
Transforming Code into Beautiful, Idiomatic Python

Notes from Raymond Hettinger's talk at pycon US 2013 video, slides.

The code examples and direct quotes are all from Raymond's talk. I've reproduced them here for my own edification and the hopes that others will find them as handy as I have!

Looping over a range of numbers

```
1 for i in [0, 1, 2, 3, 4, 5]:
2     print i**2
3
4 for i in range(6):
5     print i**2
```

Better

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

`xrange` creates an iterator over the range producing the values one at a time. This approach is much more memory efficient than `range`. `xrange` was renamed to `range` in python 3.

Looping over a collection

```
1 colors = ['red', 'green', 'blue', 'yellow']
2
3 for i in range(len(colors)):
4     print colors[i]
```

Better

```
1 for color in colors:
2     print color
```

Looping backwards

```
1 colors = ['red', 'green', 'blue', 'yellow']
2
3 for i in range(len(colors)-1, -1, -1):
4     print colors[i]
```

Better

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

Looping over a collection and indices

```
1 colors = ['red', 'green', 'blue', 'yellow']
2
3 for i in range(len(colors)):
4     print i, '--->', colors[i]
```

Better

```
1 for i, color in enumerate(colors):
2     print i, '--->', color
```

It's fast and beautiful and saves you from tracking the individual indices and incrementing them.

Whenever you find yourself manipulating indices [in a collection], you're probably doing it wrong.

Looping over two collections

```
1 names = ['raymond', 'rachel', 'matthew']
2 colors = ['red', 'green', 'blue', 'yellow']
3
4 n = min(len(names), len(colors))
5 for i in range(n):
6     print names[i], '--->', colors[i]
7
8 for name, color in zip(names, colors):
9     print name, '--->', color
```

Better

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

`zip` creates a new list in memory and takes more memory. `izip` is more efficient than `zip`. Note: in python 3 `izip` was renamed to `zip` and promoted to a builtin replacing the old `zip`.

Looping in sorted order

```
1 colors = ['red', 'green', 'blue', 'yellow']
2
3 # Forward sorted order
4 for color in sorted(colors):
5     print color
6
7 # Backwards sorted order
8 for color in sorted(colors, reverse=True):
9     print color
```

Custom Sort Order

```
1 colors = ['red', 'green', 'blue', 'yellow']
2
3 def compare_length(c1, c2):
4     if len(c1) < len(c2): return -1
5     if len(c1) > len(c2): return 1
6     return 0
7
8 print sorted(colors, cmp=compare_length)
```

Better

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

The original is slow and unpleasant to write. Also, comparison functions are no longer available in python 3.

Call a function until a sentinel value

```
1 blocks = []
2 while True:
3     block = f.read(32)
4     if block == '':
5         break
6     blocks.append(block)
```

Better

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

`iter` takes two arguments. The first you call over and over again and the second is a sentinel value.

Distinguishing multiple exit points in loops

```
1 def find(seq, target):
2     found = False
3     for i, value in enumerate(seq):
4         if value == target:
5             found = True
6             break
7     if not found:
8         return -1
9     return i
```

Better

```
1 def find(seq, target):
2     for i, value in enumerate(seq):
3         if value == target:
4             break
5     else:
6         return -1
7     return i
```

Inside of every `for` loop is an `else`.

Looping over dictionary keys

```
1 d = {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
2
3 for k in d:
4     print k
5
6 for k in d.keys():
7     if k.startswith('r'):
8         del d[k]
```

When should you use the second and not the first? When you're mutating the dictionary.

If you mutate something while you're iterating over it, you're living in a state of sin and deserve what ever happens to you.

`d.keys()` makes a copy of all the keys and stores them in a list. Then you can modify the dictionary. Note: in python 3 to iterate through a dictionary you have to explicitly write: `list(d.keys())` because `d.keys()` returns a "dictionary view" (an iterable that provide a dynamic view on the dictionary's keys). See documentation.

Looping over dictionary keys and values

```
1 # Not very fast, has to re-hash every key and do a lookup
2 for k in d:
3     print k, '--->', d[k]
4
5 # Makes a big huge list
6 for k, v in d.items():
7     print k, '--->', v
```

Better

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

`iteritems()` is better as it returns an iterator. Note: in python 3 there is no `iteritems()` and `items()` behaviour is close to what `iteritems()` had. See documentation.

Construct a dictionary from pairs

```
1 names = ['raymond', 'rachel', 'matthew']
2 colors = ['red', 'green', 'blue']
3
```

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

For python 3: `d = dict(zip(names, colors))`

Counting with dictionaries

```
1 colors = ['red', 'green', 'red', 'blue', 'green', 'red']
2
3 # Simple, basic way to count. A good start for beginners.
4 d = {}
5 for color in colors:
6     if color not in d:
7         d[color] = 0
8     d[color] += 1
9
10 # {'blue': 1, 'green': 2, 'red': 3}
```

Better

```
1 d = {}
2 for color in colors:
3     d[color] = d.get(color, 0) + 1
4
5 # Slightly more modern but has several caveats, better for advanced
  users
6 # who understand the intricacies
7 d = collections.defaultdict(int)
8 for color in colors:
9     d[color] += 1
```

Grouping with dictionaries – Part I and II

```
1 names = ['raymond', 'rachel', 'matthew', 'roger',
2          'betty', 'melissa', 'judith', 'charlie']
3
4 # In this example, we're grouping by name length
5 d = {}
6 for name in names:
7     key = len(name)
8     if key not in d:
9         d[key] = []
10    d[key].append(name)
11
```

```
12 # {5: ['roger', 'betty'], 6: ['rachel', 'judith'], 7: ['raymond', 'matthew', 'melissa', 'charlie']}
13
14 d = {}
15 for name in names:
16     key = len(name)
17     d.setdefault(key, []).append(name)
```

Better

```
1 d = collections.defaultdict(list)
2 for name in names:
3     key = len(name)
4     d[key].append(name)
```

Is a dictionary `popitem()` atomic?

```
1 d = {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
2
3 while d:
4     key, value = d.popitem()
5     print key, '-->', value
```

`popitem` is atomic so you don't have to put locks around it to use it in threads.

Linking dictionaries

```
1 defaults = {'color': 'red', 'user': 'guest'}
2 parser = argparse.ArgumentParser()
3 parser.add_argument('-u', '--user')
4 parser.add_argument('-c', '--color')
5 namespace = parser.parse_args([])
6 command_line_args = {k:v for k, v in vars(namespace).items() if v}
7
8 # The common approach below allows you to use defaults at first, then
9 # override them
10 # with environment variables and then finally override them with
11 # command line arguments.
12 # It copies data like crazy, unfortunately.
13 d = defaults.copy()
14 d.update(os.environ)
15 d.update(command_line_args)
```

Better

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

`ChainMap` has been introduced into python 3. Fast and beautiful.

Improving Clarity

- Positional arguments and indices 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

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

Better

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

Is slightly (microseconds) slower but is worth it for the code clarity and developer time savings.

Clarify multiple return values with named tuples

```
1 # Old testmod return value
2 doctest.testmod()
3 # (0, 4)
4 # Is this good or bad? You don't know because it's not clear.
```

Better

```
1 # New testmod return value, a named tuple
2 doctest.testmod()
3 # TestResults(failed=0, attempted=4)
```

A named tuple is a subclass of tuple so they still work like a regular tuple, but are more friendly.

To make a named tuple, call `namedtuple` factory function in `collections` module:

```
1 from collections import namedtuple
2 TestResults = namedtuple('TestResults', ['failed', 'attempted'])
```

Unpacking sequences

```
1 p = 'Raymond', 'Hettinger', 0x30, 'python@example.com'
2
3 # A common approach / habit from other languages
4 fname = p[0]
5 lname = p[1]
6 age = p[2]
7 email = p[3]
```

Better

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

The second approach uses tuple unpacking and is faster and more readable.

Updating multiple state variables

```
1 def fibonacci(n):
2     x = 0
3     y = 1
4     for i in range(n):
5         print x
6         t = y
7         y = x + y
8         x = t
```

Better

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

Problems with first approach

- x and y are state, and state should be updated all at once or in between lines that state is mismatched and a common source of issues

-
- ordering matters
 - it's too low level

The second approach is more high-level, doesn't risk getting the order wrong and is fast.

Simultaneous state updates

```
1 tmp_x = x + dx * t
2 tmp_y = y + dy * t
3 # NOTE: The "influence" function here is just an example function, what
  it does
4 # is not important. The important part is how to manage updating
  multiple
5 # variables at once.
6 tmp_dx = influence(m, x, y, dx, dy, partial='x')
7 tmp_dy = influence(m, x, y, dx, dy, partial='y')
8 x = tmp_x
9 y = tmp_y
10 dx = tmp_dx
11 dy = tmp_dy
```

Better

```
1 # NOTE: The "influence" function here is just an example function, what
  it does
2 # is not important. The important part is how to manage updating
  multiple
3 # variables at once.
4 x, y, dx, dy = (x + dx * t,
5                 y + dy * t,
6                 influence(m, x, y, dx, dy, partial='x'),
7                 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

Basically, just don't move data around unnecessarily.

Concatenating strings

```
1 names = ['raymond', 'rachel', 'matthew', 'roger',
2         'betty', 'melissa', 'judith', 'charlie']
3
4 s = names[0]
5 for name in names[1:]:
6     s += ', ' + name
7 print s
```

Better

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

Updating sequences

```
1 names = ['raymond', 'rachel', 'matthew', 'roger',
2         'betty', 'melissa', 'judith', 'charlie']
3
4 del names[0]
5 # The below are signs you're using the wrong data structure
6 names.pop(0)
7 names.insert(0, 'mark')
```

Better

```
1 names = collections.deque(['raymond', 'rachel', 'matthew', 'roger',
2                             'betty', 'melissa', 'judith', 'charlie'])
3
4 # More efficient with collections.deque
5 del names[0]
6 names.popleft()
7 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 responsibility!

Using decorators to factor-out administrative logic

```
1 # Mixes business / administrative logic and is not reusable
2 def web_lookup(url, saved={}):
3     if url in saved:
4         return saved[url]
5     page = urllib.urlopen(url).read()
6     saved[url] = page
7     return page
```

Better

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

Note: since python 3.2 there is a decorator for this in the standard library: `functools.lru_cache`.

Factor-out temporary contexts

```
1 # Saving the old, restoring the new
2 old_context = getcontext().copy()
3 getcontext().prec = 50
4 print Decimal(355) / Decimal(113)
5 setcontext(old_context)
```

Better

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

How to open and close files

```
1 f = open('data.txt')
2 try:
3     data = f.read()
4 finally:
5     f.close()
```

Better

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

How to use locks

```
1 # Make a lock
2 lock = threading.Lock()
3
4 # Old-way to use a lock
5 lock.acquire()
6 try:
7     print 'Critical section 1'
8     print 'Critical section 2'
9 finally:
10    lock.release()
```

Better

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

Factor-out temporary contexts

```
1 try:
2     os.remove('somefile.tmp')
3 except OSError:
4     pass
```

Better

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

`ignored` is new in python 3.4, [documentation](#). Note: `ignored` is actually called `suppress` in the standard library.

To make your own `ignored` context manager in the meantime:

```
1 @contextmanager
2 def ignored(*exceptions):
3     try:
4         yield
5     except exceptions:
6         pass
```

Stick that in your `utils` directory and you too can ignore exceptions

Factor-out temporary contexts

```
1 # Temporarily redirect standard out to a file and then return it to
   normal
2 with open('help.txt', 'w') as f:
3     oldstdout = sys.stdout
4     sys.stdout = f
5     try:
6         help(pow)
7     finally:
8         sys.stdout = oldstdout
```

Better

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

`redirect_stdout` is proposed for python 3.4, bug report.

To roll your own `redirect_stdout` context manager

```
1 @contextmanager
2 def redirect_stdout(fileobj):
3     oldstdout = sys.stdout
4     sys.stdout = fileobj
5     try:
6         yield fileobj
7     finally:
8         sys.stdout = oldstdout
```

Concise Expressive One-Liners

Two conflicting rules:

-
- Don't put too much on one line
 - 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

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

Better

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

First way tells you what to do, second way tells you what you want.