How to Stand Out in a Python Coding Interview

by James Timmins Mar 27, 2019 10 Comments best-practices

Table of Contents

You've made it past the phone call with the recruiter, and now it's time to show that you know how to solve problems with actual code. Whether it's a HackerRank exercise, a take-home assignment, or an onsite whiteboard interview, this is your moment to prove your coding interview skills.

But interviews aren't just about solving problems: they're also about showing that you can write clean production code. This means that you have a deep knowledge of Python's built-in functionality and libraries. This knowledge shows companies that you can move quickly and won't duplicate functionality that comes with the language just because you don't know it exists.

At *Real Python*, we've put our heads together and discussed what tools we're always impressed to see in coding interviews. This article will walk you through the best of that functionality, starting with Python built-ins, then Python's native support for data structures, and finally Python's powerful (and often underappreciated) standard library.

In this article, you'll learn how to:

- Use `enumerate()` to iterate over both indices and values
- Debug problematic code with `breakpoint()`
- Format strings effectively with f-strings
- Sort lists with custom arguments
- Use generators instead of list comprehensions to conserve memory
- Define default values when looking up dictionary keys
- Count hashable objects with the `collections.Counter` class
- Use the standard library to get lists of permutations and combinations

Select the Right Built-In Function for the Job

Python has a large standard library but only a small library of built-in functions, which are always available and don't need to be imported. It's worth going through each one, but until you get the chance to do so, here are a few built-in functions worth understanding how to use, and in the case of some of them, what alternatives to use instead.

Iterate With `enumerate()` Instead of `range()`

This scenario might come up more than any other in coding interviews: you have a list of elements, and you need to iterate over the list with access to both the indices and the values.

There's a classic coding interview question named FizzBuzz that can be solved by iterating over both indices and values. In FizzBuzz, you are given a list of integers. Your task is to do the following:

1. Replace all integers that are evenly divisible by `3` with `"fizz"`
2. Replace all integers divisible by `5` with `"buzz"`
3. Replace all integers divisible by both `3` and `5` with `"fizzbuzz"`

Often, developers will solve this problem with `range()`:

```
>>> numbers = [45, 22, 14, 65, 97, 72]
>>> for i in range(len(numbers)):
...     if numbers[i] % 3 == 0 and numbers[i] % 5 == 0:
...         numbers[i] = 'fizzbuzz'
...     elif numbers[i] % 3 == 0:
...         numbers[i] = 'fizz'
...     elif numbers[i] % 5 == 0:
...         numbers[i] = 'buzz'
...
>>> numbers
['fizzbuzz', 22, 14, 'buzz', 97, 'fizz']
```

Range allows you to access the elements of `numbers` by index and is a useful tool for some situations. But in this case, where you want to get each element's index and value at the same time, a more elegant solution uses `enumerate()`:

```
>>> numbers = [45, 22, 14, 65, 97, 72]
>>> for i, num in enumerate(numbers):
...     if num % 3 == 0 and num % 5 == 0:
...         numbers[i] = 'fizzbuzz'
...     elif num % 3 == 0:
...         numbers[i] = 'fizz'
...     elif num % 5 == 0:
...         numbers[i] = 'buzz'
...
>>> numbers
['fizzbuzz', 22, 14, 'buzz', 97, 'fizz']
```

For each element, `enumerate()` returns a counter and the element value. The counter defaults to `0`, which conveniently is also the element's index. Don't want to start your count at `0`? Just use the optional `start` parameter to set an offset:

```
>>> numbers = [45, 22, 14, 65, 97, 72]
>>> for i, num in enumerate(numbers, start=52):
...     print(i, num)
...
52 45
53 22
54 14
55 65
56 97
57 72
```

By using the `start` parameter, we access all of the same elements, starting with the first index, but now our count starts from the specified integer value.

Use List Comprehensions Instead of `map()` and `filter()`

> "I think dropping filter() and map() is pretty uncontroversial[.]"
>
> — *Guido van Rossum, Python's creator*

He may have been wrong about it being uncontroversial, but Guido had good reasons for wanting to remove `map()` and `filter()` from Python. One reason is that Python supports list comprehensions, which are often easier to read and support the same functionality as `map()` and `filter()`.

Let's first take a look at how we'd structure a call to `map()` and the equivalent list comprehension:

```
>>> numbers = [4, 2, 1, 6, 9, 7]
>>> def square(x):
...     return x*x
...
>>> list(map(square, numbers))
[16, 4, 1, 36, 81, 49]

>>> [square(x) for x in numbers]
[16, 4, 1, 36, 81, 49]
```

Both approaches, using `map()` and the list comprehension, return the same values, but the list comprehension is easier to read and understand.

Now we can do the same thing for the `filter()` and its equivalent list comprehension:

```
>>> def is_odd(x):
...     return bool(x % 2)
...
>>> list(filter(is_odd, numbers))
[1, 9, 7]

>>> [x for x in numbers if is_odd(x)]
[1, 9, 7]
```

Like we saw with `map()`, the `filter()` and list comprehension approaches return the same value, but the list comprehension is easier to follow.

Developers who come from other languages may disagree that list comprehensions are easier to read than `map()` and `filter()`, but in my experience beginners are able to pick up list comprehensions much more intuitively.

Either way, you'll rarely go wrong using list comprehensions in a coding interview, as it will communicate that you know what's most common in Python.

Debug With `breakpoint()` Instead of `print()`

You may have debugged a small problem by adding `print()` to your code and seeing what was printed out. This approach works well at first but quickly becomes cumbersome. Plus, in a coding interview setting, you hardly want `print()` calls peppered throughout your code.

Instead, you should be using a debugger. For non-trivial bugs, it's almost always faster than using `print()`, and given that debugging is a big part of writing software, it shows that you know how to use tools that will let you develop quickly on the job.

If you're using Python 3.7, you don't need to import anything and can just call `breakpoint()` at the location in your code where you'd like to drop into the debugger:

```
# Some complicated code with bugs

breakpoint()
```

Calling `breakpoint()` will put you into `pdb`, which is the default Python debugger. On Python 3.6 and older, you can do the same by importing `pdb` explicitly:

```
import pdb; pdb.set_trace()
```

Like `breakpoint()`, `pdb.set_trace()` will put you into the `pdb` debugger. It's just not quite as clean and is a tad more to remember.

There are other debuggers available that you may want to try, but `pdb` is part of the standard library, so it's always available. Whatever debugger you prefer, it's worth trying them out to get used to the workflow before you're in a coding interview setting.

Format strings With f-Strings

Python has a lot of different ways to handle string formatting, and it can be tricky to know what to use. In fact, we tackle formatting in depth in two separate articles: one about string formatting in general and one specifically focused on f-strings. In a coding interview, where you're (hopefully) using Python 3.6+, the suggested formatting approach is Python's f-strings.

f-strings support use of the string formatting mini-language, as well as powerful string interpolation. These features allow you to add variables or even valid Python expressions and have them evaluated at runtime before being added to the string:

```
>>> def get_name_and_decades(name, age):
...     return f"My name is {name} and I'm {age / 10:.5f} decades old."
...
>>> get_name_and_decades("Maria", 31)
My name is Maria and I'm 3.10000 decades old.
```

The f-string allows you to put `Maria` into the string and add her age with the desired formatting in one succinct operation.

The one risk to be aware of is that if you're outputting user-generated values, then that can introduce security risks, in which case Template Strings may be a safer option.

Sort Complex Lists With `sorted()`

Plenty of coding interview questions require some kind of sorting, and there are multiple valid ways you can sort items. Unless the interviewer wants you to implement your own sorting algorithm, it's usually best to use `sorted()`.

You've probably seen the most simple uses of sorting, such as sorting lists of numbers or strings in ascending or descending order:

```
>>> sorted([6,5,3,7,2,4,1])
[1, 2, 3, 4, 5, 6, 7]

>>> sorted(['cat', 'dog', 'cheetah', 'rhino', 'bear'], reverse=True)
['rhino', 'dog', 'cheetah', 'cat', 'bear]
```

By default, `sorted()` has sorted the input in ascending order, and the `reverse` keyword argument causes it to sort in descending order.

It's worth knowing about the optional keyword argument `key` that lets you specify a function that will be called on every element prior to sorting. Adding a function allows custom sorting rules, which are especially helpful if you want to sort more complex data types:

```
>>> animals = [
...     {'type': 'penguin', 'name': 'Stephanie', 'age': 8},
...     {'type': 'elephant', 'name': 'Devon', 'age': 3},
...     {'type': 'puma', 'name': 'Moe', 'age': 5},
... ]
>>> sorted(animals, key=lambda animal: animal['age'])
[
    {'type': 'elephant', 'name': 'Devon', 'age': 3},
    {'type': 'puma', 'name': 'Moe', 'age': 5},
    {'type': 'penguin', 'name': 'Stephanie, 'age': 8},
]
```

By passing in a lambda function that returns each element's age, you can easily sort a list of dictionaries by a single value of each of those dictionaries. In this case, the dictionary is now sorted in ascending order by age.

Leverage Data Structures Effectively

Algorithms get a lot of attention in coding interviews, but data structures are arguably even more important. In a coding interviewing context, picking the right data structure can have a major impact on performance.

Beyond theoretical data structures, Python has powerful and convenient functionality built into its standard data structure implementations. These data structures are incredibly useful in coding interviews because they give you lots of functionality by default and let you focus your time on other parts of the problem.

Store Unique Values With Sets

You'll often need to remove duplicate elements from an existing dataset. New developers will sometimes do so with lists when they should be using sets, which enforce the uniqueness of all elements.

Pretend you have a function named `get_random_word()`. It will always return a random selection from a small set of words:

```
>>> import random
>>> all_words = "all the words in the world".split()
>>> def get_random_word():
...     return random.choice(all_words)
```

You're supposed to call `get_random_word()` repeatedly to get 1000 random words and then return a data structure containing every unique word. Here are two common, suboptimal approaches and one good approach.

Bad Approach

`get_unique_words()` stores values in a list then converts the list into a set:

```
>>> def get_unique_words():
...     words = []
...     for _ in range(1000):
...         words.append(get_random_word())
...     return set(words)
>>> get_unique_words()
{'world', 'all', 'the', 'words'}
```

This approach isn't terrible, but it unnecessarily creates a list and then converts it to a set. Interviewers almost always notice (and ask about) this type of design choice.

Worse Approach

To avoid converting from a list to a set, you now store values in a list without using any other data structures. You then test for uniqueness by comparing new values with all elements currently in the list:

```
>>> def get_unique_words():
...     words = []
...     for _ in range(1000):
...         word = get_random_word()
...         if word not in words:
...             words.append(word)
...     return words
>>> get_unique_words()
['world', 'all', 'the', 'words']
```

This is worse than the first approach, because you have to compare every new word against every word already in the list. That means that as the number of words grows, the number of lookups grows quadratically. In other words, the time complexity grows on the order of $O(N^2)$.

Good Approach

Now, you skip using lists altogether and instead use a set from the start:

```
>>> def get_unique_words():
...     words = set()
...     for _ in range(1000):
...         words.add(get_random_word())
...     return words
>>> get_unique_words()
{'world', 'all', 'the', 'words'}
```

This may not look much different than the other approaches except for making use of a set from the beginning. If you consider what's happening within `.add()`, it even sounds like the second approach: get the word, check if it's already in the set, and if not, add it to the data structure.

So why is using a set different from the second approach?

It's different because sets store elements in a manner that allows near-constant-time checks whether a value is in the set or not, unlike lists, which require linear-time lookups. The difference in lookup time means that the time complexity for adding to a set grows at a rate of O(N), which is much better than the $O(N^2)$ from the second approach in most cases.

Save Memory With Generators

List comprehensions are convenient tools but can sometimes lead to unnecessary memory usage.

Imagine you've been asked to find the sum of the first 1000 perfect squares, starting with 1. You know about list comprehensions, so you quickly code up a working solution:

```
>>> sum([i * i for i in range(1, 1001)])
333833500
```

Your solution makes a list of every perfect square between 1 and 1,000,000 and sums the values. Your code returns the right answer, but then your interviewer starts increasing the number of perfect squares you need to sum.

At first, your function keeps popping out the right answer, but soon it starts slowing down until eventually the process seems to hang for an eternity. This is the last thing you want happening in a coding interview.

What's going on here?

It's making a list of every perfect square you've requested and summing them all. A list with 1000 perfect squares may not be large in computer-terms, but 100 million or 1 billion is quite a bit of information and can easily overwhelm your computer's available memory resources. That's what's happening here.

Thankfully, there's a quick way to solve the memory problem. You just replace the brackets with parentheses:

```
>>> sum((i * i for i in range(1, 1001)))
333833500
```

Swapping out the brackets changes your list comprehension into a generator expression. Generator expressions are perfect for when you know you want to retrieve data from a sequence, but you don't need to access all of it at the same time.

Instead of creating a list, the generator expression returns a `generator` object. That object knows where it is in the current state (for example, `i = 49`) and only calculates the next value when it's asked for.

So when `sum` iterates over the generator object by calling `.__next__()` repeatedly, the generator checks what `i` equals, calculates `i * i`, increments `i` internally, and returns the proper value to `sum`. The design allows generators to be used on massive sequences of data, because only one element exists in memory at a time.

Define Default Values in Dictionaries With `.get()` and `.setdefault()`

One of the most common programming tasks involves adding, modifying, or retrieving an item that may or may not be in a dictionary. Python dictionaries have elegant functionality to make these tasks clean and easy, but developers often check explicitly for values when it isn't necessary.

Imagine you have a dictionary named `cowboy`, and you want to get that cowboy's name. One approach is to explicitly check for the key with a conditional:

```
>>> cowboy = {'age': 32, 'horse': 'mustang', 'hat_size': 'large'}
>>> if 'name' in cowboy:
...     name = cowboy['name']
... else:
...     name = 'The Man with No Name'
...
>>> name
'The Man with No Name'
```

This approach first checks if the `name` key exists in the dictionary, and if so, it returns the corresponding value. Otherwise, it returns a default value.

While explicitly checking for keys does work, it can easily be replaced with one line if you use `.get()`:

```
>>> name = cowboy.get('name', 'The Man with No Name')
```

`get()` performs the same operations that were performed in the first approach, but now they're handled automatically. If the key exists, then the proper value will be returned. Otherwise, the default value will get returned.

But what if you want to update the dictionary with a default value while still accessing the `name` key? `.get()` doesn't really help you here, so you're left with explicitly checking for the value again:

```
>>> if 'name' not in cowboy:
...     cowboy['name'] = 'The Man with No Name'
...
>>> name = cowboy['name']
```

Checking for the value and setting a default is a valid approach and is easy to read, but again Python offers a more elegant method with `.setdefault()`:

```
>>> name = cowboy.setdefault('name', 'The Man with No Name')
```

`.setdefault()` accomplishes exactly the same thing as the snippet above. It checks if `name` exists in `cowboy`, and if so it returns that value. Otherwise, it sets `cowboy['name']` to `The Man with No Name` and returns the new value.

Take Advantage of Python's Standard Library

By default, Python comes with a lot of functionality that's just an `import` statement away. It's powerful on its own, but knowing how to leverage the standard library can supercharge your coding interview skills.

It's hard to pick the most useful pieces from all of the available modules, so this section will focus on just a small subset of its utility functions. Hopefully, these will prove useful to you in coding interviews and also whet your appetite to learn more about the advanced functionality of these and other modules.

Handle Missing Dictionary Keys With `collections.defaultdict()`

`.get()` and `.setdefault()` work well when you're setting a default for a single key, but it's common to want a default value for all possible unset keys, especially when programming in a coding interview context.

Pretend you have a group of students, and you need to keep track of their grades on homework assignments. The input value is a list of tuples with the format `(student_name, grade)`, but you want to easily look up all the grades for a single student without iterating over the list.

One way to store the grade data uses a dictionary that maps student names to lists of grades:

```
>>> student_grades = {}
>>> grades = [
...     ('elliot', 91),
...     ('neelam', 98),
...     ('bianca', 81),
...     ('elliot', 88),
... ]
>>> for name, grade in grades:
...     if name not in student_grades:
...         student_grades[name] = []
...     student_grades[name].append(grade)
...
>>> student_grades
{'elliot': [91, 88], 'neelam': [98], 'bianca': [81]}
```

In this approach, you iterate over the students and check if their names are already properties in the dictionary. If not, you add them to the dictionary with an empty list as the default value. You then append their actual grades to that student's list of grades.

But there's an even cleaner approach that uses a `defaultdict`, which extends standard `dict` functionality to allow you to set a default value that will be operated upon if the key doesn't exist:

```
>>> from collections import defaultdict
>>> student_grades = defaultdict(list)
>>> for name, grade in grades:
...     student_grades[name].append(grade)
```

In this case, you're creating a `defaultdict` that uses the `list()` constructor with no arguments as a default factory method. `list()`with no arguments returns an empty list, so `defaultdict` calls `list()` if the name doesn't exist and then allows the grade to be appended. If you want to get fancy, you could also use a lambda function as your factory value to return an arbitrary constant.

Leveraging a `defaultdict` can lead to cleaner application code because you don't have to worry about default values at the key level. Instead, you can handle them once at the `defaultdict` level and afterwards act as if the key is always present.

Count Hashable Objects With `collections.Counter`

You have a long string of words with no punctuation or capital letters and you want to count how many times each word appears.

You could use a dictionary or `defaultdict` and increment the counts, but `collections.Counter` provides a cleaner and more convenient way to do exactly that. Counter is a subclass of `dict` that uses `0` as the default value for any missing element and makes it easier to count occurrences of objects:

```
>>> from collections import Counter
>>> words = "if there was there was but if \
... there was not there was not".split()
>>> counts = Counter(words)
>>> counts
Counter({'if': 2, 'there': 4, 'was': 4, 'not': 2, 'but': 1})
```

When you pass in the list of words to `Counter`, it stores each word along with a count of how many times that word occurred in the list.

Are you curious what the two most common words were? Just use `.most_common()`:

```
>>> counts.most_common(2)
[('there', 4), ('was', 4)]
```

`.most_common()` is a convenience method and simply returns the `n` most frequent inputs by count.

Access Common String Groups With `string` Constants

It's trivia time! Is `'A' > 'a'` true or false?

It's false, because the ASCII code for `A` is 65, but `a` is 97, and 65 is not greater than 97.

Why does the answer matter? Because if you want to check if a character is part of the English alphabet, one popular way is to see if it's between `A` and `z` (65 and 122 on the ASCII chart).

Checking the ASCII code works but is clumsy and easy to mess up in coding interviews, especially if you can't remember whether lowercase or uppercase ASCII characters come first. It's much easier to use the constants defined as part of the `string` module.

You can see one in use in `is_upper()`, which returns whether all characters in a string are uppercase letters:

```
>>> import string
>>> def is_upper(word):
...     for letter in word:
...         if letter not in string.ascii_uppercase:
...             return False
...     return True
...
>>> is_upper('Thanks Geir')
False
>>> is_upper('LOL')
True
```

`is_upper()` iterates over the letters in `word`, and checks if the letters are part of `string.ascii_uppercase`. If you print out `string.ascii_uppercase` you'll see that it's just a lowly string. The value is set to the literal `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

All `string` constants are just strings of frequently referenced string values. They include the following:

- `string.ascii_letters`
- `string.ascii_uppercase`
- `string.ascii_lowercase`
- `string.digits`
- `string.hexdigits`
- `string.octdigits`
- `string.punctuation`
- `string.printable`
- `string.whitespace`

These are easier to use and, even more importantly, easier to read.

Generate Permutations and Combinations With `itertools`

Interviewers love to give real life scenarios to make coding interviews seem less intimidating, so here's a contrived example: you go to an amusement park and decide to figure out every possible pair of friends that could sit together on a roller coaster.

Unless generating these pairs is the primary purpose of the interview question, it's likely that generating all the possible pairs is just a tedious step on the way towards a working algorithm. You could calculate them yourself with nested for-loops, or you could use the powerful `itertools` library.

`itertools` has multiple tools for generating iterable sequences of input data, but right now we'll just focus on two common functions: `itertools.permutations()` and `itertools.combinations()`.

`itertools.permutations()` builds a list of all permutations, meaning it's a list of every possible grouping of input values with a length matching the `count` parameter. The `r` keyword argument lets us specify how many values go in each grouping:

```
>>> import itertools
>>> friends = ['Monique', 'Ashish', 'Devon', 'Bernie']
>>> list(itertools.permutations(friends, r=2))
[('Monique', 'Ashish'), ('Monique', 'Devon'), ('Monique', 'Bernie'),
('Ashish', 'Monique'), ('Ashish', 'Devon'), ('Ashish', 'Bernie'),
('Devon', 'Monique'), ('Devon', 'Ashish'), ('Devon', 'Bernie'),
('Bernie', 'Monique'), ('Bernie', 'Ashish'), ('Bernie', 'Devon')]
```

With permutations, the order of the elements matters, so `('sam', 'devon')` represents a different pairing than `('devon', 'sam')`, meaning that they would both be included in the list.

`itertools.combinations()` builds combinations. These are also the possible groupings of the input values, but now the order of the values doesn't matter. Because `('sam', 'devon')` and `('devon', 'sam')` represent the same pair, only one of them would be included in the output list:

```
>>> list(itertools.combinations(friends, r=2))
[('Monique', 'Ashish'), ('Monique', 'Devon'), ('Monique', 'Bernie'),
('Ashish', 'Devon'), ('Ashish', 'Bernie'), ('Devon', 'Bernie')]
```

Since the order of the values matters with combinations, there are fewer combinations than permutations for the same input list. Again, because we set `r` to 2, each grouping has two names in it.

`.combinations()` and `.permutations()` are just small examples of a powerful library, but even these two functions can be quite useful when you're trying to solve an algorithm problem quickly.

Conclusion: Coding Interview Superpowers

You can now feel comfortable using some of Python's less common, but more powerful, standard features in your next coding interview. There's a lot to learn about the language as a whole but this article should have given you a starting point to go deeper while letting you use Python more effectively when you interview.

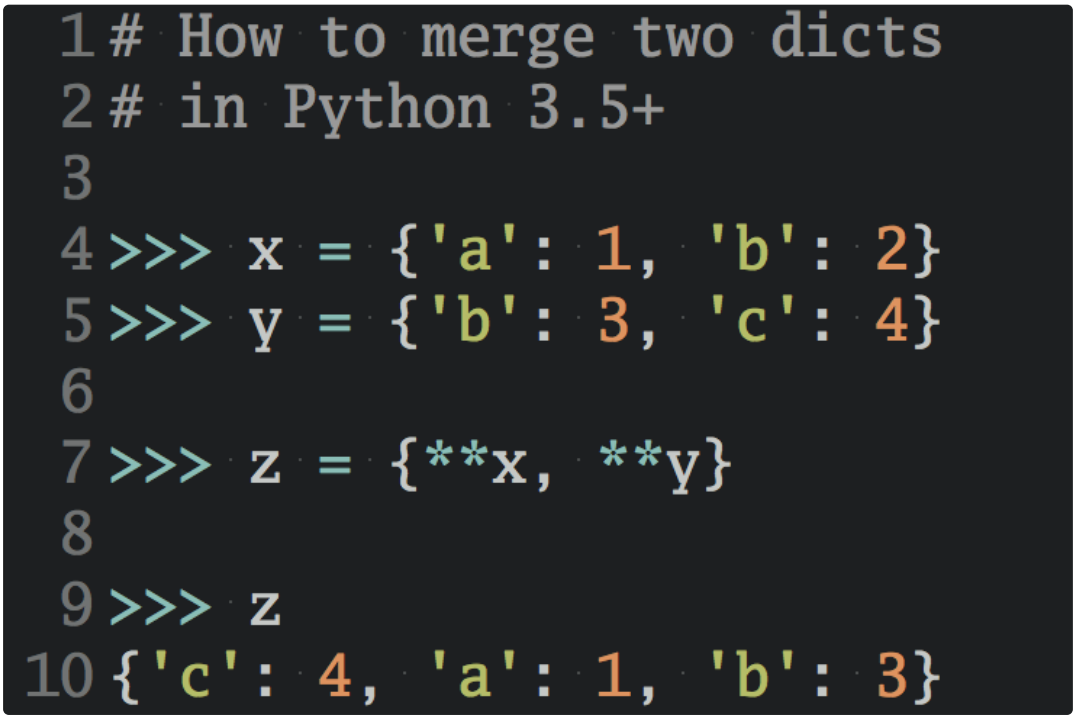
In this article, you've learned different types of standard tools to supercharge your coding interview skills:

- Powerful built-in functions
- Data structures built to handle common scenarios with barely any code
- Standard library packages that have feature-rich solutions for specific problems, letting you write better code faster

Interviewing may not be the best approximation of real software development, but it's worth knowing how to succeed in any programming environment, even interviews. Thankfully, learning how to use Python during coding interviews can help you understand the language more deeply, which will pay dividends during day-to-day development.

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.



Email Address

Send Me Python Tricks »

About James Timmins

James is a software consultant and Python developer. When he's not writing Python, he's usually writing about it in blog or book form.

» More about James

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are: