Transforming_Code_into_Beautiful_Idiomatic_Python

September 8, 2019

1 Transforming Code into Beautiful, Idiomatic Python

by Raymond Hettinger Learn to take better advantage of Python's best features and improve existing code through a series of code transformations, "When you see this, do that instead."

Raymond gave the talk but did not take this notes. Notes by Alvin Chia. The notes may include further information.

```
In [1]: # This note is taken using Python version
        !python --version
```

```
Python 3.6.4 :: Anaconda custom (64-bit)
```

```
In [2]: from IPython.display import HTML
        HTML(' <iframe width="854" height="480" src="https://www.youtube.com/embed/OSGv2VnC0go
```

```
Out[2]: <IPython.core.display.HTML object>
```

2 Looping

2.1 Looping over a range of numbers

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

```
0
1
4
9
16
25
```

Is there a better way to improve existing code?

```
In [4]: for i in range(6):
            print(i**2)
```

```
0
1
4
9
16
25
```

2.2 Looping over a collection

```
In [5]: colors = ['red', 'green', 'blue', 'yellow']

In [6]: for i in range(len(colors)-1, -1, -1):
            print(colors[i])
```

```
yellow
blue
green
red
```

Horrible code above. Write pythonic code.

```
In [7]: for color in reversed(colors):
            print(color)
```

```
yellow
blue
green
red
```

2.3 Looping over a collection and indicies

```
In [8]: for i in range(len(colors)):
            print(i,'---->',colors[i])
```

```
0 ----> red
1 ----> green
2 ----> blue
3 ----> yellow
```

How do you rewrite pythonic without using indicies?

```
In [9]: # Use enumerate
        # Fast, beautiful code
        # Whenever you use indicies to index, something is wrong.
        for i, color in enumerate(colors):
            print(i, '---->', color)
```

```
0 ----> red
1 ----> green
2 ----> blue
3 ----> yellow
```

2.4 Looping over two collection

```
In [10]: names = ['raymond', 'rachel', 'matthew']
         colors = ['red', 'green', 'blue', 'yellow']

In [11]: n = min(len(names), len(colors))
         for i in range(n):
             print(names[i], '-->', colors[i])
```

```
raymond --> red
rachel --> green
matthew --> blue
```

Why do such a thing? Because it works in every other languages they learned. What is the Python way? Use zip. Actually, it was in the very first version of Lisp if you read the original paper came out on Lisp. zip has a deep history and is a proven winning performer. The code now is clean and beautiful.

```
In [12]: for name, color in zip(names, colors):
             print(name, '-->', color)
```

```
raymond --> red
rachel --> green
matthew --> blue
```

Anything wrong with above the code? To over loop over, it manifests a third list in memory. The third list consists of tuples each of which is its own separate object. The code uses more memory than the first. How to make a program run fast? On modern processors only one thing matters is the code running on L1 cache. If you have a cache miss, the Intel optimization guide has this horrifying line in it that says the cost of a cache miss is that simple move becomes as expensive as a floating point divide. It can go from a half clock cycle to 400 to 600 clock cycles. You can lose two and half orders of magnitude by not being in cache. If these lists are really big, the zip is not going to fit in cache. For Python 2, use izip (iterators) instead of zip. For Python 3, the built in zip does the same job as izip in Python 2.x (returns a generator instead of a list)

2.5 Looping in a sorted order

```
In [13]: colors = ['red', 'green', 'blue', 'yellow']

In [14]: for color in sorted(colors):
             print(color)
```

```
blue
green
red
yellow
```

How do you reversed the sorted list?

```
In [15]: for color in sorted(colors, reverse=True):
             print(color)
```

```
yellow
red
green
blue
```

2.6 Custom sort order

```
In [16]: colors = ['red', 'green', 'blue', 'yellow']
```

```
In [17]: # Old way using cmp parameter
         def compare_length(c1, c2):
             if len(c1) < len(c2): return -1
             if len(c2) > len(c2): return 1
             return 0
         # In Python 2.x
         # In Python 3.x the cmp parameter is removed completely
         # print(sorted(colors, cmp=compare_length))
```

Horrifiying slow. You can write a shorter function and faster. How many times will be this function call. If you have a million items in a list. And you are doing sort and the number of comparision is nlogn, so it is a log of a million base 2 is 20 million comparision which a long and slow. Is there a better way? Sorted colors key equal length. The key function gets called exactly once per key. Which is better? 1 call or 20 million calls?

```
In [18]: print(sorted(colors, key=len))
```

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

2.7 Call a function until a sentinel value

```
In [19]: # blocks = []
         # while True:
         #     blocks = f.read(32)
         #     if block == '':
         #         break
         #     blocks.append(block)
```

```
In [20]: # Use iter
         # blocks = []
         #for blocks in iter(partial(f.read, 32), ''):
         #    blocks.append(block)
```

2.8 Distinguishing multiple exit points in loop

```
In [21]: # Using flags which slow down your code
         def find(seq, target):
             found = False
             for i, value in enumerate(seq):
                 if value == target:
                     found = True
                     break
             if not found:
                 return -1
             return 1
```

```
In [22]: # Better way!
         def find(seq, target):
             for i, value in enumerate(seq):
                 if value == target:
                     break
             else:
                 # what the else means the code finished the loop
                 # which is not-break
                 return -1
             return 1
```

3 Dictionary Skills

- Mastering dictionaries is a fundamental Python skill
- They are fundamental tool for expressing relationships, linking, counting, and grouping

3.1 Looping over dictionary keys

```
In [23]: d = {'matthew': 'blue', 'rachel':'green', 'raymond':'red'}
```

```
In [24]: # needs to rehash every key
         for k in d:
             print(k,'-->',d[k])
```

```
matthew --> blue
rachel --> green
raymond --> red
```

```
In [25]: # needs to rehash every key
         for k in d.keys():
             print(k,'-->',d[k])

matthew --> blue
rachel --> green
raymond --> red
```

Originally, Python items() built a real list of tuples and returned that. That could potentially takes a lot of extra memory. Then, generators were introduced to the language in general, and that method was reimplemented as an iterator-generator method named iteritems(). The original remains for backwards compatibility. One of Python 3's changes is that items() now returns iterators, and a list is never fully built. The iteritems() method is also gone, since items() in Python 3 works like viewitems() in Python 2.7

```
In [26]: # better way uses tuple unpacking but makes a list of d.items() for Python 2.x
         # For Python 2.x: dict.items() return a copy of the dictionary's list of (key,value) p
         # For Python 3.x: dict.items() return an iterator over the dictionary's (key, value) p
         for k, v in d.items():
             print(k,'-->',v)

matthew --> blue
rachel --> green
raymond --> red
```

```
In [27]: # best way in Python 2.7 to use iterator
         # For Python 2.7 dict.iteritems(): Return an iterator over the dictionary's (key, val
         # for k, v in d.iteritems():
         #     print(k,'-->',v)
```

3.2 Construct a dictionary from pairs

```
In [28]: names = ['raymond', 'rachel', 'matthew']
         colors = ['red', 'green', 'blue']

In [29]: d = dict(zip(names, colors))

In [30]: d

Out[30]: {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
```

3.3 Counting with dictionaries

```
In [31]: colors = ['red', 'green', 'red', 'blue', 'green', 'red']

In [32]: d = {}
         # basic way to loop through a dictionary
         for color in colors:
```

```
            if color not in d:
                d[color] = 0
            d[color] += 1
        d
```

```
Out[32]: {'blue': 1, 'green': 2, 'red': 3}
```

Next level improvement, use get method

```
In [33]: d = {}
         for color in colors:
             d[color] = d.get(color, 0) + 1
         d
```

```
Out[33]: {'blue': 1, 'green': 2, 'red': 3}
```

To further improve the code, use defaultdict.

```
In [34]: from collections import defaultdict
         d = defaultdict(int) #default value for integers are 0
         for color in colors:
             d[color] += 1
         d = dict(d) # convert back to d when you don't need defaultdict
         d
```

```
Out[34]: {'blue': 1, 'green': 2, 'red': 3}
```

3.4 Grouping with dictionaries

```
In [35]: names = ['raymond', 'rachel', 'matthew', 'roger', 'betty', 'melissa', 'judith', 'charl
```

```
In [36]: d = {}
         for name in names:
             key = len(name)
             if key not in d:
                 d[key] = []
             d[key].append(name)
         d
```

```
Out[36]: {5: ['roger', 'betty'],
          6: ['rachel', 'judith'],
          7: ['raymond', 'matthew', 'melissa', 'charlie']}
```

```
In [37]: # Use dict.setdefault()
         d = {}
         for name in names:
             key = len(name)
             d.setdefault(key, []).append(name)
```

```
In [38]: # Modern way, speed up the code
         from collections import defaultdict
         d = defaultdict(list)
         for name in names:
             key = len(name)
             d[key].append(name)
         d

Out[38]: defaultdict(list,
                     {5: ['roger', 'betty'],
                      6: ['rachel', 'judith'],
                      7: ['raymond', 'matthew', 'melissa', 'charlie']})
```

3.5 Is a dictionary popitem() atomic?

```
In [39]: d = {'matthew': 'blue', 'rachel':'green', 'raymond':'red'}

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

```
raymond --> red
rachel --> green
matthew --> blue
```

Yes, it is atomic. Therefore, you don't have to put locks around it. It can be used between threads to atomically pull out a task.

3.6 Linking dictionaries

```
In [40]: import os
         import argparse
         from collections import ChainMap

         defaults = {'color': 'red', 'user':'guest'}
         parser = argparse.ArgumentParser(description="Linking dictionaries")
         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}
         # third dictionary not shown - os.environ

In [41]: # Traditional way to do it - found in standard library
         # Standard defaults, if someone specified environment variables, it should update and
         # Command line arguments should take precedence over environment variables.
         # To be fast, don't copy
         d = defaults.copy()
         d.update(os.environ)
         d.update(command_line_args)
```

```
In [42]: # Use ChainMap instead
         # Precedence order: command_line_args > os.environ > defaults
         # Links all independent dictionary together without copying
         d = ChainMap(command_line_args, os.environ, defaults)

         #In Python2.7: from ConfigParser import _ChainMap as ChainMap
```

What is the purpose of collections.ChainMap? In Python 3.3 a ChainMap class was added to the collections module: A ChainMap class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls. Point out two other motivations/advantages/differences of ChainMap, compared to using a dict-update loop, thus only storing the "final" version": - More information: since a ChainMap structure is "layered", it supports answering question like: Am I getting the "default" value, or an overridden one? What is the original ("default") value? At what level did the value get overridden (borrowing @b4hand's config example: user-config or command-line-overrides)? Using a simple dict, the information needed for answering these questions is already lost. - Speed tradeoff: suppose you have N layers and at most M keys in each, constructing a ChainMap takes O(N) and each lookup O(N) worst-case[*], while construction of a dict using an update-loop takes O(NM) and each lookup O(1). This means that if you construct often and only perform a few lookups each time, or if M is big, ChainMap's lazy-construction approach works in your favor.

[*] The analysis in (2) assumes dict-access is O(1), when in fact it is O(1) on average, and O(M) worst case. See more details here.

4 Improving Clarity

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

4.1 Clarity function calls with keyword arguments

```
In [43]: def twitter_search(msg, retweets, numtweets, popular):
             pass
```

```
In [44]: # .... somewhere in the code
         # Code commonly found in client-side customer base
         # what is False? what is 20? what is True?
         twitter_search('@obama', False, 20, True)
```

```
In [45]: # Replace unreadable code with keyword arguments
         # Save microseconds of compute time or hours of programmer time?
         twitter_search('@obama', retweets=False, numtweets=20, popular=True)
```

4.2 Clarify multiple return values with named tuples

```
In [46]: def pt_in_circle():
             return 0, 4
```

```
# what are 0 and 4
pt_in_circle()
```

```
Out[46]: (0, 4)
```

Use collections.namedtuple instead. - namedtuple instances are memory efficient as regular tuples because they do not have per-instance dictionaries. - Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements. - Python Documentation here

```
In [47]: from collections import namedtuple

         # declare a namedtuple
         PointInCircle = namedtuple("PointInCircle", "x y")

         def pt_in_circle():
             # Clarity in code and readability
             return PointInCircle(x=0, y=4)

         # Readable __repr__ with a name=value style
         pt_in_circle()
```

```
Out[47]: PointInCircle(x=0, y=4)
```

```
In [48]: # Use tuple unpacking
         x, y = pt_in_circle()
```

Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules:

```
In [49]: EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade

         import csv
         for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
             print(emp.name, emp.title)
```

4.3 Unpacking sequences

```
In [50]: p = 'Raymond', 'Hettinger', 0x30, 'python@example.com'
```

```
In [51]: # Longer way using indexing to unpack
         fname = p[0]
         lname = p[1]
         age = p[2]
         email = p[3]
```

```
In [52]: # Better and readable way to unpack
         fname, lname, age, email = p
```

4.4 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

4.5 Concatenating strings

```
In [53]: names = ['raymond', 'rachel', 'matthew', 'roger', 'betty', 'melissa', 'judith', 'charl
```

```
In [54]: # c style to concatenate string
         s = names[0]
         for name in names[1:]:
             s += ', ' + name
         print(s)
```

```
raymond, rachel, matthew, roger, betty, melissa, judith, charlie
```

```
In [55]: # Use .join instead
         print(", ".join(names))
```

```
raymond, rachel, matthew, roger, betty, melissa, judith, charlie
```

4.6 Updating sequences

```
In [56]: names = ['raymond', 'rachel', 'matthew', 'roger', 'betty', 'melissa', 'judith', 'charl
         # Using indicies
         del names[0]
         names.pop()
         names.insert(0,'mark')
         names
```

```
Out[56]: ['mark', 'rachel', 'matthew', 'roger', 'betty', 'melissa', 'judith']
```

```
In [57]: # Uses deque instead
         from collections import deque
         names = deque(['raymond', 'rachel', 'matthew', 'roger', 'betty', 'melissa', 'judith',
         del names[0]
         names.popleft()
         names.appendleft('mark')
         names
```

```
Out[57]: deque(['mark', 'matthew', 'roger', 'betty', 'melissa', 'judith', 'charlie'])
```

5 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 with great responsibility!

5.1 Using decorators to factor-out administrative logic

For the function below, the administrative and business logic are all mixed together. 1. Administrative logic is cache url in a dictionary that way if I go and look the same web page over and over again, I simply remember it. 2. Business logic is opening a url and returning a web page

```
In [58]: def web_lookup(url, saved={}):
             if url in saved:
                 return saved[url]
             page = urllib.urlopen(url).read()
             saved[url] = page
             return page
```

How do we factor-out the administrative logic? By using decorators.

```
In [59]: # for python 3
         from functools import lru_cache

         @lru_cache(maxsize=100)
         def web_lookup(url):
             return urllib.urlopen(url).read()
```

```
In [60]: # for python 2, write a simple caching
         from functools import wraps
         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

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

What does wraps does? Read here

5.2 Factor-out temporary contexts

From Python docs: The usual start to using decimals is importing the module, viewing the current context with getcontext() and, if necessary, setting new values for precision, rounding, or enabled traps:

```
In [61]: from decimal import *
         getcontext()
```

```
Out[61]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clar

In [62]: getcontext().prec = 7 # Set a new precision
```

Back to the problem:

```
In [63]: # Copy and stored old context. Set new precision to compute. Then reset to previous c
         old_context = getcontext().copy()
         getcontext().prec = 50
         print(Decimal(355)/Decimal(113))
         setcontext(old_context)
```

```
3.1415929203539823008849557522123893805309734513274
```

```
In [64]: # Here is a better way. Using a local context manager, it has reusable logic
         with localcontext(Context(prec=50)):
             print(Decimal(355)/Decimal(113))
         print(Decimal(355)/Decimal(113))
```

```
3.1415929203539823008849557522123893805309734513274
3.141593
```

5.3 How to open and close file

```
In [65]: f = open('data.txt')
         try:
             data = f.read()
         finally:
             f.close()

In [66]: # A simple way
         with open('data.txt') as f:
             data = f.read()
```

5.4 How to use locks

```
In [67]: # Either this
         from threading import Thread, Lock
         lock = Lock()

In [68]: # Or this (based from youtube presentation)
         import threading

         # Make a lock
         lock = threading.Lock()

         # Old way to use a lock
```

```
        lock.acquire()
        try:
            print("Critical Session 1")
            print("Critical Session 2")
        finally:
            lock.release()
```

```
Critical Session 1
Critical Session 2
```

Separate the administration logic of getting a lock from printing by using context manager:

```
In [69]: with threading.Lock():
             print("Critical Session 1")
             print("Critical Session 2")
```

```
Critical Session 1
Critical Session 2
```

5.5 How to remove file

```
In [70]: import os
         try:
             os.remove('somefile.tmp')
         except OSError:
             pass
```

```
In [71]: # Better Alternative:
         import os
         from contextlib import suppress

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

5.6 How to redirect to stdout

```
In [72]: # C style using try, finally to redirect to stdout/stderr
         import sys
         with open('help.txt', 'w') as f:
             oldstdout = sys.stdout
             sys.stdout = f
             try:
                 help(pow)
             finally:
                 sys.stdout = oldstdout
```

A better way is to use context manager. Context manager for temporarily redirecting sys.stdout to another file or file-like object. This tools adds flexibility to existing functions or classes whose output is hardwired to stdout. For example, the output of help() normally is sent to sys.stdout. You can capture the output in a string by redirecting the output to an io.StringIO object

```
In [73]: import io
         from contextlib import redirect_stdout
         f = io.StringIO()
         with redirect_stdout(f):
             help(pow)
         f.getvalue()
```

```
Out[73]: 'Help on built-in function pow in module builtins:\n\npow(x, y, z=None, /)\n    Equiva
```

To send output of help() to a file on disk, redirect the output to a regular file

```
In [74]: from contextlib import redirect_stdout
         with open('help.txt', 'w') as f:
             with redirect_stdout(f):
                 help(pow)
```

```
In [75]: !cat help.txt
```

```
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

To send output of help() to sys.stderr

```
In [76]: from contextlib import redirect_stdout
         with redirect_stdout(sys.stderr):
             help(pow)
```

```
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

Note that the global side effect on sys.stdout means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

contexlib.redirect_stderr is similar to redirect_stdout but redirecting sys.stderr to another file or file-like object

5.7 How to use @contextmanger new in Python 3.6

Context manager docs are here

```
In [77]: # Example to use @contextmanger but not recommend to generate HTML Tag
         from contextlib import contextmanager

         @contextmanager
         def tag(name):
             print("<%s>" % name)
             yield
             print("<%s>" % name)

         with tag("h1"):
             print("foo")
```

```
<h1>
foo
<h1>
```

6 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 - One logical line = One statement

6.1 List Comprehensions and Generator Expression

```
In [78]: result = []
         for i in range(11):
             s = i ** 2
             result.append(s)
         print(sum(result))
```

```
385
```

```
In [79]: # Better way using List Comprehension Alternative
         print(sum([i ** 2 for i in range(11)]))
```

```
385
```

Why is the second alternative better? The first method shows how to approach the problem of calculating the sum. The second method shows exactly what the problem wants that is the sum of squares. It shows a single unit of though in terms of mathematics by taking sum of square of i from 1 to 10.

```
In [80]: # There's a even better way. I took an eraser and erase the square brackets.
         # It becomes generator expression
         print(sum(i ** 2 for i in range(11)))
```

```
385
```