Extracted from:

Pythonic Programming

Tips for Becoming an Idiomatic Python Programmer

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pythonic Programming

Tips for Becoming an Idiomatic Python Programmer

Dmitry Zinoviev

Edited by Adaobi Obi Tulton

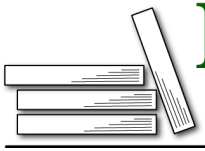
Pythonic Programming

Tips for Becoming an Idiomatic Python Programmer

Dmitry Zinoviev

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-861-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2021

This chapter contains general tips—the tips that are related to general Python programming. Following them does not make your programs faster or more correct, and not following them does not make your programs slower or faulty. These tips are about programming style and readability—that is, about being Pythonic.

There is a belief among programmers that if a program is not beautiful, it is also incorrect (for some unrelated reason). Pythonic programs are beautiful and, therefore, less likely to be incorrect. Follow these tips; make your programs beautiful.

Tip 14

Chain Comparison Operators

★[2.7, 3.4+] Remember how you could write in an algebra class that x < y < z? In Python, you can chain comparison operators, too: x < y < z. You can even write x < y <= z != w, why not? An expression like this is evaluated first by evaluating each comparison operator separately and then taking the and of the results:

```
x < y and y <= z and z != w
```

You could even evaluate bizarre expressions like x < y > z (check if y is greater than both x and z)—but they are confusing and better avoided.

Tip 15

Expand the Tabs

★[2.7, 3.4+] The character '\t' (tab space) is the only ASCII character whose visual representation is context sensitive. One can trace the symbol to the Ice Age of typewriters; the tabulator key would move the cartridge to the nearest tab stop. Advanced typewriters allowed users to define tab stops anywhere along the line, but the default ("cheap") configuration was to keep them eight spaces apart.

Modern terminals mostly follow the same rules. They display '\t' as a sequence of one through seven characters, depending on how far the next tab stop position is, which makes it hard to align the output, especially when the distance between the tab positions is not known. Use method str.expandtabs(tabsize=8) to simulate the printout before it is displayed. The method expands each tab in the string by adding enough whitespace characters to reach the nearest tab stop. You can control the distance between the stops via the tabsize parameter:

```
'column1\tcol2\tc3'.expandtabs(8)
```

```
⇛ 'column1 col2    c3'
```

You can use str.expandtabs() to calculate the width of a table or single string. Sadly, the method simulates only "cheap" typewriters; you cannot customize the individual stops.

Tip 16

Pickle It

★★[2.7, 3.4+] In a not so uncommon situation, you may want to save the intermediate results of your computations, either because another Python program will use them or because it took you a couple of hours to compute and you do not want to take any risks (see Tip 69, Checkpoint, It Saves Time, on page ?). There are many ways to accomplish this task. You can use CSV files for two-dimensional tabular data (either directly in the core Python or via the module csv). Unstructured data can be converted to JSON (module json) or XML (module xml). Networks and other graphs gain from being stored in GraphML, a specialized graph description format (module networkx). However, some (if not most) Python complex objects are not easily converted into a series of characters or bytes, or serialized. You will need another round of jumping through hoops to convert that series back into complex objects (deserialize).

Fortunately, Python supports pickling (and unpickling, something not quite possible in real life). Pickling (or dumping) is a Python-specific serialization mechanism that takes any Python object, no matter how complex, and saves it to a file. Pickle files usually have the extension .p, .pkl, or .pickle. Unpickling (loading) is about reading data from a pickle file and reconstructing the serialized objects. You can save several objects into the same file sequentially or create a list of objects that need saving, and pickle the whole list at once. Here is an example of pickling in action:

```
with open('results.p', 'wb') as pickleFile:
    pickle.dump(anyPythonObject, pickleFile)
```

Unpickling is equally simple:

```
with open('results.p', 'rb') as pickleFile:
    aPythonObject = pickle.load(pickleFile)
```

Note that the pickle file must be opened in the binary mode, both for dumping and loading.

Tip 17

Avoid range() in Loops

★★[2.7, 3.4+] The well-known built-in function range() that they very likely taught you to use in a for loop is often unnecessary. It is a tribute to C/C++/Java, the honorable predecessors of Python that actually need an index loop variable. Python does not. Here is why.

The "classic" non-destructive for loop over an iterable seq looks like this:

```
for i in range(len(seq)):
    do_something_with(seq[i])
```

The variable i is used only to access the next element from the iterable. Since Python for loop is, in fact, a foreach loop, there is a better way to accomplish this by iterating directly over the iterable (thus the name):

```
for item in seq:
    do_something_with(item)
```

The direct iteration approach is faster (by about 30%), does not require creating the unnecessary indexes, and is more readable ("*for [each] item in [that] sequence, do something with [that] item*"). But what if you really need to know the index—say, to modify each item in a mutable iterable? For that, you have the built-in function enumerate(). The function returns a generator (Tip 58, Yield, Do Not Return, on page ?) that produces tuples of items and their respective indexes in the form (i,seq[i]).

```
for i, item in enumerate(seq):
    seq[i] = do_something_with(item)
```

Naturally, you can use the index for any other purpose—for example, to process only odd-numbered items:

```
for i, item in enumerate(seq):
    if i % 2:
        do_something_with(item)
```

A C/C++/Java-trained programmer may argue that range() is required to manipulate parallel iterables. Parallel iterables contain different attributes of the i'th item at their i'th positions. Say, for example, lists x and y store the namesake coordinates of two-dimensional points. How would one calculate the distances from the points to the origin without knowing the indexes?

First and foremost, parallel lists do not belong in Python. As a matter of fact, they do not belong in any programming language younger than the original FORTRAN. You should use structs, classes, two-dimensional arrays, and other data structures that group attributes. You may be excused if the attributes originally come from different sources and require merging before further use. The built-in function zip() comes to rescue:

```
for item1, item2 in zip(seq1, seq2):
    do_something_with(item1, item2)
```

As a bonus, zip() can handle any number of parallel iterables. Tip 55, Pass Arguments Your Way, on page ? explains how to pass them to the processing functions at once:

```
for items in zip(seq1, seq2):
    do_something_with(*items)
```

To bring this example to perfection, let's assume that the parallel iterables themselves are organized in a tuple seqs. In the spirit of the tip mentioned above, pass the tuple to zip() as a whole and let Python unpack it:

```
for items in zip(*seqs):
    do_something_with(*items)
```

The solution is concise, idiomatic, and independent of the number of parallel iterables—as long as the function do_something_with() expects precisely that many parameters.

Do we even need range() at all? Yes, we do, when we need a range of evenly spaced integer numbers. For everything else, there are other tools.

Tip 18

Pass It

★[2.7, 3.4+] Some compound statements in Python (for example, class, for, while, if, and try) require that their body is not empty and contains at least one statement. Even if you do not want to provide that statement, you must. In C/C++/Java, one would use an empty statement—a block of two curly braces {}. Python does not use curly braces (at least not for this purpose). Instead, it uses indentation, and there is a problem with an empty statement defined through indentation—it is invisible, like this:

```
class EmptyClass:
      # Is there an empty statement here? Can you see it?
```

Python provides an equivalent of an empty statement called pass. You use pass when you must use a statement, but you do not care about it:

```
class EmptyClass:
    pass
```

You could use a string literal instead, something like "Do nothing here"—but that goes against *The Zen of Python*: "There should be one—and preferably only one—obvious way to do it."

Tip 19

Try It

★★[2.7, 3.4+] There are two programming approaches to coding an operation that may fail: an optimistic and a pessimistic one. The pessimistic approach explained in Tip 85, Check, Then Touch, on page ? implies that failures are frequent and costly to repair. It is cheaper to avoid them by checking some precondition.

The optimistic approach, on the contrary, implies that failures are rare. It is cheaper to try and then recover if anything goes wrong using the exception handling mechanism (try/except). One of the most illustrative examples of an optimistic scenario is checking whether a string represents a number.

Though not trivial, it is possible to describe a valid string representation of a floating-point number with an exponential part using a regular expression. However, it is much easier to pretend that a string represents such a number and attempt to convert it. If the conversion is successful, the string is a number. Otherwise, it is not (Tip 47, Discover an Infinity, on page ?):

```
def string_to_float(s):
    try:
        return float(s)
    except ValueError:
        return float('nan')
```

Another common application of the optimistic method is opening a file for reading. You can open a file for reading if it exists, is indeed a file (not a directory), it belongs to you, and is readable. You can replace this sequence

of checks with one painless attempt to open the file. If the file opens, it is openable. Otherwise, it is not:

```
try:
    with open('somefile') as infile:
        # Do something with infile
except (FileNotFoundError, IsADirectoryError, PermissionError):
    # Do something else
```

In case you anticipate some other file-related errors, add them to the tuple, but try to avoid the "blanket" exception handlers (Tip 90, Isolate Exceptions, on page ? explains why).

Whether to be an optimist and try, or be a pessimist and check, depends on the complexity of the check, the penalty of trying, and your programming philosophy. The former two can be estimated. The latter one is a matter of how you were taught. I cannot help you with that.