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

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

There are three types of programming errors: syntax errors make your program grammatically invalid, runtime errors make your program crash, logical errors make your program compute not what you want but something else.

Logical errors are the worst. They do not offend the Python interpreter or crash your programs but subtly lead to incorrect results. These results are often hard to distinguish from the expected results. The hints in this chapter explain how to avoid logical errors.

You will be reminded to call your functions. You will learn the difference between optimistic programming and pessimistic programming (and finally find out if the glass is half empty or half full). You will hopefully be scared by the function `eval()` and never use it again, even if you used it before.

This chapter is the longest in the book. Not because I am so fond of errors but because there are usually few ways to solve a problem correctly and infinitely many ways to solve it incorrectly. I tried to cover your bases to the fullest extent.

Tip 75

Call That Function

★$^{2.7, 3.4+}$ A function identifier is a reference to that function, a claim that the function has been defined and exists somewhere in the interpreter's memory. When you mention an existing function's identifier on the command line, the interpreter confirms that the function indeed exists. It also tells you the address of the function in the RAM, but you probably do not care about it:

```
def add1(x):
    return x + 1
add1
```

```
⇛ <function add1 at 0x7f9c11c28048>
```

The interpreter does not call a function unless you tell it to call the function using the function call operator, also known as the parentheses:

```
add1(10)
```

```
⇛ 11
```

Not calling a function is not a syntax or runtime error. There is nothing wrong with merely referring to the function. But in most cases, that is not what you want to do. This is the right code:

```
result = add1(10)
```

This code is possibly correct; it provides an alternative identifier to an existing function:

```
increment = add1
result = increment(10)
```

But this code is most probably wrong:

```
result = add1
```

Tip 76

Get the Hang of Local Variables

★★$^{2.7,\ 3.4+}$ Many things are confusing in Python, but few are as confusing as local and global variables. The scope of a local variable is restricted to the enclosing function. The scope of a global variable is not restricted. But what makes a variable local or global?

A variable is local in a function if it is declared in the function unless explicitly marked as `global` in the same function. And the only way to declare a variable in Python is to use it on the left-hand side (LHS) of an assignment statement—that is why Python variables are always initialized. Whether a variable in a function is global or not does not depend on whether a global variable with the same identifier already exists. If it does, the function may have a local variable with the same name that will shadow the global variable—make it invisible in the function.

Let `gv=1` be a global variable. The variable remains global in the following two functions. `f1()` treats `gv` as a global because it is marked as global, despite the assignment statement. `f2()` treats `gv` as a global because there is no assignment statement.

```
def f1():
    global gv
    gv += 1
    return gv
f1(), gv
```

```
⇛ (2, 2)
```

```
def f2():
    return gv + 1
f2(), gv
```

```
⇛ (3, 2)
```

The variable `gv` in the third example is local because of the augmented assignment statement. It attempts to shadow the namesake global variable. The attempt is unsuccessful because an augmented assignment does not create a new variable:

```
def f3():
    gv += 1
    return gv
```

```
⇒ Traceback (most recent call last):
⇒   File "<stdin>", line 1, in <module>
⇒   File "<stdin>", line 2, in f3
⇒ UnboundLocalError: local variable 'gv' referenced before assignment
```

The fourth function successfully creates the local variable, which has the same name identifier as the global variable, but a different value:

```
def f4():
    gv = -1
    return gv
f4(), gv
```

```
⇒ (-1, 2)
```

In my opinion, the last example is the most dangerous. You may think that you modify the global variable while you change the local look-alike. Want to play safe? Avoid global variables!

Tip 77

Grasp What Is Truth

★★★$^{2.7,\ 3.4+}$ The Python concept of truth and falseness goes far beyond the boolean constants True and False. Typically, a "naturally empty" object is interpreted as false: None, 0, 0j, 0.0, an empty list, an empty tuple, an empty dictionary, an empty set—in other words, something that either is nothing, is numerically zero, or has the length of zero.

You can decide how to interpret an arbitrary object's boolean value by redefining its method __len__():

```
class GlassIsHalfEmpty:
    def __init__(self, mood):
        self.mood = mood
    def __len__(self):
        return 1 if self.mood == 'pessimist' else 0

bool(GlassIsHalfEmpty('pessimist'))
```

```
⇒ True
```

```
bool(GlassIsHalfEmpty('optimist'))
```

```
⇒ False
```

The plethora of boolean values makes it unsafe to rely on the equality to True and False. Consider the pattern-matching function re.search() that returns a match object if it finds a match and None otherwise. The following code fragment fails to detect matches because a match object is not equal to True:

```
if re.search(pattern, string) == True:
    print('Found a match!') # Not going to happen
else:
    print('No match')
```

Luckily, the conditional operator if recognizes all shades of truth. It interprets the match object as true even though its value is not equal to True:

```
if re.search(pattern, string): # Anything 'non-empty'
    print('Found a match!')
else:
    print('No match')
```

Avoid checking for truth directly; let the control statements (conditionals and loops) do their interpretation.

Tip 78

Check for Range

★★2.7, 3.4+ The built-in function range() returns a namesake object. A range object is an iterable. You can use it as the sequence in a for loop (but please do not; see Tip 17, Avoid range() in Loops, on page ?). A range object is not an iterator; you cannot get its next element by calling next(), and you can iterate over a range many times without consuming it.

A range object is not a range in the algebraic meaning of the word; range(x,y) is not the same as [x,y). The closest other data structure that describes the inner world of a range is a set of numbers. Similar to a set, a range is discrete (but ordered).

An immediate consequence of this observation is that the operator in checks if its left operand is *one of the discrete numbers* in the range. It does not check if the left operand is numerically greater than or equal to the start of the range and smaller than the end of the range:

```
5 in range(10)
```

```
⇒ True
```

(Because 5 is one of the numbers 0, 1, 2, ..., 9.)

```
5.5 in range(10)
```

```
⇒ False
```

(Because 5.5 is not one of those numbers.) The right way to check if x is in an algebraic range [a,b) is to use the comparison operators (see Tip 14, Chain Comparison Operators, on page ?):

```
0 <= 5.5 < 10
```

```
⇒ True
```

Tip 79

Strip User Input

★3.4+ Everybody knows how to read user input from the console. Wait. Everybody thinks they know how to read user input from the console (Tip 9, Let input() Speak for Itself, on page ?). But there is a twist: when asked to enter something, a user may, intentionally or not, add extra spaces before or after the requested information. Honestly, an evil user may insert additional spaces in the middle, too, but handling that kind of user is beyond the scope of this book.

If you use the function input() to request a number, you will further call int() or float(), both of which are trained to discard the heading and trailing spaces. Not so with general strings. It is your responsibility to make sure that the spaces do not make their way into the system. The most common disastrous scenarios are when the user enters the new username or, especially, the password (because the password input is typically invisible and hard to control). Play it safe, strip user input with str.strip()!

```
username = input('Enter your username: ').strip()
password = input('Enter your password: ').strip()
```