8 Advanced Python Tricks Used by Seasoned Programmers

Apply these tricks in your Python code to make it more concise and performant

Erik van Baaren Jun 25 · 5 min read ★

Illustration by author

Here are eight neat Python tricks some I'm sure you haven't seen before. Apply these tricks in your Python code to make it more concise and performant!

> *Visit python3.guide to learn all about Python!*

```
people = [
{ 'name': 'John', "age": 64 },
{ 'name': 'Janet', "age": 34 },
{ 'name': 'Ed', "age": 24 },
{ 'name': 'Sara', "age": 64 },
{ 'name': 'John', "age": 32 },
{ 'name': 'Jane', "age": 34 },
{ 'name': 'John', "age": 99 },
]
```

But we don't just want to sort it by name or age, we want to sort it by both fields. In SQL, this would be a query like:

```
SELECT * FROM people ORDER by name, age
```

There's actually a very simple solution to this problem, thanks to Python's guarantee that sort functions offer a stable sort order. This means items that compare equal retain their original order.

To achieve sorting by name and age, we can do this:

```
import operator
people.sort(key=operator.itemgetter('age'))
people.sort(key=operator.itemgetter('name'))
```

Notice how I reversed the order. We first sort by age, and then by name. With `operator.itemgetter()` we get the age and name fields from each dictionary inside the list in a concise way.

This gives us the result we were looking for:

```
[
 {'name': 'Ed',   'age': 24},
 {'name': 'Jane', 'age': 34},
 {'name': 'Janet','age': 34},
```

```
  { name : "Sara", age : 64}
]
```

The names are sorted primarily, the ages are sorted if the name is the same. So all the Johns are grouped together, sorted by age.

Inspired by this StackOverflow question.

2. List Comprehensions

A list comprehension can replace ugly for loops used to fill a list. The basic syntax for a list comprehension is:

```
[ expression for item in list if conditional ]
```

A very basic example to fill a list with a sequence of numbers:

```
mylist = [i for i in range(10)]
print(mylist)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

list_comprehensions_1.py hosted with ❤ by GitHub view raw

And because you can use an expression, you can also do some math:

```
squares = [x**2 for x in range(10)]
print(squares)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

list_comprehensions_2.py hosted with ❤ by GitHub view raw

Or even call an external function:

```
def some_function(a):
    return (a + 5) / 2

```

And finally, you can use the 'if' to filter the list. In this case, we only keep the values that are dividable by 2:

```
filtered = [i for i in range(20) if i%2==0]
print(filtered)
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```


3. Check memory usage of your objects

With sys.getsizeof() you can check the memory usage of an object:

```
import sys

mylist = range(0, 10000)
print(sys.getsizeof(mylist))
# 48
```


Woah… wait… why is this huge list only 48 bytes?

It's because the range function returns a ***class*** that only behaves like a list. A range is a lot more memory efficient than using an actual list of numbers.

You can see for yourself by using a list comprehension to create an actual list of numbers from the same range:

```
import sys

myreallist = [x for x in range(0, 10000)]
print(sys.getsizeof(myreallist))
# 87632
```

4. Data classes

Since version 3.7, Python offers data classes. There are several advantages over regular classes or other alternatives like returning multiple values or dictionaries:

- a data class requires a minimal amount of code
- you can compare data classes because `__eq__` is implemented for you
- you can easily print a data class for debugging because `__repr__` is implemented as well
- data classes require type hints, reduced the chances of bugs

Here's an example of a data class at work:

```python
from dataclasses import dataclass

@dataclass
class Card:
    rank: str
    suit: str

card = Card("Q", "hearts")

print(card == card)
# True

print(card.rank)
# 'Q'

print(card)
Card(rank='Q', suit='hearts')
```

dataclass.py hosted with ❤ by **GitHub** **view raw**

An in-depth guide can be found here.

5. The `attrs` Package

Instead of data classes, you can use `attrs`. There are two reasons to choose `attrs`:

- You are using a Python version older than 3.7
- You want more features

The `attrs` package supports all mainstream Python versions, including CPython 2.7 and PyPy. Some of the extras `attrs` offers over regular data classes are validators, and converters. Let's look at some example code:

```
@attrs
class Person(object):
    name = attrib(default='John')
    surname = attrib(default='Doe')
    age = attrib(init=False)

p = Person()
print(p)
p = Person('Bill', 'Gates')
p.age = 60
print(p)

# Output:
#   Person(name='John', surname='Doe', age=NOTHING)
#   Person(name='Bill', surname='Gates', age=60)
```

attrs_test.py hosted with ❤ by GitHub view raw

The authors of `attrs` have, in fact, worked on the PEP that introduced data classes. Data classes are intentionally kept simpler (easier to understand), while `attrs` offers the full range of features you might want!

For more examples, check out the attrs examples page.

6. Merging dictionaries (Python 3.5+)

Since Python 3.5, it's easier to merge dictionaries:

```
print (merged)
# {'a': 1, 'b': 3, 'c': 4}
```

merging_dicts.py hosted with ❤ by **GitHub** **view raw**

If there are overlapping keys, the keys from the first dictionary will be overwritten.

In Python 3.9, merging dictionaries becomes even cleaner. The above merge in Python 3.9 can be rewritten as:

```
merged = dict1 | dict2
```

7. Find the Most Frequently Occurring Value

To find the most frequently occurring value in a list or string:

```
test = [1, 2, 3, 4, 2, 2, 3, 1, 4, 4, 4]
print(max(set(test), key = test.count))
# 4
```

most_frequent.py hosted with ❤ by **GitHub** **view raw**

Do you understand why this works? Try to figure it out for yourself before reading on.

You didn't try, did you? I'll tell you anyway:

- `max()` will return the highest value in a list. The `key` argument takes a single argument function to customize the sort order, in this case, it's test.count. *The function is applied to each item on the iterable.*
- `test.count` is a built-in function of list. It takes an argument and will count the number of occurrences for that argument. So `test.count(1)` will return 2 and `test.count(4)` returns 4.
- `set(test)` returns all the unique values from test, so {1, 2, 3, 4}

maximum value.

And no — I didn't invent this one-liner.

Update: a number of commenters rightfully pointed out that there's a much more efficient way to do this:

```
from collections import Counter
Counter(test).most_common(1)
# [4: 4]
```

8. Return Multiple Values

Functions in Python can return more than one variable without the need for a dictionary, a list, or a class. It works like this:

```
def get_user(id):
    # fetch user from database
    # ....
    return name, birthdate

name, birthdate = get_user(4)
```

return_multiple_variables.py hosted with ❤ by **GitHub** **view raw**

This is alright for a limited number of return values. But anything past 3 values should be put into a (data) class.

That's it! Did you learn anything new? Or do you have another trick up your sleeve that you want to share? Please let me know in the comments!

A handpicked list of the most useful and surprising Python packages from PyPI

medium.com