Python Best Practices

Written for CD/LSC/DBI/DBA by Randolph J Herber, Oct 9, 2008.

Abstract

This document tries to describe some of the issues involved in using the Python language properly. Python has typed values instead of typed variables, which means the programmer and code reviewer needs to assure that appropriate values are passed to functions and methods. Python uses white space instead of punctuation or keywords to delimit boundaries of program constructs, which means the programmer and code reviewer needs to assure that only and all appropriate code is within such program constructs. When Python detects programming errors, it usually is at execution time; therefore testing is required. Python supports object oriented design and coding.

Friends, users and programmers of Python, lend me your attention.

1 Some aphorisms pertinent to programming

1. Let the computer do the work.
2. The price of quality is eternal vigilance: trust, but verify.
3. You can not test quality into something. You only can test that the quality may be there.
4. Write code so that it reads right. Do not write surprising code. Eschew obfuscation!
5. Detect errors as soon as possible: requirements, design, coding, testing, etc. Generally, the earlier an error is detected, the cheaper and faster it is fixed.
6. Write functions and methods to do only one thing and do it well.
7. Do not repeat yourself. Refactor the code to make common code a function. Use classes or functions to pass into that function commands.[1]
8. Keep the interfaces between sections of a program to a minimum. Ideally, interfaces should occur as public, documented function, method or constructor calls only. One section should not reach into another section to examine or tweak a value or use a private function or method in another section.

2 Some observations pertinent to programming

1. Understand the program specifications.
2. Verify that specifications mean the same thing to the customer, the programmer, the tester and the user.
3. Testing verifies proper output of the program for a given set of inputs delivered in a particular way to the program. It is possible that the program generates the correct result for incorrect reasons.[2]

   Practical testing is a sampling process. It is appropriate to test common cases of both correct and erroneous input, boundary cases of correct and erroneous input and all cases of correct combinations of input where the number of combinations is “small.” If the consequences of an error in a portion of the program is significant, the testing of that portion of the program also should be significant. Testing does not put quality into a program, find all defects in a program or guarantee that the program will be fixed correctly or at all.
4. Since Python supports object oriented design and coding, having knowlege of object oriented design patterns is useful. One source of such knowlege is Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, (ISBN 0-201-63361-2). There are object oriented design patterns which are not effective.[3]

1Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, ISBN 0-201-63361-2

2http://en.wikipedia.org/wiki/Halting_problem

3AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray, ISBN 0-471-19713-0. There are later editions.

3 Philosophy of Python

The philosophy of Python is described briefly by a document called "Zen of Python[4]."

In Python, the types are associated with values and not variables.

Python uses white space and indentation to identify program structure. Code easily can move into and out of control statements merely by changing indentation.

Python encourages the immediate writing of code and defers the detection of most error conditions to execution time.

Python generally supplies only one implementation of a given data structure such as lists, sets and maps.

One person, Guido van Rossum[5], is in control of the design and style of Python.

4 Issues with Python to consider while coding

1. Always check the Python documentation[6] as you code to assure that you are using the proper method calls on an object and that the arguments are in the correct order. The O'Reilly Python Essential Reference by David Beazley[7] is a useful guide to the Python language.
2. `if` statements have less execution cost than function or method calls.

   Therefore, debugging code which is to be disabled in production should be protected from execution by an inline `if` and not an `if` in a debugging function or method.

   Do this (this is a long example):

```
import types


def isNumber(x):
  xType = type(x)
  for t in (types.FloatType,
            types.IntType,
            types.LongType):
    if type(x) == t:
      return 1
  return None


def toBeTested(self,arg):
  if DEBUG:
    if not isNumber(arg):
```

4http://www.python.org/dev/peps/pep-0020/
5http://www.python.org/uido/ and http://wiki.python.org/moin/BDFL
6http://www.python.org/doc/
7http://safari.oreilly.com/0672328623

```
      raise Exception(
        "toBeTested(%r) argument not numeric" %
        arg)
  print "in the place of more code"
```

Do not do this (this also is a long example):

```
import types

def isNumber(x):
  xType = type(x)
  for t in (types.FloatType,
            types.IntType,
            types.LongType):
    if type(x) == t:
      return 1
  return None

def debuggingTest(arg):
  if DEBUG:
    if not isNumber(arg):
      raise Exception(
        "toBeTested(%r) argument not numeric"
        % arg)

def toBeTested(arg):
  debuggingTest(arg)
  print "in the place of more code"
```

The important difference between the examples is the debuggingTest function and `if` within it.

3. Types are associated with values.

   Therefore, variables may have different types of values at different locations in the program and the type may be dependent on the execution path.
4. Functions and methods may return different types of values depending on the execution path through the method or function.
5. Methods and functions should check for proper types and values of their arguments.

   This argument validation should be turned off by `if` statements or removed only when their performance impact becomes significant.
6. Python uses white space semantically. Indentation is used to define the boundaries of multiple line language constructs.

   Marking of the intended end of such a group of statements with a comment which names the statement it ends is strongly recommended as Python does not have language constructs for that purpose.

Always run `expand` or equivalent to convert all white space to blanks so that one is not surprised by different interpretations of tab characters by different editors, compilers and operating system.

By simply changing the identation of statements, code easily can be moved *in error* into or out of 'if', 'while', 'for', 'def' and 'class' statements.

Be very vigilant about indentation. During code reviews, verify the correct end of multiple line language constructs.

7. Python has several statements for the importation of entities from other packages.

   According to the O'Reilly Python Essential Reference by David Beazley, `from` *something* `import *` is required to be at the top of program at the outermost level of the code. In fact, experiments show that the statement is accepted anywhere with a syntax warning. None the less, the The only proper location for `from` *something* `import *` is at the top of the program.

   `import` can be placed anywhere in the code. The only proper location for `import` statement is at the top of the program with the `from` statements. Placing `import` statements elsewhere is surprising and can cause varying effects depending on the execution path through the program.
8. Python has a "handy feature" to import all of the elements of a package: `from xyz import *`.

   This "feature" should not be used. Python handles the importation of identically named elements of several packages for you automatically. Python retains for you the last definition it receives. The results may be surprising.

   Another variant of the statement: `from xyz import something`.

   This variant restricts the access to only those elements of the package named. This variant requires more work on the part of the programmer to select only those parts of a package that actually are wanted from the module. The advantage is that the programmer and the code reviewer knows what was imported. It is recommended that the list of imported items be updated when an element is no longer needed.

   Using `import xyz` and then `xyz.something()` requires extra coding in the body of the program.

   This variant has several advantages. It shows exactly where an element of the package is used. If during program maintenance the usage is removed, then the importation is automatically removed as well.
9. Python uses modifiable dictionaries (hash tables) to represent objects, classes, stack frames, modules and dictionaries. This feature means that a program readily can add or remove fields, functions and methods from any of these things, which allows for great flexibility in a program. The issue with this feature is that it causes great surprises.

   Do not do it *at all*!

   Also, do not use modifiable dictionaries (hash tables) as a replacement for using class instances.

   During code reviews, be vigilant for use of this "feature."
10. Python, does not have pre- and post-increment operators, such as, ++a, –a, a– and a++.

    The post-increment forms do not exist in Python and cause compile time errors.

    Unfortunately, the pre-increment forms are not compile time errors in Python. They silently have no effect.

    During code reviews, be vigilant for use of this "feature."

```
>>> a=42
```

```
>>> ++a

42

>>> --a

42
```

11. Python helps a programmer pay attention to one problem at a time by, generally, presenting one compilation time error message per compilation attempt.
12. As in most languages with a nesting block structure, Python is easier to read if a uniform indenting style is used. An alternate way of handling multiple levels of indentation is to move the code of those higher levels of indentation to their own separate functions placed elsewhere. This reduces the size of methods and the use of indentation to show program structure resulting in more readable program.
13. As in most languages, Python code usually becomes more readable with appropriate white space around operators and expressions.
14. As with most texts, code usually becomes more readable if the width of the text is not made too wide. Column widths between 65 columns and 95 columns are recommended. Guido van Rossum's Python style guide and the commmon default terminal width suggests that the line width limit should be less than 80 characters. This is the reason that newspapers and magazine have narrower columns than books; newspapers and magazines are designed and intended to be read fast. Studies have shown similar effects on computer displays and web pages.[8]
15. Python treats strings as sequences of characters.

```
>>> for c in 'abe': print c
...
a
b
e
```

16. Python permits long variable names. These can be used to good effect to improve readability at the cost of additional typing. Use distinct variable names so that one does not have to check the right hand end of name to determine which variable it is.

    With long variable names:

```
status_unknown = 0

status_good = 1

status_bad = 3

status_code_map = { status_unknown : "unknown",

                    status_good    : "good",

                    status_bad     : "bad" }

if thingie.getStatus() == status_good:
```

8http://hid.fidelity.com/q31998/column.htm
http://edlab.tc.columbia.edu/files/eye-tracking%20article.pdf

```
xyz.useThisThingie(thingie)
```

With short variable names:

```
u=0
g=1
b=3
sc = {0:"unknown",1:"good",3:"bad"}
if t.st == g:
   xyz.doit(t)
```

17. Python does not have `protected` or `private` access to fields and methods.

    The Python users community has a conventation that field, function and methods that start with an underscore (_) are limited access elements of the class or instance object. Please, use this convention.

    The use of getter and setter methods for all class fields is recommended strongly, particularly for fields marked as intended as private. They permit the programmer to change the internal design of a class without having to change the remainder of the program, to add constraints with verification or to prohibit the modification of a field. It also can assist in debugging by permitting the easy insertion of debugging to look for setting to rare values.

    The issue being addressed by this recommendation is the minimumization of ”coupling” between sections of a program.
18. Program class should be derived from some class. If not derived from any other class, then derive from `object`.

```
class Top(object):
```

19. All Python classes should have reasonable __init__, __str__ and __repr__ methods.
20. When logging a object value, use the %r format item and the value argument of the % operator always should be a tuple. Use a trailing comma (,) in the tuple if there is one value to be logged.

```
logString = "the value: %r" % ([1,2,3],)
```

21. If you need to change the behavior of a function or method, leave the old version alone (or remove it entirely) and add a new version with a *new* name.

5 Useful Coding Tricks

See also: PEP 8[9].

Even though Guido van Rossum states in his style guide for Python, PEP 8, that “a foolish consistency is the hobgoblin of little minds,” since most programmers generally do follow those guidelines, the principal of minimum surprise means that you, the programmer, ought to follow the PEP 8 guidelines as well. These guidelines actually are quite reasonable.

See also: PEP 257[10].

This PEP describes docstrings, which a string immediately following a module, class, function or method definition and is intended to document that entity.

Use consistent content in docstrings and other comments. Data to be included when pertinent are: purpose,

9http://www.python.org/dev/peps/pep-0008/
10http://www.python.org/dev/peps/pep-0257/

methodology, location of related documentation, expected argument types and their defaults and return types.

Use a logging package to report errors and programming events. During initial testing, the logging package can write to terminal. Later, the use of a logging package will ease the redirection of logs to files or network connections or the elimination of classes of log data. It is a design decision as whether to use a logging package to removing debugging output or to use `if` statements.

1. Comments and code must agree. Fix whichever is in error. A bad comment is worse than no comment at all. Explain why a less than obvious method was chosen (e.g., why the obvious method is wrong). If code is order-dependent in some less-than-obvious way, then order dependency some be documented in the code comments.
2. Use library routines and base type methods instead of writing code.
   (a) The code is more likely to be correct the first time.
   (b) The code is likely to be compiled and not interpreted.
   (c) The language documentation will explain what the code is doing.
3. Use `in` to check for the presence of a key in a map: `if 6 in mapping: print "have a 6"`.
4. Use `is` to check the type of a value: `if x is None`. "None" is a singleton object (there is only one of this value). `is` is a test on identity of an object (are these the same object) and not on equality (do these have the same value by some definition). Generally identity tests are faster than equality tests.
5. Use `xrange` when the size of a range is large as `range` builds a list and then iterates over it and `xrange` is an iterator which computes the next value as needed. This can be particularly efficient when search over a large range and the search normally ends very early.
6. Use `if __name__ == "__main__":` to protect code that should only be executed if the program is used as a main program.
7. Use `a, b, c = b, c, a` to rotate three values. Similar code can be used to rotate other order n-tuples.
8. Use `a,b,c,d = values` to unpack lists into named variables. In this example, `values = [2,3,5,7]` Hint: this can be useful when the program wants to return more than one value from a function or method. Make a list of the values, `return [1, 17]`, and return that single value.
9. Use one or more blank lines to separate logical distinct sections of code in a method or function.
10. Try to keep functions or methods small.
11. Keep your code simple and direct. Use appropriate data structures and use classes to keep data that belongs together together. Use complicated algorithms only when that portion of the program requires them for performance. Donald Knuth's The Art of Computer Programming[11] is an excellent source of well discussed algorithms. Another excellent source of algorithms is NIST's Dictionary of Algorithms and Data Structures[12].
12. Pay attention to resource leaks and ever-growing data structures. "Did you close that file when you were finished with it? " "Empty a data structure when you are done with it."
13. When a variable is no longer meaningful, use the `del` on it. This permits more rapid garbage collections and reduces the chances of incorrect usage of the old value.
    (a) loop variable

```
for x in [1,2,3]:
  pass
del x
```

    (b) a list

```
x = [1,2,3]
del x[:]
del x
```

    (c) a dictionary

11http://www-cs-staff.stanford.edu/ũno/taocp.html
12http://www.nist.gov/dads/

```
x = {1:'a',2:'b',3:'c'}
x.clear()
del x
```

(d) a class instance

```
x = ABigClass()
del x
```

14. Particularly for user input, when appropriate, use case independent comparisons for text strings. Since Python supports Unicode and some languages do not have all the glyphs of the language in upper case or title case, convert a copy of the input string to lower case for comparison. You want to keep the original input string intact for error messages.
15. Parentheses are cheap. Use parenthesis whenever and wherever the priority of the operators in an expression *might* be unclear to a reader of the code.
16. Use long expressions with white space and parentheses as needed rather than unnecessary temporary variables. This is not a license to combine as many expressions as possible.
17. Validate inputs from outside of the control of the program. If the program is divided into major sections, then treat those sections as separate programs and validate arguments passed between those major sections.
18. When feasible, use parentheses around a multi-line expression rather than using the Python back-slash (\\) line continuation.
19. Use camel-casing for variable, function, method, class, etc. names. Use a upper case first letter for the first letter of a name for a class name and a lower case first letter for everything else.

```
exampleThingFactory = ExampleThingFactory()
exampleThing = exampleThingFactory.makeThingie
```

20. Python sequences may indexed or sliced with negative values. Negative indices index from the high end of the sequence. Extra viligance is needed when non-constant possibly negative expressions are used. Negative values only should occur when intended and not as a result of a “one-off” or similar error.
21. Regardless of which programming language or languages are used to write a program, the implementation language or language should not be visible to the user in the syntax of the input language.
    Instead, use:
    (a) a simple input design,
    (b) a standard input language such as XML[13] or
    (c) a designed small language implemented using a parser and lexer pair or a compiler generator[14].
22. Although code management systems encourage one to keep one’s changes small, freely rewrite bad code rather than looking for a tiny version of the change.
23. If the code is not following the project’s style, then it needs to be rewritten to correct its style.
24. Design the input so that it is easy to generate correctly. If it is human generated input, then, even if it makes more work for the programmer, design the input for the user.
25. Use configuration files and manifest constants. Configuration files, network data sources or configuration classes should be read or processed as soon as possible in the program’s initialization. Other values related to values in configuration data and values of manifest constants must be computed from those constants. Put all non-operating-system file path names in the configuration data.
    Consider using a separate package for the project’s manifest constants. The package should also compute common derived values from the manifest constants during the packages initialization and make those values available as well.
26. If an error is discovered in a configuration file, the error should be reported.

13http://www.rexx.com/kuhlman/pyxmlfaq.html
14http://www.lava.net/ñewsham/pyggy/

27. Beware of common programming mistakes such as failure to appropriately initialize variables, loops that are one-off in their control, failure to test correctly for equality or “sameness”, failure in control when equality or “sameness” occurs, handle the absence of input cleanly and clearly, failure to handle boundary conditions correctly such as maximum negative numbers or stepping off the end of lists, etc.
28. If requirements writer, the designer, the programmer or the tester commonly makes certain errors, then then look for those errors in the following stages of the program’s development.
29. If a new technology is used, then pay close attention to the portions of the program which use them.
30. Error messages and warnings should be as specific as practical. Include the argument name and value unless security issues preclude providing these data. Suggest a solution to the problem. The log entry for the event should include all applicable data unless security issues preclude providing these data.
31. In exception handling, use as specific as possible exceptions. Use multiple `except` clause if necessary. Name subclass exceptions before superclass exceptions. If you need to `catch` and do nothing, then comment why that is appropriate behavior. In most case, that is a program event that should be logged. E.g., a `try` block around a file close to bypass a possible I/O exception should log that I/O exception.
32. Using `super()` in a instance initialization may be dangerous.[15]
33. A class instance should be usable as soon as it has been made by the class constructor.
    If this is not possible, then use a factory method to make a class instance and then customize it.

6 Disclaimer

This document assumes that the requestor, analyst and programmer have good and sufficient reasons for wanting to use the Python programming language and libraries. Many trademarks and logos are mentioned in this document. They belong to their respective owners.

15http://fuhm.net/super-harmful/