# Simple Tips to Level Up Your Python Programming Skills

Monday, November 14th 2022

Python is on its way to becoming the world's most popular programming language (https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language). If you're just starting to learn Python, here are some simple tips that can help you use the language more effectively.

> This blog post contains runnable code snippets powered by Pyodide (https://pyodide.org/), a version of Python compiled to WASM for use in the browser.

## Table of Contents

# Use List Comprehensions

This is a pretty basic feature, but a lot of beginners coming from other languages aren't familiar with it. List comprehensions let you express a lot of complicated operations in a very concise way. As an example, let's look at a simple task - create a list of the square of all even numbers from 1 to 10 (inclusive).

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = []
for n in numbers:
  if n % 2 == 0:
    result.append(n * n)
print(result)
```

▶ RUN

Pretty simple, but we can make this even better using a list comprehension. Our task consists of three parts:

1. Iteration ( `for n in numbers` )
2. Filtering ( `if n % 2 == 0` )
3. Transformation ( `n * n` )

A list comprehension lets us do all three of these in a single expression. [1].

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print([n * n for n in numbers if n % 2 == 0])
```

▶ RUN

# Use f-Strings for Constructing Messages

Python has several options for formatting data into strings. Here are just a few.

```
from string import Template

name = "Batman"
```

```
4
5 print("I'm " + name + "!") # string concatenation
6 print("I'm %s!" % name) # %-strings
7 print("I'm {}!".format(name)) # str.format
8 print(Template("I'm $name!").substitute(name="Batman")) # Template strings
```

▶ RUN

All of these mechanisms have weaknesses. String concatenation is quite verbose, and the other methods separate the value from the place they will end up in the final string (this isn't great when you have lots of variables to splice in). Python 3.6 introduces a new feature called f-strings that solves all of these problems [2].

```
1 name = "Batman"
2 print(f"I'm {name}!")
```

▶ RUN

f-Strings let you put any Python expression in between the curly braces. In my opinion, this is the most effective pattern for building strings, since the variables appear in the string exactly where their value will appear in the constructed result. You can even add format specifiers, which makes it easy to do things like round floating points down to a fixed-digit presentation.

```
1 value = 5/9
2 print(f"Result: {value}, Rounded: {value:.2f}")
```

▶ RUN

# Use `enumerate` to Loop Over Values and Indices

Imagine I need to print out the elements of a list, while also printing out the positions of those elements. I can do this pretty easily with `range` - I simply iterate over indices, and index into the array to get the values.

```
1 fruits = ["Apples", "Oranges", "Bananas"]
2
3 for i in range(len(fruits)):
```

```
4   print(f"#{i + 1} - {fruits[i]}")
```

▶ RUN

However, there's an even cleaner way to do this - `enumerate`. `enumerate` loops over each element in an array while also providing the index to that element.

```
1 fruits = ["Apples", "Oranges", "Bananas"]
2
3 for i, fruit in enumerate(fruits):
4   print(f"#{i + 1} - {fruit}")
```

▶ RUN

# Auto-format Your Code with Black

On many projects, a significant amount of comments on pull requests tend to be style and formatting suggestions. This is a waste of programmer time and energy - instead, we can use a tool to automatically format our code in a consistent, readable way. For Python, we have a few options, each of which has a wealth of configuration options. This leads to more questions. Which formatter do we use? Which configuration options do we set?

The answer: just use Black (https://github.com/ambv/black) and forget about all of these things.

Black is an *opinionated* code formatter, meaning that it has no configuration options. Just point it at your code and run. Black will end all debates about formatting on your team. It standardizes quotes, indentation, wrapping, and much more. It will even improve your diffs by using trailing commas where appropriate [3].

## Before Formatting

```
def f( arg1,
  arg2  ):
 pass
a = ["test1", 'test2',
  'test3']; print("test")
```

## After Formatting

```
def f(arg1, arg2):
    pass


a = ["test1", "test2", "test3"]
print("test")
```

# Flatten Nested Loops with `itertools.product`

Nested loops are a pretty common pattern in Python.

```
1 for i in range(3):
2   for j in range(6):
3     for k in range(4):
4       print(i, j, k)
```

▶ RUN

Unfortunately, they add a lot of extra indentation to your code and make it harder to read. Instead, you can use `itertools.product` to flatten these loops.

```
1 from itertools import product
2
3 for i, j, k in product(range(3), range(6), range(4)):
4   print(i, j, k)
```

▶ RUN

`product` takes in any number of iterables and returns the cartesian product of those iterables. Now, you can simply loop over the cartesian product. There are a lot more useful functions like this in the `itertools` module.

# Use Named Tuples / Data Classes

Sometimes you have collections of data that you want to keep together. For example, you might have some user data that includes things like name, age, etc. Typically you would put these into a tuple. However, you then have to remember what data is in each index, and accessing the wrong index might cause an error. Instead, you might want to use a class. But classes feel like overkill for this situation, and your constructor would just be a bunch of boilerplate anyway.

It turns out there's something in between - the named tuple. These act as lightweight container classes with only fields and no methods.

```
from collections import namedtuple

Person = namedtuple('Person', ['firstname', 'lastname', 'age'])

people = [
  Person('John', 'Doe', 22),
  Person(firstname='Jane', lastname='Doe', age=34)
]

for p in people:
  print(f"{p.firstname} {p.lastname}, Age: {p.age}")
```

▶ RUN

These are great for loading data from a database or CSV.

Another option is to use data classes. This option uses a decorator to automatically create a constructor for your class, along with other useful functions. Data classes also work better with type checking (explained later in this post).

```
from dataclasses import dataclass

@dataclass
class Person:
  firstname: str
  lastname: str
```

```
  age: int

people = [
  Person('John', 'Doe', 22),
  Person(firstname='Jane', lastname='Doe', age=34)
]

for p in people:
  print(f"{p.firstname} {p.lastname}, Age: {p.age}")
```

▶ RUN

# Default Dict

A common pattern in Python is to check if something is in a dict and initialize it if it isn't there already. Here's a basic example where I'm grouping users by the letter of their first name.

```
users = ['John Doe', 'Jane Doe', 'Richard Doe']
users_by_first_letter = {}

for user in users:
  first_letter = user[0]
  if first_letter not in users_by_first_letter:
    users_by_first_letter[first_letter] = []
  users_by_first_letter[first_letter].append(user)

print(users_by_first_letter['J'])
print(users_by_first_letter['B'])
```

▶ RUN

I have to check if a list already exists at `users_by_first_letter[first_letter]` before adding the current user to the list, otherwise, I will get a `KeyError`.

There's an easier way to implement this using the special collection `defaultdict`. `defaultdict` allows us to specify a factory function that constructs a default value for a key when it is first requested. This lets us drop the `if` statement entirely.

```
from collections import defaultdict
users = ['John Doe', 'Jane Doe', 'Richard Doe']
users_by_first_letter = defaultdict(list)

for user in users:
  first_letter = user[0]
  users_by_first_letter[first_letter].append(user)

print(users_by_first_letter['J'])
print(users_by_first_letter['B'])
```

▶ RUN

# Use Type Annotations / Mypy

Python is a dynamically typed language, which means any variable can be assigned to a value of any type, and variables don't need to be defined before they are used. This offers a lot of flexibility, but at the same time can be the source of difficult-to-find bugs that only pop up in certain situations. The function below has a bug due to a typo, but this bug only shows up in rare circumstances. We might not catch this "bug" until after we deploy our code!

```
import random

regular_pokemon = "Regular Caterpie"
shiny_pokemon = "Shiny Caterpie"

def get_pokemon():
  if random.random() < (1/8192):
    return shiny_pokeomn
  else:
    return regular_pokemon

for i in range(10000):
  print(get_pokemon())
```

▶ RUN

Let's go ahead and add some type annotations. By default, the Python interpreter won't do anything with these, and will still run our code.

```python
import random

regular_pokemon: str = "Regular Caterpie"
shiny_pokemon: str = "Shiny Caterpie"

def get_pokemon() -> str:
  if random.random() < (1/8192):
    return shiny_pokeomn
  else:
    return regular_pokemon

for i in range(10000):
  print(get_pokemon())
```

▶ RUN

However, we can run this code through Mypy (a Python type-checker) (https://github.com/python/mypy) and it will catch our typo, thus letting us fix this error before we send our code to production.

```
main.py:8: error: Name "shiny_pokeomn" is not defined  [name-defined]
Found 1 error in 1 file (checked 1 source file)
```

Mypy is great, but you don't need to rush out and convert all your code right away. You can adopt it incrementally, and even adopt it on a per-function basis.

---

1. Those coming from the functional programming world will recognize this as a `filter` followed by a `map`. Python has these functions too, but they are less idiomatic. ↩
2. In other languages, this feature is called string interpolation. ↩
3. For more details on why you should use trailing commas, check out this post (https://medium.com/@nikgraf/why-you-should-enforce-dangling-commas-for-multiline-statements-d034c98e36f8). ↩

**#programming-languages (/tags/programming-languages/)** **#python (/tags/python/)**

## Similar Posts

ChatGPT is Good at Roleplaying Characters (/posts/chatgpt-role-playing/)

Lua Gotchas (/posts/lua-gotchas/)

An Introduction to Parser Combinators (/posts/intro-parser-combinators/)