

Python 101

Overview

Course: Python 101 tutorial, PyCon 2009, Chicago

Presenter: Stuart Williams (stuart of swilliams.ca)

Intended audience: Programmers who want a fast introduction to the basics 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, up to 20 minutes on more involved topics.

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 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 101 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.

```
>>> 1 0
>>> -1 1
>>> 1- 2
>>> 1 = 2 3
>>> 1 == 2 4
>>> 1 != 2 5
>>> 1 < 2 6
>>> 1 <= 1 7
>>> 1 > 2 8
>>> 1 < 3 < 5 9
>>> 1 < 3 and 3 < 5 10
>>> 1 * 2 11
>>> 1 + 2 12
>>> 1 / 2 13
>>> 1 / 2.0 14
>>> 1 // 2 15
>>> 1 // 2.0 16
>>> from __future__ import division 17
>>> 1 / 2 18
>>> 1 / 2.0 19
>>> 1 // 2 20
>>> 1 // 2.0 21
>>> 9 % 3 22
>>> 10 % 3 23

>>> int 24
>>> int(2) 25
>>> int(2.0) 26
>>> int(2.1) 27
>>> int(2.9) 28
>>> int('2') 29
>>> int('2.0') 30
>>> int('four') 31

>>> float(2) 32
>>> float('2') 33
>>> float('2.9') 34

>>> 1 / 0 35

>>> 1 + 1.0 36

>>> 1 < 2 and 2 < 3 37
>>> 1 < 2 and not (2 < 3) 38
>>> 1 < 2 and True 39
>>> 1 < 2 and False 40

>>> 2 & 4 41
>>> 2 | 4 42
```

```
>>> ~2 43
>>> 2 << 1 44
>>> 2 << 2 45
```

Exercises: Numbers

Now it's your turn. Start the Python interpreter and see what happens (and what you can learn) when you type in the following:

```
>>> abs(4) 46
>>> abs(-4) 47

>>> pow(2, 8) 48
>>> 2**8 49
>>> 2 ** 16 50
>>> 2 ** 32 51
>>> 2 ** 31 52
>>> 2 ** 30 53
>>> int(2 ** 31 - 1) 54
>>> int(2 ** 31 - 1) + 1 55

>>> round(1.01) 56
>>> round(1.99) 57
>>> round(1.50) 58
>>> 1/3.0 59
>>> third = 1/3.0 60
>>> round(third) 61
>>> round(third, 1) 62
>>> round(third, 2) 63
>>> round(third, 3) 64

>>> round(1234.56, -1) 65
>>> round(1234.56, -2) 66
>>> round(1234.56, -3) 67

>>> type(int) 68
>>> callable(int) 69
>>> int() 70
>>> 0 == int() 71
>>> 0 is int() 72
>>> type(int) 73
>>> type(int()) 74
>>> int(4.3) 75
>>> int('4') 76

>>> int('four') 77
>>> int('z') 78
>>> int('c', 16) 79
```

Strings

>>> 'hello'	80
>>> "hello"	81
>>> 'bob's your uncle'	82
>>> 'bob\'s your uncle'	83
>>> "bob's your uncle"	84
>>> 'A quote (") mark'	85
>>> 'hello	86
>>> """hello	87
... there"""	
>>> '''bob's your "uncle"'''	88
>>> """bob's your "uncle""""	89
>>> 'hello\nthere\n'	90
>>> 'h' in 'hello'	91
>>> 'h' not in 'hello'	92
>>> 'hello'[0]	93
>>> s = 'hello'	94
>>> s[0] = 'j'	95

Exercise: Strings

>>> r'hello'	96
>>> r'hello' is 'hello'	97
>>> r'hello\n'	98
>>> r'hello\n' == 'hello\n'	99
>>> len(r'hello\n')	100
>>> len('hello\n')	101
>>> 2 * 'hello'	102
>>> 2 + 'hello'	103
>>> '2' + 'hello'	104
>>> type('hello')	105
>>> u'hello'	106
>>> type(u'hello')	107

String Methods

>>> len('hello')	108
>>> min('hello')	109
>>> max('hello')	110
>>> sorted('hello')	111

>>> 'hello'.startswith('h')	112
>>> 'hello'.startswith('he')	113
>>> 'hello'.endswith('lo')	114
>>> 'hello'.upper()	115
>>> 'HELLO'.lower()	116
>>> ' hello '.strip()	117
>>> ' hello '.rstrip()	118
>>> ' hello '.lstrip()	119

Exercise: String Methods

>>> sorted('hello')	120
>>> sorted('hello', reverse=True)	121
>>> reversed('hello')	122
>>> list(reversed('hello'))	123
>>> 'hello'.upper()	124
>>> 'HELLO'.isupper()	125
>>> 'hello'.title()	126
>>> 'Hello'.istitle()	127
>>> 'hello world'.title()	128
>>> 'hello world'.title().swapcase()	129
>>> '!' in '.?!'	130

Write a predicate (boolean) expression for a sentence, checking that it starts with a capital letter and ends with punctuation.

String Formatting with %

>>> '%d' % 99	131
>>> '%10d' % 99	132
>>> '%-10d' % 99	133
>>> '%s: %d' % ('count', 99)	134
>>> 3	135
>>> print 3	136
>>> print(3)	137
>>> print(3, 2)	138
>>> from __future__ import print_function	139
>>> print 3	140

```

>>> print(3) 141
>>> print(3, 2) 142
>>> print('three', 4) 143
>>> 'hello\n' 144
>>> print('hello\n') 145
>>> print('hello\nthere\n') 146

```

Exercise: String Formatting

```

>>> '%.2f' % (1/3.0) 147
>>> '%7.2f' % (1/3.0) 148
>>> '%3.2f' % 1/3.0 149
>>> '%3.2f' % (1/3.0) 150
>>> '%7.2e' % (1/3.0) 151
>>> '%7.2g' % (1/3.0) 152
>>> '%7.2g' % (.00001/3.0) 153
>>> 'Rate is %d%%' % 17 154
>>> '%s' % 4.3 155
>>> '%r' % 4.3 156
>>> help('FORMATTING') 157

```

String Formatting with .format()

```

>>> 'Hello'.format() 158
>>> 'Hello {0}'.format() 159
>>> 'Hello {0}'.format('Stu') 160
>>> '{0} {1}'.format('Hi', 'Stu') 161
>>> '{1} {0}'.format('Hi', 'Stu') 162
>>> '{0} {1}, {0}!'.format('Hi', 'Stu') 163

>>> '{0:d}'.format(99) 164
>>> '{0:10d}'.format(99) 165
>>> '{0:>10d}'.format(99) 166
>>> '{0:<10d}'.format(99) 167
>>> '{0:^10d}'.format(99) 168

>>> '{greet} {who}'.format( 169
...     greet='Hi',
...     who='Stu')

```

Exercise String Formatting with .format()

```
>>> v = 1/3.0 170
>>> '{0:f}'.format(v) 171
>>> '{0:4.2f}'.format(v) 172
>>> '{0:7.2f}'.format(v) 173
>>> '{0:7.4f}'.format(v) 174

>>> '{0:b}'.format(2) 175
>>> '{0:b}'.format(15) 176
>>> '{0:b}'.format(16) 177
>>> '{0:x}'.format(65535) 178
>>> '{0:o}'.format(65535) 179

>>> '{0:%}'.format(0.35) 180
>>> '{0:5.2%}'.format(0.35) 181

>>> '{0:10.{1}f}'.format(v, 3) 182
>>> '{0:10.{1}f}'.format(v, 5) 183
```

Introspection, str, repr

Note that `str` is a type, a class, a builtin, not the same as the deprecated standard library module `string`, and historically a builtin function, so don't be surprised by some obsolete references out there.

```
>>> 'hello' 184
>>> 'hello'.__class__ 185
>>> type('hello') 186
>>> 'hello'.__doc__ 187
>>> print('hello'.__doc__) 188

>>> str 189
>>> help(str) 190
>>> print(str.__doc__) 191
>>> str.strip 192
>>> help(str.strip) 193
>>> print(str.strip.__doc__) 194
>>> type(str) 195
>>> type(str.strip) 196

>>> help(dir) 197
>>> dir(str) 198
>>> dir() 199
>>> import __builtin__ 200
>>> dir() 201
>>> dir(__builtin__) 202
```

Exercise: Introspection, str, repr

```
>>> dir(str.strip) 203
>>> dir('hello') 204
>>> dir(str) == dir('hello') 205

>>> help(str) 206
>>> str(3) 207
>>> type(str(3)) 208
>>> help(repr) 209
>>> repr(3) 210
>>> float(4.3) 211
>>> str(4.3) 212
>>> repr(4.3) 213
>>> str('hello') 214
>>> repr('hello') 215
```

repr adds quotes so it can be eval'ed

```
>>> '%s' % 'hello' 216
>>> '%r' % 'hello' 217

>>> '{0!s}'.format('hello') 218
>>> '{0!r}'.format('hello') 219

>>> help(eval) 220
>>> str('hello') 221
>>> eval(str('hello')) 222
>>> hello 223
>>> str('hello') == 'hello' 224
>>> repr('hello') 225
>>> eval(repr('hello')) 226
```

Tuples, Lists

```
>>> [1, 2, 3] 227
>>> type([1, 2, 3]) 228
>>> (1, 2, 3) 229
>>> type(1, 2, 3) 230
>>> n = (1, 2, 3) 231
>>> type(n) 232
>>> type(1, 2, 3) 233
>>> type((1, 2, 3)) 234
>>> list((1, 2, 3)) 235
>>> tuple([1, 2, 3]) 236
>>> [1, 'b', 3] 237
>>> (1, 'b', 3) 238
```

>>> m = [1, 2, 3]	239
>>> n = (1, 2, 3)	240
>>> m[1] = 'b'	241
>>> m	242
>>> n[1] = 'b'	243
>>> n	244
>>> m + ['d']	245
>>> m	246
>>> n	247
>>> n + 'd'	248
>>> n + ('d')	249
>>> type('d')	250
>>> type(('d'))	251
>>> type(((('d'))))	252
>>> 'd'	253
>>> ('d')	254
>>> ('d',)	255
>>> tuple('d')	256
>>> type(('d',))	257
>>> n + ('d',)	258
>>> n	259
>>> n * 2	260
>>> m * 2	261
>>> tuple()	262
>>> type(tuple())	263
>>> n = ()	264
>>> n	265
>>> type(n)	266
>>> type(())	267
>>> m	268
>>> len(m)	269
>>> min(m)	270
>>> max(m)	271
>>> sorted(m)	272
>>> reversed(m)	273
>>> list(reversed(m))	274
>>> reversed('hello')	275
>>> list(reversed('hello'))	276
>>> (p, q) = (1, 2)	277
>>> p	278
>>> q	279
>>> p, q	280
>>> p, q = 3, 4	281
>>> p, q	282
>>> t1 = (1, 2, 3)	283
>>> t1	284
>>> t2 = 1, 2, 3	285
>>> t2	286
>>> t1 == t2	287

Exercise: Tuples, Lists

```
>>> m = [1, 2, 3] 288
>>> m 289
>>> m += 'd' 290
>>> m 291
>>> m.append('e') 292
>>> m 293
>>> m.append(5, 5, 6, 6, 7) 294
>>> m.append([5, 5, 6, 6, 7]) 295
>>> m 296
>>> del m[-1] 297
>>> m 298
>>> m.extend([5, 5, 6, 6, 7]) 299
>>> m 300
>>> 5 in m 301
>>> 5 not in m 302
>>> not 5 in m 303
>>> [5, 6] in m 304
>>> m 305
>>> m.append([5, 6]) 306
>>> m 307
>>> [5, 6] in m 308

>>> n = [1, 2, 4] 309
>>> m < n 310

>>> p, q = 1, 2 311
>>> p, q 312
>>> p, q = q, p 313
>>> p, q 314

>>> x, y, z = (1, 2, 3) 315
>>> x, y, z 316
>>> x, y, z = 1, 2, 3 317
>>> x, y, z 318
>>> x, y, z = [1, 2, 3] 319
>>> x, y, z 320
>>> x, y, z = 'xyz' 321
>>> x, y, z 322

>>> 'hello there'.split() 323
>>> 'a b c d'.split() 324
>>> ', '.join(['hello', 'there']) 325
>>> ', '.join(['one', 'two', 'three']) 326

>>> m.reverse() 327
```

The `reverse` and `sort` methods mutate a `list` and return `None`.

The `reversed` and `sorted` methods don't mutate a sequence, and they return a new sequence (actually an iterator).

```
>>> m 328
>>> m.sorted() 329
>>> m 330
>>> sorted(m) 331
>>> m 332
>>> m.sort() 333
>>> m 334
>>> m.sort(reverse=True) 335
>>> m 336
```

Sequence Indexing, Slicing

```
>>> m = ['jan', 'feb', 'mar', 337
...      'apr', 'may']

>>> m[0] 338
>>> m[3] 339
>>> m[-1] 340
>>> m[-2] 341
>>> m[0:1] 342
>>> m[0:2] 343
>>> m[0:-1] 344
>>> m[0:100] 345
>>> m2 = m[:] 346
>>> m2 347
>>> m2 == m 348
>>> m2 is m 349
>>> id(m2), id(m) 350
>>> help(id) 351
>>> del m2[0] 352
>>> m 353
>>> m2 354

>>> m[0] = 'January' 355
>>> m 356
>>> m[-1] = m[-1].capitalize() 357
>>> m 358
>>> del m[2] 359
>>> m 360
>>> m = range(10) 361
>>> m 362
```

Exercise: Sequence Indexing, Slicing

```
>>> m = [0, 1, 2] 363
>>> m[1] = [10, 20] 364
>>> m 365
>>> m = [0, 1, 2] 366
>>> m[1:2] = [10, 20] 367
>>> m 368
```

Create a list of integers from 0 to 100, inclusive, and then use a slice to pull out 0, 3, 6, 9, ..., 99.

Note that indexing and slicing work on strings and tuples, too, but remember they are immutable.

List Comprehensions

```
>>> range(8) 369
>>> [e for e in range(8)] 370
>>> [2 * e for e in range(8)] 371
>>> [2 + e for e in range(8)] 372
>>> [e for e in range(8) 373
...     if e % 2 == 0]

>>> ['%d * 2 == %d' % (e, 2 * e) 374
...     for e in range(8)]

>>> ['%d * 2 == %d' % (e, 2 * e) 375
...     for e in range(8)
...     if e % 2 == 0]

>>> [e for e in range(8) if e % 3 == 0] 376
```

Exercise: List Comprehensions

Create a list of the positive multiples of 3 less than 100.

```
>>> 'a b c'.split() 377
>>> list('abc') 378
>>> [(a, b) for a in range(3) 379
...     for b in ['a', 'b', 'c']]

>>> help(enumerate) 380
>>> enumerate(['a', 'b', 'c']) 381
>>> list(enumerate(['a', 'b', 'c'])) 382
>>> [(n, c) for (n, c) in 383
...     enumerate(['a', 'b', 'c'])]
```

```

>>> help(zip) 384
>>> zip(['one', 'two', 'three'], 385
...     (1, 2, 3))

>>> zip(['one', 'two', 'three'], 386
...     (1, 2, 3, 4))

```

Decorate, Sort, Undecorate Idiom (DSU)

```

>>> months = [ 387
...     ('Jan', 1, 31), ('Feb', 2, 28),
...     ('Mar', 3, 31), ('Apr', 4, 30)]

>>> sorted(months) 388
>>> dsu = [(days, (name, order, days)) 389
...     for (name, order, days) in months]

>>> dsu 390
>>> dsu.sort() 391
>>> dsu 392
>>> [ b for (a, b) in dsu ] 393

```

Exercise: Decorate, Sort, Undecorate Idiom (DSU)

Use the DSU idiom to sort `months` alphabetically.

Objects and Variables

Restart python to get a clean slate.

```

>>> id(1) 394
>>> help(id) 395

```

Everything 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*.

```

>>> id([]) 396
>>> [] 397
>>> dir([]) 398
>>> type([]) 399
>>> i = [] 400
>>> j = i 401
>>> id(i), id(j) 402
>>> id(i) == id(j) 403
>>> type([]).__bases__ 404

>>> id('xyz') 405
>>> id('xyz2') 406
>>> type('xyz') 407
>>> 'xyz' 408
>>> s = 'xyz' 409
>>> dir() 410
>>> s[1] = 'b' 411
>>> id('xyz') 412
>>> id(s) 413
>>> t = s 414
>>> dir() 415
>>> id(t) 416
>>> id(s) == id(t) 417
>>> t = 'xyz' 418
>>> id(s) == id(t) 419

>>> type([]) 420
>>> class AClass(list): 421
...     pass

>>> ac = AClass() 422
>>> type(ac) 423
>>> ac.__class__ 424
>>> ac.__class__.__bases__ 425
>>> ac.__class__.__bases__[0] 426
>>> ac.__class__.__bases__[0].__bases__ 427

```

Exercise: Objects and Variables

Restart the python interpreter.

```

>>> dir() 428
>>> i = 1 429
>>> i 430

```

```

>>> type(i) 431
>>> id(i) 432
>>> j = 1 433
>>> id(j) 434

>>> m = [1, 2, 3] 435
>>> m 436
>>> n = m 437
>>> n 438
>>> id(m) == id(n) 439
>>> m[1] = 'two' 440
>>> m 441
>>> n 442

```

Dictionaries

```

>>> int_to_month = [ 443
...     None, 'Jan', 'Feb', 'Mar']

>>> int_to_month[2] 444
>>> month_to_int = { 445
...     'Jan': 1, 'Feb': 2, 'Mar': 3 }

>>> month_to_int['Feb'] 446
>>> month_to_int['Apr'] 447
>>> month_to_int['Apr'] = 4 448
>>> month_to_int['Apr'] 449

>>> month_to_int.has_key('Feb') 450
>>> 'Feb' in month_to_int 451
>>> del month_to_int['Feb'] 452
>>> 'Feb' in month_to_int 453

>>> import operator 454
>>> ops = { 455
...     '+': operator.add,
...     '-': operator.sub, }

>>> e = '7+3' 456
>>> e[1], e[0], e[2] 457
>>> 7+3 458
>>> operator.add(7, 3) 459
>>> ops[e[1]](e[0], e[2]) 460
>>> '7'+ '3' 461
>>> e[1], int(e[0]), int(e[2]) 462
>>> ops[e[1]](int(e[0]), int(e[2])) 463
>>> e = '9-5' 464
>>> ops[e[1]](int(e[0]), int(e[2])) 465

```

Exercise: Dictionaries

```
>>> d = {'zero': 0, 'one' : 1, 'two' : 2} 466
>>> d['two'] 467
>>> d['three'] = 3 468
>>> d.keys() 469
>>> d.values() 470
>>> d.items() 471
>>> help(d.items()) 472
>>> help(d.items) 473

>>> dict([(k, v) for v, k in enumerate( 474
...     'zero one two three'.split())])
```

Rewrite the ops code above to split on spaces instead of assuming single digit numbers.

Blocks, for loops

```
>>> print('hello') 475
>>> print('there') 476

>>> i = 0 477
>>> while i < 5: 478
...     i += 1
...     print(i)

>>> temp = 15 479
>>> if temp <= 0: 480
...     print('Freezing')
...     elif temp < 10:
...         print('Cold')
...     elif temp < 20:
...         print('Temperate')
...     else:
...         print('Warm')

>>> for i in [1, 2, 3]: 481
...     print(i)
...     print(i * 2)

>>> for i in range(3): 482
...     for j in range(3):
...         print((i, j))

>>> import sys 483
>>> from __future__ import print_function 484
>>> for i in range(3): 485
```

```

...     for j in range(3):
...         print('( %d, %d) '
...             % (i, j), end='')
...     print()

>>> for i in (1, 2, 3):
...     print(i)
486

>>> d = {'zero': 0, 'one' : 1, 'two' : 2}
>>> for k, v in d.items():
...     print('key %s -> val %s' % (k, v))
487
488

>>> for t in d.iteritems():
...     print('key %s -> value %s' % t)
489

>>> months = 'jan feb mar apr may'.split()
>>> for m in reversed(months):
...     print(m)
490
491

```

Exercise: Blocks, for loops

```

>>> if []:
...     print('list non-empty')
492

>>> if [None]:
...     print('list non-empty')
493

>>> if '':
...     print('string non-empty')
494

>>> if 'False':
...     print('string non-empty')
495

>>> d = dict(one=1, two=2, three=3)
>>> for k in d:
...     print(k)
496
497

>>> for k in sorted(d):
...     print(k)
498

>>> for k in reversed(d):
...     print(k)
499

>>> range(10)
500
>>> help(range)
501
>>> range(5, 15)
502
>>> range(5, 15, 3)
503
>>> range(15, 5, -3)
504

```

Iterables, Generator Expressions

- In a for loop the expression is evaluated to get an iterable, and then `iter()` is called to produce an iterator.
- The iterator's `next()` method is called repeatedly until `StopIteration` is raised.
- `iter(foo)`
 - calls `foo.__iter__()` if it exists
 - next checks for `foo.__getitem__()`, calls it starting at zero, handles `IndexError` by raising `StopIteration`.
- Note `iter(callable, sentinel)` behaves differently

```
>>> m = [1, 2, 3] 505
>>> reversed(m) 506
>>> it = reversed(m) 507
>>> type(it) 508
>>> dir(it) 509
>>> it.next() 510
>>> it.next() 511
>>> it.next() 512
>>> it.next() 513
>>> it.next() 514
>>> it.next() 515

>>> for i in m: 516
...     print(i)

>>> m.next() 517
>>> it = iter(m) 518
>>> it.next() 519
>>> it.next() 520
>>> it.next() 521
>>> it.next() 522

>>> m.__getitem__(0) 523
>>> m.__getitem__(1) 524
>>> m.__getitem__(2) 525
>>> m.__getitem__(3) 526

>>> it = reversed(m) 527
>>> it2 = it.__iter__() 528
>>> hasattr(it2, 'next') 529

>>> m = [2 * i for i in range(3)] 530
>>> m 531
>>> type(m) 532
```

```

>>> mi = (2 * i for i in range(3)) 533
>>> mi 534
>>> type(mi) 535
>>> hasattr(mi, 'next') 536
>>> dir(mi) 537
>>> help(mi) 538
>>> mi.next() 539
>>> mi.next() 540
>>> mi.next() 541
>>> mi.next() 542

```

Exercise: Iterables, Generator Expressions

```

>>> m = [1, 2, 3] 543
>>> it = iter(m) 544
>>> it.next() 545
>>> it.next() 546
>>> it.next() 547
>>> it.next() 548

```

```

>>> for n in m: 549
...     print(n)

```

```

>>> it = iter(m) 550
>>> it2 = iter(it) 551
>>> list(it2) 552
>>> list(it) 553

```

```

>>> it1 = iter(m) 554
>>> it2 = iter(m) 555
>>> list(it1) 556
>>> list(it2) 557
>>> list(it1) 558
>>> list(it2) 559

```

```

>>> d = {'one': 1, 'two': 2, 'three':3} 560
>>> it = iter(d) 561
>>> list(it) 562

```

```

>>> mi = (2 * i for i in range(3)) 563
>>> list(mi) 564
>>> list(mi) 565

```

```

>>> import itertools 566
>>> help(itertools) 567

```

Writing Scripts

- Start with `#!/usr/bin/env python`
- Suffix `.py`
- Python creates `.pyc`
- Use lowercase and valid python identifiers

play1.py:

```
#!/usr/bin/env python
x = 3
y = 2
print(x + y)
```

```
>>> import play0 568
>>> import play1 569
>>> dir(play1) 570
>>> play1.x 571
>>> play1.y 572
>>> play1.z 573
>>> play1.z = 99 574
>>> play1.z 575
>>> dir(play1) 576
>>> del play1.z 577
>>> dir(play1) 578

>>> help(reload) 579
>>> reload(play1) 580
```

play2.py:

```
#!/usr/bin/env python

s = 'abc'
t = 'def'
def play():
    return s + t

play()
```

```
>>> from play2 import s, t 581
>>> dir(play2) 582
>>> s, t 583
```

play3.py:

```

#!/usr/bin/env python

def play(args):
    pass    # Put code here.

def test_play():
    pass    # Put tests here.

if __name__ == '__main__':
    test_play()    # This doesn't run on import.

>>> from play3 import *
>>> dir()

```

584
585

Exercise: Writing Scripts

Edit your own `play.py` and load it.

Defining and Calling Functions

```

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

```

586

```

>>> iseven(1)
>>> iseven(2)

```

587
588

```

>>> def add(x, y): return x + y

```

589

```

>>> add(1, 2)

```

590

```

>>> def plural(w):
...     if w.endswith('y'):
...         return w[:-1] + 'ies'
...     return w + 's'

```

591

```

>>> plural('word')
>>> plural('city')
>>> plural('fish')
>>> plural('day')

```

592
593
594
595

```

>>> def fact(n):
...     """factorial(n), -1 if n < 0"""
...     if n < 0:
...         return -1
...     if n == 0:
...         return 1
...     return n * fact(n - 1)

```

596

```

>>> fact.__doc__ 597
>>> fact(-1) 598
>>> fact(0) 599
>>> fact(1) 600
>>> fact(2) 601
>>> fact(3) 602
>>> fact(4) 603
>>> fact(10) 604
>>> fact(20) 605
>>> fact(30) 606
>>> fact(100) 607
>>> fact(500) 608
>>> fact(990) 609
>>> fact(1000) 610

>>> import sys 611
>>> sys.getrecursionlimit() 612

```

Exercise: Defining and Calling Functions

Write a function `triple(n)` in `triple.py` such that `triple(3)` returns 9. Include tests, preferably writing them first.

Import `triple.triple` and try it out.

Extend the plural function above to handle proper nouns that end in 'y', for example the correct plural of "Harry" is "Harrys".

Write a recursive predicate function `palindrome(s)` which returns whether or not the string passed to it is the same forwards and backwards. Write test cases first.

Generators

```

>>> def list123(): 613
...     yield 1
...     yield 2
...     yield 3

>>> list123 614
>>> list123() 615
>>> list(list123()) 616

>>> it = list123() 617
>>> it 618
>>> type(it) 619
>>> it.next() 620

```

```

>>> it.next() 621
>>> it.next() 622
>>> it.next() 623

>>> for i in list123(): 624
...     print(i)

>>> def list123(): 625
...     for i in [1, 2, 3]:
...         yield i

>>> list123() 626
>>> list(list123()) 627

```

Exercise: Generators

Write a generator that returns all the even numbers between 0 and 100.

Extend the generator to take start and stop parameters similar to the builtin `range` function.

Call by Object Reference

```

>>> def f1(i): 628
...     print('Old: %d, ' % i, end='')
...     i = i + 1
...     print('New: %d' % i)

>>> j = 3 629
>>> j 630
>>> f1(j) 631
>>> j 632

>>> def f2(m): 633
...     print('Old: %s' % m)
...     m.append(3)
...     print('New: %s' % m)

>>> n = [0, 1, 2] 634
>>> n 635
>>> f2(n) 636
>>> n 637

```

Classes and Instances

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:

1. locals, enclosing functions (or module if not in a function)
2. module, including `global`
3. built-ins

All namespaces changes (assignment, import, def, del) happen in the local scope.

<http://docs.python.org/tutorial/classes.html#python-scopes-and-name-spaces>

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

```
>>> class Point(object):                                     638
...     """Example point class"""
...     def __init__(self, x=0, y=0):
...         # Note that self exists by now
...         self.x, self.y = x, y
...         # (usually this comment would just be a blank line...)
...     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

>>> p1 = Point()                                           639
>>> p1                                                       640
>>> p1.translate(2, 4)                                       641
>>> p1                                                       642
>>> p1.translate(-3.5, 4.9)                                  643
>>> p1                                                       644
>>> p2 = Point(1, 2)                                         645
>>> p2                                                       646

>>> p1.__repr__()                                           647
>>> repr(p1)                                                 648
```

```

>>> dir(Point) 649
>>> dir(p1) 650
>>> set(dir(p1)) - set(dir(Point)) 651
>>> p1.__dict__ 652

>>> class Record(object): 653
...     pass

>>> r = Record() 654
>>> r.fname, r.lname = 'Jane', 'Doe' 655
>>> r.fname 656

```

Exercise: Classes and Instances

Write a class `Employee` that tracks first and last name, age, and manager.

Review: Classes

- Class creates a new namespace and a new class object, and wires them for inheritance.
- Calling the class object creates an instance.
- If attribute lookup finds a method then a method object is created. It handles sending `self` to the function.
- Classes can be used as simple records.
- Modules and functions also have writable slots.

Exceptions

```

>>> int('four') 657
>>> try: 658
...     int('four')
... except:
...     print('caught it')

>>> try: 659
...     int('four')
... except Exception as e:
...     print('caught this:\n%s\n%s'
...           % (e, repr(e)))
...     savee = e

```

```
>>> dir(savee)
>>> savee.args
```

660
661

The End

Thanks!