Transforming Code into Beautiful, Idiomatic Python

Raymond Hettinger

@raymondh

When you see this, do that instead!

- Replace traditional index manipulation with Python's core looping idioms

- Learn advanced techniques with for-else clauses and the two argument form of iter()

- Improve your craftmanship and aim for clean, fast, idiomatic Python code

Looping over a range of numbers

```
for i in [0, 1, 2, 3, 4, 5]:
    print i**2
```

```
for i in range(6):
    print i**2
```

```
for i in xrange(6):
    print i**2
```

Looping over a collection

```
colors = ['red', 'green', 'blue', 'yellow']


for i in range(len(colors)):
    print colors[i]


for color in colors:
    print color
```

Looping backwards

```
colors = ['red', 'green', 'blue', 'yellow']


for i in range(len(colors)-1, -1, -1):
    print colors[i]


for color in reversed(colors):
    print color
```

Looping over a collection and indicies

```
colors = ['red', 'green', 'blue', 'yellow']


for i in range(len(colors)):
    print i, '-->', colors[i]


for i, color in enumerate(colors):
    print i, '-->', colors[i]
```

Looping over two collections

```
names = ['raymond', 'rachel', 'matthew']
colors = ['red', 'green', 'blue', 'yellow']
```

```
n = min(len(names), len(colors))
for i in range(n):
    print names[i], '-->', colors[i]
```

```
for name, color in zip(names, colors):
    print name, '-->', color
```

```
for name, color in izip(names, colors):
    print name, '-->', color
```

Looping in sorted order

```
colors = ['red', 'green', 'blue', 'yellow']


for color in sorted(colors):
    print color


for color in sorted(colors, reverse=True):
    print color
```

Custom sort order

```
colors = ['red', 'green', 'blue', 'yellow']


def compare_length(c1, c2):
    if len(c1) < len(c2): return -1
    if len(c1) > len(c2): return 1
    return 0


print sorted(colors, cmp=compare_length)


print sorted(colors, key=len)
```

Call a function until a sentinel value

```
blocks = []
while True:
    block = f.read(32)
    if block == '':
        break
    blocks.append(block)
```

```
blocks = []
for block in iter(partial(f.read, 32), ''):
    blocks.append(block)
```

Distinguishing multiple exit points in loops

```
def find(seq, target):
    found = False
    for i, value in enumerate(seq):
        if value == tgt:
            found = True
            break
    if not found:
        return -1
    return i

def find(seq, target):
    for i, value in enumerate(seq):
        if value == tgt:
            break
    else:
        return -1
    return i
```

Dictionary Skills

- Mastering dictionaries is a fundamental Python skill

- They are fundament tool for expressing relationships, linking, counting, and grouping

Looping over dictionary keys

```
d = {'matthew': 'blue', 'rachel': 'green', 'raymond':
'red'}
```

```
for k in d:
    print k
```

```
for k in d.keys():
    if k.startswith('r'):
        del d[k]
```

```
d = {k : d[k] for k in d if not k.startswith('r')}
```

Looping over a dictionary keys and values

```
for k in d:
    print k, '-->', d[k]
```

```
for k, v in d.items():
    print k, '-->', v
```

```
for k, v in d.iteritems():
    print k, '-->', v
```

Construct a dictionary from pairs

```
names = ['raymond', 'rachel', 'matthew']
colors = ['red', 'green', 'blue']
```

```
d = dict(izip(names, colors))
{'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
```

```
d = dict(enumerate(names))
{0: 'raymond', 1: 'rachel', 2: 'matthew'}
```

Counting with dictionaries

```
colors = ['red', 'green', 'red', 'blue', 'green', 'red']

d = {}
for color in colors:
    if color not in d:
        d[color] = 0
    d[color] += 1


{'blue': 1, 'green': 2, 'red': 3}

d = {}
for color in colors:
    d[color] = d.get(color, 0) + 1

d = defaultdict(int)
for color in colors:
    d[color] += 1
```

Grouping with dictionaries -- Part I

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']
```

```
d = {}
for name in names:
    key = len(name)
    if key not in d:
        d[key] = []
    d[key].append(name)
```

```
{5: ['roger', 'betty'], 6: ['rachel', 'judith'],
 7: ['raymond', 'matthew', 'melissa', 'charlie']}
```

Grouping with dictionaries -- Part II

```
d = {}
for name in names:
    key = len(name)
    d.setdefault(key, []).append(name)
```

```
d = defaultdict(list)
for name in names:
    key = len(name)
    d[key].append(name)
```

Is a dictionary popitem() atomic?

```
d = {'matthew': 'blue', 'rachel': 'green', 'raymond':
'red'}


while d:
    key, value = d.popitem()
    print key, '-->', value
```

Linking dictionaries

```
defaults = {'color': 'red', 'user': 'guest'}
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args([])
command_line_args = {k:v for k, v in
                     vars(namespace).items() if v}
```

```
d = defaults.copy()
d.update(os.environ)
d.update(command_line_args)
```

```
d = ChainMap(command_line_args, os.environ, defaults)
```

Improving Clarity

- Positional arguments and indicies are nice
- Keywords and names are better
- The first way is convenient for the computer
- The second corresponds to how human's think

Clarify function calls with keyword arguments

```
twitter_search('@obama', False, 20, True)
```

```
twitter_search('@obama', retweets=False, numtweets=20,
popular=True)
```

Clarify multiple return values with named tuples

```
doctest.testmod()
(0, 4)
```

```
doctest.testmod()
TestResults(failed=0, attempted=4)
```

```
TestResults = namedtuple('TestResults', ['failed',
'attempted'])
```

Unpacking sequences

```
p = 'Raymond', 'Hettinger', 0x30, 'python@example.com'


fname = p[0]
lname = p[1]
age = p[2]
email = p[3]


fname, lname, age, email = p
```

Updating multiple state variables

```
def fibonacci(n):
    x = 0
    y = 1
    for i in range(n):
        print x
        t = y
        y = x + y
        x = t
```

```
def fibonacci(n):
    x, y = 0, 1
    for i in range(n):
        print x
        x, y = y, x+y
```

Tuple packing and unpacking

- Don’t under-estimate the advantages of updating state variables at the same time
- It eliminates an entire class of errors due to out-of-order updates
- It allows high level thinking: “chunking”

Simultaneous state updates

```
tmp_x = x + dx * t
tmp_y = y + dy * t
tmp_dx = influence(m, x, y, dx, dy, partial='x')
tmp_dy = influence(m, x, y, dx, dy, partial='y')
x = tmp_x
y = tmp_y
dx = tmp_dx
dy = tmp_dy
```

```
x, y, dx, dy = (x + dx * t,
                y + dy * t,
                influence(m, x, y, dx, dy, partial='x'),
                influence(m, x, y, dx, dy, partial='y'))
```

Efficiency

- An optimization fundamental rule
- Don’t cause data to move around unnecessarily
- It takes only a little care to avoid O(n**2) behavior instead of linear behavior

Concatenating strings

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']
```

```
s = names[0]
for name in names[1:]:
    s += ', ' + name
print s
```

```
print ', '.join(names)
```

Updating sequences

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']


del names[0]
names.pop(0)
names.insert(0, 'mark')


names = deque(['raymond', 'rachel', 'matthew', 'roger',
               'betty', 'melissa', 'judith', 'charlie'])


del names[0]
names.popleft()
names.appendleft('mark')
```

Decorators and Context Managers

- Helps separate business logic from administrative logic

- Clean, beautiful tools for factoring code and improving code reuse

- Good naming is essential.

- Remember the Spiderman rule: With great power, comes great respsonsibility!

Using decorators to factor-out administrative logic

```
def web_lookup(url, saved={}):
    if url in saved:
        return saved[url]
    page = urllib.urlopen(url).read()
    saved[url] = page
    return page
```

```
@cache
def web_lookup(url):
    return urllib.urlopen(url).read()
```

Caching decorator

```
def cache(func):
    saved = {}
    @wraps(func)
    def newfunc(*args):
        if args in saved:
            return newfunc(*args)
        result = func(*args)
        saved[args] = result
        return result
    return newfunc
```

Factor-out temporary contexts

```
old_context = getcontext().copy()
getcontext().prec = 50
print Decimal(355) / Decimal(113)
setcontext(old_context)
```

```
with localcontext(Context(prec=50)):
    print Decimal(355) / Decimal(113)
```

How to open and close files

```
f = open('data.txt')
try:
    data = f.read()
finally:
    f.close()
```

```
with open('data.txt') as f:
    data = f.read()
```

How to use locks

```
# Make a lock
lock = threading.Lock()


# Old-way to use a lock
lock.acquire()
try:
    print 'Critical section 1'
    print 'Critical section 2'
finally:
    lock.release()


# New-way to use a lock
with lock:
    print 'Critical section 1'
    print 'Critical section 2'
```

Factor-out temporary contexts

```
try:
    os.remove('somefile.tmp')
except OSError:
    pass
```

```
with ignored(OSError):
    os.remove('somefile.tmp')
```

Context manager: ignored()

```
@contextmanager
def ignored(*exceptions):
    try:
        yield
    except exceptions:
        pass
```

Factor-out temporary contexts

```
with open('help.txt', 'w') as f:
    oldstdout = sys.stdout
    sys.stdout = f
    try:
        help(pow)
    finally:
        sys.stdout = oldstdout
```

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

Context manager: redirect_stdout()

```
@contextmanager
def redirect_stdout(fileobj):
    oldstdout = sys.stdout
    sys.stdout = fileobj
    try:
        yield fieldobj
    finally:
        sys.stdout = oldstdout
```

Concise Expressive One-Liners

Two conflicting rules:

1. Don’t put too much on one line
2. Don’t break atoms of thought into subatomic particles

Raymond’s rule:

- One logical line of code equals one sentence in English

List Comprehensions and Generator Expressions

```
result = []
for i in range(10):
    s = i ** 2
    result.append(s)
print sum(result)
```

```
print sum([i**2 for i in xrange(10)])
```

```
print sum(i**2 for i in xrange(10))
```

Q & A