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

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

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

“Python comes with a great collection of data types and data structures”—you will often see this phrase and its variations throughout this chapter. Choosing the right representation for your data may be a matter of “life and death.” In the best case, a wrongly chosen data type may cause significant performance degradation. In the worst case, it may cause logical errors and lead to incorrect results.

This chapter provides tips about both “traditional” and “obscure” data structures and types, including standard containers, counters, and various numbers. You will get advice on how to work with complex and rational numbers and infinities, easily create modules, transform lists, count items, and appreciate the immutability of tuples. The chapter also includes suggestions about advanced class design (class attributes and customized object print-outs).

According to Niklaus Wirth, the inventor of Pascal and Modula and Turing Award winner, *Algorithms + Data Structures = Programs [Wir78]*. This chapter addresses all three aspects of the famous equation.

Tip 33

Construct a One-Element Tuple

★$^{2.7,\ 3.4+}$ Creating a one-element tuple is a pain. Let's first use our common sense and try to make a one-element tuple similar to one-element lists and one-element sets:

```
type([0])
```

```
⇒ <class 'list'>
```

```
type({0})
```

```
⇒ <class 'set'>
```

```
type((0))
```

```
⇒ <class 'int'>
```

Bummer. The result is not a tuple but an integer number—the first and the only element itself. That is because the parentheses in Python have several uses: they participate in creating a tuple (in cooperation with commas), define functions, define subclasses, invoke functions, and change the order of evaluation, to name a few. In the last example, the outer pair of the parentheses invokes the function, and the inner pair…changes the order of evaluation!

To tell Python that a tuple is born, add a comma after the first element. It is the comma that builds a tuple, not the parentheses.

```
type((0,))
```

```
⇒ <class 'tuple'>
```

```
a = 0, # Just a comma!
type(a)
```

```
⇒ <class 'tuple'>
```

Isn't it exciting? So, perhaps, even the inner parentheses are redundant? Can we eliminate them? Let's calculate the length of a one-element tuple.

```
len(0,)
```

```
⇒ Traceback (most recent call last):
⇒   File "<stdin>", line 1, in <module>
⇒ TypeError: object of type 'int' has no len()
```

The error is that the comma also has several uses. It creates tuples, but it also separates arguments in a function call and parameters in a function definition. In this example, Python thought about the second use and treated 0 as the first argument. It is getting curiouser and curiouser. Perhaps, just stay away from one-element tuples. You can replace a one-element container that is not expandable with a single scalar variable.

Tip 34

Improve Readability with Raw Strings

★$^{2.7,\ 3.4+}$ Raw strings are strings prefixed with the letters r or R outside of the quotation marks. Within a raw string, the escape character, backslash '\', does not have a special meaning. It is not an escape character anymore, it is merely a backslash. Respectively, all special compound characters, such as '\n' and '\v', lose their special meaning and become two-character strings:

```
print(r'\n' + '\n' + r'\n', len('\v'), len(r'\v'))
```

```
⇒ \n
⇒ \n 1 2
```

Consider a string that has many backslashes as such—for example, a regular expression. In a "cooked" (not raw) string, each backslash must be prefixed by another backslash, creating a forest of barely decipherable backslashes, very much like this paragraph itself:

```
regex = '\\n\\\.\\\\n'
print(regex)
```

```
⇒ \n\\.\\n
```

(For your reference, this regular expression matches a string that consists of a line break, followed by a literal backslash, followed by a period, followed by another literal backslash, and by one more line break.) Raw strings make this code more readable:

```
regex = r'\n\\.\\n'
print(regex)
```

```
⇒ \n\\.\\n
```

But what if you want to have a special escaped character in a raw string? That is not directly possible. Either revert to the "cooked" strings or combine a raw and a "cooked" string with string concatenation:

```
mixed_string = '\n' + r'\\.' + '\n'
print(mixed_string)
```

```
⇒
⇒ \\.
⇒
```

Last but not least, for a bizarre reason, a backslash at the end of a raw string still acts as an escape character. The string r'\' is not a single backslash, it is an unterminated string.

```
r'\'
```

```
⇒   File "<stdin>", line 1
⇒     r'\'
⇒        ^
⇒ SyntaxError: EOL while scanning string literal
```

Tip 35

Unpack Lists and Tuples

★★$^{2.7,\ 3.4+}$ You can extract individual items from a sequence (such as tuple, list, or string) using the indexing operator:

```
seq = 1, 2, 3, 4
x  = seq[0]
y  = seq[1]
z1 = seq[2]
z2 = seq[3]
```

Another way is to resort to multiple assignment (also known as a simultaneous assignment). Naturally, the number of items on the left must match the sequence size.

```
x, y, z1, z2 = seq
```

Multiple assignment works best if the number of items in the sequence is known and does not change, because you have to list the variables on the left-hand side of the assignment, and those variables must match the sequence element-wise.

But wait, there is a catch. You can use the operator “star” (“*”) to collect the remaining items from the sequence, even if you are not sure about the sequence size. It suffices to know that the sequence has at least several items—say, two. And there may be more of them, but maybe not. The following statement unpacks a sequence into the variables x (the first element), y (the second element), and z (the rest of the elements as a list). The list is empty if the sequence has only two items:

```
x, y, *z = seq
print(x, y, z, sep=' | '))
```

```
⇛ 1 | 2 | [3, 4]
```

The starred variable on the left-hand side does not have to be the last. It can be anywhere in the middle and even at the beginning of the statement. But you cannot use more than one star; otherwise, matching is not possible:

```
*x, y, z = seq
print(x, y, z, sep=' | ')
```

```
⇛ [1, 2] | 3 | 4
```

```
*x, y, z = seq[:2] # Take the first two elements
print(x, y, z, sep=' | ')
```

```
⇛ [] | 1 | 2
```

And a little string example:

```
start, *rest, end = 'Hello, world'
print(start, ''.join(rest), end, sep=' | ')
```

```
⇛ H | ello, worl | d
```

Just what one would expect.

Tip 36

Print a List

★★$^{2.7, 3.4+}$ If you tried to print a Python list in a human-readable way—without all those square brackets, commas, and quotation marks—you know that print(l) is not an ideal solution:

```
l = list('hello') + list(range(5))
print(l)
```

```
⇛ ['h', 'e', 'l', 'l', 'o', 0, 1, 2, 3, 4]
```

What you need is a way to convert each list item to a string with str() and to combine the strings with a delimiter of your choice (say, whitespace) and the str.join() method. A list comprehension is an ideal tool for the job:

```
print(' '.join(str(x) for x in l))
```

```
⇒ h e l l o 0 1 2 3 4
```

The missing square brackets around what looks like a list comprehension are not a mistake. Instead of list comprehension, I used a comprehension expression to give you another exposure to this underappreciated mechanism (Tip 20, Embrace Comprehensions, on page ?). If the lack of the brackets scares you, put them back:

```
print(' '.join([str(x) for x in l]))
```

```
⇒ h e l l o 0 1 2 3 4
```

If the list is recursive (contains other compound items, such as lists, tuples, and sets, or any combinations of them), you may combine printing with flattening. Tip 37, Flatten That List, on page ? explains how.