, 5)	351
>>> m	352
>>> m.insert(4, 'e')	353
>>> m	354
>>> m.remove('e')	355
>>> m	356
>>> m.remove('e')	357
>>> m = range(6)	358
>>> m.reverse()	359
>>> m	360
>>> m.sorted()	361
>>> m	362
>>> sorted(m)	363
>>> m	364
>>> sorted(m, reverse=True)	365
>>> m	366
>>> m.sort()	367
>>> m	368
>>> m.sort(reverse=True)	369
>>> m	370

Exercise: Tuples and Lists

>>> m = range(4)	371
>>> m	372
>>> m.append(5)	373
>>> m	374
>>> m.extend([6, 7, 8])	375
>>> m	376
>>> r = m.pop()	377
>>> r, m	378
>>> m.pop(), m	379
>>> m.pop(), m	380
>>> m.pop(), m	381
>>> m	382
>>> m.extend([4, 5, 6])	383
>>> m	384
>>> m.pop(0), m	385
>>> m.pop(0), m	386
>>> m	387
>>> m.append([7, 8, 9])	388
>>> m	389
>>> m.append(('a', 'b'))	390
>>> m	391

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'])	392
>>> ', '.join(['one', 'two', 'three'])	393
>>> 'www.test.com'.partition('.')	394
>>> 'hello there'.partition('.')	395
>>> 'www.test.com'.rpartition('.')	396
>>> 'hello there'.split()	397
>>> 'hello there world'.split()	398
>>> 'www.python.org'.split('.')	399
>>> 'www.python.org'.split('.', 1)	400
>>> 'www.python.org'.rsplit('.', 1)	401
>>> lines = 'hello\nthere\nworld'	402
>>> lines.split()	403
>>> lines = lines + '\n'	404
>>> lines	405
>>> lines.split()	406
>>> lines.splitlines()	407
>>> lines.splitlines(True)	408
>>> lines.splitlines()	409

Exercises: More String Methods

>>> ('red orange yellow').split()	410
>>> ', '.join('red orange yellow'.split())	411
>>> ('red orange yellow'.replace(' ', ', '))	412
>>> m = range(20)	413
>>> m	414
>>> n = m[:]	415
>>> n == m	416
>>> n is m	417
>>> n[0] = 'first'	418
>>> n == m	419
>>> m	420
>>> m[0:-1:2]	421
>>> m[::2]	422
>>> m[:: -2]	423
>>> m[:: -1]	424
>>> del m[0:-1:2]	425
>>> m	426
>>> m = range(16)	427
>>> m	428
>>> ['low even'] * 5	429

```

>>> m[0:10:2] = ['low even'] * 5 430
>>> m 431
>>> del m[0:10:2] 432
>>> m.index(10) 433
>>> m[0:m.index(10)] 434
>>> m[0:m.index(10)] = [] 435
>>> m 436
>>> s1 = slice(2, 4) 437
>>> m[s1] 438
>>> s1 = slice(0, 4, 2) 439
>>> m[s1] 440

```

Nested Sequences

```

>>> m = [ ['one', 'two', 'three'], 441
...       ['One', 'Two', 'Three']]

>>> m 442
>>> m[0] 443
>>> m[1] 444
>>> m[0][0] 445

```

Decorate, Sort, Undecorate (DSU) and Alternatives

```

>>> months = [('Jan', 1, 31), 446
...           ('Feb', 2, 28),
...           ('Mar', 3, 31),
...           ('Apr', 4, 30)]

>>> dsu = [(days, (name, order, days)) 447
...        for (name, order, days) in months]

>>> dsu 448
>>> dsu.sort() 449
>>> dsu 450
>>> [ b for (a, b) in dsu ] 451

>>> print(min.__doc__) 452
>>> from operator import itemgetter 453
>>> print(itemgetter.__doc__) 454
>>> days = itemgetter(2) 455
>>> min(months, key=days) 456
>>> max(months, key=days) 457

```

Exercise: 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) 458
>>> [element for element in range(8)] 459
>>> [2 * element for element in range(8)] 460
>>> [2 + element for element in range(8)] 461

>>> [e for e in range(8) if e % 3 == 0] 462

>>> [(c, j) 463
...     for c in ['a', 'b']
...     for j in range(4)]

>>> [(i, j) 464
...     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)) 465
>>> it 466
>>> list(it) 467
```

Exercise: List Comprehensions and Functional Programming

Write a list comprehension or generator expression with three loops.

```
>>> def iseven(n): 468
...     return n % 2 == 0

>>> [e for e in range(10) if iseven(e)] 469
>>> filter(iseven, range(10)) 470
>>> print(filter.__doc__) 471
>>> map(iseven, range(10)) 472
>>> print(map.__doc__) 473

>>> import operator 474
>>> from functools import reduce 475
>>> reduce(operator.add, range(10)) 476
>>> reduce(operator.mul, range(10)) 477
>>> print(reduce.__doc__) 478
```

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() 479
>>> s = 'xyz' 480
>>> dir() 481
>>> id(s) 482
>>> type(s) 483
>>> s 484

>>> t = 'xyz' 485
>>> dir() 486
>>> id(s) == id(t) 487

>>> u = t 488
>>> dir() 489
>>> id(u) == id(t) 490

>>> m = [1, 2, 3] 491
>>> m 492
>>> dir() 493
>>> m[1] = 20 494
>>> m 495
>>> m[1] = s 496
>>> m 497
>>> id(s) == id(m[1]) 498
>>> m[1] = 'xyz' 499
>>> m 500
>>> id(s) == id(m[1]) 501
>>> n = m 502
>>> dir() 503
>>> id(n) == id(m) 504
>>> m 505
>>> m[1] = 'abc' 506
>>> m 507
>>> n 508
>>> m == n 509
```

Exercise: Objects

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

```
>>> m = range(10) 510
>>> m 511
>>> m[0] 512
>>> m.__getitem__(0) 513
>>> m[1] = 'one' 514
>>> m 515
>>> m.__setitem__(2, 'two') 516
>>> m 517
>>> m[0:10:2] 518
>>> m.__getitem__(slice(0, 10, 2)) 519
>>> m[3] = 'three' 520
>>> m 521
>>> m.__setitem__(slice(4, 6), 522
...     ['four', 'five'])
>>> m 523
```

Dictionaries (1)

```
>>> d = dict([(k, v) for v, k in 524
...     enumerate(
...     'zero one two three '
...     'four five six'.split())])

>>> s = 'zero one two three'\ 525
...     'four five six'

>>> m = s.split() 526
>>> m 527
>>> enumerate(m) 528
>>> list(enumerate(m)) 529
>>> n = list(enumerate(m)) 530
>>> n 531
>>> [(k, v) for v, k in n] 532
>>> dict([(k, v) for v, k in n]) 533

>>> d 534
>>> d.items() 535

>>> d = { 536
...     'zero': 0,
...     'one': 1,
...     'two': 2,
```

```

...     'three': 3,
...     'four': 4,
...     'five': 5,
...     'six': 6,
...     }

>>> d                                     537
>>> from operator import itemgetter       538
>>> sorted(d.items(), key=itemgetter(1))   539

>>> {'Jan': 1, 'Feb': 2, 'Mar': 3}        540
>>> dict([('Jan', 1),                      541
...       ('Feb', 2), ('Mar', 3)])

>>> dict(Jan=1, Feb=2, Mar=3)              542

>>> 'Jan Feb Mar'.split()                 543
>>> range(1, 4)                            544
>>> zip('Jan Feb Mar'.split(),            545
...     range(1, 4))

>>> dict(zip('Jan Feb Mar'.split(),        546
...         range(1, 4)))

>>> list(enumerate('Jan Feb Mar'.split())) 547
>>> [(k, v+1) for v, k in enumerate(       548
...     'Jan Feb Mar'.split())]

>>> d = dict([(k, v+1) for v, k in         549
...     enumerate('Jan Feb Mar'.split())])

```

Exercise: Dictionaries (1)

```

>>> dict(Jan=1, Feb=2)                     550
>>> dict(Jan=Janvier, Feb=Fevrier)         551
>>> dict(Jan='Janvier', Feb='Fevrier')     552
>>> dict(1=Jan, 2=Feb)                     553

>>> months = 'january february march april may june july august '\
>>>         'september october november december'  554
>>>                                     555

```

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'} 556
>>> {(1,): 'one', (2,): 'two'} 557
>>> {[1]: 'one', [2]: 'two'} 558

>>> d = dict([(k, v) for v, k in 559
...     enumerate('zero one two three '
...     'four five six'.split())])

>>> d 560
>>> [item for item in d] 561
>>> list(iter(d)) 562

>>> [t for t in d.items()] 563
>>> [t for t in d.iteritems()] 564

>>> d.keys() 565
>>> d.values() 566
```

Exercise: Dictionaries (2)

```
>>> d = dict.fromkeys(['one', 'two'], 99) 567
>>> d2 = dict(three=3, four=4) 568
>>> d 569
>>> d2 570
>>> d.update(d2) 571
>>> d 572
>>> d2.clear() 573
>>> d2 574
>>> d['one'] 575
>>> d['nine'] 576
>>> d.get('one') 577
>>> d.get('nine') 578
>>> d.get('nine', 0) 579
>>> d 580
>>> d.setdefault('nine', 9) 581
>>> d 582
>>> d.pop('nine') 583
>>> d 584
>>> d.pop('ten') 585
>>> d.pop('ten', 10) 586
>>> d 587
>>> d.popitem() 588
>>> d 589
```

Write a histogram function that takes a list of elements and returns a dictionary. The `get` method can help make the function more concise (which isn't always better).

Files

```
>>> f = open('sample.txt') 590
>>> f 591
>>> f.read() 592
>>> f 593
>>> len(f.read()) 594
>>> f 595
>>> f = open('sample.txt') 596
>>> f 597
>>> len(f.read()) 598
>>> f = open('sample.txt') 599
>>> dir(f) 600
>>> hasattr(f, '__iter__') 601
>>> hasattr(f, 'next') 602

>>> f.next() 603
>>> f.next() 604
>>> f.next() 605
>>> f.next() 606
>>> f.next() 607
>>> from __future__ import print_function 608
>>> for line in open('sample.txt'): 609
...     print(line, end='')

>>> f = open('sample.txt') 610
>>> for line in f: 611
...     print(line, end='')

>>> f.close() 612
>>> del f 613
>>> from __future__ import with_statement 614
>>> with open('sample.txt') as f: 615
...     for line in f:
...         print(line, end='')

>>> f 616
```

Other file operations

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

Exercise: Files (and Dictionaries)

Write code which reads a file and produces 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 text 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 = 7 617
>>> def test(): 618
...     print(x)

>>> test() 619
>>> x = 8 620
>>> test() 621

>>> def test2(): 622
...     x = x + 1
...     print(x + 1)

>>> test2() 623

>>> def test3(): 624
...     x = 19
...     print(x)

>>> x 625
>>> test3() 626
>>> x 627

>>> def test4(): 628
...     global x
...     x = x + 1
...     print(x)
```

```
>>> x
>>> test4()
>>> x
```

629
630
631

“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).”

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 (its) 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. `c.method_name` attribute access creates a *method object* if `method_name` is a method (in `ClassName` or superclasses). A method object binds the 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 __str__(self):
        return 'Point(%f, %f)' \
            % (self.x, self.y)
```

```

    __repr__ = __str__

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

>>> from point1 import Point                                632
>>> p1 = Point()                                           633
>>> p1                                                       634
>>> p1.translate(2, 4)                                     635
>>> p1                                                       636
>>> p1.translate(-3.5, 4.9)                                637
>>> p1                                                       638
>>> p2 = Point(1, 2)                                       639
>>> p2                                                       640

>>> dir(p1)                                                 641
>>> p1.__dict__                                             642

>>> points = [p1, p2]                                       643
>>> points                                                  644
>>> from operator import attrgetter                         645
>>> sorted([p1, p2],                                       646
...         key=attrgetter('x'),
...         reverse=True)

>>> max(points,                                           647
...         key=attrgetter('x'))

```

Exercise: A Simple Class

Design and write a 10 - 15 line python module which implements a class `Employee` with first name, last name, and age. Give it a constructor to which one may supply initial values. Override methods to customize the result of `str` and `repr` on instances of the class. Start adding a test function to your file which exercises the features of the class.

More of a Simple Class

point2.py:

```

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

    def __str__(self):
        return 'Point(%f, %f)' \
            % (self.x, self.y)

    __repr__ = __str__

    def translate(self,
                  dx=None, dy=None):
        if dx:
            self.x += dx
        if dy:
            self.y += dy

    def __sub__(self, other):
        from math import sqrt
        return 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)

```

Exercise: 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)
```

665
666
667
668
669
670
671
672
673
674

Exercise: Subclassing

Design and write 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.

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`)? Try both.

Write 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 seems to return them in random order. Think about how you would design that subclass.

Dynamic Classes (advanced)

```
>>> class MyClass(object):                                     675
...     pass

>>> dir(MyClass)                                             676
>>> list(MyClass.__dict__)                                   677

>>> myc1 = MyClass()                                       678
>>> myc1                                                    679
>>> type(myc1)                                             680
>>> myc1.__dict__                                          681

>>> def myhello(self):                                       682
...     print('Called hello()')

>>> myc2 = MyClass()                                       683
>>> myc2.hello = myhello                                    684
>>> myc2.hello()                                           685
>>> MyClass.hello = myhello                                686
>>> myc2.hello()                                           687
>>> myc2.__dict__                                          688
>>> myc2.hello                                             689
>>> del myc2.hello                                         690
>>> myc2.__dict__                                          691
>>> myc2.hello                                             692
>>> myc2.hello()                                           693
>>> myc2.hello.im_class                                    694
>>> myc2.hello.__func__                                    695
>>> myc2.hello.__self__                                    696
>>> myc2                                                    697

>>> def myinit1(self):                                       698
...     print('Called myinit1()')
...     self.field = 'a value'
```

```

>>> MyClass.__init__ 699
>>> MyClass.__init__ = myinit1 700
>>> MyClass.__init__ 701
>>> myc3 = MyClass('bob') 702
>>> myc3.field 703

>>> def myinit2(self, ival): 704
...     print('Called myinit2()')
...     self.field = ival

>>> MyClass.__init__ 705
>>> MyClass.__init__ = myinit2 706
>>> MyClass.__init__ 707
>>> myc4 = MyClass('one') 708
>>> myc4.__dict__ 709
>>> myc4.field 710
>>> myc5 = MyClass() 711

>>> class MyClass(object): 712
...     def __init__(self, ival):
...         print('Called myinit()')
...         self.field = ival
...     #
...     def hello(self):
...         print('Called hello()')

```

Dynamic Classes with type (advanced)

```

>>> print(type.__doc__) 713
>>> MyClass2 = type( 714
>>>     'MyClass2', 715
>>>     (object,), 716
>>>     { '__init__': myinit2, 717
...     'hello': myhello})

>>> myc2 = MyClass2('two') 718
>>> type(myc2) 719
>>> myc2.__class__ 720
>>> myc2.__dict__ 721
>>> myc2.field 722
>>> myc2.hello() 723

```

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):
    """Documentation here."""

    def __init__(self, initial_value):
        self.ifiield = 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,

Bullet list ends without a blank line; unexpected unindent.

handles `IndexError` by raising `StopIteration`.

- Note `iter(callable, sentinel)` does something else.

```

>>> class MyIt(object):                                     724
...     pass

>>> myit = MyIt()                                         725
>>> iter(myit)                                           726
>>> def mygetitem(self, n):                               727
...     print('Called mygetitem(%d)' % n)
...     return [0, 1, 2][n]

>>> MyIt.__getitem__ = mygetitem                         728
>>> iter(myit)                                           729
>>> list(iter(myit))                                     730
>>> 1 in myit                                           731
>>> x, y, z = myit                                       732

```

```

>>> myit2 = iter([1, 2, 3]) 733
>>> 2 in myit2 734
>>> 2 in myit2 735

>>> class ListOfThree(object): 736
...     def __iter__(self):
...         self.count = 0
...         return self
...     #
...     def next(self):
...         if self.count < 3:
...             self.count += 1
...             return self.count
...         raise StopIteration

>>> lot = ListOfThree() 737
>>> lotit = iter(lot) 738
>>> lotit.next() 739
>>> lotit.next() 740
>>> lotit.next() 741
>>> lotit.next() 742

>>> list(lotit) 743
>>> list(lotit) 744

```

Exercise: Iterators

Write a subclass of `dict` whose iterator returns its keys, as does `dict`, but in sorted order. Do not use `yield`.

Write 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)) 745
>>> it 746
>>> hasattr(it, 'next') 747
>>> dir(it) 748

```

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)): 749
...     print(i)

```

```

>>> def list123():
...     yield 1
...     yield 2
...     yield 3
750

>>> list123
751
>>> list123()
752
>>> it = list123()
753
>>> it.next()
754
>>> it.next()
755
>>> it.next()
756
>>> it.next()
757

>>> def even(limit):
...     for i in range(0, limit, 2):
...         print('Yielding %d' % i)
...         yield i
...     print('done loop, falling out')
758

>>> it = iter(even(3))
759
>>> it
760
>>> it.next()
761
>>> it.next()
762
>>> it.next()
763

>>> for i in even(3):
...     print(i)
764

>>> list(even(10))
765
>>> def paragraphs(lines):
...     result = ''
...     for line in lines:
...         if line.strip() == '':
...             yield result
...             result = ''
...         else:
...             result += line
...     yield result
766

>>> list(paragraphs(open('sample.txt')))
767
>>> len(list(paragraphs(open('sample.txt'))))
768

```

Exercise: Generators

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. Start by writing a good set of test cases, but 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):
...     if i in [2, 4]:
...         print('continue (%d)' % i)
...         continue
...     print(i)
... else:
...     print('Else')
```

769

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

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

770

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

771
772

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

773
774

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

775
776
777

Functions

```
>>> def f3(arg1, arg2, kwarg1=0, kwarg2=0): 778
...     print('arg1: %s, arg2: %s, '
...           'kwarg1: %s, kwarg2: %s'
...           % (arg1, arg2, kwarg1, kwarg2))

>>> f3(1, 2) 779
>>> f3(1, 2, 3) 780
>>> f3(1, 2, kwarg2=4) 781
>>> f3(1, kwarg1=3) 782

>>> def f4(arg1, arg2, kwarg1=0, kwarg2=0, 783
...         *args, **kwargs):
...     print('arg1: %s, arg2: %s, '
...           'kwarg1: %s, kwarg2: %s'
...           % (arg1, arg2, kwarg1, kwarg2))
...     if args: # tuple confuses %:
...         print('args: %s' % str(args))
...     if kwargs:
...         print('kwargs: %s' % (kwargs))

>>> f4(1, 2) 784
>>> f4(arg1=1, arg2=2) 785
>>> f4(arg2=1, arg1=2) 786
>>> f4(1, 2, 3) 787
>>> f4(1, 2, kwarg2=4) 788
>>> f4(1, kwarg1=3) 789
>>> f4(1, 2, 3, 4, 5, 6) 790
>>> f4(1, 2, 3, 4, keya=7, keyb=8) 791
>>> f4(1, 2, 3, 4, 5, 6, keya=7, keyb=8) 792

>>> tuple12 = 1, 2 793
>>> tuple12 794
>>> tuple56 = 5, 6 795
>>> tuple56 796
>>> dict78 = dict(keya=7, keyb=8) 797
>>> dict78 798
>>> f4(*tuple12) 799
>>> f4(**dict78) 800
>>> f4(1, 2, **dict78) 801
>>> f4(1, 2, *tuple56, **dict78) 802
>>> f4(1, 2, 3, 4, *tuple56, **dict78) 803

>>> import operator 804
>>> ops = {'+': operator.add, 805
...        '-': operator.sub}

>>> e = '7 + 3'.split() 806
>>> e[1], e[0], e[2] 807
>>> ops[e[1]](int(e[0]), int(e[2])) 808
```

Exercise: 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((Doe, Jane))`
- `Employee('Jane Doe', assistant=Employee('John Doe'))`

Exceptions: try/except/else/finally

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

809

```
>>> try:
...     int('4')
... except ValueError as e:
...     print('Caught error:', e)
... else:
...     print('Else: (fell off try)')
... finally:
...     print('Finally is now')
```

810

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 btw. 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

- 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!

The End

Thanks!