The Zen of Python and Its Application

Readability Promotes Reusable Code

CSCI 3155 Principles of Programming Languages

Jessica Lynch
Noah Dillon
Max Harris

Python is highly readable due to its complete set of style guidelines and Pythonic idioms relayed in detail in the Process Python Enhancement Proposal (PEP) #8, “Style Guide for Python Code” created by GvR, Warsaw and Coghlan in 2001. On August 19, 2004, “The Zen of Python” by Pythoneer Tim Peters was approved as Informational PEP 20. While PEP 8 is all about the rules for structuring your code, PEP 20 conveys the inspiration behind how you write your Python code. You can access The Zen of Python by visiting https://www.python.org/dev/peps/pep-0020 or via the Python terminal or command prompt by typing >>import this and pressing Enter. The below list of Python Zen points will print to the screen unnumbered.

```
1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!
```

Python according to its creator Guido van Rossum provides a higher level of readability aimed at the creation of reusable code. Guido writes in his original Python Style Guide essay, "… code can hardly be considered reusable if it's not readable." In industry, reusable code is coveted to help achieve time and memory efficiency. Why reinvent code that has already been written? This is not a hypothetical question. Here are the main reasons programmers opt to write their own version of pre-existing code:

```
(1) The code (despite doing its job) is badly written, or in other words, is a hack job!
(2) The code is hard to understand (perhaps, just too complex and/or missing comments) even
    doing its job.
(3) The programmer is in too big of a hurry and can code up quickly what s/he needs.
```

Let's focus on statements (1) and (2) because they often bring about statement (3). Statements (1) and (2) in the case of Python are the result of one or more violations of the Python style guidelines (PEP 8) which are summarized in the Zen of Python (PEP 20).

The first seven Zen points of PEP 20 address the beautiful nature of Python code if written well, that is, according to the Zen of Python. Beautiful code is achieved through a combination of explicit, simple, sparse and flat attributes to ensure readability and therefore the potential for reusability! To illustrate, we will analyze the below code examples based on the task of walking a directory tree containing multiple directories to generate a list of all file paths of files contained within. Our first example violates several of the Zen points 1 - 7 and we progressively improve upon this example with our second (better) and third (best) examples.

```
#Example 1:
import os.path as op

def generate_file_list( filepath ):
   pathList = []
   path0    = filepath
   dirList0 = os.listdir( path0 )

   for p0 in dirList0:
      path1 = op.join( path0, p0 )
      if op.isdir( path1 ):
         dirList1 = os.listdir( path1 )
         for p1 in dirList1:
            path2 = op.join( path1, p1 )
            if op.isdir( path2 )
               dirList2 = os.listdir( path2 )

               for p2 in dirList2:
                  path3 = op.join( path2, p2 )
                  if op.isdir( path3 ):
                     dirList3 = os.listdir( path3 )

                     for p3 in dirList3:
                        path4 = op.join( path3, p3 )
                        if op.isdir( path4 ):
                           dirList4 = os.listdir( path4 )

                           for p4 in dirList4:
                              pathList.append( op.join( path4, p4 ) )

                        else:
                          pathList.append( op.join( path3, p3 ) )

                  else:
                     pathList.append( op.join( path2, p2 ) )

             else:
                pathList.append( op.join( path1, p1 ) )

      else:
         pathList.append( op.join( path0, p0 ) )

   return pathList
```

Example 1, although it may save memory by avoiding recursion, fails to adhere to Zen points 1, 3, 5 and 6, which furthermore, hinders readability and fails to adhere to Zen point 7.

```
#Example 2:
import os.path as op

def generate_file_list( filepath ):
   pathList = []
   for root, dirs, files in os.walk( filepath ):
      for filename in files:
         pathList.append( op.join(root, filename) )
      for dir in dirs:
         generate_file_list( dir )
   return pathList
```

Example 2 is a lot more readable due to its simplicity achieved through less lines of code. However, it does a lot of unnecessary work behind the scenes. Python's os.walk module is useful to lessen lines of code, but when dealing with big data i.e. millions of lines of code to traverse and analyze, the implementation of this module which uses recursion can be expensive and time consuming.

```
#Example 3:
import os.path as op

def generate_file_list( filepath ):
   pathList = []
   if op.isdir( filepath ):
      for p0 in os.listdir( filepath ):
         path1 = op.join( filepath, p0 )
         if op.isdir( path1 ):
            pathList += generate_file_list( path1 )
         else:
            pathList.append( path1 )
   return pathList
```

Example 3 is in one way or another beautiful, explicit, simple, flat and sparse, and is therefore readable making it reusable code which adheres to the Zen of Python points 1 through 7.

Zen point 8: "Special cases are not special enough to break the rules" tells us to find another way despite the temptation of hacking together a solution. The rules are there to provide structure and organization and therefore help to secure efficiency. Breaking them therefore, as we know, can result in hard-to-read code that cannot be reused. For example, the rejected PEP 315 tried to add the functionality of do while loops to Python. However, the implementation is more complicated and less readable.

```
#Conventional way
while(True):
    <setup code>
    if(<end condition>):
        break
    <loop code>
```

```
#Proposed way
do:
   <setup code>
while(<end condition>):
   <loop code>
```

One can also argue that if there is already a good way to do something, why implement another way. This goes against another PEP 20 Zen point that “there should be one – and preferably only one – obvious way to do it.” Zen point 9: “Although practicality beats purity”, on the other hand, addresses the rare case where the rules can be ignored to ensure a more straight-forward answer. As is written in PEP 8, “A Foolish Consistency is the Hobgoblin of Little Minds.” Know when to be inconsistent – sometimes the style guide just doesn’t apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don’t hesitate to ask!

Zen point 10: “Errors should never pass silently” and zen point 11: “Unless explicitly silenced” address the significance of implementing error-checking and error-handling in our code.

```
#Bad example:
try:
    <erroneous code>
except:
    pass

#Good example:
try:
    <erroneous code>
except:
    try:
        <fixing code>
    except:
         print <error>
         raise
```

Zen point 12: “In the face of ambiguity, refuse the temptation to guess” means that when something is not clear, look it up and figure it out. Make sure you know and fully comprehend before coming to a conclusion. In the example below, we illustrate how easily a guess about a try except method can have negative results. When we make assumptions, we risk producing erroneous code.

```
#Bad example where the variable err leaks
try:
   with open(fn, 'r') as f:
      lines = list(f)
except( IOError, OSError), err:
   log_error( err )

#Good example where the variable err no longer leaks
try:
   with open(fn, 'r') as f:
      lines = list(f)
except( IOError, OSError) as err:
   log_error( err )
```

Zen point 13 says "There should be one – and preferably only one – obvious way to do it." The below examples illustrate both impractical and practical approaches.

```
#Bad (impractical) example
i = 0
while i < len(array):
    print array[i]
    i+=1

#Good (practical and obvious) example
for element in array:
    print element
```

This is a counter to Perl's motto: In your Python code, there should be one "best" way of doing something. However, this one best approach may not be obvious at first unless you are Guido van Rossum to which Zen point 14 eludes, "Although, that way may not be obvious at first unless you're Dutch." "May not be obvious at first" means that you can always find a way to do something but the first thing you think of probably won't be the most efficient.

Zen point 15: "Now is better than never" and 16: "Although never is often better than *right* now" remind us that sometimes a working but less than eloquent solution needs implemented immediately with plans to update and refactor it later. However, when this is done, precautions need taken to ensure the robustness of the code being implemented. Adequate testing needs applied and so forth to catch any and all bugs. Follow through with updating and refactoring must be carried out to comply with the principles of code design.

Zen points 16: “If the implementation is hard to explain, it’s a bad idea” and 17: “If the implementation is easy to explain, it may be a good idea” address the importance of writing readable code. The below examples help illustrate both code that is hard to explain and code that is easy to explain. The former hinders readability and therefore potential for reusability. However, just because code is easy to explain does not mean it is the obvious solution or even correct. Zen point 17 stresses the importance of checking our work and not settling on the first simple solution before we fully know and comprehend the problem.

```python
#Code that is hard to explain
def hard():

    import xml.dom.minidom
    document = xml.dom.minidom.parseString(
        '''<menagerie><cat>Fluffers</cat><cat>Cisco</cat></menagerie>''')
    menagerie = document.childNodes[0]
    for node in menagerie.childNodes:
        if node.childNodes[0].nodeValue== 'Cisco' and node.tagName == 'cat':
            return node

#Code that is easy to explain
def easy(maybe):

    import lxml
    menagerie = lxml.etree.fromstring(
        '''<menagerie><cat>Fluffers</cat><cat>Cisco</cat></menagerie>''')
    for pet in menagerie.find('./cat'):
        if pet.text == 'Cisco':
            return pet
```

The last Zen point 19: “Namespaces are one honking great idea – let’s do more of those!” encourages us to optimize use of namespaces in our Python code. A namespace is a mapping from names to objects, with the property that there is zero relation between names in different namespaces. In Python, namespaces are defined by the individual modules, and since modules can be contained in hierarchical packages, then name spaces are hierarchical too. They are usually implemented as Python dictionaries, although this is abstracted away. Namespaces are typically employed for the purpose of grouping symbols and identifiers around a particular functionality helping to create a higher level of organization while promoting readability and the potential for reusability.

In general when a module is imported then the names defined in the module are defined via that module's name space, and are accessed in from the calling modules by using the fully qualified name.

```
# Example 1: Assume moduleA defines two functions : func1() and func2() and one class : clas
import moduleA

moduleA.func1()
moduleA.func2()
a = moduleA.class1()
```

The "from … import …" can be used to insert the relevant names directly into the calling module's namespace, and those names can be accessed from the calling module without the qualified name as shown below. However, you risk overwriting any pre-existing names with no warning. In order to avoid this, apply an alias by using from module import name as nickname as shown below.

```
# Example 2: Assume moduleA defines two functions : func1() and func2() and one class : clas
from moduleA import func1 as f1

func1()
func2() # this will fail as an undefined name, as will the full name moduleA.func2()
a = class1() # this will fail as an undefined name, as will the full name moduleA.class1()
```

In conclusion, the hallmark of Python design guidelines addressed in detail in PEP 8 and in a summarized version in PEP 20 is the promotion of reusable code by ensuring readability. We can combine various attributes of beautiful code and good coding practices to achieve readability which facilitates the developer's understanding of the code being reused. The promotion of code reuse can decrease resource constrain, strengthen communication among engineering teams, and increase productivity. Such benefits should inspire all programmers to write beautiful code.

Resources: http://docs.python-guide.org/en/latest/writing/style/
http://ruben.verborgh.org/blog/2013/02/21/programming-is-an-art/

https://www.python.org/doc/essays/foreword/ (Guido van Rossum's original Python Style Guide essay)
http://neopythonic.blogspot.com/ (Guido's blog)
http://www.stat.washington.edu/~hoytak/blog/whypython.html
http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide
http://artifex.org/~hblanks/talks/2011/pep20_by_example.html (Blanks, Hunter)