

Pydon'ts

Write elegant  code

From the international Python speaker

RODRIGO GIRÃO SERRÃO

Pydon'ts

Write elegant Python code

Rodrigo Girão Serrão

23-08-2021

Contents

Foreword	8
Pydon't disrespect the Zen of Python	9
Zen of Python	10
References	11
Does elegance matter?	12
Introduction	12
Beware, opinions ahead	13
Elegance is not a dispensable luxury	13
Don't write that, that's unreadable	13
The right tool for the job	14
Optimise for rewrite	15
Conclusion	15
References	16
Code style matters	17
Introduction	18
Code style	18
Tools for your tool belt	19
Conclusion	22
References	22
Naming matters	24
Introduction	24
Naming conventions	25
PEP 8 recommendations	25
Standard names	27
Verbosity	28
Picking a name	31
Context is key	33
Practical example	33
Conclusion	35
References	36

Chaining comparison operators	37
Introduction	38
Chaining of comparison operators	38
Pitfalls	39
Ugly chains	40
Examples in code	41
Conclusion	42
References	42
Assignment expressions and the walrus operator :=	43
Walrus operator and assignment expressions	43
Examples in code	44
Conclusion	47
References	47
Truthy, Falsy, and bool	48
“Truthy” and “Falsy”	50
The <code>__bool__</code> dunder method	51
Remarks	52
Examples in code	53
Conclusion	56
References	56
Deep unpacking	57
Introduction	57
Assignments	58
Examples in code	60
Conclusion	62
References	62
Unpacking with starred assignments	64
Starred Assignment	65
Examples in code	66
References	67
EAFP and LBYL coding styles	69
EAFP and LBYL	69
EAFP instead of LBYL?	70
Conclusion	74
References	74
Zip up	75
Introduction	75
How <code>zip</code> works	76
Zip is lazy	77
Three is a crowd	78
Mismatched lengths	78

Create a dictionary with <code>zip</code>	79
Examples in code	79
Conclusion	82
References	83
Enumerate me	84
Introduction	84
How <code>enumerate</code> works	85
Optional start argument	86
Unpacking when iterating	86
Examples in code	88
Conclusion	93
References	94
str and repr	95
<code>str</code> and <code>repr</code>	97
The <code>__str__</code> and <code>__repr__</code> dunder methods	97
Examples in code	98
Conclusion	100
References	100
Structural pattern matching tutorial	101
Introduction	102
Structural pattern matching Python could already do	102
Your first <code>match</code> statement	102
Pattern matching the basic structure	103
Matching the structure of objects	105
<code>__match_args__</code>	106
Wildcards	107
Naming sub-patterns	109
Traversing recursive structures	110
Conclusion	112
References	112
Structural pattern matching anti-patterns	114
Introduction	115
There should be only one obvious way to do it	115
A short and sweet <code>if</code> statement	115
Be smart(er)	116
Basic mappings	117
Conclusion	122
References	123
Watch out for recursion	124
Introduction	125
Watch out for recursion	125
Examples in code	128

Conclusion	134
References	135
Sequence indexing	136
Introduction	136
Sequence indexing	137
Maximum legal index and index errors	138
Negative indices	138
Indexing idioms	140
To index or not to index?	140
Best practices in code	141
Conclusion	144
Idiomatic sequence slicing	146
Introduction	146
Slicing syntax	147
What to slice?	148
Slicing from the beginning	149
Slicing until the end	150
Slicing with negative indices	150
Slicing and range	151
Idiomatic slicing patterns	152
Empty slices	153
More empty slices	154
Examples in code	154
Conclusion	157
References	158
Mastering sequence slicing	159
Introduction	159
Slicing step	160
Sequence copying	167
Manipulating mutable sequences	167
More idiomatic slicing	170
Conclusion	173
References	174
Inner workings of sequence slicing	175
Introduction	176
The slice class	176
Getting items from sequences	178
Setting items, deleting items, and container emulation	179
Comma-separated indices and slices	179
Examples in code	180
Conclusion	184
References	184

Boolean short-circuiting	186
Introduction	186
Return values of the and and or operators	187
Short-circuiting	188
Short-circuiting in plain English	190
all and any	191
Short-circuiting in chained comparisons	191
Examples in code	191
Conclusion	200
References	200
The power of reduce	202
Introduction	202
How reduce works	203
The rabbit hole of the built-in reductions	204
Why bother?	205
Far-fetched reductions	205
The identity element...	207
Why some people dislike reduce	209
Examples in code	209
Conclusion	211
References	211
Usages of underscore	213
Introduction	213
Recovering last result in the session	214
Prefixes and suffixes for variable names	215
Underscore as a sink	221
Matching everything in the new match statement	223
String localisation	224
Improve number readability	225
Conclusion	226
References	226
name dunder attribute	228
Introduction	228
What is <code>__name__</code> ?	229
The module attribute <code>__name__</code>	229
<code>__name__</code> as an object type attribute	231
Examples in code	233
Conclusion	237
References	237
Bite-sized refactoring	238
Introduction	238
Refactoring	239
What to refactor?	240

Case study	240
Conclusion	247
References	248
String translate and maketrans methods	249
Introduction	249
str.translate	250
Non-equivalence to str.replace	252
Generic translation tables	253
str.maketrans	254
Examples in code	256
Conclusion	260
References	261
Boost your productivity with the REPL	262
Introduction	262
REPL	263
Just fire up the REPL	263
REPL mechanics	263
The last result	267
Getting help from within the REPL	268
Tips for quick hacks	269
Other tools	271
Conclusion	272
References	272
set and frozenset	273
Introduction	273
(Mathematical) sets	274
(Common) Operations on sets	275
Differences between set and frozenset	278
What are sets used for?	280
Examples in code	281
Conclusion	285
References	285
List comprehensions 101	286
Introduction	286
What is a list comprehension?	287
Anatomy of a list comprehension	287
Example list comprehensions without filtering	288
Example list comprehensions with filtering	288
Equivalence with for loops	289
Nesting for loops	291
Nesting if statements	292
Arbitrary nesting	293
List comprehensions instead of map and filter	294

Examples in code	295
Conclusion	297
References	298
Conditional expressions	299
Introduction	299
What is a conditional expression?	300
Rationale	302
Short-circuiting	303
Conditional expressions and <code>if</code> statements	305
Precedence	308
Conditional expressions that evaluate to Booleans	309
Examples in code	310
Conclusion	311
References	312
Pass-by-value, reference, and assignment	313
Introduction	315
Is Python pass-by-value?	315
Is Python pass-by-reference?	316
Python object model	317
Python is pass-by-assignment	321
Making copies	322
Examples in code	325
Conclusion	329
References	329
String formatting comparison	331
Introduction	331
String formatting rationale	332
Three string formatting methods	333
Value conversion	334
Alignment	335
Named placeholders	336
Accessing nested data structures	336
Parametrised formatting	337
Custom formatting	338
Examples in code	339
Conclusion	340
References	341
Closing thoughts	342

Foreword

I'm glad that you are reading these words.

For a long time now, teaching and sharing knowledge has been a passion of mine. If you are reading this book, then it probably means that you hope to find new knowledge in here, knowledge that I am sharing with you. This brings me great joy.

I have been writing Python since I was 15 years old, which means I have now written Python for more than a third of my life (if you do the maths, that means I am at least 22.5 years old at the time of writing). This is not necessarily something that *you* find impressive, but I do appreciate that fact about myself.

Python was not my first programming language, and I remember picking it up as a friend of mine recommended it to me. Now, many years later, I still enjoy writing Python code, whether for work-related reasons or for my own projects (just take a look at <https://mathspp.com/blog/tag:python>). In programming, much like in mathematics – my main area of expertise –, there is a sense of elegance in the code (or proofs) we write. As I learned more and more about programming in general and Python in particular, I developed a sense for what *I* consider to be elegant Python programs. This is one of the things I intend to share in this book: tips on how to write beautiful Python programs.

Of course, the notion of elegance is a subjective one, so it may very well be the case that what I find elegant is not what *you* find elegant, and that is perfectly fine. In general, neither one of us will be wrong.

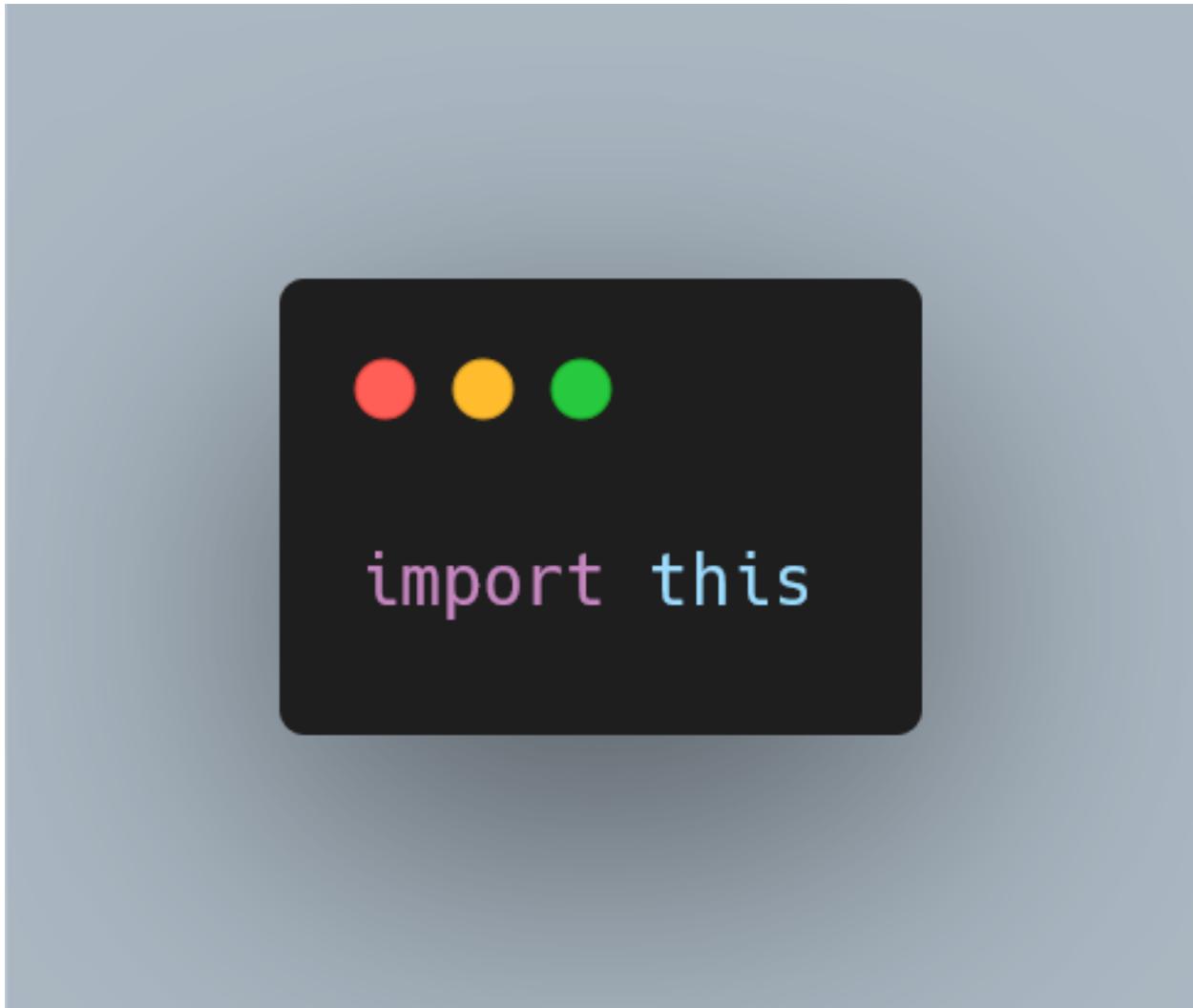
Tied to my effort of sharing my interpretation of what elegant Python programs look like, I also want you to learn about all the nooks and crannies of the core language. Python is a very, very, rich language, and the more you learn about it, the more well equipped you will be to use it to its full potential. That is why every chapter focuses on exploring a single feature of the core language of Python, which is always accompanied by usage examples of said feature. Some times we will look at how Python's own Standard Library makes use of that feature, other times I will show some of my own code, and other times I will even come up with random examples.

For now, this book is being expanded as I write one chapter per week and publish it to <https://mathspp.com/blog/pydonts>, where you can read the contents of this book for free.

If you would like to share your feedback, let me know of any typos or errors you found, or otherwise just get in touch, feel free to drop a line to rodrigo@mathspp.com.

— Rodrigo, <https://mathspp.com>

Pydon't disrespect the Zen of Python



(Thumbnail of the original article at <https://mathspp.com/blog/pydons/pydont-disrespect-the-zen-of->

[python.\)](#)

To kick-off the Pydon't series we start with a set of guidelines that all Pythonistas should be aware of: the [Zen of Python](#).

The [Zen of Python](#) is like a meta style guide. While you have things like [PEP 8](#) that tell you how you should format your code, how to name your variables, etc., the [Zen of Python](#) provides you with the guidelines that you should follow when thinking about (Python) code and when designing a program.

Zen of Python

You can read the [Zen of Python](#) by executing `import this` in your REPL, which should print the following text:

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Take a look at those guidelines and try to appreciate their meaning. If you want to write truly Pythonic code, then you should try to embrace these guidelines as much as possible.

Digging in the reference of [PEP 20 – The Zen of Python](#) shows that Tim Peters (a major contributor to Python in its earlier days) thinks that these guidelines are “*fundamental idiomatic recommendations for operating within the spirit of the [Python] language*”, which goes to show that these recommendations are serious and should not be taken lightly - if you are willing to go the extra mile.

If you've seen the [Kung Fu Panda](#), think of it this way: the [Zen of Python](#) is to Python programmers what the *Dragon Scroll* is to kung fu practitioners: Po was only able to take his kung fu skills to the next level, becoming truly amazing, after embracing the Dragon Scroll. You will only become a *true* Pythonista after you embrace the [Zen of Python](#).

My advice would be to read this from time to time, and to try and remember the [Zen of Python](#) while you code and while you go over code that has already been written (by you or someone else). I don't know about *you*, but whenever I write a (text) document, like a letter or a blog post, I never get it right on the first try. I usually write a first draft and then go over it, editing as I see fit: sometimes reworking whole sections. Writing code is the same: chances are, the first thing you write can be greatly improved upon.

This Pydon't was more of a "meta" Pydon't, with subjective advice on how to code. This might seem useless to you at first, but the more you dwell on it the more helpful it will become. The next Pydon'ts will show you objective, practical tips on how to write more Pythonic code.

References

- PEP 20 – The Zen of Python, <https://www.python.org/dev/peps/pep-0020/>
- "The Way of Python" mailing thread, https://groups.google.com/g/comp.lang.python/c/B_VxeTBCIM0/m/L8W9KlsiriUJ
- Tim Peters (software engineer), Wikipedia [https://en.wikipedia.org/wiki/Tim_Peters_\(software_engineer\)](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer))

Does elegance matter?

“

ELEGANCE IS NOT
A DISPENSABLE LUXURY,
BUT A CRUCIAL MATTER
THAT DECIDES BETWEEN
SUCCESS AND FAILURE.

— Edsger W. Dijkstra

(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/does-elegance-matter>.)

Introduction

At the time of writing this Pydon’t, I am finishing the preparation of [my Python conference talk “Pydon’ts” at EuroPython](#).

For that matter, today’s Pydon’t will be a bit different. Usually, I write about using Python’s core features to write idiomatic, expressive, elegant Python code. In this Pydon’t I will share with you *why* this is important.

Beware, opinions ahead

Idiomatic code, readable code, “Pythonic” code, elegant code, these are all subjective things. That means that whatever I write about these topics will never be 100% consensual. In other words, you might disagree.

I am fine with the fact that there are people who disagree with me, and I do invite you to make yourself heard, maybe by writing me or leaving a comment on the blog – diversity of points of view is enriching.

I just want to let you know that this Pydon’t might not be a good read for you if you can’t stand the fact that other people might think differently from you ☺.

Elegance is not a dispensable luxury

Let me be honest with you: when I was preparing my talk, and preparing this Pydon’t, I thought that part of my argument about why elegance is important was going to draw from my experience as a mathematician – when doing mathematics, people usually strive for writing elegant proofs, constructing simple arguments, finding even simpler counter-examples to others’ arguments, etc.

Then, I found a quote by a respectable computer scientist that made this connection for me, and I felt much more confident with the parallel I was trying to establish. Edsger W. Dijkstra, a Dutch computer scientist, after which the “[Dijkstra algorithm][dijkstra-alg]” was named, wrote:

“How do we convince people that in programming simplicity and clarity – in short: what mathematicians call “elegance” – are not a dispensable luxury, but a crucial matter that decides between success and failure?”

✉ Edsger W. Dijkstra, “Selected Writings on Computing: A Personal Perspective”, p347.

I think Dijkstra’s quote says it all: simple and clear code is elegant code. And this these are very desirable properties to have in code. In fact, a little bit further down the page, Dijkstra adds

“[...] in the case of software unreliability is the greatest cost factor. It may sound paradoxical, but a reliable (and therefore simple) program is much cheaper to develop and use than a (complicated and therefore) unreliable one.”

From my experience, people mistake *beginner-level* code for *simple* and *clear* code, and that is something very dangerous, in my opinion.

Don’t get me wrong, there is nothing inherently wrong with beginner-level code, we all write it when we are learning. What is wrong is when you hold yourself back, or when others hold you back, because they force you to write beginner-level code: which is code that only uses the most basic tools in the language.

Don’t write that, that’s unreadable

Think about whether or not you have been in this situation before, or have witnessed it: you write a beautiful couple of lines of code that does *exactly* what needed to be done. Then a peer of yours walks by (or worse, someone hierarchically above you), glances at your code, and says “don’t write code like that, that’s unreadable”. And as it turns out, they just said that because you used a feature that they don’t know of! And because

they don't know the feature that you used, they *glanced* at your code, didn't feel *comfortable* reading it, and determined it was unreadable!

I don't know the Chinese language. If I open a book written in Chinese, do you think it makes sense that I look at the pages and assert "this is unreadable!"? What I need to do is acknowledge that I *can't* read Chinese, not that Chinese is unreadable. If you don't know the language, it doesn't make sense that you say a piece of it is unreadable. The same thing goes for Python, or any other programming language.

Here is an [APL](#) expression:

```
(a b c) ← 1 ¯4 3
(2×a)÷-b(+,-)0.5*(b*2) ¯4×a×c
1 3
```

This computes the two solutions to the equation $x^2 + -4x + 3 = 0$. Is that piece of code unreadable? If you *know* APL, then the answer is "no", it is not unreadable, because it is a very natural way of writing the quadratic formula in APL. However, if you do *not* know APL then your impulse will be to say that it *is* unreadable, when in fact you mean that you *do not know how to read it*.

In short, something is unreadable when you know what all the constituent pieces mean, and yet the pieces are put together in a way that doesn't convey the global meaning well. This could be due to the usage of the wrong constituent pieces, because pieces are missing, because there are superfluous pieces, etc.

Therefore, if you don't know all the features that are being used in a piece of Python code, I claim that you are not in a position to look at it and say that the code is unreadable. First, you have to take the time to learn what all the different pieces are. Only then you can look at that line of code and determine if the pieces were well put together or not.

Of course this takes *time* and is an effort that must be done consciously, and that is why most people don't even bother. Please, don't be like most people.

The right tool for the job

To back me up, I have the words of legendary [Tim Peters](#), the author of [The Zen of Python](#). Addressing some new features of Python, someone complained about "the additional complexities to reading other's code", to which Tim Peters replied

"Here's the plan: When someone uses a feature you don't understand, simply shoot them. This is easier than learning something new, and before too long the only living coders will be writing in an easily understood, tiny subset of Python 0.9.6 <wink>."

✉ Tim Peters

I wasn't around writing Python code when we were at version 0.9.6, but I'm guessing there wasn't too much back then. Maybe your plain `for` loops and `if` statements.

If you want to improve, you have to learn. That's just the way it is.

This is what my Pydon'ts try to address. I try to teach you all the features there are to know about the core Python, and then also show you examples of code where those are put to good use, so that you extend your knowledge of Python.

And it is through the process of continuous learning that you will be able to write elegant code. As you learn more syntax, more functions, more modules, more data types, etc, you get to know more tools to solve your problems. And the more tools you have on your tool belt, the more likely you are to know about *the perfect tool for the job* when you need to do something new.

If you have the right tool for the job, then you can use it to solve your problem. Simplicity, clarity, elegance; all those will follow naturally.

If you don't know about the tools, your solution can end up being worse in a variety of ways, for example:

- (much) longer – you have to write more code to make up for the fact that you don't know of better functions, modules, syntax, etc.;
- (much) slower – the correct tool has probably been optimised for its use cases;
- paradoxically, harder to maintain – the wrong tools won't be able to convey the same meaning in the same way, and thus that piece of code takes longer to understand, debug, tweak, etc.

By opposition, writing elegant and Pythonic code will make it

- more succinct;
- clearer;
- easier to maintain;
- faster;

Optimise for rewrite

At this point I am assuming that you are interested in this notion of continuous learning – and by that, I don't mean to say that you should devour the full Python documentation in the next week. You just agree with me in that trying to learn as much as possible about the language will naturally lead you to write better code.

If you are continuously learning, then it will happen that you look at an old piece of code of yours and go "funny, I know a much better way to do that". Because you try to write simple and elegant code, you can look at your code and quickly understand what that piece of code is doing, hence, you can modify your old code to include the new, improved solution. But that just made your code even clearer and more elegant. And this is a loop that has the potential to go on forever.

Idiomatic code is code whose semantics are easy to understand, that is, idiomatic code is code that conveys its intent really well. If you do things right, elegant code and idiomatic code go hand-in-hand, and if you strive to write code like that, you are essentially opening up the doors for future rewrites of your code. This is a good thing, because code that can be changed is code that can be improved over time.

Would you rather write code that is easy to understand and modify, or would you rather write code that everyone (including your future self) is scared of going near to?

Conclusion

Part of the elegance in your Python programs will come naturally from learning about the features that Python has to offer, about the built-ins, the modules in the standard library, etc.

By taking those and giving your best shot at employing them in the correct situations, using them to solve the problems that they were made for, you will be well on your path to writing idiomatic, elegant code. And in writing code like that, you make it easier on yourself, and on others, to continuously keep improving your code and, as a consequence, your project/product.

So, next time someone looks at your code to flatten lists of lists and says “that’s unreadable”:

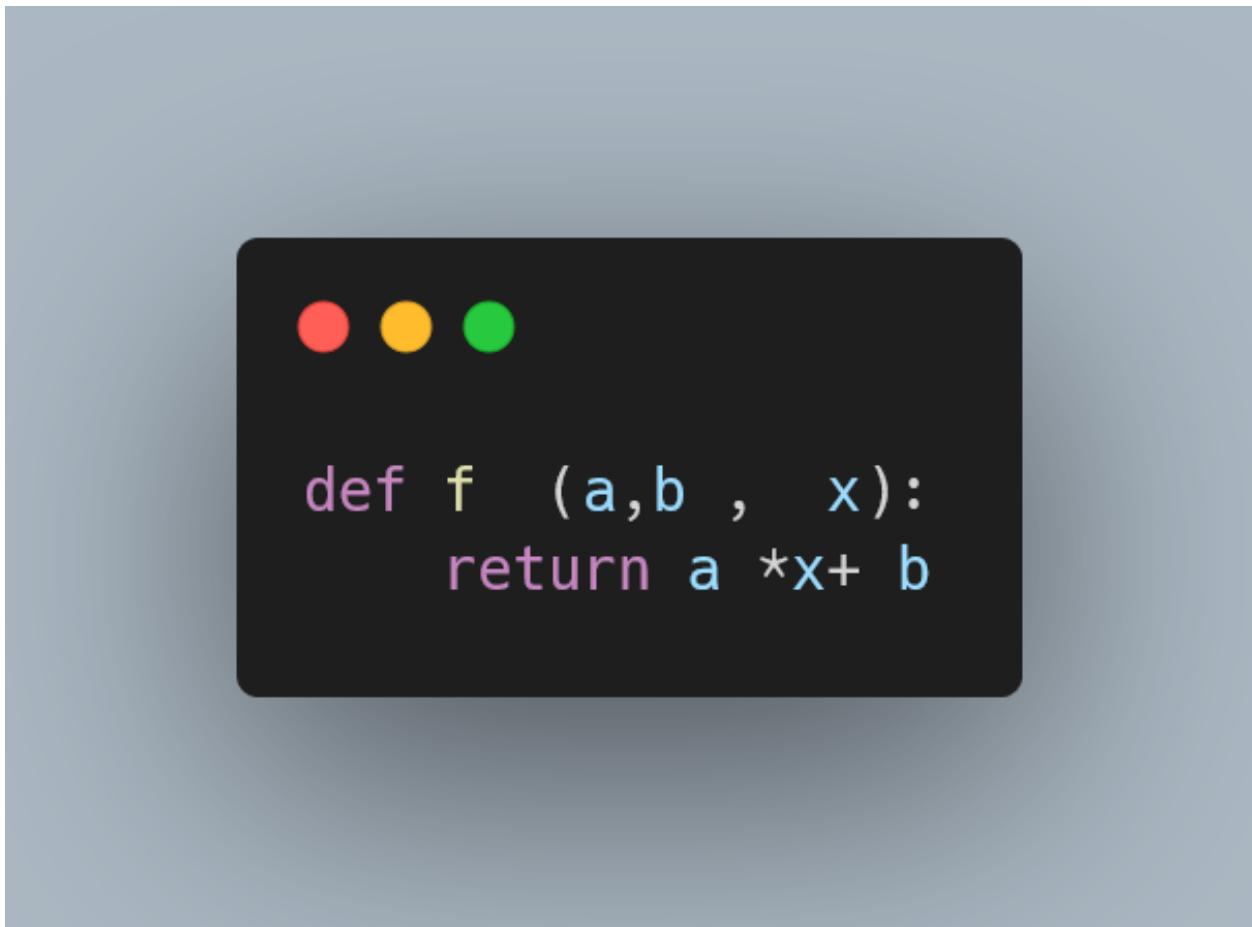
```
>>> import itertools
>>> list_of_lists = [[1, 2, 3], [4], [5, 6]]
>>> list(itertools.chain(*list_of_lists))
[1, 2, 3, 4, 5, 6]
```

just tell them to go read the docs.

References

- Edsger W. Dijkstra (2012), “Selected Writings on Computing: A personal Perspective”, p.347, Springer Science & Business Media;
- Aaron W. Hsu (2017), “Beginner Patterns vs Anti-patterns in APL”, FnConf’17, <https://www.youtube.com/watch?v=v7Mt0GYHU9A> [last accessed 06-07-2021];
- Tim Peters (2002), Python mailing list thread, <https://mail.python.org/pipermail/python-list/2002-December/134521.html> [last accessed 06-07-2021];

Code style matters



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/code-style-matters>.)

Introduction

The overall style of your code can have a great impact on the readability of your code. And code is more often read than written, so you (and others!) have a lot to benefit from writing well styled code.

In this Pydon't, you will:

- understand the importance of having a consistent style; and
- learn about tools that help you with your code style.

By the way, this week I wrote a shorter and lighter Pydon't, as I am still investing lots of time [preparing for Euro Python 2021](#) at the time of writing... I hope you still find it useful!

Code style

Consistency

Humans are creatures of habit. From the fact that the first leg that goes into your trousers is always the same, to the fact that you always start brushing your teeth on the same side.

These habits automate routines that do not require much attention, so that you can spend your precious brain power on other things.

As far as my experience goes, the same can be said about your coding style: if you write with a consistent code style, it becomes easier to read because you already expect a given structure; you are only left with acquiring the information within that structure.

Otherwise, if your style isn't consistent, you have to spend more precious brain power parsing the structure of what you are reading and only then apprehend the information within that structure.

[PEP 8](#) is a document whose purpose is to outline a style guide for those who write Python code. It has plenty of useful recommendations. However, right after the introduction, PEP 8 reads

“A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

However, know when to be inconsistent – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!”

This is very important: PEP 8 is a style guide that contains *recommendations*, not laws or strict rules. And what is more, notice that there is a strong focus on *consistency*. Using your own (possibly weird) style consistently is better than using no style at all. That's if you are working alone; in a project, it is a good idea to decide on a particular style beforehand.

Whitespace matters

When I'm teaching Python, I often do some sort of live coding, where I explain things and type examples, that I often ask students to type as well. I have noticed that people that are *just starting* with Python will

often copy the words that I'm typing, but won't respect my whitespace usage.

It is a bit of an exaggeration, but I might type

```
def f(a, b, x):  
    return a*x + b
```

and then they will type things like

```
def f  (a,b , x):  
    return a *x+ b
```

Python is the language where whitespace matters (because Python uses indentation to nest structures), but whitespace turns out to be important in more places than just those.

For example, above, we can see that the misuse of blank spaces makes the second definition of `f` much more hectic and aesthetically unpleasant. And if there is one thing we know, is that **elegance matters**.

If you skim through PEP 8 you will find that *most* recommendations there are about whitespace. Number of empty lines around functions, classes, methods; whitespace around operators and keywords; whitespace before/after commas; etc. Take a look at PEP 8 and gradually try to incorporate some recommendations into your coding.

For example, PEP 8 suggests that you use whitespace to help the reader parse the priority of mathematical operations in an expression. Above, I wrote

```
return a*x + b
```

surrounding `+` with blanks, to indicate that the `a*x` (notice the lack of blanks around `*`) has higher priority. Of course, the number of blank spaces I use doesn't alter the order of operations, but it helps you see the order of things.

And the cool thing is that the more used to writing like this you are, the more helpful it becomes!

Again, my suggestion is that you take a look at PEP 8 and pick a couple of recommendations you enjoy and try incorporating those into your coding. When those sink in, add a couple more. And just roll with that. Easier than trying to change everything all at once.

Tools for your tool belt

On a happier note, there are *many* tools you can use that help you format your code and keep it neat and tidy.

Auto-formatters

`black`

A class of tools that you can use is what are known as (auto-)formatters, of which `black` is a prime example (see their repo [here](#)).

Auto-formatters like `black` take your code and reformat it so that it fits within the style that the tool supports/you configure.

For example, let me create the file `my_f.py` and paste this code in there:

```
## In my_f.py
def f(a, b, x):
    return a * x + b
```

Now let me run `black` on that file:

```
> python -m black my_f.py
reformatted my_f.py
All done!
1 file reformatted.
```

Now, I open my file and this is what is inside:

```
## In my_f.py
def f(a, b, x):
    return a * x + b
```

As we can see, `black` took my code and just reformatted it to the style that `black` adheres to. `black`'s style is fairly similar to PEP 8's style and `black` is a great tool if you just want to have something automatically helping you reformat your code, so that you don't have to think too much about it.

`black` is as easy to install as any other Python module:

```
python -m pip install black
```

There are many tools like `black` out there; another common option is `pycodestyle`.

`pycodestyle` `pycodestyle` checks if your style is similar to what PEP 8 recommends. In fact, `pycodestyle` used to be called `pep8`, but was renamed so that people understand that:

1. PEP 8 isn't a set of rigid rules; and
2. `pycodestyle` doesn't match PEP 8's recommendations 100%.

Let me modify the file `my_f.py` to the following:

```
## In my_f.py
import os, time
def f(a, b, x):
    return a * x + b
```

If I run `pycodestyle`, this is what I get as output:

```
> python -m pycodestyle my_f.py
my_f.py:2:10: E401 multiple imports on one line
my_f.py:3:1: E302 expected 2 blank lines, found 0
```

We can see that `pycodestyle` complained about a couple of things:

1. the fact that I merged `import os` and `import time`; and
2. the fact that there aren't enough empty lines separating the imports from `f`.

A big difference between `black` and `pycodestyle` is that `black` does reformat your code, whereas `pycodestyle` just complains.

Installing `pycodestyle` is just a matter of typing

```
python -m pip install pycodestyle
```

For both tools, and for most of the similar tools out there, you can configure them to ignore types of errors, or ignore sections of your code, etc. Just go read their documentation!

Level up (aka linters)

(Auto-)Formatters are helpful, but there are other tools out there that have even more potential: linters.

Linters are tools that analyse your code and help you find things like

- stylistic issues (like the formatters do);
- programming errors;
- some types of bugs;
- etc.

These tools can be incredibly helpful, for example, to manage all the imports in a big project. I often find myself importing some modules and using them. Later, [I refactor my code](#), and I stop needing those imports. When I do that, I *always* forget to check if the imports are still needed or no longer relevant. Linters can, for example, flag unused imports.

An example of a linter is `flake8` (you can find it [here](#)). If I use `flake8` on my `my_f.py` file, here is what I get:

```
> python -m flake8 my_f.py
my_f.py:2:1: F401 'os' imported but unused
my_f.py:2:1: F401 'time' imported but unused
my_f.py:2:10: E401 multiple imports on one line
my_f.py:3:1: E302 expected 2 blank lines, found 0
```

You can see that now `flake8` is complaining about the fact that I am importing things that I don't use at all! Not only that, but the two bottom lines are identical to `pycodestyle`'s output above... And that's because `flake8` uses `pycodestyle` within itself.

You can install `flake8` with

```
python -m pip install flake8
```

Another fairly common alternative for a linter is `pylint` ([pylint's page](#)). Running it on the same `my_f.py` file, I get some more warnings:

```
> python -m pylint my_f.py
*****
Module my_f
my_f.py:1:0: C0114: Missing module docstring (missing-module-docstring)
my_f.py:2:0: C0410: Multiple imports on one line (os, time) (multiple-imports)
my_f.py:3:0: C0103: Function name "f" doesn't conform to snake_case naming style (invalid-name)
my_f.py:3:0: C0103: Argument name "a" doesn't conform to snake_case naming style (invalid-name)
my_f.py:3:0: C0103: Argument name "b" doesn't conform to snake_case naming style (invalid-name)
```

```
my_f.py:3:0: C0103: Argument name "x" doesn't conform to snake_case naming style (invalid-name)
my_f.py:3:0: C0116: Missing function or method docstring (missing-function-docstring)
my_f.py:2:0: W0611: Unused import os (unused-import)
my_f.py:2:0: W0611: Unused import time (unused-import)
```

Your code has been rated at **-20.00/10**

We can see that pylint was more unforgiving, complaining about the fact that I did not include docstrings and complaining about my 1-letter names. This might be something you appreciate! Or not!

I reckon personal taste plays a big role in picking these tools.

Installing pylint can be done through

```
python -m pip install pylint
```

Conclusion

As far as these tools are concerned, I suggest you pick something that is fairly consensual for your personal projects, so that it doesn't hurt you too much when you contribute to other projects. For open source projects, you will often be asked to follow a given style, and there may or may not be tools that help you reformat your code to follow that style.

This Pydon't was not supposed to be a thorough review of all the possibilities there are out there, I only touched upon a couple of popular alternatives, so that might be a decent indicator of things that are consensual.

By the way, many IDEs these days have integrated support for these linters, making it even easier to harness their helpful suggestions.

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"Pay attention to the style with which you write code and pick a suite of tools to help you if you want/need."

This Pydon't showed you that:

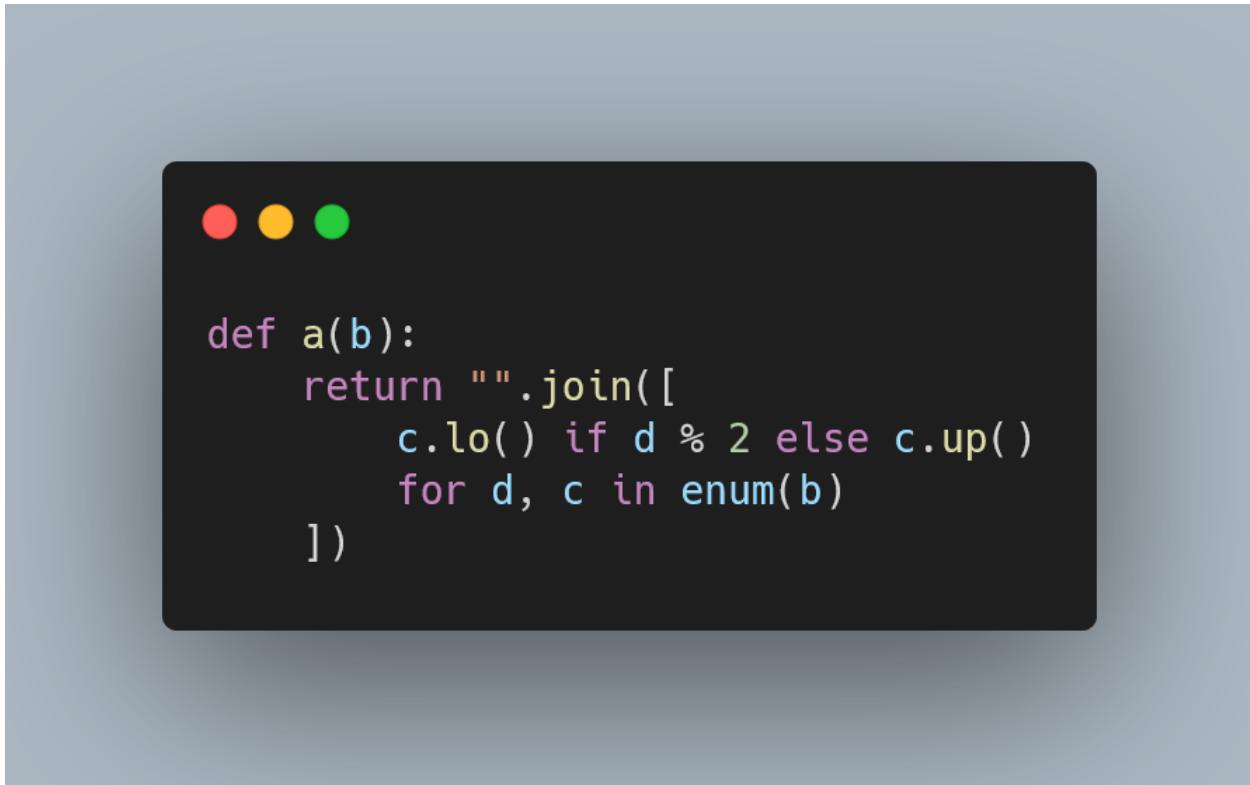
- coding style has an impact in code readability;
- tools like black and pycodestyle can help you fix the style of your code; and
- linters like flake8 and pylint can give further insights into some types of errors/bugs/problems your programs might have.

References

- PEP 8 – Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008> [last accessed 20-07-2021];
- black – The Uncompromising Code Formatter, <https://github.com/psf/black> [last accessed 20-07-2021];

- pycodestyle, <https://pycodestyle.pycqa.org/en/latest/intro.html> [last accessed 20-07-2021];
- pydocstyle, <http://www.pydocstyle.org/en/stable/> [last accessed 20-07-2021];
- flake8, <https://flake8.pycqa.org/en/latest/index.html> [last accessed 20-07-2021];
- pylint, <https://www pylint.org/> [last accessed 20-07-2021];

Naming matters



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/naming-matters>.)

Introduction

Names are like real estate in our brains. They are labels we give to things, concepts, ideas, objects, so that it is easier for us to refer to those things, but they take up space. As it turns out, we can only hold a very small number of different ideas in our heads, so these labels are very expensive...

We might as well do the best job we can to manage them as well as possible!

In this Pydon't, you will:

- learn about some naming conventions;
- learn the suggested naming conventions for Python code; and
- understand some do's and don'ts for naming variables, functions, methods, etc, in Python.

Naming conventions

When we talk about names, there are two things that need to be discussed. One of them is the actual name that you give to things, and the other is the way in which you write the name: the casing of the letters and how consecutive words are separated.

These are often referred to as naming conventions, and there are a few of them. I will present them here, so that I can refer to them later in the Pydon't.

The list that follows is *not* comprehensive, in that there are more naming conventions out there. However, they are not relevant for this Pydon't.

- CAPSLOCK – all letters of all words are upper case and there is nothing to separate consecutive words;
- CAPS_LOCK_WITH_UNDERSCORES – like the one above, but with underscores separating words;
- lowercase – all letters of all words are lower case and there is nothing to separate consecutive words;
- snake_case – like the one above, but with underscores separating words;
- PascalCase – all words are put together, but their initials are capitalised to help you know where one word ends and the other begins; and
- camelCase – like the one above, except the very first word starts with a lower case letter as well.

On top of these naming conventions, sometimes leading and/or trailing underscores can be added to the mix. That isn't strictly related to the naming conventions by themselves, but it is related to the way Python uses names. In case you need a refresher, I wrote a Pydon't that teaches you [all the usages of underscore](#) and, in particular, it tells you what the underscores do for you if in the beginning and/or end of a name.

PEP 8 recommendations

[PEP 8](#) is a document – a Python Enhancement Proposal – that contains a style guide for Python, and it is the most widely accepted and used style guide for Python. In case you don't know it, it might be worth taking a look at it.

[PEP 8](#) starts by acknowledging that “the naming conventions of Python's library are a bit of a mess”, so bear in mind that if you start working on some project that already uses a specific naming convention, you should stick to it. Remember that being consistent is more important than following the PEP 8 guide.

PascalCase

You can find the PascalCase convention often on classes. That is the most common use case for this convention.

What this means is that your classes will look like:

```

class Shape:
    # ...

class Circle(Shape):
    # ...

and

class GameArena:
    # ...

class HumanPlayer:
    # ...

class NPC:
    # ...

class AIPlayer:
    # ...

```

Notice that the NPC and AIPlayer classes are actually using acronyms: NPC stands for non-playable character and AI stands for artificial intelligence. PEP 8 recommends that you capitalise all letters of an acronym in a PascalCase name. Sometimes this makes it look like we are using the CAPSLOCK convention.

Other common use cases for the PascalCase convention include exceptions – which shouldn't surprise you because exceptions come from classes –, and type variables.

snake_case

The snake_case naming convention is the bread and butter of naming in Python. Variables, functions, methods, arguments, they all use the snake_case convention.

So, prefer

```
def cool_function(cool_argument, optional_info):
    # ...
```

to things like

```
def coolFunction(coolargument, optionalinfo):
    # ...
```

```
def CoolFunction(coolArgument, optionalInfo):
    # ...
```

```
def COOL_FUNCTION(cool_argument, optional_info):
    # ...
```

As an additional note, PEP 8 also recommends that you append an underscore to a name that you want to use, when that name is a keyword. So, for example, you cannot have a variable called `for`, but you could call

it for_.

CAPS_LOCK_WITH_UNDERSCORES

This naming convention, that might look a bit clunky to you, is actually used to represent global constants. Python doesn't have support for variables that are truly constant – in the sense that trying to change them would throw an error – and so we use this widely adopted convention that variables that are used as global constants are named with the CAPS_LOCK_WITH_UNDERSCORES convention.

Generally, you will find these “constants” in the beginning of a file.

For example, I often have a couple of paths defined this way:

```
IMG_BIN = "images"  
LOG_FILE = "logs/app.log"  
  
## ...
```

Standard names

There are a few cases where certain names are the golden standard in those situations.

`self`

A great example of that is the name of the first argument to instance methods. The first argument of such a method is always `self`.

Therefore, do

```
class Square:  
    def __init__(self, side_length):  
        # ...
```

instead of

```
class Square:  
    def __init__(square, side_length):  
        # ...  
  
class Square:  
    def __init__(a, b):  
        # ...  
  
class Square:  
    def __init__(bananas, side_length):  
        # ...
```

Notice that all three alternatives above (that I claim you should avoid) are actually functional. Here is an example:

```
>>> class Square:  
...     def __init__(a, b):  
...         a.side_length = b  
...  
>>> sq = Square(10)  
>>> sq.side_length  
10
```

However, they look utterly wrong to any (!?) Python programmer worth their salt. Ok, let's not get fundamental here, it's just a matter of respecting the one Python convention that is probably the most widely used.

`cls`

In a similar setting, `cls` is the widely accepted name for class methods.

Class methods are *not* the regular methods you define when you implement a custom class. Those are called instance methods. Class methods are instance methods decorated with `@classmethod`.

Why is that? Well, `class` is a keyword so we can't really have that as the parameter name. And for some reason, people started sticking to `cls` instead of something like `class_`. So, nowadays, class methods have their first parameter be `cls`.

A great example of a class method is the method `dict.fromkeys`, which you call to initialise a dictionary in a different way:

```
>>> dict.fromkeys("abc")  
{'a': None, 'b': None, 'c': None}
```

Class methods are often used to implement different ways of building instances of your classes, and that's precisely what is happening in the previous example: we are creating a dictionary (or, in other words, an instance of the class `dict`) in a different way from what is the usual way.

Verbosity

Having discussed some of the most widely spread conventions when dealing with names in Python, I will now share my experience regarding good naming principles.

One thing that is often object of many worries is the length of the name you are picking. Should you pick a long name that contains much information but is a pain to type? Should you pick a short name that is easy to type but a pain to recall what it is for?

Balance is key, always.

Remember that Python imposes a soft limit on the maximum length of a line, so if your variables look like

```
number_of_times_user_tried_to_login_unsuccessfully = 2
```

then you won't be able to do much in each line of code you write.

However, if you go down the other extreme, you end up with names that are one, two, three characters long, and those names won't tell you anything.

There are two metrics that you can use to help you decide how long a name should be:

1. the number of times a name is used; and
2. the distance between definition of the name and its usage.

What do these things mean?

If a name is used very often, because it is a function that you call all the time or maybe a variable that you need to access very frequently, then that name can be shorter, because you are always aware of the existence and purpose of that name.

On the other hand, if a name is *rarely* used, then the name should be longer, because you will need more help to remember what is the purpose of that name.

The reasoning behind bullet point 2. is similar. If you define a name and use it right after it was defined, then its purpose will be fresh in your memory, and you might be able to afford a shorter name.

On the other hand, if you define a name – like a function or a variable – and only use it far down the file, or even in other files or modules, then it is probably best if you use a longer, more descriptive name.

One-char names

At one of the ends of the spectrum are one-character names. One-character names consist of a letter, either uppercase or lowercase, or the underscore.

One-character names should generally be avoided, because they contain little to no information about what they refer to. However, there are a couple of exceptions that make some sense in their given contexts.

Whenever you need to assign to a variable, even though you don't need its value, you could use a sink, and the one-character name `_` is the recommended variable name for a sink. A sink is a variable that is assigned to even though we don't care about its value. An example of a sink shows up in `unpacking`, when you care about the first and last elements of a list, but not about the middle:

```
>>> l = [42, 10, 20, 73]
>>> first, *_, last = l
>>> first
42
>>> last
73
```

In numerical contexts, `n` is also a common name for an integer and `x` for a real number (a float). This might seem silly, but it is recommended that you do not use `n` for values that might not be whole integers. People get so used to these conventions that breaking them might mean that understanding your code will take much longer.

`c` and `z` are also occasionally used for complex numbers, but those are conventions that come from the world of mathematics. In other words, these conventions are more likely to be followed by people that are close to mathematics/mathematicians.

Still along the lines of conventions drawn from mathematics, `i`, `j`, and `k`, in this order, are often used for (integer) indices. For example, you often see the following beginning of a for loop:

```
for i in range(n):
    pass
```

j and k are then used for nested loops, or for when i is already referring to another fixed index:

```
for i in range(...):
    for j in range(...):
        for k in range(...):
            pass
```

Other common one-character names include the first letters of built-in types: d for dictionaries, s for sets or strings, and l for lists.

However, l is a particularly bad one-character name, and so are I (capital i) and O, (capital o), because for many fonts, these characters are easily mistaken by the numbers one and zero.

One-character names should only be used if the surrounding context *clearly* validates what the names refer to. For example, s will be a terrible one-character name if you are dealing with sets and strings in the same place.

Finally, you can try to use other short names to replace the one-character names. For example, idx instead of i makes it clearer that we are talking about an index, and char instead of c makes it clearer that we are talking about a character.

Abbreviations

Abbreviations need to be used sparingly. They might make sense if it is a widely recognise abbreviation... But that, itself, is a dangerous game to play, because you cannot know what abbreviations the readers of your code might know.

Something that might be safer is to use abbreviations that are relative to the domain knowledge of the code. For example, if your code handles a network of logistics drones, at some point it might make sense to use "eta" – which stands for "estimated time of arrival" – for a variable name that holds the *estimated time of arrival* of a drone. But then again, try to reason about whether the readers of your code will be familiar with the domain-specific lingo or not.

While this first guideline is fairly subjective, there is one type of abbreviation that is definitely a terrible idea, and that's non-standard abbreviations. If you can't Google that abbreviation and get its meaning in the first couple of results, then that's not a standard abbreviation, at all.

For example, taking the long variable name from above and abbreviating it is a *bad* idea:

```
## Don't
number_of_times_user_tried_to_login_unsuccessfully = 2

## but this is even worse:
ntutlu = 2
```

This also has the inconvenience that it is not a name that you can *pronounce*, and that makes it harder for you to talk about your code with others.

On the flip side, there is also a very specific situation in which non-standard abbreviations can make sense, and that is in short-lived scopes. A prototypical example arises from iterating over a collection:

```
data_sets = [ds for ds in data_sets if not is_complete(ds)]
```

Notice that there is a container, a list, with a name in the plural: `data_sets`. Then, as we traverse through that container, what do we expect each element to be? Because the container is called `data_sets`, we immediately expect it to contain, well, data sets. Therefore, each element is naturally thought of as a single data set. As an extension to that, the name `ds` – that abbreviates “data set” – is a perfectly acceptable name for the consecutive elements of `data_sets`, as that name only lives inside the list comprehension.

Sentences

Rather than having names like

```
number_of_times_user_tried_to_login_unsuccessfully = 2
```

or

```
def compute_number_of_unsuccessful_login_attempts():
    pass
```

consider shortening those names, and instead include a comment that gives further context, if needed. As you will see, more often than not, you don’t even need the extra comment:

```
## Number of unsuccessful attempts made by the user:
unsuccessful_logins = 2
```

I mean, we are clearly working with a number, so we can just write:

```
## Unsuccessful attempts made by the user:
unsuccessful_logins = 2
```

We also know we are talking about unsuccessful attempts, because that’s in the variable name:

```
## Attempts made by the user:
unsuccessful_logins = 2
```

We can, either stop at this point, or remove the comment altogether if the user is the only entity that could have made login attempts.

For functions, include the extra context in the docstring. This ensures that that helpful context is shown to you/users when using calling your function. Nowadays, IDEs will show the docstring of the functions we are calling in our code.

Picking a name

When picking the actual name for whatever it is that you need to name, remember to:

- pick a name that is consistent in style/wording with your surroundings;
- use always the same vocabulary and spelling;

```

## Bad:
first_color = "red"
last_colour = "blue"

## Good:
first_colour = "red"
last_colour = "blue"
## (or use `color` in both)

## Bad:
item.has_promotion = True
item.discount_percentage = 30

## Good:
item.has_discount = True      # or item.is_discounted, for example.
item.discount_percentage = 30

```

- use a name that reflects what we are dealing with, instead of a generic name that reflects the type of the data.

```

## Bad:
num = 18
string = "Hello, there."

## Good:
legal_age = 18
greeting = "Hello, there."

```

For variables, you can also consider a name that reflects a major invariant property of the entity you are working with. “Invariant” means that it doesn’t change. This is important, otherwise you will have a name that indicates something when the value itself is something else. I’ll show you an example of this by the end of the Pydon’t.

Naming functions and methods

A guideline that is specific for functions/methods is that they should be named with verbs. This reflects the action that the function/method will do when called and makes it clear that things *happen* when the function/method is called.

Naming variables

Similarly, variables are better named with nouns, when they refer to entities.

For Boolean variables (also known as predicates), adjectives might be a good choice as well, in the sense that the value of the Boolean reflects the presence or absence of that adjective.

For example:

```
if painting.colourful:  
    print("I like colours!")
```

Notice that the noun “painting” leads us into assuming we are talking about some object that may be an instance of some class `Painting` that was created earlier, and `painting.colourful` leads us into assuming that that’s a Boolean value indicating whether or not the painting is colourful.

Notice how redundant the paragraph above was. When the names used in the code are good, English explanations become easily too verbose. That’s a good thing, it means that the code speaks for itself.

Having variables be names/adjectives *and* functions be verbs improves readability when you call functions on your own variables, because the function (the verb) will be acting on the variables (the nouns). That’s exactly how English and most natural languages work, and thus we are writing our code in a way that is similar to natural languages.

Context is key

This has been mentioned heavily throughout this Pydon’t, but I want it to be highlighted even more, so there’s a heading devoted to just this: context is key.

Remember that the context in which you are writing your code will impact a lot the names that you pick.

Contexts that matter include the domain(s) that your code belongs to (are you writing software to handle bank transactions, to manage a network of logistics drones, or are you implementing a game?), the specific module and functions you are in, and whether or not you are inside a statement like a loop, a list comprehension, or a `try: ... except: ...` block.

As an example of how the domain you are working in can drastically affect your naming, consider the following example, drawn from my experience with mathematics. Sometimes it is useful to be able to add polynomials, and therefore you might want to implement that function:

```
def poly_addition(poly1, poly2):  
    pass
```

However, if you are in the context of a module that specialises in working with polynomials, then that function’s signature could probably be boiled down to:

```
def add(p, q):  
    pass
```

(`p` and `q` are common names for polynomials in mathematics.)

See? Context is key.

Practical example

In my [Pydon’ts talk](#) and in the [Pydon’t about refactoring](#), I showed a piece of code written by a beginner and then proceeded to refactor it little by little. One of the steps was renaming things.

Here is said piece of code:

```

def myfunc(a):
    empty = []
    for i in range(len(a)):
        if i % 2 == 0:
            empty.append(a[i].upper())
        else:
            empty.append(a[i].lower())

    return "".join(empty)

```

This is what the code does:

```

>>> myfunc("abcdef")
'AbCdEf'
>>> myfunc("ABCDEF")
'AbCdEf'
>>> myfunc("A CDEF")
'A CdEf'

```

It alternates the casing of the characters of the argument.

As an exercise for you, try improving the names in the piece of code above before you keep reading.

Ok, have you had a go at improving the names?

Here are all of the names that show up in the function above:

- `myfunc` is the function name;
- `a` is the parameter of the function;
- `empty` is a list that grows with the new characters of the result; and
- `i` is the index into the argument string.

Here is a suggestion of improvement:

```

def alternate_casing(text):
    letters = []
    for idx in range(len(text)):
        if idx % 2 == 0:
            letters.append(text[idx].upper())
        else:
            letters.append(text[idx].lower())

    return "".join(letters)

```

`myfunc` is now `alternate_casing`

`myfunc` was a generic name for a function and it gave you no information whatsoever as to what the function did.

Instead, we can pick a name like `alternate_casing` that tells you that this function will alternate the casing of its argument.

Notice that we did not go for something like

```
def alternate_casing_starting_with_uppercase(...):  
    pass
```

That implementation detail is better suited for the docstring of the function.

a is now text

Our function accepts a generic string as input. There is nothing particularly special or interesting about this string, so perfectly good names include `text` and `string`.

I opted for `text` because it gives off the feeling that we will be working with human-readable strings.

empty is now letters

The variable name `empty` here is a great counter-example of one of the guidelines presented before. This Pydon't suggested that you give variable names according to important *invariant* properties of your objects.

Well, `[]` is clearly an empty list, and so the author decided to name this variable as `empty`, which actually looks sensible. However, three lines down, that list is appended to consecutively, so it stops being empty rather quickly. Therefore, the list being empty is not an *invariant* property and also not a good name.

The name `letters` is more appropriate than `empty`, but you might argue that it might be misleading – after all, we put all characters in there, not just the letters.

What name would you use, then?

i is now idx

`i` is a very typical name for an index and I don't think there was anything wrong with it. I have a personal preference for the 110% explicit `idx` for an index, and that is why I went with it.

Conclusion

Having gone through this Pydon't, you might be thinking that most of the guidelines in here are fairly subjective, and you are right!

I know it can be frustrating to not have objective rules to pick names for your variables, functions, etc... But you know what they say! Naming things is the hardest problem you have to solve in programming.

Don't fret, with experience you will become better and better at using good names in your code, and remember, Python reads almost like English, so the names you pick should help with that.

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"While naming can be hard, there are guidelines to help you make the best decisions possible."

This Pydon't showed you that:

- consistency with existing code is paramount in naming things;

- pick up all the hints you can from the context to determine the best names;
- [PEP 8](#) suggests some naming conventions for your code,
 - most notably `snake_case` for almost everything; and
 - `PascalCase` for classes and exceptions.
- `CAPS_WITH_UNDERSCORE` is a widely accepted convention for global constants;
- variables should be named with nouns, Boolean variables sometimes with adjectives, and functions with verbs;

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [don't forget to subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- PEP 8 – Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008> [last accessed 28-07-2021];
- Stack Overflow, “What’s an example use case for a Python classmethod?”, <https://stackoverflow.com/q/5738470/2828287> [last accessed 28-07-2021];
- testdriven.io, “Clean Code”, <https://testdriven.io/blog/clean-code-python/#naming-conventions> [last accessed 10-08-2021]

Chaining comparison operators

```
>>> a = 3
>>> l = [3, 5]
>>> if a in l == True:
...     print("Yeah :D")
... else:
...     print("Hun!?")
...
Hun!?
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/chaining-comparison-operators>.)

Introduction

In this Pydon't we will go over the chaining of comparison operators:

- how they work;
- useful usages; and
- weird cases to avoid.

Chaining of comparison operators

One of the things I enjoy about Python is that some of its features make so much sense that you don't even notice that you are using a feature until someone points out that such code wouldn't work in other languages. One such example is comparison chaining! Look at this snippet of code and tell me if it doesn't look natural:

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> if a < b < c:
...     print("Increasing seq.")
...
Increasing seq.
```

When Python sees two comparison operators in a row, like in `a < b < c`, it behaves as if you had written something like `a < b` and `b < c`, except that `b` only gets evaluated once (which is relevant if `b` is an expression like a function call).

In my opinion, this features makes a *lot* of sense and does not look surprising. Instead, now I feel kind of sad that most languages do not have support for this behaviour.

Another example usage is for when you want to make sure that three values are all the same:

```
>>> a = b = 1
>>> c = 2
>>> if a == b == c:
...     print("all same")
... else:
...     print("some are diff")
...
some are diff
>>> c = 1
>>> if a == b == c:
...     print("all same")
... else:
...     print("some are diff")
...
all same
```

Did you know that you can actually chain an arbitrary number of comparison operators? For example, `a == b == c == d == e` checks if all five variables are the same, while `a < b < c < d < e` checks if you have a strictly increasing sequence.

Pitfalls

Even though this feature looks very sensible, there are a couple of pitfalls you have to look out for.

Non-transitive operators

We saw above that we can use `a == b == c` to check if `a`, `b` and `c` are all the same. How would you check if they are all different?

If you thought about `a != b != c`, then you just fell into the first pitfall!

Look at this code:

```
>>> a = c = 1
>>> b = 2
>>> if a != b != c:
...     print("a, b, and c all different:", a, b, c)
a, b, and c all different: 1 2 1
```

The problem here is that `a != b != c` is `a != b` and `b != c`, which checks that `b` is different from `a` and from `c`, but says nothing about how `a` and `c` relate.

From the mathematical point of view, `!=` isn't transitive, i.e., knowing how `a` relates to `b` and knowing how `b` relates to `c` *doesn't* tell you how `a` relates to `c`. As for a transitive example, you can take the `==` equality operator. If `a == b` and `b == c` then it is also true that `a == c`.

Non-constant expressions or side-effects

Recall that in a chaining of comparisons, like `a < b < c`, the expression `b` in the middle is only evaluated *once*, whereas if you were to write the expanded expression, `a < b` and `b < c`, then `b` would get evaluated twice.

If `b` contains an expression with side-effects or if it is something that isn't constant, then the two expressions are not equivalent and you should think about what you are doing.

This snippet shows the difference in number of evaluations of the expression in the middle:

```
>>> def f():
...     print("hey")
...     return 3
...
>>> if 1 < f() < 5:
...     print("done")
...
hey
```

```

done
>>> if 1 < f() and f() < 5:
...     print("done")
...
hey
hey
done

```

This snippet shows that an expression like `1 < f() < 0` can actually evaluate to True when it is unfolded:

```

>>> l = [-2, 2]
>>> def f():
...     global l
...     l = l[::-1]
...     return l[0]
>>> if 1 < f() and f() < 0:
...     print("ehh")
...
ehh

```

The syntax `l[::-1]` is a “slice” that reverses a list. I’ll be writing about list slicing soon, so [stay tuned](#) for that!

Of course that `1 < f() < 0` should never be True, so this just shows that the chained comparison and the unfolded one aren’t always equivalent.

Ugly chains

This feature looks really natural, but some particular cases aren’t so great. This is a fairly subjective matter, but I personally don’t love chains where the operators aren’t “aligned”, so chains like

- `a == b == c`
- `a < b <= c`
- `a <= b < c`

look really good, but in my opinion chains like

- `a < b > c`
- `a <= b > c`
- `a < b >= c`

don’t look that good. One can argue, for example, that `a < b > c` reads nicely as “check if `b` is larger than both `a` and `c`”, but you could also write `max(a, c) < b` or `b > max(a, c)`.

Now there’s some other chains that are just confusing:

- `a < b` is True
- `a == b` in `l`
- `a in l` is True

In Python, `is`, `is not`, `in`, and `not in` are comparison operators, so you can also chain them with the other operators. This creates weird situations like

```
>>> a = 3
>>> l = [3, 5]
>>> if a in l == True:
...     print("Yeah :D")
... else:
...     print("Hun!?")
...
Hun!?
```

Here is a breakdown of what is happening in the previous example:

- `a in l == True` is equivalent to `a in l and l == True`;
- `a in l` is *True*, *but*
- `l == True` is *False*, so
- `a in l == True` unfolds to `True and False` which is *False*.

The one who wrote `a in l == True` probably meant `(a in l) == True`, but that is also the same as `a in l`.

Examples in code

Inequality chain

Having a simple utility function that ensures that a given value is between two bounds becomes really simple, e.g.

```
def ensure_within(value, bounds):
    return bounds[0] <= value <= bounds[1]
```

or if you want to be a little bit more explicit, while also ensuring `bounds` is a vector with *exactly* two items, you can also write

```
def ensure_within(value, bounds):
    m, M = bounds
    return m <= value <= M
```

Equality chain

Straight from Python's `enum` module, we can find a helper function (that is not exposed to the user), that reads as follows:

```
def _is_dunder(name):
    """Returns True if a __dunder__ name, False otherwise."""
    return (len(name) > 4 and
            name[:2] == name[-2:] == '__' and
```

```
name[2] != '__' and  
name[-3] != '__')
```

This function checks if a string is from a “dunder” method or not.

“Dunder” comes from “double underscore” and just refers to some Python methods that some classes have, and that allow them to interact nicely with many of Python’s built-in features. These methods are called “dunder” because their names start and end with __. You have seen the __str__ and __repr__ dunder methods in the “[str and repr](#)” Pydon’t and the __bool__ dunder method in the “[Truthy, falsy, and bool](#)” Pydon’t. I will be writing about dunder methods in general in a later Pydon’t, so feel free to [subscribe](#) to stay tuned.

The first thing the code does is check if the beginning and the ending of the string are the same and equal to “__”:

```
>>> _is_dunder("__str__")  
True  
>>> _is_dunder("__bool__")  
True  
>>> _is_dunder("__dnd__")  
False  
>>> _is_dunder("_____underscores__")  
False
```

Conclusion

Here’s the main takeaway of this article, for you, on a silver platter:

“Chaining comparison operators feels so natural, you don’t even notice it is a feature. However, some chains might throw you off if you overlook them.”

This Pydon’t showed you that:

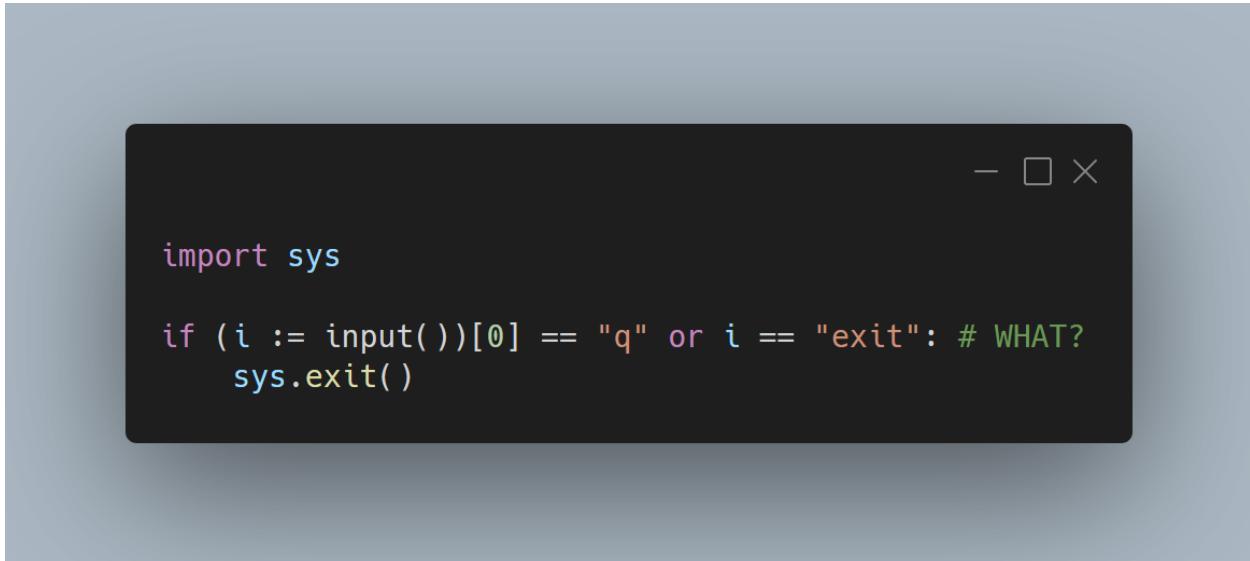
- you can chain comparisons, and do so arbitrarily many times;
- chains with expressions that have side-effects or with non-deterministic outputs are not equivalent to the extended version; and
- some chains using is or in can look really misleading.

References

- Python 3 Documentation, The Python Language Reference <https://docs.python.org/3/reference/expressions.html#comparisons>;
- Python 3 Documentation, The Python Standard Library, enum, <https://docs.python.org/3/library/enum.html>;
- Reddit, comment on “If they did make a python 4, what changes from python 3 would you like to see?”, https://www.reddit.com/r/Python/comments/ltaf3y/if_they_did_make_a_python_4_what_changes_from/gowuau5?utm_source=share&utm_medium=web2x&context=3.

Online references last consulted on the 1st of March of 2021.

Assignment expressions and the walrus operator :=



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/assignment-expressions-and-the-walrus-operator>.)

Walrus operator and assignment expressions

The walrus operator is written as `:=` (a colon and an equality sign) and was first introduced in Python 3.8. The walrus operator is used in *assignment expressions*, which means assignments can now be used as a part of an expression, whereas before Python 3.8 the assignments were only possible as statements.

An assignment statement assigns a value to a variable name, and that is it. With an assignment expression, that value can then be immediately reused. Here is an example of the difference:

```
>>> a = 3
>>> print(a)
```

```
3
>>> print(b = 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'b' is an invalid keyword argument for print()
>>> print(b := 3)
3
>>> b
3
```

As shown in [PEP 572](#), a good usage of assignment expressions can help write better code: code that is clearer and/or runs faster.

Assignment expressions should be avoided when they make the code too convoluted, even if it saves you a couple of lines of code. You don't want to [disrespect the Zen of Python](#), and the Zen of Python recommends writing readable code.

The snippet of code below features what is, in my opinion, a fairly unreadable usage of an assignment expression:

```
import sys

if (i := input()[0] == "q" or i == "exit":
    sys.exit()
```

I think a better alternative would have been

```
import sys

i = input()
if i[0] == "q" or i == "exit":
    sys.exit()
```

The second alternative (without `:=`) is much easier to read than the first one, even though using `:=` saved one line of code.

However, good uses of assignment expressions can

- make your code faster,
- make it more readable/expressive, and
- make your code shorter.

Examples in code

Here are a couple of examples of good usages of assignment expressions.

Controlling a while loop with initialisation

Consider the following `while` loop:

```
inp = input()
while inp:
    eval(inp)
    inp = input()
```

This code can be used to create a very basic Python repl inside your Python program, and the REPL stops once you give it an empty input, but notice that it features some repetition. First, you have to initialise `inp`, because you want to use it in your `while` condition, but then you also have to update `inp` inside your `while` loop.

With an assignment expression, the above can be rewritten as:

```
while inp := input(" >> "):
    eval(inp)
```

This not only makes the code shorter, but it makes it more expressive, by making it blatantly clear that it is the user input provided by `input()` that is controlling the `while` loop.

Reducing visual noise

Say you want to count the number of trailing zeroes in an integer. An easy way to do so would be to convert the integer to a string, find its length, and then subtract the length of that same string with all its trailing zeroes removed. You could write it like so:

```
def trailing_zeroes(n):
    s = str(n)
    return len(s) - len(s.rstrip("0"))
```

However, for a function so simple and so short, it kind of looks sad to have such a short `s = str(n)` line represent half of the body of the `trailing_zeroes` function. With an assignment expression, you can rewrite the above as

```
def trailing_zeroes(n):
    return len(s := str(n)) - len(s.rstrip("0"))
```

The function above can be read as “*return the length of the string `s` you get from `n`, minus the length of `s` without trailing zeroes*”, so the assignment expression doesn’t hurt the readability of the function and, in my opinion, improves it. Feel free to disagree, of course, as this is not an objective matter.

Reuse computations in list comprehensions

Suppose you are writing a list comprehension with an `if` filter, but the filter test in the comprehension uses a value that you also want to use in the list itself. For example, you have a list of integers, and want to keep the factorials of the numbers for which the factorial has more than 50 trailing zeroes.

You could do this like so:

```
from math import factorial as fact

l = [3, 17, 89, 15, 58, 193]
facts = [fact(num) for num in l if trailing_zeroes(fact(num)) > 50]
```

The problem is that the code above computes the factorial for each number twice, and if the numbers get big, this can become really slow. Using assignment expressions, this could become

```
from math import factorial as fact

l = [3, 17, 89, 15, 58, 193]
facts = [f for num in l if trailing_zeroes(f := fact(num)) > 50]
```

The use of `:=` allows to reuse the expensive computation of the factorial of `num`.

Two other similar alternatives, without assignment expressions, would be

```
from math import factorial as fact

l = [3, 17, 89, 15, 58, 193]
## Alternative 1
facts = [fact(num) for num in l]
facts = [num for num in facts if trailing_zeroes(num) > 50]
## Alternative 2
facts = [num for num in map(fact, l) if trailing_zeroes(num) > 50]
```

Notice that the second one can be more memory efficient if your list `l` is large: the first alternative first computes the *whole* list of factorials, whereas the second alternative only computes the factorials as they are needed. (I'll write more about this in a later Pydon't, [subscribe](#) so you don't miss it!)

Flattening related logic

Imagine you reach a point in your code where you need to pick an operation to do to your data, and you have a series of things you would like to try. But you also would like to stick to the first one that works. As a very simple example, suppose we have a string that may contain an email or a phone number, and you would like to extract the email and, in case you find none, you look for the phone number. (For the sake of simplicity, let's assume phone numbers are 9 digits long and let's also consider simple `.com` emails with only letters.)

You could do something like:

```
import re

string = input("Your contact info: >> ")
email = re.search(r"\b(\w+@\w+\.\com)\b", string)
if email:
    print(f"Your email is {email.group(1)}")
else:
    phone = re.search(r"\d{9}", string)
    if phone:
        print(f"Your phone is {phone.group(0)}")
    else:
        print("No info found...")
```

Notice the code above is nested, but the logic is flat: we look for successive things and stop as soon as we find something. With assignment expressions this could be rewritten as:

```

import re

string = input("Your contact info: >> ")
if email := re.search(r"\b(\w+@\w+\.\com)\b", string):
    print(f"Your email is {email.group(1)}")
elif phone := re.search(r"\d{9}", string):
    print(f"Your phone is {phone.group(0)}")
else:
    print("No info found...")

```

Conclusion

Assignment expressions allow the binding of a name to a part of an expression, which can be used to great benefit in clarifying the flow of some programs or saving time on expensive computations, for example. Bad usages of assignment expressions, however, can make code very unreadable and is therefore crucial to judge whether or not an assignment expression is a good fit for a particular task.

References

- Python 3 Documentation, What's New in Python, What's new in Python 3.8 - Assignment expressions, <https://docs.python.org/3/whatsnew/3.8.html#assignment-expressions>.
- PEP 572 – Assignment Expressions, <https://www.python.org/dev/peps/pep-0572>.
- Real Python, “Assignment Expressions: The Walrus Operator”, <https://realpython.com/lessons/assignment-expressions/>.

Online references consulted on the 26th of January of 2021.

Truthy, Falsy, and bool

```
● ● ●

l = [3, 1, 6]
if l:                      # if len(l) > 0:
    print(l)

n = 0
if n:                      # if n != 0:
    l.append(n)

msg = "Hello, world!"
if msg:                     # if len(msg) > 0:
    print(msg)

d = {"a": 2, "b": 6}
if d:                       # if len(d) > 0:
    print(d.keys())
```

“Truthy” and “Falsy”

Quoting the Python documentation,

“Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below [or, and and not].”

What does that mean? It means that we can use any Python object we want whenever a boolean value is expected. Boolean values (True and False) are used in conditions, which pop up in if statements and while statements, as well as in expressions that make use of the Boolean operators or, and and not.

As a very basic example, consider this Python session:

```
>>> if True:  
...     print("Hello, World!")  
...  
Hello, World!  
>>> if False:  
...     print("Go away!")  
...  
>>>
```

This piece of code should not surprise you, as it is very standard Python code: there are a couple of if statements that make use of explicit Boolean values. The next step is using an expression that *evaluates* to a Boolean value:

```
>>> 5 > 3  
True  
>>> if 5 > 3:  
...     print("Hello, World!")  
...  
Hello, World!
```

The *next* step is using an object that is **not** a Boolean value, which is what this blog post is all about:

```
>>> l = [1, 2, 3]  
>>> if l:  
...     print(l)  
...  
[1, 2, 3]
```

This is the part that could be surprising if you have never encountered it. The reason this if statement is getting executed is because the list [1, 2, 3] is *Truthy*, that is, the list [1, 2, 3] can be interpreted as True in a Boolean context. How can you know if an object is “Truthy” or “Falsy”? The simplest way is to use the built-in bool function that converts any Python object to a Boolean:

```
>>> bool(l)  
True
```

The way this works is really simple! There are a couple of rules that specify how this works, but these simple rules can even be simplified further with a simpler heuristic:

“A value of a given type if Falsy when it is “empty” or “without any useful value”.”

Examples of built-in types and their Falsy values include the empty list, empty set, empty tuple, empty dictionary, the number 0, None and the empty string. For example:

```
>>> bool([])
False
>>> bool("")
False
```

Of course, “without any useful value” definitely depends on what you intend to do with the value you have, so I should really specify the objective rules:

- By default, a value is Truthy (that is, is interpreted as True).
- An object has a Falsy value (that is, is interpreted as False) if calling `len` on it returns 0.

Notice that the previous rule tells us that, in general, types that are containers or sequences (types of objects for which it generally makes sense to use `len` on), are considered Falsy when they are empty, i.e., when they have length equal to zero. But there is one more case that gives a Falsy value:

The `__bool__` dunder method

- An object has a Falsy value (that is, is interpreted as False) if it defines a `__bool__` method that returns `False`.

`__bool__` is a *dunder* method (dunder stands for double underscore) that you can use to tell your objects if they are Truthy or Falsy in Boolean contexts, by implementing it in your own classes. (You have seen [other dunder methods](#) already.)

If you are not acquainted with Python’s dunder methods, you may want to [subscribe](#) to the Pydon’t newsletter, I will write more about them later. Until then, you may want to have a look at the Python 3 Docs and what they say about the [data model](#).

Here is a simple example showing an object that is always taken to be Truthy:

```
>>> class A:
...     pass
...
>>> a = A()
>>> if a:
...     print("Hello, World!")
...
Hello, World!
```

On the opposite end, we can consider a class whose objects will always be taken to be Falsy:

```
>>> class A:
...     def __bool__(self):
...         return False
...
>>> a = A()
```

```
>>> if a:  
...     print("Go away!")  
...
```

In general, your use case may be such that your object sometimes is Truthy and sometimes is Falsy.

Finally, it is very important to state the order in which the rules apply!

When given an arbitrary Python object that needs to be tested for a truth value, Python first tries to call `bool` on it, in an attempt to use its `__bool__` dunder method. If the object does not implement a `__bool__` method, then Python tries to call `len` on it. Finally, if that also fails, Python defaults to giving a Truthy value to the object.

Remarks

Now a couple of remarks about the functioning of Truthy and Falsy values.

A note about containers with falsy objects

We said that things like the empty list, zero, and the empty dictionary are Falsy. However, things like a list that only contains zeroes or a dictionary composed of zeroes and empty lists are not Falsy, because the containers themselves are no longer empty:

```
>>> bool([])  
False  
>>> bool({})  
False  
>>> bool(0)  
False  
>>> bool([0, 0, 0]) # A list with zeroes is not an empty list.  
True  
>>> bool({0: []}) # A dict with a 0 key is not an empty dict.  
True
```

A note about checking for None

As mentioned above, `None` is Falsy:

```
>>> bool(None)  
False  
>>> if None:  
...     print("Go away!")  
...
```

This seems about right, as `None` is the go-to value to be returned by a function when the function does nothing.

Imagine someone implemented the following function to return the integer square root of a number, returning `None` for negative inputs (because negative numbers do not have a square root in the usual sense):

```
import math
def int_square_root(n):
    if n < 0:
        return None
    return math.floor(math.sqrt(n))
```

When you use the function above you know it returns `None` if the computation fails, so now you might be tempted to use your newfound knowledge about the Falsy value of `None`, and you might write something like the following, to check if the computation succeeded:

```
n = int(input("Compute the integer square root of what? >> "))
int_sqrt = int_square_root(n)
if not int_sqrt:
    print("Negative numbers do not have an integer square root.")
```

Now, what happens if `n` is 0 or 0.5?

```
>>> n = 0.5
>>> int_sqrt = int_square_root(n)
>>> if not int_sqrt:
...     print("Negative numbers do not have an integer square root.")
...
Negative numbers do not have an integer square root
```

Which is clearly wrong, because `n = 0.5` is certainly positive. Let us inspect `int_sqrt`:

```
>>> int_sqrt
0
```

The problem is that `int_square_root` returned a meaningful value (that is, it did not return `None`) but that meaningful value is still Falsy. When you want to check if a function returned `None` or not, do not rely on the Truthy/Falsy value of the return value. Instead, check explicitly if the return value is `None` or not:

```
## Use                                # Avoid
if returned is None:                 # if not returned:
    # ...                           #   #
if returned is not None:            # if returned:
    # ...                           #   # ...
```

This recommendation is to avoid problems like the one outlined above.

Examples in code

Now I will show you some examples of places where using the Truthy and Falsy values of Python objects allows you to write more Pythonic code.

2D point

Let us implement a simple class to represent points in a 2D plane, which could be an image, a plot or something else. Retrieving what we already had [in the article about `__str__` and `__repr__`](#), we can add a

`__bool__` method so that the origin (the point `Point2D(0, 0)`) is Falsy and all other points are Truthy:

```

# From https://mathspp.com/blog/pydonts/pydont-confuse-str-and-repr
class Point2D:
    """A class to represent points in a 2D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        """Provide a good-looking representation of the object."""
        return f"({self.x}, {self.y})"

    def __repr__(self):
        """Provide an unambiguous way of rebuilding this object."""
        return f"Point2D({repr(self.x)}, {repr(self.y)})"

    def __bool__(self):
        """The origin is Falsy and all other points are Truthy."""
        return self.x or self.y

print(bool(Point2D(0, 1))) # True
print(bool(Point2D(0, 0))) # False
print(bool(Point2D(1, 0))) # True
print(bool(Point2D(4, 2))) # True

```

Notice how we defined the Truthy/Falsy value of the `Point2D` in terms of the Truthy/Falsy values of its components! We want the `Point2D` to be Falsy when `self.x` is 0 and `self.y` is also 0, which means a `Point2D` is Truthy if any of `self.x` or `self.y` are Truthy (that is, different from 0)!

Handling error codes or error messages

It is quite common for functions to return “error codes”: integers that encode specific things that did not go quite right, or for such functions to return error messages as strings when things don’t go right. These error codes are usually such that returning 0 means everything went ok, while different other integers can mean all sorts of problems.

If you are calling such a function, you can use the Truthy value of strings and/or integers to check if something went wrong, and to handle it accordingly.

As a generic example, this is the pattern we are looking for:

```

return_value, error_code = some_nice_function()
if error_code:
    # Something went wrong, act accordingly.

## Alternatively, something like:

```

```
return_value, error_msg = some_other_nice_function()
if error_msg:
    print(error_msg)
    # Something went wrong, act accordingly.
```

Processing data

It is also very common to use Truthy and Falsy values to measure if there is still data to be processed.

For example, when I talked about the walrus operator `:=`, we saw a `while` loop vaguely similar to this one:

```
input_lines = []
while (s := input()):
    input_lines.append(s)
## No more lines to read.
print(len(input_lines))
```

This `while` loop essentially reads input lines *while* there are lines to be read. As soon as the user inputs an empty line "", the loop stops and we print the number of lines we read:

```
>>> input_lines = []
>>> while (s := input()):
...     input_lines.append(s)
...
Line 1
Line 2

>>> print(len(input_lines))
2
```

Another common pattern is when you have a list that contains some data that you have to process, and such that the list itself gets modified as you process the data.

Consider the following example:

```
import pathlib

def print_file_sizes(dir):
    """Print file sizes in a directory, recurse into subdirs."""

    paths_to_process = [dir]
    while paths_to_process:
        path, *paths_to_process = paths_to_process
        path_obj = pathlib.Path(path)
        if path_obj.is_file():
            print(path, path_obj.stat().st_size)
        else:
            paths_to_process += path_obj.glob("*)")
```

This is not necessarily the way to go about doing this, *but* notice the `while` statement, and then the `if: ... else: ...` block that either prints something, or extends the `paths_to_process` list.

Conclusion

- Python's Truthy and Falsy values allow you to rewrite common conditions in a way that is more readable and, therefore, Pythonic.
- You can implement your own Truthy and Falsy values in custom classes by implementing the `__bool__` dunder method.
- You should also be careful when checking if a given variable is `None` or not, and avoid using the Falsy value of `None` in those particular cases.

References

- Python 3 Documentation, The Python Language Reference, Data model, `bool`, https://docs.python.org/3/reference/datamodel.html#object.__bool__.
- Python 3 Documentation, The Python Standard Library, Truth Value Testing, <https://docs.python.org/3/library/stdtypes.html#truth-value-testing>.
- Python 3 Documentation, The Python Standard Library, Built-in Functions, `bool`, <https://docs.python.org/3/library/functions.html#bool>.
- PEP 8 – Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008/>.
- Python 3 Documentation, The Python Standard Library, File and Directory Access, `pathlib`, <https://docs.python.org/3/library/pathlib.html>.
- Stack Overflow, Listing of all files in directory?, <https://stackoverflow.com/a/40216619/2828287>.
- Stack Overflow, How can I check file size in Python?, <https://stackoverflow.com/a/2104107/2828287>.
- freeCodeCamp, Truthy and Falsy Values in Python: A Detailed Introduction, <https://www.freecodecamp.org/news/truthy-and-falsy-values-in-python/>.

Online references last consulted on the 9th of February of 2021.

Deep unpacking



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/deep-unpacking>.)

Introduction

In this Pydon't we will go over deep unpacking: - what it is; - how it works; - how to use it to improve code readability; and - how to use it to help debug your code.

Learning about deep unpacking will be **very** helpful in order to pave the road for [structural matching](#), a feature to be introduced in Python 3.10.

Assignments

Before showing you how deep unpacking works, let's have a quick look at two other nice features about Python's assignments.

Multiple assignment

In Python, multiple assignment is what allows you to write things like

```
>>> x = 3
>>> y = "hey"
>>> x, y = y, x      # Multiple assignment to swap variables.
>>> x
'hey'
>>> y
3
```

or

```
>>> rgb_values = (45, 124, 183)
>>> # Multiple assignment unpacks the tuple.
>>> r, g, b = rgb_values
>>> g
124
```

With multiple assignment you can assign, well, multiple variables at the same time, provided the right-hand side has as many items as the left-hand side expects.

Starred assignment

Starred assignment, that I covered in depth in [this Pydon't](#), allows you to write things like

```
>>> l = [0, 1, 2, 3, 4]
>>> head, *body = l
>>> print(head)
0
>>> print(body)
[1, 2, 3, 4]
>>> *body, tail = l
>>> print(tail)
4
>>> head, *body, tail = l
>>> print(body)
[1, 2, 3]
```

With starred assignment you can tell Python that you are not sure how many items the right-hand side will have, but all of them can be stored in a single place.

Deep unpacking

Deep unpacking, or nested unpacking, is similar to multiple assignment in a sense. Multiple assignment allows you to match the length of an iterable, on the right-hand side of an assignment, and get each element into a variable. In a similar fashion, deep unpacking allows you to match the *shape* of what is on the right-hand side of an assignment; in particular, if there are nested iterables, you can unpack those iterables at once.

For example, using multiple assignment twice in a row, you could do this:

```
>>> colour_info = ("AliceBlue", (240, 248, 255))
>>> name, rgb_values = colour_info
>>> name
'AppleBlue'
>>> r, g, b = rgb_values
>>> g
248
```

But if you already know you want to get to the separate RGB values, you could use deep unpacking:

```
>>> colour_info = ("AliceBlue", (240, 248, 255))
>>> name, (r, g, b) = colour_info
>>> name
'AppleBlue'
>>> g
248
```

Notice how we group the `r`, `g`, and `b` variables with `()` to create a tuple, mimicking the shape of the `colour_info` variable. If we had simply written `name, r, g, b = colour_info` then Python would think we are trying to do multiple assignment, and would expect `colour_info` to have four items to unpack:

```
>>> colour_info = ("AliceBlue", (240, 248, 255))
>>> name, r, g, b = colour_info
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 4, got 2)
```

Our use of parenthesis in `(r, g, b)` tells Python we actually want to go into the nested structure of `colour_info`.

This might be clearer if we actually include the outer set of parenthesis that is usually omitted:

```
>>> colour_info = ("AliceBlue", (240, 248, 255))
>>> (name, (r, g, b)) = colour_info
>>> name
'AppleBlue'
>>> g
248
```

Now if we put the left-hand side of the assignment, `(name, (r, g, b))`, next to the value it is getting, it becomes very clear what values go where:

```
>>> (name, (r, g, b)) = ("AliceBlue", (240, 248, 255))
```

Did you know that in Python 2 you could use deep unpacking in function signatures? For example, this would be valid Python 2 code:

```
def print_some_colour_info(name, (r, g, b)):  
    print name + " has g value of " + str(g)  
  
# Prints 'AliceBlue has g value of 248'  
print_some_colour_info("AliceBlue", (240, 248, 255))
```

This was removed with [PEP 3113](#).

In loops

Deep unpacking can also be used in the implicit assignments of `for` loops, it doesn't have to be in explicit assignments with an equals sign! The examples below will show you that.

Deep unpacking, when used well, can improve the readability of your code – by removing indexing clutter and by making the intent more explicit – and can help you test your code for some errors and bugs.

Nothing better than showing you some code, so you can see for yourself.

Examples in code

Increasing expressiveness

Given the RGB values of a colour, you can apply a basic formula to convert it to greyscale, which weighs the R, G, and B components differently. We could write a function that takes the colour information like we have been using, and then computes its greyscale value:

```
def greyscale(colour_info):  
    return 0.2126*colour_info[1][0] + 0.7152*colour_info[1][1] + \  
        0.0722*colour_info[1][2]
```

(This formula we are using,

```
[ 0.2126R + 0.7152G + 0.0722B ~ , ]
```

is usually the first step of a slightly more involved formula, but it will be good enough for our purposes.)

Now you can use your function:

```
colour = ("AliceBlue", (240, 248, 255))  
print(greyscale(colour)) # prints 246.8046
```

But I think we can all agree that the function definition could surely be improved. The long formula with the additions and multiplications doesn't look very nice. In fact, if we use deep unpacking to extract the `r`, `g`, and `b` values, the formula will be spelled out pretty much like if it were the original mathematical formula I showed:

```
def greyscale(colour_info):  
    name, (r, g, b) = colour_info
```

```

    return 0.2126*r + 0.7152*g + 0.0722*b

colour = ("AliceBlue", (240, 248, 255))
print(greyscale(colour)) # still prints 246.8046

```

Of course, more cunning or suspicious readers might say “That is all well and good, but you could have just defined the function to take the separate `r`, `g`, and `b` values as arguments from the get-go.”. And those people are right! You could have defined your function to be

```

def greyscale(r, g, b):
    return 0.2126*r + 0.7152*g + 0.0722*b

```

But sometimes you are writing code that interacts with other people’s code, and sometimes there are already types and formats of data that are in use, and it is just simpler to adhere to whatever the standards are.

Now imagine that you have a list with some colours and want to compute the greyscales. You can use deep unpacking in a `for` loop (and in a list comprehension too):

```

colours = [
    ("AliceBlue", (240, 248, 255)),
    ("Aquamarine", (127, 255, 212)),
    ("DarkCyan", (0, 139, 139)),
]
greyscales = [
    round(0.2126*r + 0.7152*g + 0.0722*b, 2)
    for name, (r, g, b) in colours
]
print(greyscales) # [246.8, 224.68, 109.45]

```

Catching bugs

I said earlier that deep unpacking can also help you find bugs in your code. It is not hard to believe that the `colours` list of the previous example could have come from some other function, for example a function that scrapes the webpage I have been checking, and creates those tuples with colour information.

Let us pretend for a second that my web scraper isn’t working 100% well yet, and so it ended up producing the following list, where it read the RGB values of two colours into the same one:

```

colours = [
    ("AliceBlue", (240, 248, 255, 127, 255, 212)),
    ("DarkCyan", (0, 139, 139)),
]

```

If we were to apply the original `greyscale` function to `colours[0]`, the function would just work:

```

def greyscale(colour_info):
    return 0.2126*colour_info[1][0] + 0.7152*colour_info[1][1] + \
        0.0722*colour_info[1][2]

colours = [

```

```
        ("AliceBlue", (240, 248, 255, 127, 255, 212)),  
        ("DarkCyan", (0, 139, 139)),  
    ]
```

```
print(greyscale(colours[0])) # 246.8046
```

However, if you were to use the function that uses deep unpacking, then this would happen:

```
def greyscale(colour_info):  
    name, (r, g, b) = colour_info  
    return 0.2126*r + 0.7152*g + 0.0722*b  
  
colours = [  
    ("AliceBlue", (240, 248, 255, 127, 255, 212)),  
    ("DarkCyan", (0, 139, 139)),  
]  
  
print(greyscale(colours[0])) # ValueError: too many values to unpack (expected 3)
```

Deep unpacking expects the shapes to be correct, and so the part `(r, g, b)` tells Python it expects a nested iterable with three elements, but all of a sudden Python tries to give it six numbers and it complains! Hitting this error, you would realise something is weird in your code and eventually you will find the bug!

All in all, deep unpacking (or the chance to use it) isn't something you come across very often, but when you do, it is nice knowing how to use it to your advantage.

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“Use deep unpacking to improve readability and to keep the shape of your variables in check.”

This Python showed you that:

- Python's assignments have plenty of interesting features;
- deep unpacking can prevent cluttering your code with hardcoded indexing;
- deep unpacking improves the readability of your code; and
- some bugs related to iterable shape can be caught if using deep unpacking.

References

- PEP 634 – Structural Pattern Matching: Specification, <https://www.python.org/dev/peps/pep-0634/>;
- PEP 3113 – Removal of Tuple Parameter Unpacking, <https://www.python.org/dev/peps/pep-3113/>;
- Multiple assignment and tuple unpacking improve Python code readability, https://treyhunner.com/2018/03/tuple-unpacking-improves-python-code-readability/#Using_a_list-like_syntax;
- Unpacking Nested Data Structures in Python, <https://dbader.org/blog/python-nested-unpacking>;
- W3Schools, HTML Color Names, https://www.w3schools.com/colors/colors_names.asp;

- Wikipedia, Grayscale, Converting color to grayscale, https://en.wikipedia.org/wiki/Grayscale#Converting_color_to_grayscale.

Online references last consulted on the 23rd of February of 2021.

Unpacking with starred assignments



```
l = [1, 2, 3, 4, 5]

# head, tail = l[0], l[1:]
head, *tail = l
print(head)      # 1
print(tail)      # [2, 3, 4, 5]
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/unpacking-with-starred-assignments>.)

Starred Assignment

It is fairly common to have a list or another iterable that you want to split in the first element and then the *rest*. You can do this by using slicing in Python, but the most explicit way is with *starred assignments*.

This feature was introduced in [PEP 3132 – Extended Iterable Unpacking](#) and allows for the following:

```
>>> l = [1, 2, 3, 4, 5]
>>> head, *tail = l
>>> head
1
>>> tail
[2, 3, 4, 5]
```

This starred assignment is done by placing one * to the left of a variable name in a multiple assignment, and by having any iterable on the right of the assignment. All variable names get a single element and the variable name with the “star” (the asterisk *) gets all other elements as a list:

```
>>> string = "Hello!"
>>> *start, last = string
>>> start
['H', 'e', 'l', 'l', 'o']
>>> last
'!'
```

You can have more than two variable names on the left, but **only one** asterisk:

```
>>> a, b, *c, d = range(5) # any iterable works
>>> a
0
>>> b
1
>>> c
[2, 3]
>>> d
4
```

When you use the starred assignment, the starred name might get an empty list,

```
>>> a, *b = [1]
>>> a
1
>>> b
[]
```

and an error is issued if there are not enough items to assign to the names that are not starred:

```
>>> a, *b = []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected at least 1, got 0)
```

Examples in code

Here are a couple of examples in some actual code, to give more context.

reduce from functools

Imagine you wanted to implement a function akin to the `reduce` function from `functools` (you can read its documentation [here](#)).

Here is how an implementation might look like, using slices:

```
def reduce(function, list_):
    """Reduce the elements of the list by the binary function."""

    if not list_:
        raise TypeError("Cannot reduce empty list.")
    value = list_[0]
    list_ = list_[1:]
    while list_:
        value = function(value, list_[0])
        list_ = list_[1:]
    return value
```

And here is an equivalent implementation using starred assignment:

```
def reduce(function, list_):
    """Reduce the elements of the list by the binary function."""

    if not list_:
        raise TypeError("Cannot reduce empty list.")
    value, *list_ = list_
    while list_:
        val, *list_ = list_
        value = function(value, val)
    return value
```

The usage of the starred assignment here makes it abundantly clear that we wish to unpack the list into an item to be used now and the rest to be used later.

Another similar example, but with the starred name in the beginning, follows.

Credit card check digit

The [Luhn Algorithm](#) is used to compute a check digit for things like credit card numbers or bank accounts.

Let's implement a function that verifies if the check digit is correct, according to the Luhn Algorithm, and using starred assignment to separate the check digit from all the other digits:

```
def verify_check_digit(digits):
    """Use the Luhn algorithm to verify the check digit."""
```

```

*digits, check_digit = digits
weight = 2
acc = 0
for digit in reversed(digits):
    value = digit * weight
    acc += (value // 10) + (value % 10)
    weight = 3 - weight # 2 -> 1 and 1 -> 2
return (9 * acc % 10) == check_digit

## Example from Wikipedia.
print(verify_check_digit([7, 9, 9, 2, 7, 3, 9, 8, 7, 1, 3])) # True

```

Maybe it is not obvious to you what the function does just by looking at it, but it should be very clear that the line `*digits, check_digit = digits` splits the list `digits` into the items in the beginning and the final digit.

How would you implement the function above, using slices and indexing? An example could be like so:

```

def verify_check_digit(digits):
    """Use the Luhn algorithm to verify the check digit."""

    weight = 2
    acc = 0
    for digit in reversed(digits[:-1]):
        value = digit * weight
        acc += (value // 10) + (value % 10)
        weight = 3 - weight # 2 -> 1 and 1 -> 2
    return (9 * acc % 10) == digits[-1]

## Example from Wikipedia.
print(verify_check_digit([7, 9, 9, 2, 7, 3, 9, 8, 7, 1, 3])) # True

```

This also works, but looks a bit more confusing. Notice we have two similar indexing operations, but one is actually a slice while the other is a proper indexing.

In the `for` loop we have a `reversed(digits[:-1])` while in the return value we have `... == digits[-1]`. If I am not paying enough attention, I won't notice those are different things. Of course it is *my* fault that I am not paying enough attention, but when I'm writing code, I prefer for my code to be as clear as possible: I don't want the reader to spend too much time reading the code, I prefer them to spend time studying the algorithms.

References

- PEP 3132 – Extended Iterable Unpacking, <https://www.python.org/dev/peps/pep-3132/>
- Python 3.9.1 Documentation, The Python Standard Library, Functional Programming Modules, `functools`, <https://docs.python.org/3/library/functools.html#functools.reduce> [consulted on the 12th of January of 2021].

- Luhn Algorithm, Wikipedia, https://en.wikipedia.org/wiki/Luhn_algorithm.

EAFP and LBYL coding styles



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/eafp-and-lbyl-coding-styles>.)

EAFP and LBYL

“EAFP” is an acronym that stands for “Easier to Ask for Forgiveness than Permission”, a coding practice that is more or less the opposite of the “LBYL”, which stands for “Look Before You Leap”.

LBYL means you first check if a given operation can be made successfully, and then proceed to do it. For example, if you want to ask the user for a number whose default value will be 1, you can use the code

```

print("Type a positive integer (defaults to 1):")
s = input(" >> ")
if s.isnumeric():
    n = int(s)
else:
    n = 1

```

(In the code above, we use the method `str.isnumeric` to check if the string is a valid integer. Try running `print(str.isnumeric.__doc__)` in your Python REPL.)

With EAFP, you first try to perform whatever operation it is you want to do, and then use a `try` block to capture an eventual exception that your operation might throw in case it is not successful. In our example, this means we simply try to convert `s` into an integer and in case a `ValueError` exception is raised, we set the default value:

```

print("Type a positive integer (defaults to 1):")
s = input(" >> ")
try:
    n = int(s)
except ValueError:
    n = 1

```

We use `except ValueError` because a `ValueError` is the exception that is raised if you try to convert to integer a string that doesn't contain an integer:

```

>>> int("345")
345
>>> int("3.4")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.4'
>>> int("asdf")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'asdf'

```

EAFP instead of LBYL?

Writing code that follows the EAFP style can be advantageous in several situations, and I will present them now.

Avoid redundancy

Sometimes, coding with EAFP in mind allows you to avoid redundancy in your code. Imagine you have a dictionary from which you want to extract a value associated with a key, but that key might not exist.

With LBYL, you would do something like:

```

d = {"a": 1, "b": 42}
print("What key do you want to access?")
key = input(" >> ")
if key in d:
    print(d[key])
else:
    print(f"Cannot find key '{key}'")

```

If the key that was entered exists in the dictionary, this code performs two accesses to the dictionary: the first checks if key exists as a key, and the second retrieves its value. This is more or less like you opening a box to see if it contains something and closing it. Then, if the box was not empty, you open it again and remove whatever is inside. Would you do this in real life?

With EAFP, you can open the box and immediately empty it if you find something inside:

```

d = {"a": 1, "b": 42}
print("What key do you want to access?")
key = input(" >> ")
try:
    print(d[key])
except KeyError:
    print(f"Cannot find key '{key}'")

```

Still aligned with the EAFP mindset is a method that you should know about: `dict.get!` This operation I described is so common that dictionaries even come with a method that have a EAFP-like behaviour for when you want to take a value associated with a key, and use a default value if the key is not present:

```

d = {"a": 1, "b": 42}
print("What key do you want to access?")
key = input(" >> ")
print(d.get(key, None))

```

Try running the code above and type in keys that don't exist in your dictionary `d`. Notice that `None` gets printed in those cases.

EAFP can be faster

If failing is expected to happen not very often, then EAFP is faster: you just run a piece of code (your operation) instead of two (the “look” and the “leap”).

As an example, let's go over the code from the example image above, using the `timeit` module to see what option is faster when the input *can* be converted to an integer:

```

>>> import timeit
>>> eafp = """s = "345"
... try:
...     n = int(s)
... except ValueError:
...     n = 0"""

```

```
>>> timeit.timeit(eafp)
0.1687019999999393
```

Here we define `s` as an integer immediately so that the timing does not have to take into account the time it takes for me to type an integer. Also, the `timeit` function is running the code [a bunch](#) of times and I don't want to have to type one million integers in the console.

Now, compare it with the LBYL approach:

```
>>> lbyl = """s = "345"
... if s.isnumeric():
...     n = int(s)
... else:
...     n = 0"""
>>> timeit.timeit(lbyl)
0.30682630000001154
```

The LBYL approach took almost twice the time. If you can make it so that the operation fails very rarely, then you are saving time by using a EAFP approach.

LBYL may still fail

When interacting with the environment, for example with the Internet or with the OS, in between the time it takes for you to do your safety check and then perform the operation, circumstances may change and your operation may no longer be viable.

For example, imagine you have a script that is reading some files. You can only read a file that exists, obviously, so an LBYL approach could entail writing code like

```
import pathlib

print("What file should I read?")
filepath = input(" >> ")
if pathlib.Path(filepath).exists():
    with open(filepath, "r") as f:
        contents = f.read()
    # Do something with the contents.
else:
    print("Woops, the file does not exist!")
```

If your script is in a computer that can be accessed by several users, or if there are other scripts working with the file system, your `if` statement might evaluate to `True` because the file was found, but then an external agent might delete the file and your `with` statement fails, raising an error and breaking your code. If you are writing critical code, this possibility has to be taken into account. Or if the code you're executing after the check takes a long time to run.

If you use an EAFP approach, the code either reads the file or doesn't, but both cases are covered:

```
print("What file should I read?")
filepath = input(" >> ")
```

```

try:
    with open(filepath, "r") as f:
        contents = f.read()
except FileNotFoundError:
    print("Woops, the file does not exist!")
else:
    # Do something with the contents.
    pass

```

The `else` in the `try` block above ensures you only run the code that processes the contents if you are able to read the file. (I'll write a Pydon't about this, don't worry!)

Catch many types of fails

If you are trying to perform a complex operation that might fail in several ways, it might be easier to just enumerate the exceptions that might be raised instead of writing a really, really long `if` statement that performs all the necessary checks in advance.

For example, if you want to call a third party function that might throw several different exceptions, it is fairly simple to write an elegant `try` block that covers all the cases that might arise.

Imagine you have a function that takes a string, representing an integer, and then returns its inverse, but the person who wrote it performs no checks: just assumes the string represents an integer, converts it with `int` and then divides 1 by that integer:

```

def get_inverse(num_str):
    return 1 / int(num_str)

```

You want to use that function in your code after asking for user input, but you notice the user might type something that is not an integer, or the user might type a 0, which then gives you a `ZeroDivisionError`. With an EAFP approach, you write:

```

print("Type an integer:")
s = input(" >> ")
try:
    print(get_inverse(s))
except ValueError:
    print("I asked for an integer!")
except ZeroDivisionError:
    print("0 has no inverse!")

```

How would you do this with LBYL? Maybe

```

print("Type an integer:")
s = input(" >> ")
if s.isnumeric() and s != "0":
    print(get_inverse(s))
elif not s.isnumeric():
    print("I asked for an integer!")

```

```
else:  
    print("0 has no inverse!")
```

But now you are using the function `isnumeric` twice. And `isnumeric` doesn't even work for negative integers. And what if the user types something like " 3"? `isnumeric` fails, but this is still an integer that `int` can convert! Or what if the user types "000"? This still evaluates to 0... I hope you get my point by now.

Conclusion

EAFP code is a very good alternative to LBYL code, even being superior in various alternatives, like the ones I mentioned above. When writing code, try to weigh the different pros and cons of the several approaches you can take, and don't forget to consider writing EAFP code!

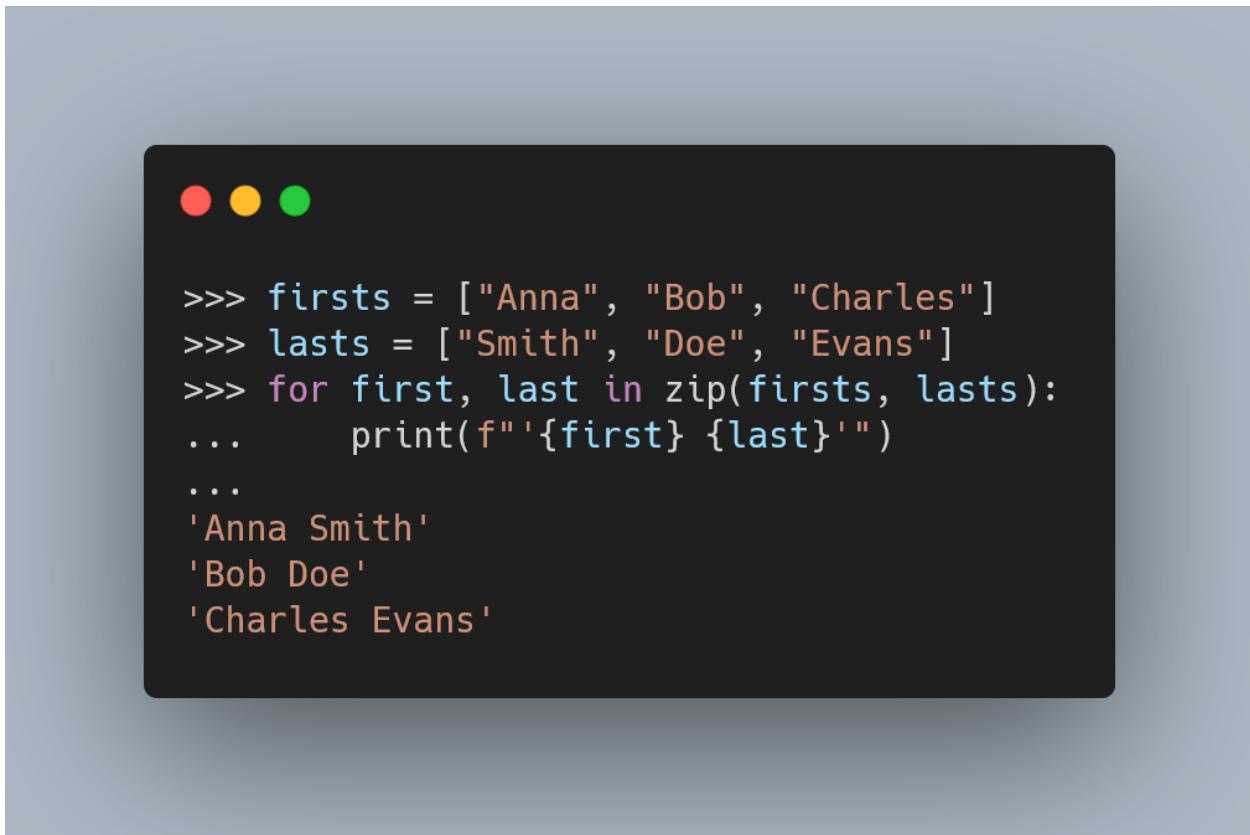
EAFP is not the absolute best way to go in *every single situation*, but EAFP code can be very readable and performant!

References

- PEP 463 – Exception-catching expressions, <https://www.python.org/dev/peps/pep-0463/>
- Python 3 Documentation, The Python Standard Library, Debugging and Profiling, `timeit`, <https://docs.python.org/3/library/timeit.html>
- Python 3 Documentation, The Python Tutorial, Errors and Exceptions, <https://docs.python.org/3/tutorial/errors.html>.
- Microsoft Devblogs, Idiomatic Python: EAFP versus LBYL, <https://devblogs.microsoft.com/python/idiomatic-python-eafp-versus-lbyl/>.
- Stack Overflow, “What is the EAFP principle in Python?”, <https://stackoverflow.com/questions/11360858/what-is-the-eafp-principle-in-python>.
- Stack Overflow, “Ask forgiveness not permission - explain”, <https://stackoverflow.com/questions/11360858/what-is-the-eafp-principle-in-python>.

Online references consulted on the 19th of January of 2021.

Zip up



(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/zip-up>.)

Introduction

One of the things I appreciate most about Python, when compared to other programming languages, is its `for` loops. Python allows you to write very expressive loops, and part of that is because of the built-in `zip` function.

In this article you will

- see what `zip` does;
- get to know a new feature of `zip` that is coming in Python 3.10;
- learn how to use `zip` to create dictionaries; and
- see some nice usage examples of `zip`.

How `zip` works

In a simple `for` loop, you generally have an iterator `it` and you just write something like

```
for elem in it:  
    # Do something with elem  
    print(elem)
```

An “iterator” is something that can be traversed linearly, of which a list is the simplest example. Another very common iterator used in Python’s `for` loops is a `range`:

```
for n in range(10):  
    print(n**2)
```

Sometimes you will have two or more iterators that contain related information, and you need to loop over those iterators to do something with the different bits of information you got.

In the example below, we have a list of first and last names of people and we want to print the full names. The naïve solution would be to use a `range` to traverse all the indices and then index into the lists:

```
>>> firsts = ["Anna", "Bob", "Charles"]  
>>> lasts = ["Smith", "Doe", "Evans"]  
>>> for i in range(len(firsts)):  
...     print(f'{firsts[i]} {lasts[i]}')  
...  
'Anna Smith'  
'Bob Doe'  
'Charles Evans'
```

This does the job, but a `for` loop like this only hints at the fact that you are probably going to access the values in `firsts`, because you wrote

```
range(len(firsts))
```

but turns out you also want to access the items in `lasts`. This is what `zip` is for: you use it to pair up iterables that you wanted to traverse at the same time:

```
>>> firsts = ["Anna", "Bob", "Charles"]  
>>> lasts = ["Smith", "Doe", "Evans"]  
>>> for first, last in zip(firsts, lasts):  
...     print(f'{first} {last}')  
...  
'Anna Smith'
```

```
'Bob Doe'  
'Charles Evans'
```

Notice that you can specify two iterating variables in the `for` loop, in our case `first` and `last`, and each variable will take the successive values of the respective iterator.

This is a special case of an unpacking assignment, because `zip` is actually producing tuples with the names in them:

```
>>> firsts = ["Anna", "Bob", "Charles"]  
>>> lasts = ["Smith", "Doe", "Evans"]  
>>> for z in zip(firsts, lasts):  
...     print(z)  
...  
('Anna', 'Smith')  
('Bob', 'Doe')  
('Charles', 'Evans')
```

What we are doing is taking that tuple and assigning each portion:

```
>>> firsts = ["Anna", "Bob", "Charles"]  
>>> lasts = ["Smith", "Doe", "Evans"]  
>>> for z in zip(firsts, lasts):  
...     first, last = z  
...     print(f'{first} {last}')  
...  
'Anna Smith'  
'Bob Doe'  
'Charles Evans'
```

But instead of the intermediate step, we unpack right in the `for` statement. This unpacking, tied with good naming of variables, allows `for` loops to be read in plain English.

For example, the loop from before was

```
for first, last in zip(firsts, lasts):
```

and that can be read as

“For each `first` and `last` [name] in the lists `firsts` and `lasts`...”

Zip is lazy

One thing to keep in mind is that `zip` doesn't create the tuples immediately. `zip` is lazy, and that means it will only compute the tuples when you ask for them, for example when you iterate over them in a `for` loop (like in the examples above) or when you convert the `zip` object into a list:

```
>>> firsts = ["Anna", "Bob", "Charles"]  
>>> lasts = ["Smith", "Doe", "Evans", "Rivers"]  
>>> z = zip(firsts, lasts)  
>>> z
```

```
<zip object at 0x0000019F56702680>
>>> list(z)
[('Anna', 'Smith'), ('Bob', 'Doe'), ('Charles', 'Evans')]
```

zip being lazy also means that zip by itself isn't *that* similar to a list. For example, you cannot ask what is the length of a zip object:

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

Three is a crowd

We have seen zip with two arguments, but zip can take an arbitrary number of iterators and will produce a tuple of the appropriate size:

```
>>> firsts = ["Anna", "Bob", "Charles"]
>>> middles = ["Z.", "A.", "G."]
>>> lasts = ["Smith", "Doe", "Evans"]
>>> for z in zip(firsts, middles, lasts):
...     print(z)
...
('Anna', 'Z.', 'Smith')
('Bob', 'A.', 'Doe')
('Charles', 'G.', 'Evans')

>>> prefixes = ["Dr.", "Mr.", "Sir"]
>>> for z in zip(prefixes, firsts, middles, lasts):
...     print(z)
...
('Dr.', 'Anna', 'Z.', 'Smith')
('Mr.', 'Bob', 'A.', 'Doe')
('Sir', 'Charles', 'G.', 'Evans')
```

Mismatched lengths

zip will always return a tuple with as many elements as the arguments it received, so what happens if one of the iterators is shorter than the others?

If zip's arguments have unequal lengths, then zip will keep going until it exhausts one of the iterators. As soon as one iterator ends, zip stops producing tuples:

```
>>> firsts = ["Anna", "Bob", "Charles"]
>>> lasts = ["Smith", "Doe", "Evans", "Rivers"]
>>> for z in zip(firsts, lasts):
...     print(z)
```

```
...
('Anna', 'Smith')
('Bob', 'Doe')
('Charles', 'Evans')
```

Starting with Python 3.10, `zip` will be able to receive a keyword argument named `strict` that you can use to tell `zip` to error if the lengths of the iterators do not match:

```
>>> firsts = ["Anna", "Bob", "Charles"]
>>> lasts = ["Smith", "Doe", "Evans", "Rivers"]
>>> # strict=True available in Python >= 3.10
>>> for z in zip(firsts, lasts, strict=True):
...     print(z)
...
('Anna', 'Smith')
('Bob', 'Doe')
('Charles', 'Evans')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is longer than argument 1
```

Notice that `zip` only errors *when* it finds the length mismatch, it doesn't do the check in the beginning: this is because the arguments to `zip` may themselves be lazy iterators.

(Lazy) iterators will be covered in further Pydon'ts, so be sure to [subscribe](#) to the Pydon't newsletter to stay tuned!

In general, `zip` is used with iterators that are *expected* to have the same length. If that is the case – if you expect your iterators to have the same length – then it is a good idea to always set `strict=True`, because that will help you catch bugs in your code.

Create a dictionary with `zip`

You can create dictionaries in Python by feeding key-value pairs to the `dict` function, which means `zip` is a prime way of creating dictionaries when you have all the keys in an iterator and all the values in another iterator:

```
>>> firsts = ["Anna", "Bob", "Charles"]
>>> lasts = ["Smith", "Doe", "Evans"]
>>> dict(zip(firsts, lasts))
{'Anna': 'Smith', 'Bob': 'Doe', 'Charles': 'Evans'}
```

Examples in code

Now you will see some usages of `zip` in actual Python code.

Snake game

A friend of mine is learning Python and he started creating a replica of the game of Snake. There is a certain point in the game where he has a menu and he wants to display thumbnails of the “maps” that can be played on, and he has those images in a list called `lvlpictures`. At the same time, he has the positions of where those images should go in a list called `self.buttons`. In order to display the thumbnails in the correct positions, he has to call a function called `blit`, which expects the image and the position the image should go to.

Here are the loops he wrote before and after knowing about `zip`.

```
# Before:  
for i in range(len(lvlpictures)):  
    self.surface.blit(lvlpictures[i], (self.buttons[i][0]+2, self.buttons[i][1]+2))  
  
# Then he learned about `zip`:  
for pic, btn in zip(lvlpictures, self.buttons):  
    self.surface.blit(pic, (btn[0] + 2, btn[1] + 2))
```

Notice that using `zip` makes your code shorter and it also makes more clear the intent of processing the pictures and the buttons together. Finally, when Python 3.10 is released, he may even add the `strict=True` keyword argument, because he expects `lvlpictures` and `self.buttons` to have the same length.

Matching paths

If you are not aware of it, then you might be interested in knowing that Python has a module named `pathlib` that provides facilities to deal with filesystem paths.

When you create a path, you can then check if it matches a given pattern:

```
>>> from pathlib import PurePath  
>>> PurePath('a/b.py').match('*.py')  
True  
>>> PurePath('/a/b/c.py').match('b/*.py')  
True  
>>> PurePath('/a/b/c.py').match('a/*.py')  
False
```

If you take a look at this `match` function, you find this:

```
class PurePath(object):  
    # ...  
  
    def match(self, path_pattern):  
        """  
        Return True if this path matches the given pattern.  
        """  
        # code omitted for brevity  
        for part, pat in zip(reversed(parts), reversed(pat_parts)):  
            if not fnmatch.fnmatchcase(part, pat):
```

```

        return False
    return True

```

The code omitted does some checks that allow the function to tell right away that there is no match. The for loop that I am showing you makes use of `zip` to pair each part of the path with each part of the pattern, and then we check if those match with `fnmatch.fnmatchcase`.

Try adding a couple of prints here:

```

class PurePath(object):
    # ...

    def match(self, path_pattern):
        """
        Return True if this path matches the given pattern.
        """
        # code omitted for brevity

        print(parts)      # added by hand to check what is going on.
        print(pat_parts) # same here.
        for part, pat in zip(reversed(parts), reversed(pat_parts)):
            if not fnmatch.fnmatchcase(part, pat):
                return False
        return True

```

And then rerun the examples from the documentation:

```

>>> from pathlib import PurePath
>>> PurePath('a/b.py').match('*.py')
['a', 'b.py']    # parts
['*.py']         # pat_parts
True
>>> PurePath('/a/b/c.py').match('b/*.py')
['\\', 'a', 'b', 'c.py']
['b', '*.py']
True
>>> PurePath('/a/b/c.py').match('a/*.py')
['\\', 'a', 'b', 'c.py']
['a', '*.py']
False

```

It should become clearer what `parts` and `pat_parts` actually do, and it should become clearer why we `zip` them up together.

This is a nice example of when using `strict=True` makes no sense, because it may happen that the path and the pattern have a different number of parts, and that is perfectly fine.

Writing a CSV file

The Python Standard Library comes with a module, `csv`, to read and write CSV files. Among other things, it provides you with the classes `DictReader` and `DictWriter` for when you want to use the header of the CSV file to read the data rows like dictionaries or for when you have the data rows as dictionaries.

Here is an example of how you might take several dictionaries and write them as a CSV file:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

The `fieldnames` variable will establish the header of the CSV file and is then used by the `writerow` method to know the order in which the values of the dictionary should be written in the file.

The `writeheader` function is the function that writes the header of the CSV file, and here is what it looks like:

```
class DictWriter:
    # ...

    def writeheader(self):
        header = dict(zip(self.fieldnames, self.fieldnames))
        return self.writerow(header)
```

Basically, what this function is doing is using `zip` to transform the header names into a dictionary where the keys and the values are the same, pretending that the header is just a regular data row:

```
>>> fieldnames = ['first_name', 'last_name']
>>> dict(zip(fieldnames, fieldnames))
{'first_name': 'first_name', 'last_name': 'last_name'}
```

Therefore, the `writeheader` function just needs to create this dictionary and can then defer the actual *writing* to the `writerow` function.

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“`zip` is your friend whenever you need to traverse two or more iterables at the same time.”

This Pydon't showed you that:

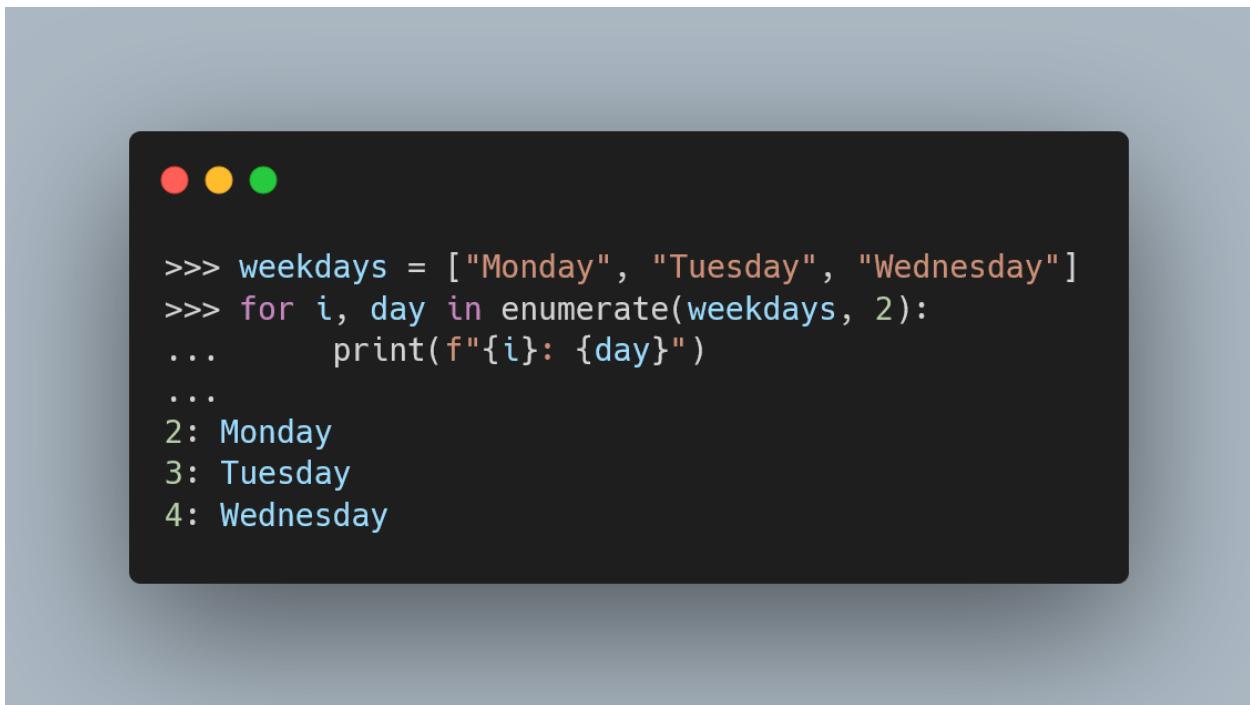
- `zip` can be used to traverse several iterables at the same time;

- `zip` by itself returns a `zip` object which must then be iterated or converted explicitly to a `list` if you want the tuples it produces;
- if the arguments to `zip` have uneven lengths, `zip` will stop as soon as one of the iterators is exhausted;
- starting with Python 3.10, you can use the keyword argument `strict=True` to tell `zip` to error if the arguments to `zip` have different lengths; and
- `zip` can provide for a really simple way to create dictionaries.

References

- Python 3 Documentation, The Python Standard Library, `zip`, docs.python.org/3/library/functions.html#zip [last accessed 30-03-2021];
- Python 3.10 Documentation, The Python Standard Library, `zip`, docs.python.org/3.10/library/functions.html#zip [last accessed 30-03-2021];
- Python 3 Documentation, The Python Standard Library, `csv`, docs.python.org/3/library/csv.html [last accessed 30-03-2021].
- Python 3 Documentation, The Python Standard Library, `pathlib`, docs.python.org/3/library/pathlib.html [last accessed 30-03-2021].

Enumerate me



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/enumerate-me>.)

Introduction

Following up on last week's [Pydon't about zip](#), today we are talking about `enumerate`.

One of the things I appreciate most about Python, when compared to other programming languages, is its `for` loops. Python allows you to write very expressive loops, and some of that expressiveness comes from the built-in `enumerate` function.

In this article you will

- see what `enumerate` does;

- take a look at its underrated optional start argument;
- learn a couple of neat use cases for enumerate;
- see some nice examples of code using enumerate.

How enumerate works

Python newcomers are usually exposed to this type of `for` loop very early on:

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

This leads them to “learning” this anti-pattern of `for` loops to go over, say, a list:

```
>>> words = ["Hey", "there"]
>>> for i in range(len(words)):
...     print(f'{words[i]} has {len(words[i])} letters.')
...
'<Hey> has 3 letters.'
'<there> has 5 letters.'
```

The Pythonic way of writing such a loop is by iterating directly over the list:

```
>>> words = ["Hey", "there"]
>>> for word in words:
...     print(f'{word} has {len(word)} letters.')
...
'<Hey> has 3 letters.'
'<there> has 5 letters.'
```

However, the final step in this indices vs. elements stand-off comes when you need to know the index of each element but also access the element at the same time. The naïve approach would be to loop over the range of the length and then index to get the element:

```
for i in range(len(words)):
    word = words[i]
    # ...
```

or, if you read [my Pydon't on zip](#) and are feeling imaginative, you could also do

```
for i, word in zip(range(len(words)), words):
    # ...
```

but the Pythonic way of doing so is by using the built-in `enumerate`:

```
>>> words = ["Hey", "there"]
>>> for i, word in enumerate(words):
...     print(f'Word #{i}: <{word}> has {len(word)} letters.')
```

```
...
'Word #0: <Hey> has 3 letters.'
'Word #1: <there> has 5 letters.'
```

Optional start argument

The `enumerate` function can also accept an optional argument that specifies the first index it returns. For example, if we are counting words (like in the example above), we might want to start counting from 1:

```
>>> words = ["Hey", "there"]
>>> for i, word in enumerate(words, 1):
...     print(f"Word #{i}: <{word}> has {len(word)} letters.")
...
'Word #1: <Hey> has 3 letters.'
'Word #2: <there> has 5 letters.'
```

This optional argument can come in really handy as it saves you from having to manually offset the index.

By the way, the argument has to be an integer but can be negative:

```
>>> for i, v in enumerate("abc", start=-3243):
...     print(i)
...
-3243
-3242
-3241
```

Can you come up with a *sensible* situation where it would make sense to use `enumerate` with a negative integer as the optional argument? Comment down below if you come up with something nice!

Unpacking when iterating

The `enumerate` function produces a lazy generator, which means the items you iterate over only become available as you need them. This prevents Python from spending *a lot* of memory if you use `enumerate` on a large argument, e.g. a really long list.

This laziness of `enumerate` is something common in Python, for example `range` and `zip` are also lazy. Be sure to [subscribe](#) to the newsletter so you don't miss the Pydon'ts where I cover these concepts.

The items that `enumerate` returns are 2-item tuples, where the first element is the index and the second element is the value:

```
>>> for tup in enumerate("abc"):
...     print(tup)
...
(0, 'a')
(1, 'b')
(2, 'c')
```

What we usually do is unpack that tuple right in the loop statement, with something like

```
for i, letter in enumerate("abc"):  
    # use i and letter for whatever
```

which is roughly equivalent to

```
for tup in enumerate("abc"):  
    i, letter = tup  
    # use i and letter for whatever
```

Deep unpacking

Things can get even more interesting when you use enumerate, for example, on a zip:

```
>>> # Page where each chapter starts and the final page of the book.  
>>> pages = [5, 17, 31, 50]  
>>> for i, (start, end) in enumerate(zip(pages, pages[1:])), start=1:  
...     print(f"{i}: {end-start} pages long.")  
  
'1: 12 pages long.'  
'2: 14 pages long.'  
'3: 19 pages long.'
```

(Here I explicitly named the start= argument in the enumerate so that it was visually easier to separate it from the argument to zip.)

This code snippet takes a list of pages where chapters of a book start and prints the length of each chapter. Notice how enumerate returns tuples with indices and values, but those values are extracted from a zip, which itself returns tuples:

```
>>> # Page where each chapter starts and the final page of the book.  
>>> pages = [5, 17, 31, 50]  
>>> for tup in enumerate(zip(pages, pages[1:])), start=1:  
...     print(tup)  
  
(1, (5, 17))  
(2, (17, 31))  
(3, (31, 50))
```

What we do is use deep unpacking to access all these values directly:

```
>>> # Page where each chapter starts and the final page of the book.  
>>> pages = [5, 17, 31, 50]  
>>> for tup in enumerate(zip(pages, pages[1:])), start=1:  
...     i, (start, end) = tup  
...     print(f"{i}: {end-start} pages long.")  
  
'1: 12 pages long.'
```

```
'2: 14 pages long.'
'3: 19 pages long.'
```

If you don't know what deep unpacking is or how it works, go ahead and take a look at my [Pydon't about unpacking](#).

Examples in code

Now you will see some usages of `enumerate` in *real* Python code.

Vanilla `enumerate`

I took a look at the Python Standard Library and by and large the most common usage of `enumerate` is just a vanilla `enumerate(iter)` to access iterable values and indices at the same time. Let me share a textbook example with you:

The [doctest module](#) allows you to write simple tests for your code inside the docstrings for your functions, classes, etc. The way you write these tests is in the form of an interactive session in the REPL. `doctest` then locates those “interactive sessions” in your docstrings and plays them to see if the actual output of the code matches what your docstring showed.

If you open your Python REPL, you will see that it starts with the prompt `>>>` which has a blank space after the triple `>`. You *cannot* delete that blank space, it is part of the prompt. When parsing a docstring to extract the actual tests, the parser performs a check to see if the prompts have that leading blank space or not, and here is the code that does it:

```
## from Lib\doctest.py in Python 3.9
class DocTestParser:
    # ...

    def _check_prompt_blank(self, lines, indent, name, lineno):
        """
        Given the lines of a source string (including prompts and
        leading indentation), check to make sure that every prompt is
        followed by a space character. If any line is not followed by
        a space character, then raise ValueError.
        """
        for i, line in enumerate(lines):
            if len(line) >= indent+4 and line[indent+3] != ' ':
                raise ValueError('line %r of the docstring for %s '
                                'lacks blank after %s: %r' %
                                (lineno+i+1, name,
                                 line[indent:indent+3], line))
```

Notice how the top `for` loop uses `enumerate` to traverse the lines of the interactive examples. If, inside the loop, we encounter a line that does not have the extra blank space after `>>>` then we raise a `ValueError` where we use `i` to compute the actual line number where the error occurred, which is the `lineno+i+1` bit in the second to last line.

Want to see this in action? Try running this short script:

```
def sum_nats(n):
    """Sums the first n natural numbers.

    >>> sum_nats(1)
    1
    >>> sum_nats(10)
    55
    >>>sum_nats(100)
    5050
    """

    return int(n*(n+1)/2)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Notice how I intentionally wrote the third example without a space between >>> and `sum_nats(100)`. Running this script should throw a `ValueError` at your face, that should go away when you put a blank space there.

Using the optional argument

Line numbers in docstring tests

If you were paying attention, maybe you noticed that the `enumerate` usage of the previous example called for the optional argument of `enumerate`!

Take a look at the code again:

```
## from Lib\doctest.py in Python 3.9
class DocTestParser:
    # ...

    def _check_prompt_blank(self, lines, indent, name, lineno):
        # docstring elided.
        for i, line in enumerate(lines):
            if len(line) >= indent+4 and line[indent+3] != ' ':
                raise ValueError('line %r of the docstring for %s '
                                'lacks blank after %s: %r' %
                                (lineno+i+1, name,
                                 line[indent:indent+3], line))
```

Notice that in the string formatting at the end we compute `lineno+i+1` to raise the error message with the correct line number for the prompt that was faulty... But this is the same as rewriting the loop to use the `start=` argument:

```
class DocTestParser:
```

```

# ...
def _check_prompt_blank(self, lines, indent, name, lineno):
    # docstring elided.
    for line_n, line in enumerate(lines, start=lineno+1):
        if len(line) >= indent+4 and line[indent+3] != ' ':
            raise ValueError('line %r of the docstring for %s '
                             'lacks blank after %s: %r' %
                             (line_n, name,
                              line[indent:indent+3], line))

```

Counting days of the week

Definitely not as frequent as the plain `enumerate(iter)` usage, but there were also quite some places that made use of the optional argument to avoid computing unnecessary offsets.

An interesting use I found was in the [calendar module](#), in the function `calendar.Calendar.termonthdays2`. The function `calendar.Calendar.termonthdays2` does the following: - you give it an year and a month, e.g. 2021 and 4 (for April); and - it returns a generator with the days of the month paired with the days of the week (0 to 6). (There's the little caveat that the iterator returns sequences of whole weeks, so it may pad the results in the beginning and/or end.)

Here is an example:

```

>>> for arg in c.Calendar().termonthdays2(2021, 4):
...     print(arg)
...
(0, 0)
(0, 1)
(0, 2)
(1, 3)
(2, 4)
(3, 5)
(4, 6)
(5, 0)
(6, 1)
(7, 2)
## ... cut for brevity
(28, 2)
(29, 3)
(30, 4)
(0, 5)
(0, 6)

```

The numbers on the left show the day of the month and the days on the right encode the day of the week, where 0 is Monday, up to 6 which is Sunday. The 6th of April of 2021 (the day I wrote this article on) was a Tuesday, which is encoded by the (6, 1) in the output above.

Here is the code that implements `termonthdays2`:

```
## from Lib\calendar.py in Python 3.9
class Calendar(object):
    # ...

    def itermonthdays2(self, year, month):
        """
        Like itermonthdays(), but will yield (day number, weekday number)
        tuples. For days outside the specified month the day number is 0.
        """
        for i, d in enumerate(self.itermonthdays(year, month), self.firstweekday):
            yield d, i % 7
```

This function relies heavily on `itermonthdays(year, month)` that just returns a sequence of month days with some leading and/or trailing zeroes, so that the sequence represents the whole weeks in which that month fell.

For example, look at my desktop calendar for the month of April of 2021:

April 2021							^	▼
Su	Mo	Tu	We	Th	Fr	Sa		
28	29	30	31	1	2	3		
4	5	6	7	8	9	10		
11	12	13	14	15	16	17		
18	19	20	21	22	23	24		
25	26	27	28	29	30	1		

If I tell the `Calendar` class to start counting weeks on Sundays (day 6), like my desktop calendar does, here is what `itermonthdays` produces:

```
>>> for d in c.Calendar(6).itermonthdays(2021, 4):
...     print(d)
...
0
0
0
0
1
2
```

```
3
4
## ... cut for brevity
30
0
```

The first four 0 are the four March days that show up in the top week and the final 0 corresponds to the 1st of May that is shown in the bottom right corner of my calendar.

In order to return these days together with the respective day of the week, `enumerate` is being fed a `start=` argument, which is `self.firstweekday`, to sync up the days of the month to what the `Calendar` sees as the first day of the week.

Filtering the indices

A really neat usage of `enumerate` I found while probing the Python Standard Library was to filter a list in search for the indices of the elements that satisfy a certain predicate.

For example, say you have a list of integers and you have a function that tells you if a number is odd or not:

```
>>> nums = [4071, 53901, 96045, 84886, 5228, 20108, 42468, 89385, 22040, 18800, 4071]
>>> odd = lambda x: x%2
```

What code do you write to figure out the *indices* of the numbers that are odd? Notice that solutions making use of `nums.index` in general won't work because the list may contain duplicates (cf. `nums[0]` and `nums[-1]` above).

In a helper file in the library with, and I quote, "Shared OS X support functions.", I found a really elegant solution:

```
>>> [i for i, n in enumerate(nums) if odd(n)]
[0, 1, 2, 7, 10]
```

Of course the file I am talking about (`Lib\osx_support.py`) didn't have this code, but it did have the pattern I just showed. The actual code there is the following:

```
## from Lib\osx_support.py in Python 3.9
def compiler_fixup(compiler_so, cc_args):
    #
    indices = [i for i,x in enumerate(compiler_so) if x.startswith('-isysroot')]
    # ...
```

While I have *no* clue what the code is doing from the semantic point of view, we can clearly see that `indices` is collecting the indices of the elements in `compiler_so` that start with `"-isysroot"`.

Making the most out of the tuples

Another interesting usage of the `enumerate` function I found was to create dictionaries directly. For example, if we take a look at the [mailbox module](#) we can find a line of code that is building a table of contents as a dictionary, where the keys are the integers given by `enumerate` and the values are tuples built by `zip`:

```
## from Lib\mailbox.py in Python 3.9
class mbox(_mboxMMDF):
    # ...
    def _generate_toc(self):
        """Generate key-to-(start, stop) table of contents."""
        starts, stops = [], []
        last_was_empty = False
        self._file.seek(0)
        while True:
            # process self._file
            self._toc = dict(enumerate(zip(starts, stops)))
            # ...

```

Notice how the code initialises empty lists `starts` and `stops`, which are then populated inside the `while` loop I deleted because it was fairly long and would distract us from the main point: the line

```
self._toc = dict(enumerate(zip(starts, stops)))
```

Because `enumerate` returns 2-item tuples, `dict` can take that and build a dictionary. Curiously enough, we actually want `starts` and `stops` to be paired up together, so we end up with calling `enumerate` on a `zip`, so this is what the result could look like:

```
>>> starts = [1, 10, 21, 30]
>>> stops = [9, 15, 28, 52]
>>> dict(enumerate(zip(starts, stops)))
{0: (1, 9), 1: (10, 15), 2: (21, 28), 3: (30, 52)}
```

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“enumerate is your best friend if you need to traverse an iterator to deal with its data and also need access to information about its index.”

This Pydon't showed you that:

- `enumerate` gives you access to an iterable's elements *and* indices at the same time;
- `enumerate` by itself returns a *lazy* `enumerate` object that must be then iterated or converted explicitly to a list (or something else that suits your needs) if you want its values;
- `enumerate` takes a second argument to set an offset for the indexing;
 - and, in particular, that argument can be a negative integer;
- the result of `enumerate` can be fed directly to `dict` to create a dictionary whose keys are the indices;
- using `enumerate` we get a nice idiom to find the indices of an iterable that point to the elements that satisfy a given condition; and
- coupling `zip`, `enumerate`, and deep unpacking allows you to loop over several iterables elegantly.

References

- Python 3 Documentation, The Python Standard Library, `enumerate`, docs.python.org/3/library/functions.html#enumerate [last accessed 06-04-2021];
- Python 3 Documentation, The Python Standard Library, `calendar.Calendar`, docs.python.org/3/library/calendar.html#calendar.Calendar [last accessed 06-04-2021].
- Python 3 Documentation, The Python Standard Library, `doctest`, docs.python.org/3/library/doctest.html [last accessed 06-04-2021].
- Python 3 Documentation, The Python Standard Library, `mailbox`, docs.python.org/3/library/mailbox.html [last accessed 06-04-2021].

str and repr



```
class SomeClass:
    def __str__(self):
        # ..?

    def __repr__(self):
        # ..?
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/str-and-repr.>)

str and repr

Python has two built-in mechanisms that allow you to convert an object to a string, so that you can look at it and print it. I am talking about the `str` class and the built-in `repr` function.

There is often confusion as to what the differences between these two built-ins are, but the difference is simple and clear. The `str` class is used when you want to convert something to the string type, and is also used when you need a readable representation of your object. On the other hand, the `repr` function is used to create an *unambiguous* representation of its argument.

End users generally use `str` because they want to print readable and good looking text, whereas developers may use `repr` because they need to debug code and need to make sure they know what they are looking at. For example, take a look at the following interactive session:

```
>>> print(3)
3
>>> print("3")
3
>>> 3
3
>>> "3"
'3'
```

The `print` function calls `str` on its argument and then displays it, so both the integer 3 and the string "3" get printed the same way: you have no way to tell if the original object is an integer or a string. After that, you see that simply writing the integer 3 and the string "3" in the REPL returns an unambiguous representation of the object: you can tell integers and strings apart, because the REPL is using `repr` under the hood to show objects. `repr` is also used when your object is inside a container, like a list or a dictionary, because containers usually defer their `str` behaviour to `repr`, as you can see by looking at [PEP 3140](#) and at the following session:

```
>>> [3, "3"]
[3, '3']
>>> print([3, "3"])
[3, '3']
>>> str([3, "3"]) == repr([3, "3"])
True
```

The `__str__` and `__repr__` dunder methods

When you are defining your own classes in Python you will probably want to specify how your objects should look when printed, given that the default behaviour in Python is not very helpful:

```
>>> class A:
...     pass
...
>>> a = A()
>>> print(a)
<__main__.A object at 0x012DF640>
```

```
>>> a
<__main__.A object at 0x012DF640>
```

If you want to display your objects properly, you will want to implement the `__str__` and `__repr__` dunder methods (*dunder* stands for *double underscore*), and the implementations should follow the use case of `str` and `repr` outlined above: the implementation of `__str__` should provide a nice, readable representation of your object and `__repr__` should represent unambiguously your object, preferably by providing an expression that could be used to rebuild the object.

If you are not acquainted with Python's dunder methods, you may want to [subscribe](#) to the Pydon't newsletter, I will write more about them later. Until then, you may want to have a look at the Python 3 Docs and what they say about the [data model](#).

When implementing custom classes, I suggest you start by implementing `__repr__`, as `__str__` will default to calling `__repr__` if no custom implementation is given, but only implementing `__str__` still leaves you with rather unhelpful representations of your objects.

If you just implement `__str__`:

```
>>> class A:
...     def __str__(self):
...         return "A"
...
>>> a = A()
>>> a
<__main__.A object at 0x01600760>
>>> print(a)
A
```

If you just implement `__repr__`:

```
>>> class A:
...     def __repr__(self):
...         return "A"
...
>>> a = A()
>>> a
A
>>> print(a)
A
```

Examples in code

`datetime`

Python's `datetime` module supplies classes for manipulating dates and times. A simple date could be created like so:

```
>>> import datetime
>>> date = datetime.datetime(2021, 2, 2)
```

Now that we have your date object of type `datetime.datetime`, we can see what its `repr` looks like and compare it to its `str` version:

```
>>> print(repr(date))
datetime.datetime(2021, 2, 2, 0, 0)
>>> print(str(date))
2021-02-02 00:00:00
```

We can see that `repr(date)` could be used to create the same exact object:

```
>>> date == datetime.datetime(2021, 2, 2, 0, 0)
True
>>> date == eval(repr(date))
True
```

Whereas `str(date)` creates a nice-looking representation of the date in question. Notice that from its `str` we can't even tell that we were dealing with a `datetime.datetime` object.

2D point

An example custom usage of the `__str__` and `__repr__` dunder methods could come into play if you were to implement a simple class that represents 2D points, for example because you have to deal with images or a game or maps, or whatever your use case is.

Ignoring all other methods you would certainly implement, your class could look like this:

```
class Point2D:
    """A class to represent points in a 2D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        """Provide a good-looking representation of the object."""
        return f"({self.x}, {self.y})"

    def __repr__(self):
        """Provide an unambiguous way of rebuilding this object."""
        return f"Point2D({repr(self.x)}, {repr(self.y)})"

p = Point2D(0, 0) # the origin.
print(f"To build the point {p} in your code, try writing {repr(p)}.")
```

Running this code prints `To build the point (0, 0) in your code, try writing Point2D(0, 0).` to your console. Your end user may be accustomed to 2D points, and thus they may need nothing more than the standard `(x, y)` representation of a 2D point. During debugging, the `Point2D` prefix is useful because it helps you distinguish between a tuple and a custom `Point2D` instance.

Conclusion

When implementing custom classes you will probably want to give a custom implementation of the `__repr__` dunder method, and also a `__str__` if you need your instances to look good when printed to the end user. `__str__` and `str` are used when you need good looking strings, while the purpose of `__repr__` and `repr` is to create unambiguous representations of your objects.

References

- Python 3 Documentation, The Python Language Reference, Data model, `repr` and `str`, https://docs.python.org/3/reference/datamodel.html#object.__repr__.
- Python 3 Documentation, The Python Standard Library, Built-in Functions, <https://docs.python.org/3/library/functions.html>.
- Python 3 Documentation, The Python Standard Library, Built-in Types, `str`, <https://docs.python.org/3/library/stdtypes.html#str>.
- PEP 3140 – `str(container)` should call `str(item)`, not `repr(item)`, <https://www.python.org/dev/peps/pep-3140/>.
- Stack Overflow, “Purpose of Python’s `repr`”, <https://stackoverflow.com/questions/1984162/purpose-of-pythons-repr>.
- dbader.org, “Python String Conversion 101: Why Every Class Needs a “`repr`””, <https://dbader.org/blog/python-repr-vs-str>.

Online references last consulted on the 2nd of February of 2021.

Structural pattern matching tutorial



```
colour = (25, 56, 200)

match colour:
    case r, g, b:
        print("No alpha.")
    case r, g, b, alpha:
        print(f"Alpha is {alpha}")

# Prints 'No alpha.'
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/structural-pattern-matching-tutorial>.)

Introduction

Structural pattern matching is coming to Python, and while it may look like a plain switch statement like many other languages have, Python's `match` statement was not introduced to serve as a simple switch statement.

PEPs [634](#), [635](#), and [636](#) have plenty of information on what structural pattern matching is bringing to Python, how to use it, the rationale for adding it to Python, etc. In this article I will try to focus on using this new feature to write beautiful code.

At the time of writing, Python 3.10 is still a pre-release, so you have to look [in the right place](#) if you want to download Python 3.10 and play with it.

Structural pattern matching Python could already do

Structural pattern matching isn't completely new in Python. For a long time now, we have been able to do things like starred assignments:

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
```

And we can also do deep unpacking:

```
>>> name, (r, g, b) = ("red", (250, 23, 10))
>>> name
'red'
>>> r
250
>>> g
23
>>> b
10
```

I covered these in detail in [“Unpacking with starred assignments”](#) and [“Deep unpacking”](#), so go read those Pydon'ts if you are unfamiliar with how to use these features to write Pythonic code.

The `match` statement will use ideas from both starred assignments and deep unpacking, so knowing how to use them is going to be helpful.

Your first match statement

For your first `match` statement, let's implement the factorial function. A factorial function is a textbook example when introducing people to recursion, and you could write it like so:

```

def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
factorial(5)      # 120

```

Instead of using an `if` statement, we could use a `match`:

```

def factorial(n):
    match n:
        case 0 | 1:
            return 1
        case _:
            return n * factorial(n - 1)
factorial(5)

```

Notice a couple of things here: we start our `match` statement by typing `match n`, meaning we will want to do different things depending on what `n` is. Then, we have `case` statements that can be thought of the different possible scenarios we want to handle. Each `case` must be followed by a pattern that we will try to match `n` against.

Patterns can also contain alternatives, denoted by the `|` in `case 0 | 1`, which matches if `n` is either `0` or `1`. The second pattern, `case _`, is the go-to way of matching *anything* (when you don't care about *what* you are matching), so it is acting more or less like the `else` of the first definition.

Pattern matching the basic structure

While `match` statements can be used like plain `if` statements, as you have seen above, they really shine when you are dealing with structured data:

```

def normalise_colour_info(colour):
    """Normalise colour info to (name, (r, g, b, alpha))."""

    match colour:
        case (r, g, b):
            name = ""
            a = 0
        case (r, g, b, a):
            name = ""
        case (name, (r, g, b)):
            a = 0
        case (name, (r, g, b, a)):
            pass
        case _:
            raise ValueError("Unknown colour info.")
    return (name, (r, g, b, a))

```

```

# Prints ('', (240, 248, 255, 0))
print(normalise_colour_info((240, 248, 255)))
# Prints ('', (240, 248, 255, 0))
print(normalise_colour_info((240, 248, 255, 0)))
# Prints ('AliceBlue', (240, 248, 255, 0))
print(normalise_colour_info(("AliceBlue", (240, 248, 255))))
# Prints ('AliceBlue', (240, 248, 255, 0.3))
print(normalise_colour_info(("AliceBlue", (240, 248, 255, 0.3))))

```

Notice here that each case contains an expression like the left-hand side of an unpacking assignment, and when the structure of colour matches the structure that the case exhibits, then the names get assigned to the variable names in the case.

This is a great improvement over the equivalent code with `if` statements:

```

def normalise_colour_info(colour):
    """Normalise colour info to (name, (r, g, b, alpha))."""

    if not isinstance(colour, (list, tuple)):
        raise ValueError("Unknown colour info.")

    if len(colour) == 3:
        r, g, b = colour
        name = ""
        a = 0
    elif len(colour) == 4:
        r, g, b, a = colour
        name = ""
    elif len(colour) != 2:
        raise ValueError("Unknown colour info.")
    else:
        name, values = colour
        if not isinstance(values, (list, tuple)) or len(values) not in [3, 4]:
            raise ValueError("Unknown colour info.")
        elif len(values) == 3:
            r, g, b = values
            a = 0
        else:
            r, g, b, a = values
    return (name, (r, g, b, a))

```

I tried writing a decent, equivalent piece of code to the one using structural pattern matching, but this doesn't look that good. Someone else has suggested, [in the comments](#), another alternative that also doesn't use `match`. That suggestion looks better than mine, but is much more complex and larger than the alternative with `match`.

The `match` version becomes even better when we add type validation to it, by asking for the specific values to actually match Python's built-in types:

```

def normalise_colour_info(colour):
    """Normalise colour info to (name, (r, g, b, alpha))."""

    match colour:
        case (int(r), int(g), int(b)):
            name = ""
            a = 0
        case (int(r), int(g), int(b), int(a)):
            name = ""
        case (str(name), (int(r), int(g), int(b))):
            a = 0
        case (str(name), (int(r), int(g), int(b), int(a))):
            pass
        case _:
            raise ValueError("Unknown colour info.")
    return (name, (r, g, b, a))

# Prints ('AliceBlue', (240, 248, 255, 0))
print(normalise_colour_info(("AliceBlue", (240, 248, 255))))
# Raises # ValueError: Unknown colour info.
print(normalise_colour_info(("Red", (255, 0, "0"))))

How do you reproduce all this validation with if statements..?

```

Matching the structure of objects

Structural pattern matching can also be used to match the structure of class instances. Let us recover the Point2D class I have used as an example in a couple of posts, in particular the [Pydon't](#) about `__str__` and `__repr__`:

```

class Point2D:
    """A class to represent points in a 2D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        """Provide a good-looking representation of the object."""
        return f"({self.x}, {self.y})"

    def __repr__(self):
        """Provide an unambiguous way of rebuilding this object."""
        return f"Point2D({repr(self.x)}, {repr(self.y)})"

```

Imagine we now want to write a little function that takes a Point2D and writes a little description of where the point lies. We can use pattern matching to capture the values of the `x` and `y` attributes and, what is more,

we can use short `if` statements to help narrow down the type of matches we want to succeed!

Take a look at the following:

```
def describe_point(point):
    """Write a human-readable description of the point position."""

    match point:
        case Point2D(x=0, y=0):
            desc = "at the origin"
        case Point2D(x=0, y=y):
            desc = f"in the vertical axis, at y = {y}"
        case Point2D(x=x, y=0):
            desc = f"in the horizontal axis, at x = {x}"
        case Point2D(x=x, y=y) if x == y:
            desc = f"along the x = y line, with x = y = {x}"
        case Point2D(x=x, y=y) if x == -y:
            desc = f"along the x = -y line, with x = {x} and y = {y}"
        case Point2D(x=x, y=y):
            desc = f"at {point}"

    return "The point is " + desc

# Prints "The point is at the origin"
print(describe_point(Point2D(0, 0)))
# Prints "The point is in the horizontal axis, at x = 3"
print(describe_point(Point2D(3, 0)))
# Prints "# The point is along the x = -y line, with x = 3 and y = -3"
print(describe_point(Point2D(3, -3)))
# Prints "# The point is at (1, 2)"
print(describe_point(Point2D(1, 2)))
```

__match_args__

Now, I don't know if you noticed, but didn't all the `x=` and `y=` in the code snippet above annoy you? Every time I wrote a new pattern for a `Point2D` instance, I had to specify what argument was `x` and what was `y`. For classes where this order is not arbitrary, we can use `__match_args__` to tell Python how we would like `match` to match the attributes of our object.

Here is a shorter version of the example above, making use of `__match_args__` to let Python know the order in which arguments to `Point2D` should match:

```
class Point2D:
    """A class to represent points in a 2D space."""

    __match_args__ = ["x", "y"]
    def __init__(self, x, y):
```

```

    self.x = x
    self.y = y

def describe_point(point):
    """Write a human-readable description of the point position."""

    match point:
        case Point2D(0, 0):
            desc = "at the origin"
        case Point2D(0, y):
            desc = f"in the vertical axis, at y = {y}"
        case Point2D(x, 0):
            desc = f"in the horizontal axis, at x = {x}"
        case Point2D(x, y):
            desc = f"at {point}"

    return "The point is " + desc

# Prints "The point is at the origin"
print(describe_point(Point2D(0, 0)))
# Prints "The point is in the horizontal axis, at x = 3"
print(describe_point(Point2D(3, 0)))
# Prints "# The point is at (1, 2)"
print(describe_point(Point2D(1, 2)))

```

Wildcards

Another cool thing you can do when matching things is to use wildcards.

Asterisk *

Much like you can do things like

```

>>> head, *body, tail = range(10)
>>> print(head, body, tail)
0 [1, 2, 3, 4, 5, 6, 7, 8] 9

```

where the `*body` tells Python to put in `body` whatever does *not* go into `head` or `tail`, you can use `*` and `**` wildcards. You can use `*` with lists and tuples to match the *remaining* of it:

```

def rule_substitution(seq):
    new_seq = []
    while seq:
        match seq:
            case [x, y, z, *tail] if x == y == z:
                new_seq.extend(["3", x])
            case [x, y, *tail] if x == y:

```

```

        new_seq.extend(["2", x])
    case [x, *tail]:
        new_seq.extend(["1", x])
    seq = tail
    return new_seq

seq = ["1"]
print(seq[0])
for _ in range(10):
    seq = rule_substitution(seq)
    print("".join(seq))

"""
Prints:
1
11
21
1211
111221
312211
13112221
1113213211
31131211131221
13211311123113112211
11131221133112132113212221
"""

```

This builds the sequence I showed above, where each number is derived from the previous one by looking at its digits and describing what you are looking at. For example, when you find three equal digits in a row, like "222", you rewrite that as "32" because you are seeing three twos. With the `match` statement this becomes much cleaner. In the case statements above, the `*tail` part of the pattern matches the remainder of the sequence, as we are only using `x`, `y`, and `z` to match in the beginning of the sequence.

Plain dictionary matching

Similarly, we can use `**` to match the remainder of a dictionary. But first, let us see what is the behaviour when matching dictionaries:

```

d = {0: "oi", 1: "uno"}
match d:
    case {0: "oi"}:
        print("yeah.")
## prints yeah.

```

While `d` has a key `1` with a value `"uno"`, and that is not specified in the only case statement, there is a match and we enter the statement. When matching with dictionaries, we only care about matching the structure that was explicitly mentioned, and any other extra keys that the original dictionary has are ignored. This is

unlike matching with lists or tuples, where the match has to be perfect if no wildcard is mentioned.

Double asterisk **

However, if you want to know what the original dictionary had that was not specified in the match, you can use a `**` wildcard:

```
d = {0: "oi", 1: "uno"}  
match d:  
    case {0: "oi", **remainder}:  
        print(remainder)  
## prints {1: 'uno'}
```

Finally, you can use this to your advantage if you want to match a dictionary that contains **only** what you specified:

```
d = {0: "oi", 1: "uno"}  
match d:  
    case {0: "oi", **remainder} if not remainder:  
        print("Single key in the dictionary")  
    case {0: "oi"}:  
        print("Has key 0 and extra stuff.")  
## Has key 0 and extra stuff.
```

You can also use variables to match the values of given keys:

```
d = {0: "oi", 1: "uno"}  
match d:  
    case {0: zero_val, 1: one_val}:  
        print(f"0 mapped to {zero_val} and 1 to {one_val}")  
## 0 mapped to oi and 1 to uno
```

Naming sub-patterns

Sometimes you may want to match against a more structured pattern, but then give a name to a part of the pattern, or to the whole thing, so that you have a way to refer back to it. This may happen especially when your pattern has alternatives, which you add with `|`:

```
def go(direction):  
    match direction:  
        case "North" | "East" | "South" | "West":  
            return "Alright, I'm going!"  
        case _:  
            return "I can't go that way..."  
  
print(go("North"))      # Alright, I'm going!  
print(go("asfasdf"))    # I can't go that way...
```

Now, imagine that the logic to handle that “going” somewhere is nested inside something more complex:

```

def act(command):
    match command.split():
        case "Cook", "breakfast":
            return "I love breakfast."
        case "Cook", *wtv:
            return "Cooking..."
        case "Go", "North" | "East" | "South" | "West":
            return "Alright, I'm going!"
        case "Go", *wtv:
            return "I can't go that way..."
        case _:
            return "I can't do that..."

print("Go North")      # Alright, I'm going!
print("Go asdfasdf")  # I can't go that way...
print("Cook breakfast") # I love breakfast.
print("Drive")         # I can't do that...

```

And, not only that, we want to know where the user wants to go, in order to include that in the message. We can do this by leaving the options up, but then capturing the result of the match in a variable:

```

def act(command):
    match command.split():
        case "Cook", "breakfast":
            return "I love breakfast."
        case "Cook", *wtv:
            return "Cooking..."
        case "Go", "North" | "East" | "South" | "West" as direction:
            return f"Alright, I'm going {direction}!"
        case "Go", *wtv:
            return "I can't go that way..."
        case _:
            return "I can't do that..."

print("Go North")      # Alright, I'm going North!
print("Go asdfasdf")  # I can't go that way...

```

Traversing recursive structures

Another type of situation in which structural pattern matching is expected to succeed quite well is in handling recursive structures.

I have seen great examples of this use-case in the references I included below, and will now share one of my own.

Imagine you want to transform a mathematical expression into prefix notation, e.g. "3 * 4" becomes "* 3 4" and 1 + 2 + 3 becomes + 1 + 2 3 or + + 1 2 3 depending on whether + associates from the left or

from the right.

You can write a little `match` to deal with this:

```
import ast

def prefix(tree):
    match tree:
        case ast.Expression(expr):
            return prefix(expr)
        case ast.Constant(value=v):
            return str(v)
        case ast.BinOp(lhs, op, rhs):
            match op:
                case ast.Add():
                    sop = "+"
                case ast.Sub():
                    sop = "-"
                case ast.Mult():
                    sop = "*"
                case ast.Div():
                    sop = "/"
                case _:
                    raise NotImplementedError()
            return f"{sop} {prefix(lhs)} {prefix(rhs)}"
        case _:
            raise NotImplementedError()

print(prefix(ast.parse("1 + 2 + 3", mode="eval")))      # + + 1 2 3
print(prefix(ast.parse("2**3 + 6", mode="eval")))      # + * 2 3 6
# Prints '- + 1 * 2 3 / 5 7', take a moment to digest this one.
print(prefix(ast.parse("1 + 2*3 - 5/7", mode="eval")))
```

Careful with the hype

Now, here is a word of caution: `match` isn't the best solution always. Looking up to the prefix notation example above, perhaps there are better ways to transform each possible binary operator to its string representation..? The current solution spends two lines for each different operator, and if we add support for many more binary operators, that part of the code will become unbearably long.

In fact, we can (and probably should) do something else about that. For example,

```
import ast
```

```
def op_to_str(op):
    ops = {
        ast.Add: "+",
        ast.Sub: "-",
        ast.Mult: "*",
        ast.Div: "/",
        ast.FloorDiv: "//"
    }
    return ops.get(op, str(op))
```

```

        ast.Mult: "*",
        ast.Div: "/",
    }
    return ops.get(op.__class__, None)

def prefix(tree):
    match tree:
        case ast.Expression(expr):
            return prefix(expr)
        case ast.Constant(value=v):
            return str(v)
        case ast.BinOp(lhs, op, rhs):
            sop = op_to_str(op)
            if sop is None:
                raise NotImplementedError()
            return f"{sop} {prefix(lhs)} {prefix(rhs)}"
        case _:
            raise NotImplementedError()

print(prefix(ast.parse("1 + 2 + 3", mode="eval")))      # + + 1 2 3
print(prefix(ast.parse("2*3 + 6", mode="eval")))        # + * 2 3 6
# Prints '- + 1 * 2 3 / 5 7', take a moment to digest this one.
print(prefix(ast.parse("1 + 2*3 - 5/7", mode="eval")))

```

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“Structural pattern matching introduces a feature that can simplify and increase the readability of Python code in many cases, but it will not be the go-to solution in every single situation.”

This Pydon't showed you that:

- structural pattern matching with the `match` statement greatly extends the power of the already-existing starred assignment and structural assignment features;
- structural pattern matching can match literal values and arbitrary patterns
- patterns can include additional conditions with `if` statements
- patterns can include wildcards with `*` and `**`
- `match` statements are very powerful when dealing with the structure of class instances
- `__match_args__` allows to define a default order for arguments to be matched in when a custom class is used in a case
- built-in Python classes can be used in `case` statements to validate types

References

- PEP 622 – Structural Pattern Matching, <https://www.python.org/dev/peps/pep-0622/>;

- PEP 634 – Structural Pattern Matching: Specification, <https://www.python.org/dev/peps/pep-0634/>;
- PEP 635 – Structural Pattern Matching: Motivation and Rationale, <https://www.python.org/dev/peps/pep-0635/>;
- PEP 636 – Structural Pattern Matching: Tutorial, <https://www.python.org/dev/peps/pep-0636/>;
- Dynamic Pattern Matching with Python, <https://gvanrossum.github.io/docs/PyPatternMatching.pdf>;
- Python 3.10 Pattern Matching in Action, YouTube video by “Big Python”, <https://www.youtube.com/watch?v=SYTVSeTgL3s>.

Structural pattern matching anti-patterns

```
● ● ●

# The match statement is hard to read,
# because the cases are long and very similar.
def rule30(bits):
    """Implement the 'rule 30' automaton."""
    match bits:
        case 0, 0, 0 | 1, 0, 1 | 1, 1, 0 | 1, 1, 1:
            return 0
        case 0, 0, 1 | 0, 1, 0 | 0, 1, 1 | 1, 0, 0:
            return 1

# A bit of research shows a closed formula:
def rule30(bits):
    p, q, r = bits
    return (p + q + r + q*r) % 2
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/structural-pattern-matching-anti->

patterns.)

Introduction

Structural pattern matching is coming to Python, and while it may look like a plain switch statement like many other languages have, Python's `match` statement was not introduced to serve as a simple switch statement.

In [this article](#) I explored plenty of use cases for the new `match` statement, and in this blog post I will try to explore some use cases for which a `match` is *not* the answer. This article will assume you know how structural pattern matching works in Python, so if you are unsure how that works feel free to read my [“Pattern matching tutorial for Pythonic code”](#).

At the time of writing, Python 3.10 is still a pre-release, so you have to look [in the right place](#) if you want to download Python 3.10 and play with it.

There should be only one obvious way to do it

As per [the Zen of Python](#),

“There should be one- and preferably only one –obvious way to do it.”

The introduction of the `match` statement may seem to violate this principle... However, you have to remember that the point of the `match` statement is *not* to serve as a basic switch statement, as we already have plenty of alternatives for that – if `match` was supposed to be a simple switch, then it would probably be called “switch” and not “match”. No, the point of the `match` statement is to do *structural pattern matching*, so there's plenty of basic types of matching and casing that can be done with the traditional tools that we have been using up until Python 3.10.

Below I will share some of these tools with you, and I'll try to describe the situations in which they are helpful. Of course, this is much easier to understand with code so there will be plenty of code examples along the way.

A short and sweet if statement

The [Collatz conjecture](#) is a mathematical conjecture that says that the following function terminates for any positive integer given as input:

```
def collatz_path(n):
    path = [n]
    while n != 1:
        match n % 2:
            case 0:
                n //= 2
            case 1:
                n = 3*n + 1
        path.append(n)
    return path
```

Which gives the following two example outputs:

```
>>> collatz_path(8)
[8, 4, 2, 1]
>>> collatz_path(15)
[15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

If we look at the usage of `match` above, we see it basically served as a simple switch to match either 0 or 1, the only two values that the operation `n % 2` could result in for a positive integer `n`. Notice that if we use a plain `if` we can write exactly the same code *and* save one line of code:

```
def collatz_path(n):
    path = [n]
    while n != 1:
        if n % 2:
            n = 3*n + 1
        else:
            n //= 2
        path.append(n)
    return path
```

We saved one line of code and reduced the maximum depth of our indentation: with the `match` we had code that was indented four times, whereas the implementation with the `if` only has three levels of depth. When you only have a couple of options and you are checking for explicit equality, a short and sweet `if` statement is most likely the way to go.

Be smart(er)

Sometimes you will feel like you have to list a series of cases and corresponding values, so that you can map one to the other. However, it might be the case that you could make your life *much* simpler by looking for an alternative algorithm or formula and implementing that instead. I'll show you an example.

In case you never heard of it, [Rule 30](#) is an “elementary cellular automaton”. You can think of it as a rule that receives three bits (three zeroes/ones) and produces a new bit, depending on the three bits it received. Automatons are really, really, interesting, but discussing them is past the point of this article. Let us just look at a possible implementation of the “Rule 30” automaton:

```
def rule30(bits):
    match bits:
        case 0, 0, 0:
            return 0
        case 0, 0, 1:
            return 1
        case 0, 1, 0:
            return 1
        case 0, 1, 1:
            return 1
        case 1, 0, 0:
```

```

        return 1
case 1, 0, 1:
    return 0
case 1, 1, 0:
    return 0
case 1, 1, 1:
    return 0

```

This seems like a sensible use of the `match` statement, except that we just wrote 16 lines of code... Ok, you are right, let us put together the rules that return the same values, that should make the code shorter:

```

def rule30(bits):
    match bits:
        case 0, 0, 0 | 1, 0, 1 | 1, 1, 0 | 1, 1, 1:
            return 0
        case 0, 0, 1 | 0, 1, 0 | 0, 1, 1 | 1, 0, 0:
            return 1

```

Yup, much better. But now we have four options on each case, and I have to squint to figure out where each option starts and ends, and the long strings of zeroes and ones aren't really that pleasant to the eye... Can we make it better..?

With just a little bit of research you can find out that the “Rule 30” can be written as a *closed formula* that depends on the three input bits, which means we don't have to `match` the input bits with all the possible inputs, we can just *compute* the output:

```

def rule30(bits):
    p, q, r = bits
    return (p + q + r + q*r) % 2

```

You might argue that this formula obscures the relationship between the several inputs and their outputs. You are right in principle, but having the explicit “Rule 30” written out as a `match` doesn't tell you much about why each input maps to each output either way, so why not make it short and sweet?

Basic mappings

Getting from dictionaries

There are many cases in which you just want to take a value in and map it to something else. As an example, take this piece of code that takes an expression and writes it in prefix notation:

```

import ast

def prefix(tree):
    match tree:
        case ast.Expression(expr):
            return prefix(expr)
        case ast.Constant(value=v):
            return str(v)

```

```

case ast.BinOp(lhs, op, rhs):
    match op:
        case ast.Add():
            sop = "+"
        case ast.Sub():
            sop = "-"
        case ast.Mult():
            sop = "*"
        case ast.Div():
            sop = "/"
        case _:
            raise NotImplementedError()
    return f"{sop} {prefix(lhs)} {prefix(rhs)}"

case _:
    raise NotImplementedError()

print(prefix(ast.parse("1 + 2 + 3", mode="eval")))      # + + 1 2 3
print(prefix(ast.parse("2**3 + 6", mode="eval")))      # + * 2 3 6
# Final one prints '- + 1 * 2 3 / 5 7', take a moment to grok it.
print(prefix(ast.parse("1 + 2*3 - 5/7", mode="eval")))

```

Notice the inner `match` to convert the `op` inside a `BinOp` to a string? For starters, that nested `match` takes up too much vertical space and distracts us from what really matters, which is the traversal of the recursive structure of the tree. This means we could actually refactor that bit as a utility function:

```

import ast

def op_to_str(op):
    match op:
        case ast.Add():
            sop = "+"
        case ast.Sub():
            sop = "-"
        case ast.Mult():
            sop = "*"
        case ast.Div():
            sop = "/"
        case _:
            raise NotImplementedError()
    return sop

def prefix(tree):
    match tree:
        case ast.Expression(expr):
            return prefix(expr)
        case ast.Constant(value=v):
            return str(v)

```

```

case ast.BinOp(lhs, op, rhs):
    return f"{op_to_str(op)} {prefix(lhs)} {prefix(rhs)}"
case _:
    raise NotImplementedError()

print(prefix(ast.parse("1 + 2 + 3", mode="eval")))      # + + 1 2 3
print(prefix(ast.parse("2**3 + 6", mode="eval")))      # + * 2 3 6
# Final one prints '- + 1 * 2 3 / 5 7', take a moment to grok it.
print(prefix(ast.parse("1 + 2*3 - 5/7", mode="eval")))

```

This makes it easier to read and interpret the `prefix` function, but now we have another problem that really annoys me: a simple but long function, the `op_to_str` function. For every type of operator you support, your function grows by two lines... If you replace the `match` with a chain of `if` and `elif` statements you only save one line at the top...

The fix I suggested [in the original article](#) was using a dictionary to map the type of `op` to its string representation:

```

def op_to_str(op):
    ops = {
        ast.Add: "+",
        ast.Sub: "-",
        ast.Mult: "*",
        ast.Div: "/",
    }
    return ops.get(op.__class__, None)

```

This usage pattern of a dictionary is quite common in Python, using the `get` method to compute the mapping of a value to another value. In case you are wondering, you can use the second argument of the `get` function to provide for a default value, which might be useful if the dictionary hasn't listed every single possible value or in case you want to have a fallback value.

getattr

Another useful mechanism that we have available is the `getattr` function, which is part of a trio of Python built-in functions: `hasattr`, `getattr` and `setattr`.

I will be writing about this trio in a future Pydon't; be sure to [subscribe](#) to the Pydon't newsletter so you don't miss it! For now, I'll just show you briefly what `getattr` can do for you.

I am writing an [APL](#) interpreter called [RGSP](#), and there is a function named `visit_F` where I need to map APL primitives like `+` and `-` to the corresponding Python function that implements it. These Python functions, implementing the behaviour of the primitives, live in the `functions.py` file. If I were using a `match` statement, here is what this `visit_F` could look like:

```

import functions

def visit_F(self, func):
    """Fetch the callable function."""

```

```

name = func.token.type.lower() # Get the name of the symbol.
match name:
    case "plus":
        function = functions.plus
    case "minus":
        function = functions.minus
    case "reshape":
        function = functions.reshape
    case _:
        function = None
if function is None:
    raise Exception(f"Could not find function {name}.")
return function

```

This is a similar problem to the one I showed above, where we wanted to get a string for each type of operator we got, so this could actually be written with the dictionary mapping. I invite you to do it, as a little exercise.

However, here's the catch: I have still a long way to go in my [RGSP](#) project, and I already have a couple dozen of those primitives, so my `match` statement would be around 40 lines long, if I were using that solution, or 20 lines long if I were using the dictionary solution, with a key, value pair per line.

Thankfully, Python's `getattr` can be used to get an *attribute* from an object, if I have the name of that attribute. It is no coincidence that the value of the `name` variable above is supposed to be exactly the same as the name of the function defined inside `functions.py`:

```

import functions

getattr(functions, "plus", None)      # returns functions.plus
getattr(functions, "reshape", None)    # returns functions.reduce
getattr(functions, "fasfadf", None)    # returns None

```

With the `getattr` function that Python provides, my `visit_F` stays with a constant size, regardless of how many functions I add to the `functions.py` file:

```

def visit_F(self, func):
    """Fetch the callable function."""

    name = func.token.type.lower()      # Get the name of the symbol.
    function = getattr(functions, name, None)
    if function is None:
        raise Exception(f"Could not find function {name}.")
    return function

```

The `getattr` function can also be used to get attributes from an instance of a class, e.g.,

```

class Foo:
    def __init__(self, a, b):
        self.a = a
        self.b = b

```

```

foo = Foo(3, 4)
print(getattr(foo, "a"))      # prints 3
bar = Foo(10, ";")
print(getattr(bar, ";"))      # prints ;

```

This goes to show that it is always nice to know the tools you have at your disposal. Not everything has very broad use cases, but that also means that the more specialised tools are the ones that make the most difference when they are brought in.

Speaking of knowing your tools, the last use case in this article for which `match` is a bad alternative is related to calling different functions when your data has different types.

Single-dispatch generic functions

If you have programming experience in a programming language like Java, you will be familiar with the concept of overloading a function: you implement the same function several times, but you get to specify the behaviour of the function for different types of arguments and/or number of arguments.

For example, you might want to implement a function to pretty-print a series of different types of objects:

```

def pretty_print(arg):
    if isinstance(arg, complex):
        print(f"{arg.real} + {arg.imag}i")
    elif isinstance(arg, (list, tuple)):
        for i, elem in enumerate(arg):
            print(i, elem)
    elif isinstance(arg, dict):
        for key, value in arg.items():
            print(f"{key}: {value}")
    else:
        print(arg)

```

Which then works like so:

```

>>> pretty_print(3)
3
>>> pretty_print([2, 5])
0 2
1 5
>>> pretty_print(3+4j)
3.0 + 4.0i

```

You can see that the branching introduced by the `if` statement is merely to separate the different types that the `arg` could have, and while the handling logic might be different, the final purpose is always the same: to pretty-print an object. But what if the code to handle each type of argument took 10 or 20 lines? You would be getting a really long function with what would essentially be embedded subfunctions.

You can separate all these subfunctions by making use of the `functools.singledispatch` decorator:

```
import functools
```

```

@functools.singledispatch
def pretty_print(arg):
    print(arg)

@pretty_print.register(complex)
def _(arg):
    print(f"{arg.real} + {arg.imag}i")

@pretty_print.register(list)
@pretty_print.register(tuple)
def _(arg):
    for i, elem in enumerate(arg):
        print(i, elem)

@pretty_print.register(dict)
def _(arg):
    for key, value in arg.items():
        print(f"{key}: {value}")

```

And this can then be used exactly like the original function:

```

>>> pretty_print(3)
3
>>> pretty_print([2, 5])
0 2
1 5
>>> pretty_print(3+4j)
3.0 + 4.0i

```

The `pretty_print` example isn't the *best* example because you spend as many lines decorating as in defining the actual subfunctions, but this shows you the pattern that you can now be on the lookout for. You can read more about `singledispatch` [in the docs](#).

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“The new `match` statement *is* great, but that does not mean the `match` statement will be the best alternative always and, in particular, the `match` statement is generally being misused if you use it as a simple switch.”

This Pydon't showed you that:

- `match` isn't necessarily always the best way to implement control flow;
- short and basic `match` statements could be vanilla `if` statements;
- sometimes there is a way to *compute* what you need, instead of having to list many different cases and their respective values;

- built-in tools like `dict.get` and `getattr` can also be used to fetch different values depending on the matching key; and
- you can use `functools.singledispatch` when you need to execute different subfunctions when the input has different types.

References

- PEP 622 – Structural Pattern Matching, <https://www.python.org/dev/peps/pep-0622/>;
- PEP 634 – Structural Pattern Matching: Specification, <https://www.python.org/dev/peps/pep-0634/>;
- PEP 635 – Structural Pattern Matching: Motivation and Rationale, <https://www.python.org/dev/peps/pep-0635/>;
- PEP 636 – Structural Pattern Matching: Tutorial, <https://www.python.org/dev/peps/pep-0636/>;
- PEP 443 – Single-dispatch generic functions, <https://www.python.org/dev/peps/pep-0443/>;
- Python 3 Documentation, The Python Standard Library, `getattr`, <https://docs.python.org/3/library/functions.html#getattr>;
- Python 3 Documentation, The Python Standard Library, `functools.singledispatch`, <https://docs.python.org/3/library/functools.html#functools.singledispatch>;
- Wikipedia, “Collatz Conjecture”, https://en.wikipedia.org/wiki/Collatz_conjecture;
- WolframAlpha, “Rule 30”, <https://www.wolframalpha.com/input/?i=rule+30>;
- Wikipedia, “Rule 30”, https://en.wikipedia.org/wiki/Rule_30;

Watch out for recursion



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/watch-out-for-recursion>.)

Introduction

In this Pydon't I am going to talk a little bit about when and why recursion might not be the best strategy to solve a problem. This discussion will entail some particularities of Python, but will also cover broader topics and concepts that encompass many programming languages. After this brief discussion, I will show you some examples of recursive Python code and its non-recursive counterparts.

Despite what I said I'll do, don't take me wrong: the purpose of this Pydon't is *not* to make you dislike recursion or to say that recursion sucks. I *really* like recursion and I find it very elegant.

Watch out for recursion

Now that you know what is the purpose of this Pydon't, let me mention some things that can influence the suitability of recursion to solve problems.

RecursionError

The first thing we will discuss is the infamous recursion depth limit that Python enforces.

If you have no idea what I am talking about, then either - you never wrote a recursive function in your life, or - you are really, *really* good and never made a mistake in your recursive function definitions.

The recursion depth limit is something that makes your code raise a `RecursionError` if you make too many recursive calls. To see what I am talking about, just do the following in your REPL:

```
>>> def f():
...     return f()
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in f
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
>>>
```

In many cases, this limit *helps*, because it helps you find recursive functions for which you did not define the base case properly.

There are, however, cases in which 1000 recursive calls isn't enough to finish your computations. A classical example is that of the factorial function:

```
>>> def fact(n):
...     if n == 0:
...         return 1
...     return n*fact(n-1)
...
```

```

>>> fact(10)
3628800
>>> fact(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in fact
  File "<stdin>", line 5, in fact
  File "<stdin>", line 5, in fact
  [Previous line repeated 995 more times]
  File "<stdin>", line 2, in fact
RecursionError: maximum recursion depth exceeded in comparison

```

Our function is properly defined but by default Python does not allow us to make sufficient recursive calls.

If you must, you can always set your own recursion depth:

```

>>> import sys
>>> sys.setrecursionlimit(3000)
>>> fact(2000)
33162... # (omitted for brevity)
>>> sys.getrecursionlimit()
3000

```

Just be careful with it. I never tried, but you are likely not to be interested in having Python run out of memory because of your obscenely large amount of recursive calls.

Hence, if your function is such that it will be constantly trying to recurse more than the recursion depth allowed, you might want to consider a different solution to your problem.

No tail recursion elimination

In some programming languages, the factorial function shown above could be tweaked – so as to perform a tail call – and that would prevent some problems while saving memory: tail calls happen when the recursive call is the very last thing that is done inside the function, which more or less means that you do not need to keep any information whatsoever about the context you are in when you recurse.

In the factorial function above, after recursing with `fact(n-1)` we still have to perform a multiplication before returning from the function. If we rewrote the function to carry the partial factorial as an accumulator, we could have a factorial function that performs tail calls:

```

>>> def fact(n, partial=1):
...     if n <= 1:
...         return partial
...     return fact(n-1, n*partial)
...
>>> fact(10)
3628800

```

As you can see, the very last thing done inside the `fact` function is to call itself, so in theory Python could “forget everything about its surroundings” when making the recursive call, and save a lot of memory in the

process.

In practice, Python does not do this *intentionally*, and I refer you to the two articles on the [Neopythonic](#) blog (by Guido van Rossum) in the references to read more on why Python does not have such a feature.

Converting recursive functions into tail recursive functions is an interesting exercise and I challenge you to do so, but you won't get speed gains for it. However, it is very easy to remove the recursion of a tail recursive function, and I will show you how to do it in the [examples below](#).

Branching overlap

Another thing to take into account when considering a recursive solution to a problem is: is there going to be much overlap in the recursive calls?

If your recursive function branches in its recursive calls *and* the recursive calls overlap, then you may be wasting plenty of time recalculating the same values over and over again. More often than not this can be fixed easily, but just because a problem *probably* has a simple solution, it doesn't mean you can outright ignore it.

A classical example of recursion that leads to plenty of wasted computations is the Fibonacci sequence example:

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

A simple modification to this function shows that there are *many* recursive calls being made:

```
call_count = 0
def fibonacci(n):
    global call_count
    call_count += 1
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
print(call_count)  # 177
```

If your function is more involved, then the time you waste on recalculations can become unbearable.

Depth-first versus breadth-first

Something else to take into consideration when writing recursive solutions to your problems is that recursive solutions are inherently depth-first in nature, whereas your problem might warrant a breadth-first solution.

This is unlikely to be a large concern, but it just goes to show that sometimes, even though a solution has a very clear recursive solution, you are better off with not implementing a purely-recursive solution.

A very good example of this distinction popped up [when I solved the water bucket riddle](#): I wanted to write code that solved (a more generic version of) that riddle where you have a bucket that can hold A litres, another one that holds B litres, and you have to move water around to get one of the buckets to hold exactly T litres. The solution can be easily expressed in recursive terms, but my implementation actually used a while loop and a BFS algorithm.

If you don't know what this means, the best thing to do is to google it. For example, visit the Wikipedia pages on [Depth-first Search](#) and [Breadth-first Search](#). In a short and imprecise sentence, Depth-First Search (DFS) means that when you are traversing some structure, you prioritise exploring in depth, and only then you look around, whereas in Breadth-First Search (BFS) you first explore the level you are at, and only then go a level deeper.

Examples in code

I will now show some recursive code that can incur in some of the problems mentioned above, and will also share non-recursive versions of those same pieces of code.

Factorials

The toy example of the factorial is great because it lends itself to countless different implementations, and the ideas that these implementations exhibit can then be adapted to more complex recursions.

The main characteristic here is that the recursion of the factorial is a “linear” recursion, where each call only performs a single recursive call, and each recursive call is for a simpler problem.

The vanilla recursion follows:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n-1)
```

Like we have seen above, we could use an accumulator to write a tail recursive version of the factorial, even though Python won't optimise that in any way:

```
def factorial(n, partial=1):
    if n <= 1:
        return partial
    return factorial(n-1, n*partial)
```

Now that we have this function written in a tail recursive way, we can actually remove the recursion altogether following a simple recipe:

```
def factorial(n):
    partial = 1
    while n > 1:
        n, partial = n-1, n*partial
    return partial
```

This is a generic transformation you can do for any tail recursive function and I'll present more examples below.

Still on the factorial, because this is a linear recursion (and a fairly simple one, yes), there are many ways in which this function can be rewritten. I present a couple, pretending for a second that `math.factorial` doesn't exist:

```
import math
def factorial(n):
    return math.prod(i for i in range(1, n+1))

import functools, operator
def factorial(n):
    return functools.reduce(operator.mul, [i for i in range(1, n+1)])

def factorial(n):
    fact = 1
    for i in range(1, n+1):
        fact *= i
    return fact
```

If you are solving a problem and come up with different solutions, don't be afraid to try them out.

More on tail recursion

Let me show you a couple of simple recursive functions, their tail recursive equivalents and then their non-recursive counterparts. I will show you the generic transformation, so that you too can rewrite any tail recursive function as an imperative one with ease.

List sum

You can implement your own `sum` recursively:

```
def sum(l):
    if not l:
        return 0
    return l[0] + sum(l[1:])
```

If you carry a partial sum down the recursive calls, you can make this tail recursive:

```
def sum(l, partial=0):
    if not l:
        return partial
    return sum(l[1:], l[0] + partial)
```

From the tail recursive function to the `while` solution is simple:

```
def sum(l):
    partial = 0
    while l:
```

```

    l, partial = l[1:], l[0] + partial
    return partial

```

Notice what happened: - the default value of the auxiliary variable becomes the first statement of the function; - you write a while loop whose condition is the complement of the base case condition; - you update your variables just like you did in the tail recursive call, except now you assign them explicitly; and - after the while you return the auxiliary variable.

Of course there are simpler implementations for the `sum`, the point here is that this transformation is *generic* and *always works*.

Sorting a list

Here is another example where we sort a list with selection sort. First, “regular” recursion:

```

def selection_sort(l):
    if not l:
        return []
    m = min(l)
    idx = l.index(m)
    return [m] + selection_sort(l[:idx]+l[idx+1:])

```

Now a tail recursive version:

```

def selection_sort(l, partial=None): # partial=[] is bad!
    if partial is None:
        partial = []
    if not l:
        return partial
    m = min(l)
    idx = l.index(m)
    selection_sort(l[:idx]+l[idx+1:], partial + [m])

```

In the above we just have to be careful with something: the default value of `partial` is supposed to be the empty list, but you should avoid mutable types in your arguments’ default values, so we go with `None` and then the very first thing we do is set `partial = []` in case it was `None`.

Finally, applying the recipe, we can remove the recursion:

```

def selection_sort(l):
    partial = []
    while l:
        m = min(l)
        idx = l.index(m)
        l, partial = l[:idx]+l[idx+1:], partial + [m]
    return partial

```

Traversing (a directory)

The Depth-first versus Breadth-first distinction is more likely to pop up when you have to *traverse* something.

In this example, we will traverse a full directory, printing file names and file sizes. A simple, purely recursive solution follows:

```
import pathlib

def print_file_sizes(path):
    """Print file sizes in a directory."""

    path_obj = pathlib.Path(path)
    if path_obj.is_file():
        print(path, path_obj.stat().st_size)
    else:
        for path in path_obj.glob("*"):
            print_file_sizes(path)
```

If you apply that function to a directory tree like this one,

```
- file1.txt
- subdir1
| - file2.txt
| - subdir2
| - file3.txt
| - subdir3
| - deep_file.txt
```

then the first file you will see printed is `deep_file.txt`, because this recursive solution traverses your file-system depth first. If you wanted to traverse the directory breadth-first, so that you first found `file1.txt`, then `file2.txt`, then `file3.txt`, and finally `deep_file.txt`, you could rewrite your function to look like the following:

```
import pathlib

def print_file_sizes(dir):
    """Print file sizes in a directory, recurse into subdirs."""

    paths_to_process = [dir]
    while paths_to_process:
        path, *paths_to_process = paths_to_process
        path_obj = pathlib.Path(path)
        if path_obj.is_file():
            print(path, path_obj.stat().st_size)
        else:
            paths_to_process += path_obj.glob("*")
```

This example that I took from my “[Truthy, Falsy, and bool](#)” Pydon’t uses the `paths_to_process` list to keep track of the, well, paths that still have to be processed, which mimics recursion without actually having to recurse.

Keeping branching in check

Overlaps

When your recursive function branches out a lot, and those branches overlap, you can save some computational effort by saving the values you computed so far. This can be as simple as having a dictionary inside which you check for known values and where you insert the base cases.

This technique is often called memoisation and will be covered in depth in a later Pydon't, so [stay tuned!](#)

```
call_count = 0

fibonacci_values = {0: 0, 1: 1}
def fibonacci(n):
    global call_count
    call_count += 1

    try:
        return fibonacci_values[n]
    except KeyError:
        fib = fibonacci(n-1) + fibonacci(n-2)
        fibonacci_values[n] = fib
        return fib

print(fibonacci(10))
print(call_count)  # 19
```

Notice that this reduced the recursive calls from 177 to 19. We can even count the number of times we have to perform calculations:

```
computation_count = 0

fibonacci_values = {0: 0, 1: 1}
def fibonacci(n):
    try:
        return fibonacci_values[n]
    except KeyError:
        global computation_count
        computation_count += 1
        fib = fibonacci(n-1) + fibonacci(n-2)
        fibonacci_values[n] = fib
        return fib

print(fibonacci(10))
print(computation_count)  # 9
```

This shows that saving partial results can really pay off!

Writing recursive branching as loops

To show you how you can rewrite a recursive, branching function as a function that uses `while` loops we will take a look at another sorting algorithm, called merge sort. The way merge sort works is simple: to sort a list, you start by sorting the first and last halves separately, and then you merge the two sorted halves.

Written recursively, this might look something like this:

```
def merge(l1, l2):
    result = []
    while l1 and l2:
        if l1[0] < l2[0]:
            h, *l1 = l1
        else:
            h, *l2 = l2
        result.append(h)

    result.extend(l1)  # One of the two lists is empty,
    result.extend(l2)  # the other contains the larger elements.
    return result

def merge_sort(l):
    """Sort a list recursively with the merge sort algorithm."""

    # Base case.
    if len(l) <= 1:
        return l
    # Sort first and last halves.
    m = len(l)//2
    l1, l2 = merge_sort(l[:m]), merge_sort(l[m:])
    # Now put them together.
    return merge(l1, l2)
```

If you don't want to have all this recursive branching, you can use a generic list to keep track of all the sublists that are still to be sorted:

```
def merge(l1, l2):
    """Merge two lists in order."""

    result = []
    while l1 and l2:
        if l1[0] < l2[0]:
            h, *l1 = l1
        else:
            h, *l2 = l2
        result.append(h)

    result.extend(l1)  # One of the two lists is empty,
```

```

        result.extend(l2)  # the other contains the larger elements.
    return result

def merge_sort(l):
    """Sort a list with the merge sort algorithm."""

    # Save all sorted sublists.
    already_sorted = []
    # Keep track of sublists that need sorting:
    to_sort = [l]
    while to_sort:
        # Pick a list to be sorted.
        lst, *to_sort = to_sort
        # Base case.
        if len(lst) <= 1:
            already_sorted.append(lst)
        else:
            # Split in halves to sort each half.
            m = len(lst) // 2
            to_sort.append(lst[:m])
            to_sort.append(lst[m:])

    # Merge all the sublists.
    while len(already_sorted) > 1:
        l1, l2, *already_sorted = already_sorted
        # Factored out the `merge` to keep this short.
        already_sorted.append(merge(l1, l2))

    return already_sorted[0]

```

If you don't really know what the `h, *l1 = l1, h, *l2 = l2, lst, *to_sort = to_sort` and `l1, l2, *already_sorted = already_sorted` lines are doing, you might want to have a look at [this Pydon't about unpacking with starred assignments](#).

In this particular example, *my* translation of the merge sort to a non-recursive solution ended up being noticeably larger than the recursive one. This just goes to show that you need to judge all situations by yourself: would this be worth it? Is there an imperative implementation that is better than this direct translation? The answers to these questions will always depend on the programmer and the context they are in.

This also shows that the way you *think* about the problem has an effect on the way the code looks: even though this last implementation is imperative, it is a direct translation of a recursive implementation and so it may not look as good as it could!

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

“Pydon’t recurse mindlessly.”

This Pydon’t showed you that:

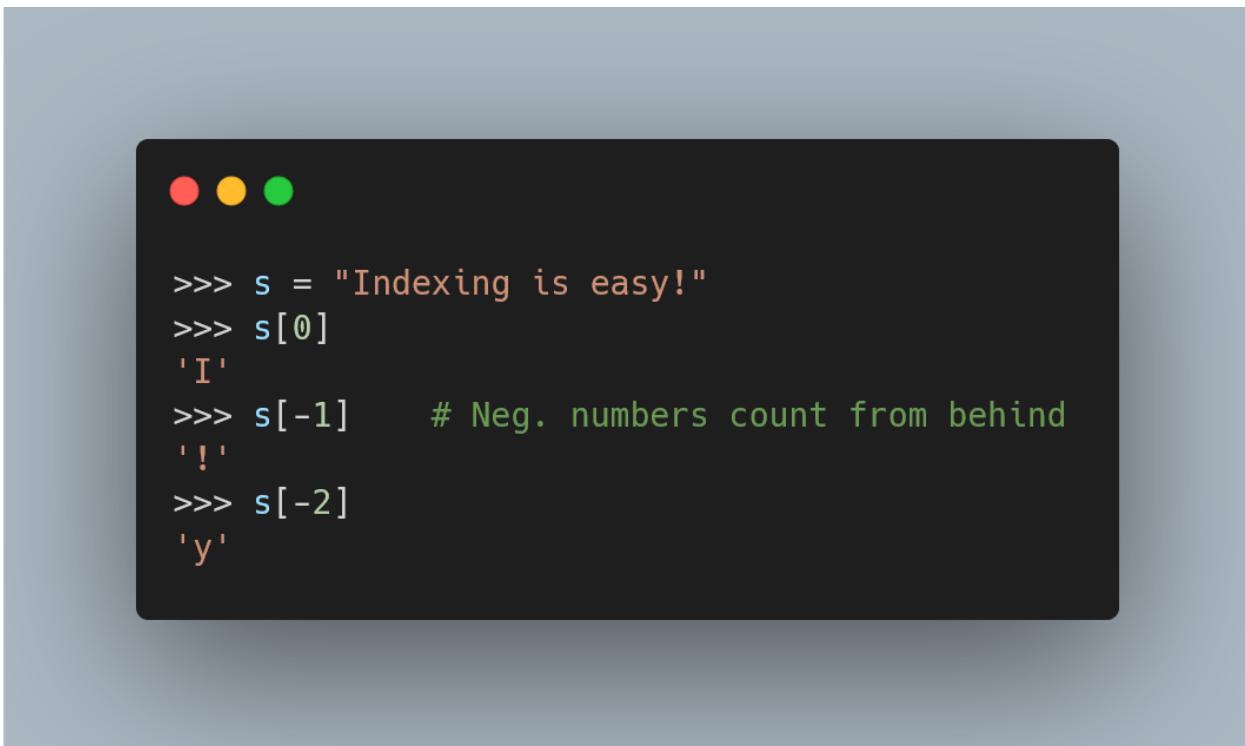
- Python has a hard limit on the number of recursive calls you can make and raises a `RecursionError` if you cross that limit;
- Python does not optimise tail recursive calls, and probably never will;
- tail recursive functions can easily be transformed into imperative functions;
- recursive functions that branch can waste a lot of computation if no care is taken;
- traversing something with pure recursion tends to create depth first traversals, which might not be the optimal way to solve your problem; and
- direct translation of recursive functions to imperative ones and vice-versa will probably produce sub-optimal code, so you need to align your mindset with what you want to accomplish.

References

- Stack Overflow, “What is the maximum recursion depth in Python, and how to increase it?”, <https://stackoverflow.com/questions/3323001/what-is-the-maximum-recursion-depth-in-python-and-how-to-increase-it>.
- Stack Overflow, “Does Python optimize tail recursion?”, <https://stackoverflow.com/questions/13591970/does-python-optimize-tail-recursion>.
- Neopythonic, Tail Recursion Elimination, <http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html>.
- Neopythonic, Final Words on Tail Calls, <http://neopythonic.blogspot.com/2009/04/final-words-on-tail-calls.html>.
- Documentation, The Python Standard Library, Functional Programming Modules, operator, <https://docs.python.org/3/library/operator.html>.

Online references last consulted on the 16th of February of 2021.

Sequence indexing



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/sequence-indexing.>)

Introduction

Sequences in Python, like strings, lists, and tuples, are objects that support indexing: a fairly simple operation that we can use to access specific elements. This short article will cover the basics of how sequence indexing works and then give you some tips regarding anti-patterns to avoid when using indices in your Python code.

In this article you will:

- learn the basic syntax for indexing sequences;
- learn how negative indices work;
- see some tools that are often used to work with sequences and indices;
- learn a couple of tricks and things to avoid when indexing;

Sequence indexing

First and foremost, I am talking about *sequence indexing* here to distinguish the type of indexing you do to access the values of a dictionary, where you use *keys* to index into the dictionary and retrieve its values. In this article we will be talking about using *integers* to index *linear sequences*, that is, sequences that we can traverse from one end to the other, in an ordered fashion.

A very simple example of such a sequence is a string:

```
>>> s = "Indexing is easy!"
>>> s
'Indexing is easy!'
```

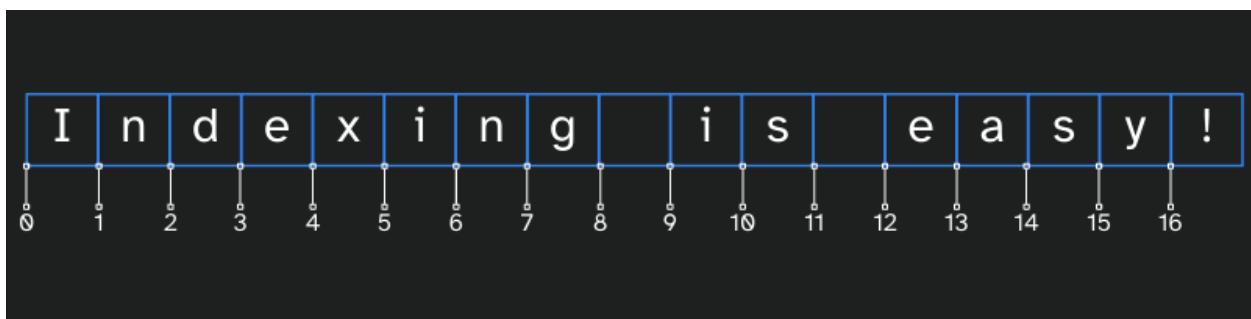
To index a specific character of this string I just use square brackets and the integer that corresponds to the character I want. Python is 0-indexed, which means it starts counting indices at 0. Therefore, the very first element of a sequence can be obtained with [0]. In our example, this should give a capital "I":

```
>>> s = "Indexing is easy!"
>>> s[0]
'I'
```

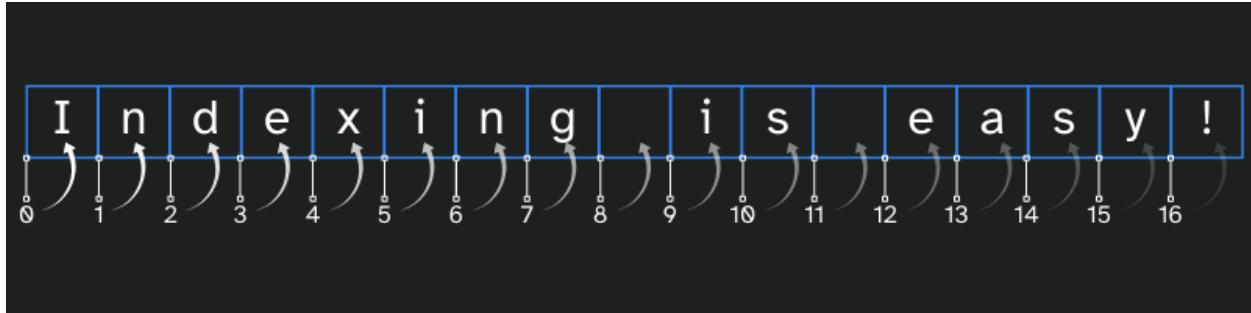
Then, each following character is obtained by increasing the index by 1:

```
>>> s = "Indexing is easy!"
>>> s[1]
'n'
>>> s[2]
'd'
>>> s[3]
'e'
>>> s[4]
'i'
>>> s[5]
'n'
>>> s[6]
'g'
>>> s[7]
'i'
>>> s[8]
's'
>>> s[9]
' '
>>> s[10]
'i'
>>> s[11]
's'
>>> s[12]
' '
>>> s[13]
'e'
>>> s[14]
'a'
>>> s[15]
's'
>>> s[16]
'y'
>>> s[17]
'!'
```

Here is a figure that shows how to look at a sequence and figure out which index corresponds to each element:



Imagine vertical bars that separate consecutive elements, and then number each of those vertical bars, starting with the leftmost bar. Each element gets the index associated with the bar immediately to its left:



Maximum legal index and index errors

Because indices start at 0, the last legal index to a sequence is the index that is equal to the length of the sequence, *minus one*:

```
>>> s = "Indexing is easy!"  
>>> len(s)  
17  
>>> s[16]  
'!'  
>>> s[17]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range
```

As you can see above, if you use an index that is too large (read: greater than or equal to the length of the sequence) Python will raise an `IndexError`, warning you about your usage of an integer that is too large for that specific indexing operation.

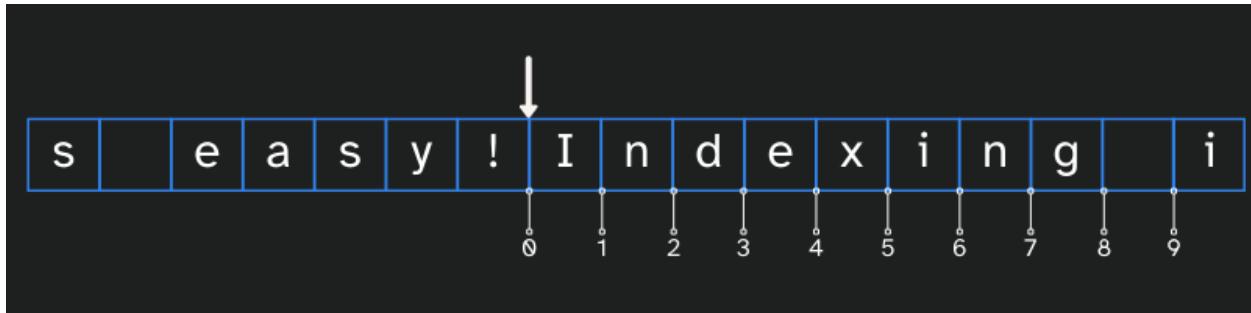
Negative indices

If the last legal index is the length of the sequence minus 1, then there is an obvious way to access the last item of a sequence:

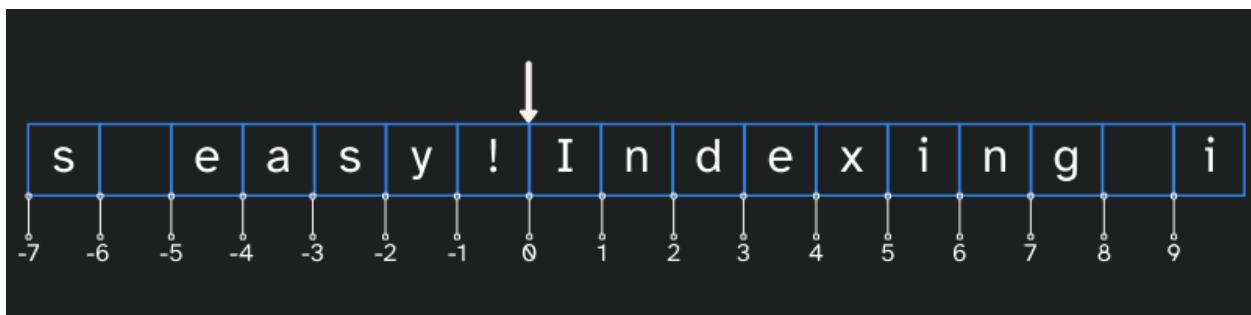
```
>>> s = "Indexing is easy!"  
>>> s[len(s)-1]  
'!'  
>>> l = [12, 45, 11, 89, 0, 99]  
>>> l[len(l)-1]  
99
```

However, Python provides this really interesting feature where you can use negative indices to count from the end of the sequence. In order to figure out which negative index corresponds to which element, think

about writing the sequence to the left of itself:



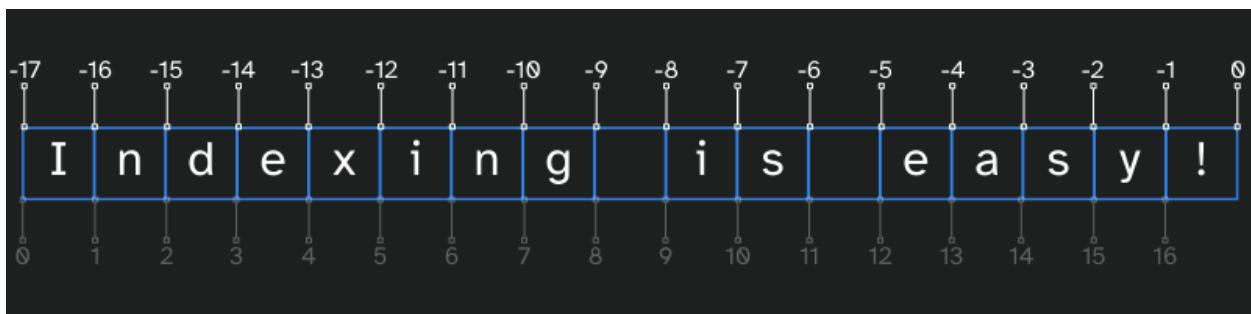
Then you just have to continue the numbering from the right to the left, therefore making use of negative numbers:



From the figure above you can see that the index -1 refers to the last element of the sequence, the index -2 refers to the second to last, etc:

```
>>> s = "Indexing is easy!"  
>>> s[-1]  
'!'  
>>> s[-2]  
'y'
```

We can also take a look at all the negative indices that work for our specific sequence:



Another way to look at negative indices is to pretend there is a `len(s)` to their left:

Negative index	Corresponding positive index
-1	<code>len(s) - 1</code>
-2	<code>len(s) - 2</code>
-3	<code>len(s) - 3</code>
...	...
<code>-len(s)</code>	<code>len(s) - len(s)</code> (same as 0)

And a couple of examples:

```
>>> s = "Indexing is easy!"
>>> s[-5]
'e'
>>> s[len(s)-5]
'e'
>>> s[-13]
'x'
>>> s[len(s)-13]
'x'
>>> len(s)
17
>>> s[-17]
'I'
>>> s[len(s)-17]
'I'
```

Indexing idioms

Having seen the basic syntax for indexing, there are a couple of indices that would be helpful if you were able to read them immediately for what they are, without having to think about them:

Index operation	Interpretation
<code>s[0]</code>	First element of <code>s</code>
<code>s[1]</code>	Second element of <code>s</code>
<code>s[-1]</code>	Last element of <code>s</code>
<code>s[-2]</code>	Second to last element of <code>s</code>

To index or not to index?

Just a quick note on something that I trip over every now and then.

Python has many useful built-ins and built-in data types. Of them, strings, lists and tuples are indexable with integers. Sets are not.

You should also be careful about things that you *think* are like lists, but really are *not*. These include enumerate, zip, map, and other objects. None of these are indexable, none of these have a len value, etc. Pay attention to that!

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> e = enumerate(l)
>>> e[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'enumerate' object is not subscriptable
>>> z = zip(l)
>>> z[3]
## ...
TypeError: 'zip' object is not subscriptable
>>> m = map(str, l)
>>> m[3]
## ...
TypeError: 'map' object is not subscriptable
```

Best practices in code

A looping pattern with range

Because of the way both range and indices work, one can understand that range(len(s)) will generate all the legal indices for s:

```
>>> s = "Indexing is easy!"
>>> list(range(len(s)))      # use list() to print the values
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>> s[0]
'I'
>>> s[16]
'!'
```

A consequence of this is that beginners, and people who are more distracted or used to other programming languages, end up employing a very common anti-pattern in for loops.

To exemplify this, suppose we wanted to write a fairly naïve program to find the unique letters in our string. Here is what the anti-pattern would look like:

```
>>> s = "Indexing is easy!"
>>> uniques = []
>>> for idx in range(len(s)):
...     if s[idx] not in uniques:
...         uniques.append(s[idx])
...
>>> uniques
['I', 'n', 'd', 'e', 'x', 'i', 'g', ' ', 's', 'a', 'y', '!']
```

This is a naïve solution to the problem of “find unique characters”, you probably want to use a Python set for a more efficient implementation :)

The problem here is that the `for` loop is being done in a roundabout way: we have access to a sequence (the string) that we could *iterate over*, but instead we find its length, so that we can use `range` to compute its legal indices, which we then *iterate over*, only to then access the elements of the sequence through their indices.

This way of writing `for` loops is similar to the way one would write `for` loops in other programming languages, if you were to iterate over the elements of an array.

However, we are using Python, not any other language. One of the things I enjoy the most about Python’s `for` loops is that you can access directly the consecutive elements of a sequence. Hence, we can actually rewrite our `for` loop slightly, but in a way that makes it much more elegant:

```
>>> s = "Indexing is easy!"  
>>> uniques = []  
>>> for letter in s:  
...     if letter not in uniques:  
...         uniques.append(letter)  
...  
>>> uniques  
['I', 'n', 'd', 'e', 'x', 'i', 'g', ' ', 's', 'a', 'y', '!']
```

What I really like about these types of loops is that if your variables are named correctly, the statements express your intent very clearly. The line `for letter in s:` is read as

“For each letter in (the string) `s`...”

This type of `for` loop iterates directly over the values you care about, which is often what you want. If you care about the indices, then be my guest and use `range(len(s))!`

Another anti-pattern to be on the lookout for happens when you need to work with the indices *and* the values. In that case, you probably want to use the `enumerate` function. I tell you all about that function [in a Pydon’t of its own](#), so go check that if you haven’t.

Large expressions as indices

When you are dealing with sequences and with indices for those sequences, you may end up needing to perform some calculations to compute new indices that interest you. For example, suppose you want the middle element of a string and you don’t know about `//` yet:

```
>>> s = "Indexing is easy!"  
>>> s[len(s)/2]      # len(s)/2 isn't an integer!!  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: string indices must be integers  
>>> len(s)/2  
8.5  
>>> import math  
>>> s[math.floor(len(s)/2)]
```

```
' '
>>> s[len(s)//2]      # Pro-tip: the operation // is ideal here
'
```

Where am I going with this?

Take a look at the expression you just used:

```
s[math.floor(len(s)/2)]
```

Maybe it is me getting old, but I struggle a bit to read that because of the [] enclosing the expression which then has a couple of () that I also have to parse, to figure out what goes where.

If you have *large* expressions to compute indices (and here, *large* will be subjective), inserting those expressions directly inside [] may lead to long lines of code that are then complicated to read and understand. If you have lines that are hard to understand, then you probably need to comment them, creating even more lines of code.

Another alternative is to create a *well-named* variable to hold the result of the computation of the new index:

```
>>> s = "Indexing is easy!"
>>> mid_char_idx = math.floor(len(s)/2)
>>> s[mid_char_idx]
'
```

For this silly example, notice that the new variable name is almost as long as the expression itself! *However*, s[mid_char_idx] is very, very, easy to read and does not need any further comments.

So, if you have large expressions to compute indices, think twice before using them to index directly into the sequence at hands and consider using an intermediate variable with a descriptive name.

Unpacking with indexing

You will find yourself often working with small groups of data, for example pairs of things that you keep together in a small list for ease of use. For example, the first and last names of a person:

```
>>> names = ["Mary", "Doe"]
```

Now you have this little function that creates a formal or informal greeting for a given name:

```
>>> names = ["Mary", "Doe"]
>>> def greet(names, formal):
...     if formal:
...         return "Hello Miss " + names[1]
...     else:
...         return "Hey there " + names[0]
...
>>> greet(names, True)
'Hello Miss Doe'
>>> greet(names, False)
'Hey there Mary'
```

Something you might consider and that adds a bit of clarity to your code is unpacking the `names` before you reach the `if` statement:

```
def greet(names, formal):
    first, last = names
    if formal:
        return "Hello Miss " + last
    else:
        return "Hey there " + first
```

Why would this be preferable, if I just added a line of code? It makes the *intent* of the code *much* more obvious. Just from looking at the function as is, you can see from the first line `first, last = names` that `names` is supposed to be a pair with the first and last names of a person and then the `if: ... else: ...` is very, very easy to follow because we see *immediately* that we want to use the last name if we need a formal greeting, and otherwise (`else`) we use the first name.

Furthermore, the action of unpacking (like so:)

```
first, last = names
```

forces your `greet` function to expect pairs as the `names` variable, because a list with less or more elements will raise an error:

```
>>> first, last = ["Mary", "Anne", "Doe"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

We are assuming we really are working with *pairs*, so if the `greet` function gets something that is not a pair, this error is useful in spotting a problem in our code. Maybe someone didn't understand how to use the function and called it with the first name of the person?

```
>>> greet("Mary", True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in greet
ValueError: too many values to unpack (expected 2)
```

This would help you find a location where the `greet` function was not being properly used.

I have written at length about unpacking in Python (another favourite feature of mine!) so feel free to read my articles on [unpacking with starred assignments](#) and on [deep-unpacking](#).

Conclusion

Here's the main takeaway of this article, for you, on a silver platter:

"Indexing is simple and powerful, but sometimes when indexing looks like the answer, there is another Python feature waiting to be used."

This Pydon't showed you that:

- Indexing in Python is 0-based;
- Python allows negative indices in sequences;
- Using indices in a `for` loop to access the elements of a sequence is an anti-pattern in Python;
- Using large expressions when indexing bloats your code and you are better off with a descriptive variable, even if that variable has a long name;
- If you know the exact structure of the sequence you are dealing with, unpacking might be preferable to indexing.

Idiomatic sequence slicing

```
● ● ●

>>> c = [0, 1, 2, 3]
# The usage of the slice `[:]` makes a copy of `c`,
# while `a = b` means that `a` and `b` will be the same object.
>>> a = b = c[::]
>>> print(a is b, a is c, b is c)
True False False
>>> a.append(4)
>>> a, b, c
([0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3])
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydons/idiomatic-sequence-slicing>.)

Introduction

Last time we went over **sequence indexing** in Python to cover the basics for our next topic of discussion: sequence slicing. Slicing is a “more advanced” way of accessing portions of sequences (like lists and tuples). I say it is more advanced just because indexing is the simplest form of accessing sequence items; as you will see, indexing isn’t that complicated either.

As it turns out, much can be said about sequence slicing, so I will split all of the contents into two Pydon’ts, this and the next one.

In this Pydon’t you will:

- learn the slicing syntax;
- learn how slicing works with 1 and 2 parameters;
- relate slices to the `range` built-in;
- master slicing with negative indices;
- learn to write Pythonic and idiomatic slices; and
- a couple of common use cases where slicing is *not* the way to go.

In the next Pydon't we will continue on this train of thought and cover the more advanced material related to slicing. In particular, you will

- learn about the stride parameter in slicing;
- learn about slice assignment;
- see how slicing can be used to copy sequences;
- learn some more idiomatic slicing patterns;
- uncover the two layers of syntactic sugar surrounding list slicing; and
- learn how to implement slicing for your custom objects.

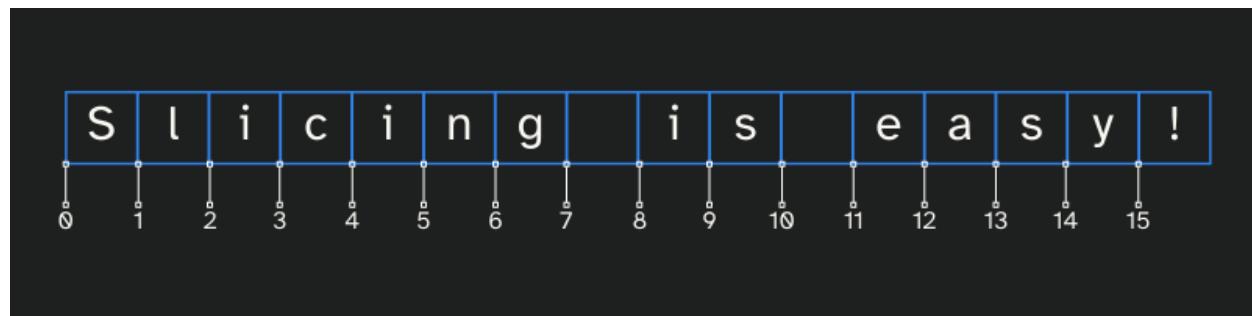
Throughout both Pydon'ts we will try to keep an eye out for how slices are actually used in real-world Python code, namely in the Python Standard Library.

If you don't want to miss the next Pydon't on the more advanced slicing topics, you can either [subscribe](#) to the Pydon'ts newsletter or grab your copy of the [Pydon'ts book](#) right now.

Slicing syntax

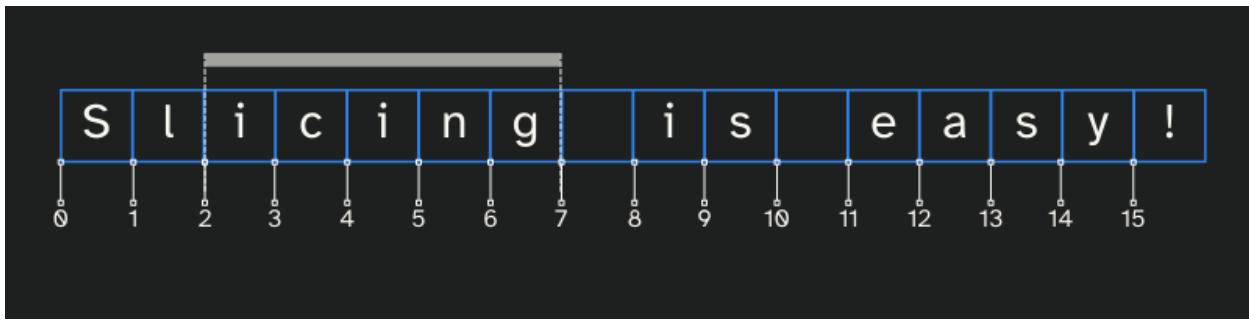
Slicing in Python is the act of accessing a sequence of elements that are extracted from successive positions of a larger sequence. Just think of an actual knife cutting through the sequence you are working with (which could be a string, list, tuple, etc) and extracting a smaller piece of your sequence.

For example, if we were working with the string "Slicing is easy!", which I present below.



Together with the characters of the string, we have the little numbers indicating the index of each character. Each little number gives the index for the box right in front of it. This is the representation I go to in my head whenever I have to reason about indices in Python, especially when I am working with negative indices. (Just take a quick look at [this Pydon't](#) if you need to jog your memory on how indexing is done in Python.)

Now, we could be interested in extracting the portion "icing" from the string:



How would we do that in Python? If you didn't know how slicing worked, you could come up with a solution involving a `for` loop and a `range`:

```
>>> s = "Slicing is easy!"
>>> subs = ""
>>> for i in range(2, 7):
...     subs += s[i]
...
>>> subs
'icing'
```

This is all good, but there is a much shorter syntax for this type of operation, the **slicing syntax**.

When you want to slice a sequence, you need to use brackets `[]` and a colon `:` to separate the start and end points. The key here is in figuring out what the start and end points are, but that is just a matter of looking at the figure above *or* at the solution with the `range(2, 7)`:

```
>>> s = "Slicing is easy!"
>>> s[2:7]
'icing'
```

This is the very first important point to make about slicing: the start and end points give you the bars that enclose what you will extract, which, in other words, means that the start point (2, in the previous example) is the index of the first element that *is included* in the slice, whereas the end point is the index of the first element that *is not included* in the slice:

```
>>> s = "Slicing is easy!"
>>> s[2:7]
'icing'
>>> s[7]
'
```

Now is a good time to fire up your Python interpreter, define `s` as the string `"Slicing is easy!"`, and work out a couple of slices for yourself.

What to slice?

Just in case it wasn't clear earlier, here are just some of the things that you can slice in Python:

```

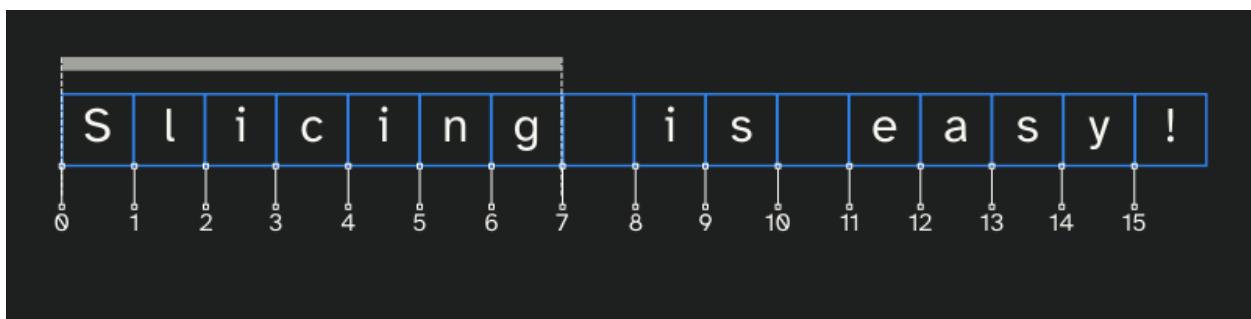
>>> "Hello"[1:3]                      # strings
'el'
>>> [True, False, 1, "hey"][1:3]       # lists
[False, 1]
>>> (True, False, 1, "hey")[1:3]       # tuples
(False, 1)
>>> range(10)[1:3]                   # ranges
range(1, 3)
>>> # etc...

```

However, we will be using string examples for most of the Pydon't, just for the sake of consistency.

Slicing from the beginning

Now assume that we wanted to extract "Slicing" from our string.



If we go back to our naïve range solution, most of us would write the following:

```

>>> s = "Slicing is easy!"
>>> subs = ""
>>> for i in range(7):
...     subs += s[i]
...
>>> subs
'Slicing'

```

Notice that, unlike when we used `range(2, 7)` for "icing", now our `range` only has one argument, the end point. That is because `range` interprets the missing starting index as 0.

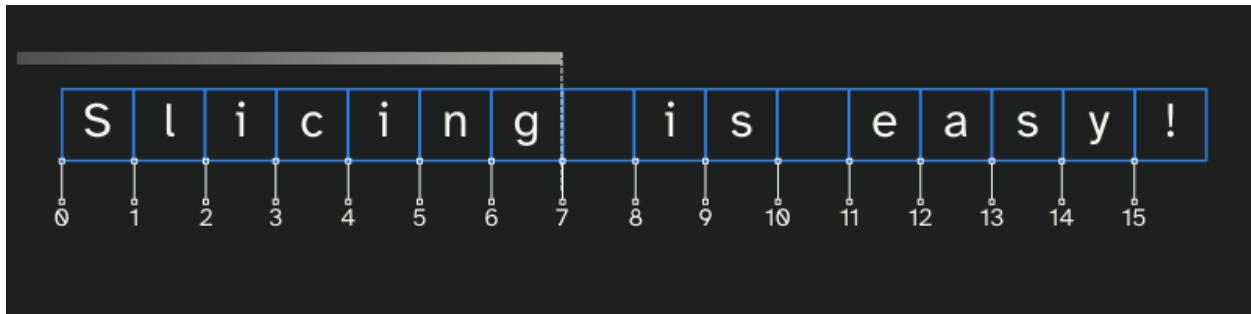
When we are slicing, we can do a similar thing! If we want to extract a portion from the beginning of a sequence, the Pythonic way of writing that slice is *without* specifying the explicit 0 as a start point. Therefore, both alternatives below work, but the second one is the preferred.

```

>>> s = "Slicing is easy!"
>>> s[0:7]      # Works ...
'Slicing'
>>> s[:7]       # ... but this is preferred!
'Slicing'

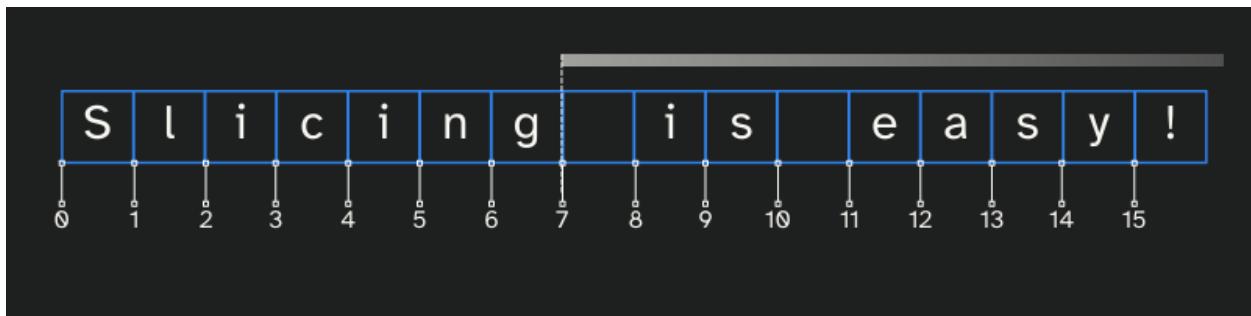
```

In terms of the figures I have been sharing, think of it like this: it's like you never tell Python where the slicing starts, so the bar that is hovering the string ends up covering the whole beginning of the string, stopping at the position you indicate.



Slicing until the end

Similarly to omitting the start point, you can omit the end point of your slice. To predict what will happen if we do so, we just have to look at the figure above and create a new one with the bar pointing in the other direction:



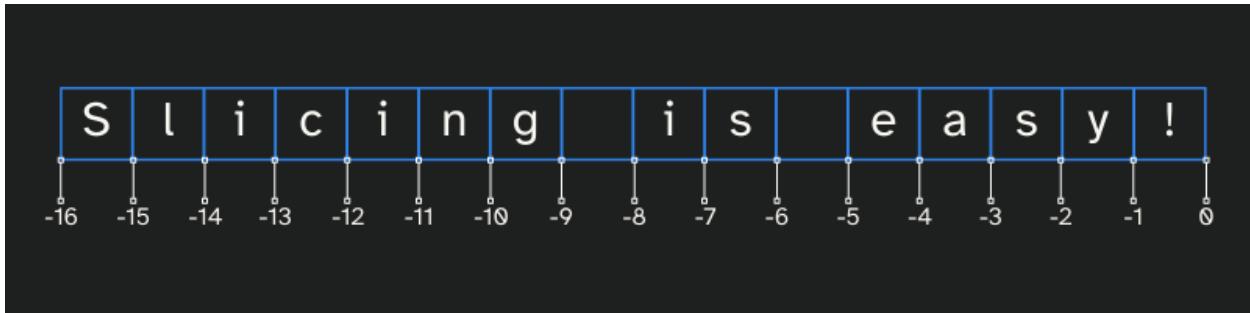
Therefore, if we don't indicate the end point for the slice, we extract all elements from the point specified, onwards. Naturally, we can specify the end point of the slice to be the length of the sequence, but that adds too much visual noise:

```
>>> s = "Slicing is easy!"  
>>> s[7:len(s)]      # Works...  
' is easy!'  
>>> s[7:]            # ... but this is preferred!  
' is easy!'
```

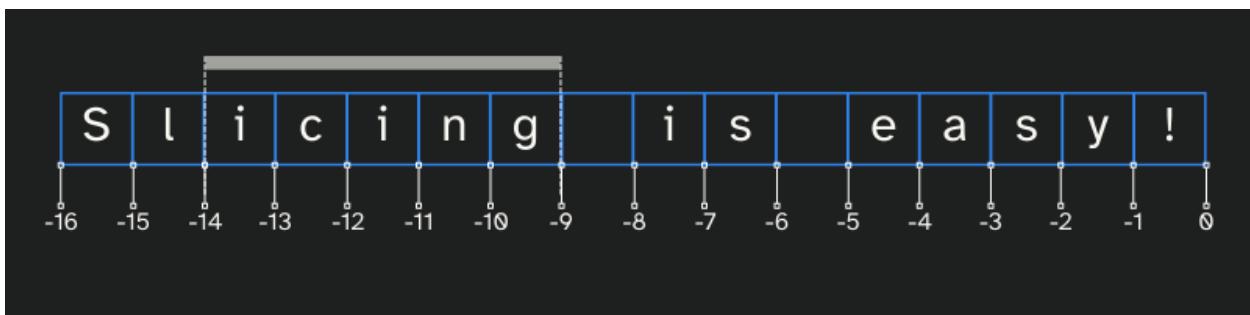
Slicing with negative indices

If you've read my previous Pydon't, you will have seen how indexing works with negative indices. Slicing can also use them, and the logic remains the same: we just draw the bar that selects the elements that are between the relevant indices.

To illustrate this, here is the representation of the negative indices of the string we have been using so far:



Now, regardless of the fact that the numbers are negative, if you had to tell me where to draw two vertical bars in order to enclose the substring "icing", what positions would you point to? You would probably tell me "Draw the bars on positions -14 and -9", and that would be absolutely correct!



In fact, using -14 and -9 would work in my naïve `range` solution but also – and most importantly – with the `slice` syntax:

```
>>> s = "Slicing is easy!"  
>>> subs = ""  
>>> for i in range(-14, -9):  
...     subs += s[i]  
...  
>>> subs  
'icing'  
>>> s[-14:-9]      # Also works and is preferred!  
'icing'
```

Slicing and `range`

At this point you should start to notice a pattern, and that is that the parameters you insert in your slices seem to be governing the *indices* that Python uses to fetch elements from your sequence, if those indices were generated with the `range` function. If you are looking at a slice and you have no clue what items are going to be picked up by it, try thinking about the slice in this way, with the `range`. It might help you.

Idiomatic slicing patterns

Now that you have taken a look at some basic slicing with positive and negative indices, and now that you know you can omit the first or the last parameters of your slices, you should really learn about *four* different slice patterns that are really idiomatic. Don't worry, I'll show you which *four* patterns I am talking about.

Suppose you have a variable `n` that is a positive integer (it may help to think of it as a small integer, like 1 or 2), and suppose `s` is some sequence that supports slicing. Here are the four idiomatic slicing patterns I am talking about:

- `s[n:]`
- `s[-n:]`
- `s[:n]`
- `s[:-n]`

Why are these “idiomatic” slicing patterns? These are idiomatic because, with a little practice, you stop looking at them as “slice `s` starting at *position blah* and ending at *position blah blah*”, and you will start looking at them for their semantic meaning.

Open your Python interpreter, set `s = "Slicing is easy!"` and `n = 2`, and see what the four slices above return. Experiment with other values of `n`. Can you give an interpretation for what each slice means?

Go ahead...

Here is what the slicing patterns mean.

`s[n:]`

If `n` is not negative (so 0 or more), then `s[n:]` means “skip the first `n` elements of `s`”:

```
>>> s = "Slicing is easy!"  
>>> s[2:]  
'icing is easy!'  
>>> s[3:]  
'cing is easy!'  
>>> s[4:]  
'ing is easy!'
```

`s[-n:]`

If `n` is **positive** (so 1 or more), then `s[-n:]` means “the last `n` elements of `s`”:

```
>>> s = "Slicing is easy!"  
>>> s[-2:]  
'y!'  
>>> s[-3:]  
'sy!'  
>>> s[-4:]  
'asy!'
```

Be careful with $n = 0$, because $-0 == 0$ and that means we are actually using the *previous* slicing pattern, which means “skip the first n characters”, which means we skip nothing and return the whole sequence:

```
>>> s = "Slicing is easy!"  
>>> s[-0:]  
'Slicing is easy!'
```

`s[:n]`

If n is not negative (so 0 or more), then $s[:n]$ can be read as “the first n elements of s ”:

```
>>> s = "Slicing is easy!"  
>>> s[:2]  
'Sl'  
>>> s[:3]  
'Sli'  
>>> s[:4]  
'Slic'
```

`s[:-n]`

Finally, if n is positive (so 1 or more), then $s[:-n]$ means “drop the last n elements of s ”:

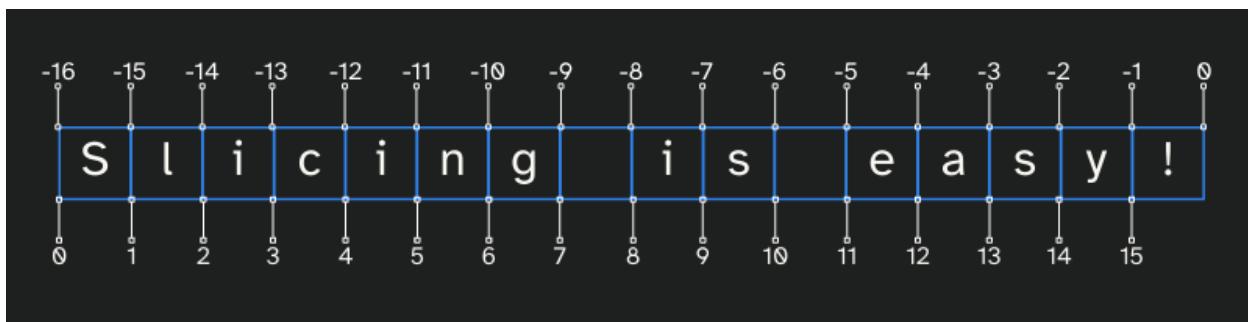
```
>>> s = "Slicing is easy!"  
>>> s[:-2]  
'Slicing is eas'  
>>> s[:-3]  
'Slicing is ea'  
>>> s[:-4]  
'Slicing is e'
```

Like with the $s[-n:]$ pattern, we need to be careful with $n = 0$, as the idiom $s[:-n]$ doesn't really apply, and we should be looking at the previous idiom.

Empty slices

Something worthy of note that may confuse some but not others, is the fact that if you get your start and end points mixed up, you will end up with empty slices, because your start point is to the right of the end point... And because of negative indices, it is *not* enough to check if the start point is less than the end point.

Take a look at the figure below:



Now try to work out why all of these slices are empty:

```
>>> s = "Slicing is easy!"
>>> s[10:5]
// 
>>> s[-6:-10]
// 
>>> s[-9:3]
// 
>>> s[10:-10]
//
```

All it takes is looking at the figure above, and realising that the end point is relative to an index that is *to the left* of the start point.

More empty slices

Another really interesting thing about slicing is that if you use numbers that are *too high* or *too low*, slicing still works and doesn't raise `IndexError` or something like that. In a way, this makes sense, and goes in line with the interpretation of the idioms we presented above.

If `s[50:]` is "skip the first 50 elements of `s`", and if `s` only has 16 elements, how many will there be left? Zero, naturally, so it is no surprise that `s[50:]` returns an empty string when `s = "Slicing is easy!"`:

```
>>> s = "Slicing is easy!"
>>> s[50:]
//
```

This segues nicely into the first common usage pattern of slices.

Examples in code

Ensuring at most `n` elements

Imagine someone is writing a spellchecker, and they have a function that takes a misspelled word and returns the top 5 closest suggestions for what the user meant to type.

Here is what that function could look like:

```
def compute_topSuggestions(misspelled, k, corpus):
    similar = find_similar(misspelled, corpus)
    ordered = rankSuggestions_by_similarity(misspelled, similar)
    top_k = []
    for i in range(min(k, len(ordered))):  
        top_k.append(ordered[i])
    return top_k
```

The final loop there is to make sure you return *at most k results*. However, the person who wrote this piece of code did not read this Pydon't! Because if they had, they would know that you can use slicing to extract *at most k elements* from ordered:

```
def compute_topSuggestions(misspelled, k, corpus):
    similar = find_similar(misspelled, corpus)
    ordered = rankSuggestions_by_similarity(misspelled, similar)
    return ordered[:k]
# ^ Idiom! Read as "return at most `k` from beginning"
```

A very similar usage pattern arises when you want to return at most k from the end, but you already knew that because you read about the four slicing idioms I shared earlier.

This usage pattern of slicing can show up in many ways, as this is just us employing slicing because of the semantic meaning this particular idiom has. Above, we have seen four different idioms, so just keep those in mind with working with sequences!

Start of a string

Slicing is great, I hope I already convinced you of that, but slicing is not the answer to all of your problems!

A common use case for slices is to check if a given sequence starts with a predefined set of values. For example, we might want to know if a string starts with the four characters ">>> ", which are the characters that mark the REPL Python prompt. The doctest Python module, for example, does a similar check, so we will be able to compare our solution to doctest's.

You just learned about slicing and you know that s[:4] can be read idiomatically as “the first four characters of s”, so maybe you would write something like

```
def check_prompt(line):
    if line[:4] == ">>> ":
        return True
    return False
```

or, much more elegantly,

```
def check_prompt(line):
    return line[:4] == ">>> "
```

However, it is important to note that this is not the best solution possible, because Python strings have an appropriate method for this type of check: the `startswith` function.

Therefore, the best solution would be

```
def check_prompt(line):
    return line.startswith("">>>> ")
```

This is better because this is a tested and trusted function that does exactly what you need, so the code expresses very clearly what you want. What is more, if you later change the prompt, you don't need to remember to also change the index used in the slice.

If we take a look at the actual source code for doctest, what they write is

```
## Inside _check_prefix from Lib/doctest.py for Python 3.9.2
## ...
if line and not line.startswith(prefix):
    # ...
```

As we can see here, they are using the `startswith` method to see if `line` starts with the `prefix` given as argument.

Similar to `startswith`, strings also define an `endswith` method.

Removing prefixes and suffixes

Similar to the example from above, another common usage pattern for slices is to remove prefixes or suffixes from sequences, more notably from strings. For example, most of the code I present in these Pydon'ts starts with "`>>>`" because of the REPL prompt. How could I write a short function to strip a line of code of this prompt? I am really hyped about slicing, so obviously I'll do something like

```
>>> def strip_prompt(line):
...     if line.startswith("">>>> "):
...         return line[4:]
...     else:
...         return line
...
>>> strip_prompt("">>>> 3 + 3")
'3 + 3'
>>> strip_prompt("6")
'6'
```

Or even better, I might do it in a generic way and suppress the `else`:

```
>>> def strip_prefix(line, prefix):
...     if line.startswith(prefix)
...         return line[len(prefix):]
...     return line
...
>>> prompt = "">>>> "
>>> strip_prefix("">>>> 3 + 3", prompt)
'3 + 3'
>>> strip_prefix("6", prompt)
'6'
```

However, I already have Python 3.9 installed on my machine, so I *should* be using the string methods that Python provides me with:

```
def strip_prefix(line, prefix):
    return line.removeprefix(prefix)
```

Of course, at this point, defining my own function is redundant and I would just go with

```
>>> prompt = ">>> "
>>> ">>> 3 + 3".removeprefix(prompt)
'3 + 3'
>>> "6".removeprefix(prompt)
'6'
```

In case you are interested, Python 3.9 also added a `removesuffix` method that does the analogous, but at the end of strings.

This just goes to show that it is nice to try and stay more or less on top of the features that get added to your favourite/most used programming languages. Also (!), this also shows that one has to be careful when looking for code snippets online, e.g. on StackOverflow. StackOverflow has amazing answers... that get outdated, so always pay attention to the most voted answers, but also the most recent ones, those could contain the more modern approaches.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"The relationship between slicing and indexing means there are four really nice idiomatic usages of slices that are well-worth knowing."

This Pydon't showed you that:

- slicing sequences lets you access series of consecutive elements;
- you can slice strings, lists, tuples, ranges, and more;
- if the start parameter is omitted, the slice starts from the beginning of the sequence;
- if the end parameter is omitted, the slice ends at the end of the sequence;
- slicing is the same as selecting elements with a `for` loop and a `range` with the same parameters;
- much like with plain indexing, negative integers can be used and those count from the end of the sequence;
- `s[n:]`, `s[-n:]`, `s[:n]`, and `s[:-n]` are four idiomatic slicing patterns that have a clear semantic meaning:
 - `s[n:]` is "skip the first n elements of `s`";
 - `s[-n:]` is "the last n elements of `s`";
 - `s[:n]` is "the first n elements of `s`";
 - `s[:-n]` is "skip the last n elements of `s`";
- slices with parameters that are too large produce empty sequences;
- if the parameters are in the wrong order, empty sequences are produced; and
- some operations that seem to ask for slicing might have better alternatives, for example using `startswith`, `endswith`, `removeprefix`, and `removesuffix` with strings.

References

- Python 3 Documentation, The Python Language Reference, Expressions – Slicings, <https://docs.python.org/3/reference/expressions.html#slicings> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – The Standard Type Hierarchy, <https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – Emulating Container Types, <https://docs.python.org/3/reference/datamodel.html#emulating-container-types> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, slice, <https://docs.python.org/3/library/functions.html#slice> [last accessed 20-04-2021];
- “Effective Python – 90 Specific Ways to Write Better Python”; Slatkin, Brett; ISBN 9780134853987.

Mastering sequence slicing

```
● ● ●

>>> c = [0, 1, 2, 3]
# The usage of the slice `[:]` makes a copy of `c`,
# while `a = b` means that `a` and `b` will be the same object.
>>> a = b = c[::]
>>> print(a is b, a is c, b is c)
True False False
>>> a.append(4)
>>> a, b, c
([0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3])
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/mastering-sequence-slicing>.)

Introduction

In the previous Pydon't we looked at using sequence slicing to manipulate sequences, such as strings or lists. In this Pydon't we will continue on the subject of slicing sequences, but we will look at the more advanced topics. In particular, in this Pydon't you will

- learn about the step parameter in slicing;
- see how slicing can be used to copy sequences;
- learn about slice assignment;
- learn some more idiomatic slicing patterns;

As it turns out, there is *A LOT* to say about sequence slicing, so I will have to split this Python's yet again, and next time we will finish off the subject of slicing sequences with:

- uncovering the two layers of syntactic sugar surrounding sequence slicing; and
- seeing how to implement slicing for your custom objects.

Slicing step

The next stop in your journey to mastering slicing in Python is knowing about the lesser-used third parameter in the slice syntax: the step.

Just the step

The step is a third integer that you can add to your slicing syntax that allows you to pick *evenly spaced* elements from the sequence. If `s` is a sequence, then you've seen how to do slicing with the syntax `s[start:end]`, and the step comes after those two parameters: `s[start:end:step]`. The easiest way to understand how the step parameter works is by omitting the `start` and `end` parameters, like so: `s[::-step]`.

Here are a couple of examples:

```
>>> "a0b0c0d0"[::-2]
'abcd'
>>> "a00b00c00d00"[::-3]
'abcd'
>>> "a000b000c000d000"[::-4]
'abcd'
```

As you can see, in all of the examples above, we used the step parameter to skip all the zeroes in between the letters.

- When the step parameter is 2, you split the sequence in groups of 2 and pick the first element of each group.
- When the step parameter is 3, you split the sequence in groups of 3 and pick the first element of each group.
- When the step parameter is 4, you split the sequence in groups of 4 and pick the first element of each group.
- ...

You get the idea, right? Once again, notice that the step parameter in slicing is closely related to the third (optional) argument that the built-in `range` function accepts.

Start, stop and step

When you also specify start and stop positions, Python will first figure out the section of the sequence that is encompassed by the start and stop parameters, and only then it uses the step to pick the correct elements from the sequence.

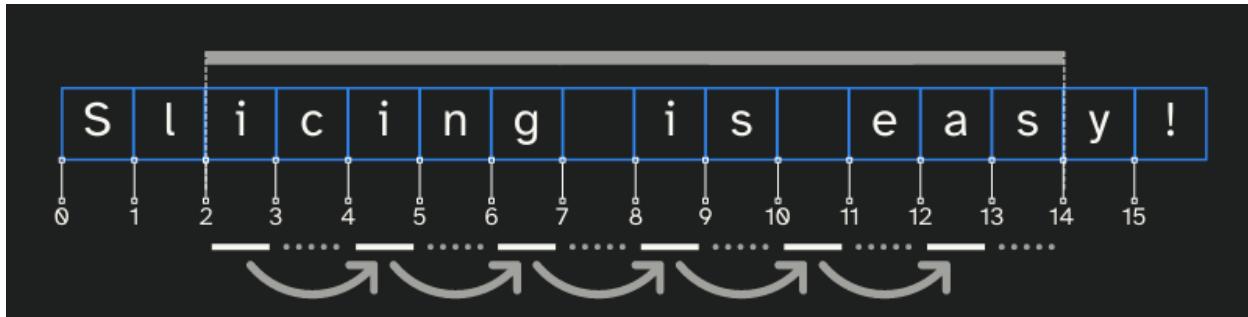
For example, can you explain yourself this result:

```
>>> s = 'Slicing is easy!'
>>> s[2:14:2]
'iigi a'
```

What happens first is that `s[2:14]` tells Python that we only want to work with a part of our original string:

```
>>> s = 'Slicing is easy!'
>>> s[2:14]
'icing is eas'
```

Then the step parameter kicks in and tells Python to only pick a few elements, like the figure below shows:



That is why we get the result that we got:

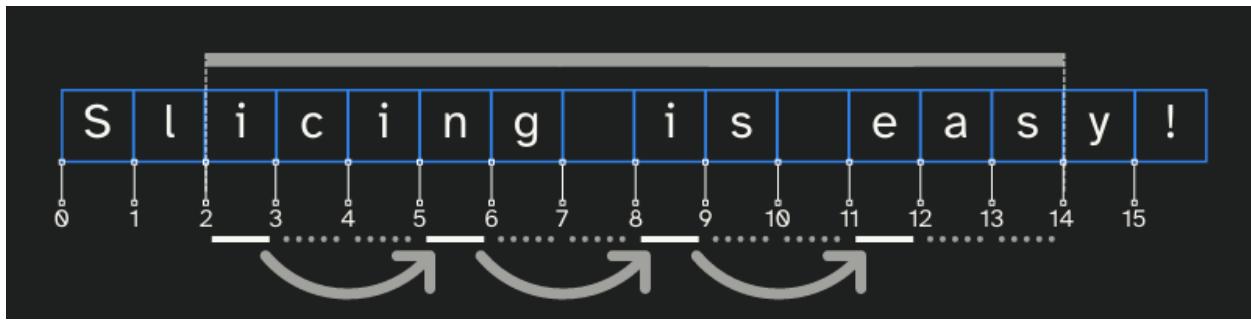
```
>>> s = 'Slicing is easy!'
>>> s[2:14:2]
'iigi a'
## We can also first get the substring and then use the `step`...
>>> sub = s[2:14]
>>> sub[::-2]
'iigi a'
## ... or do both consecutively without an intermediate variable:
>>> s[2:14][::-2]
'iigi a'
```

Give it one more go and try to figure out what the result of

```
>>> s = "Slicing is easy!"
>>> s[2:14:3]
## ...?
```

is.

To help you with that, here's a figure that schematises the slice above:



This is why we get

```
>>> s = "Slicing is easy!"
>>> s[2:14:3]
'inie'
```

Unit and empty steps

Much like with the start and stop parameters, the step parameter can be omitted. This is what we do when we write a slice of the form `[start:stop]`: by not including the step parameter we use its default value, which is 1. This is the same as writing `[start:stop:]`, which is also the same as writing `[start:stop:1]`. In short, all these three slices are equivalent:

- `s[start:stop]`;
- `s[start:stop:]`; and
- `s[start:stop:1]`.

Here is just one example:

```
>>> s = "Slicing is easy!"
>>> s[2:14]
'icing is eas'
>>> s[2:14:]
'icing is eas'
>>> s[2:14:1]
'icing is eas'
```

This is exactly the same as with `range`. When the third argument isn't specified, it is taken to be 1 by default.

Negative step

We have seen how a positive step parameter behaves, now we will see how a negative one does. This is where things really get confusing, and at this point it really is easier to understand how the slicing works if you are comfortable with how `range` works with three arguments.

When you specify a slice with `s[start:stop:step]`, you will get back the elements of `s` that are in the indices pointed to by `range(start, stop, step)`. If `step` is negative, then the `range` function will be counting from `start` to `stop` backwards. This means that `start` needs to be larger than `stop`, otherwise there is nothing to count.

For example, `range(3, 10)` gives the integers 3 to 9. If you want the integers 9 to 3 you can use the step -1, but you also need to swap the start and stop arguments. Not only that, but you also need to tweak them a bit. The start argument is the *first* number that is included in the result and the stop argument is the first number that *isn't*, so if you want the integers from 9 to 3, counting down, you need the start argument to be 9 and the stop argument to be 2:

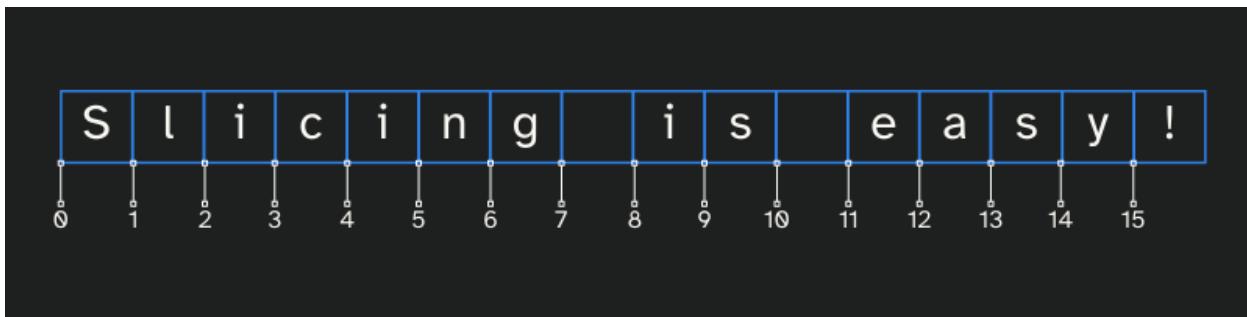
```
>>> list(range(3, 10))
[3, 4, 5, 6, 7, 8]
>>> list(range(3, 10, -1))      # You can't start at 3 and count *down* to 10.
[]
>>> list(range(10, 3, -1))      # Start at 10 and stop right before 3.
[10, 9, 8, 7, 6, 5, 4]
>>> list(range(9, 2, -1))      # Start at 9 and stop right before 2
[9, 8, 7, 6, 5, 4, 3]
```

If you are a bit confused, that is normal. Take your time to play around with `range` and get a feel for how this works.

Using the `range` results from above and the figure below, you should be able to figure out why these slices return these values:

```
>>> s[3:10]
'cing is'
>>> s[3:10:-1]
''
>>> s[10:3:-1]
' si gni'
>>> s[9:2:-1]
'si gnic'
```

Use the `range` results above and this figure to help you out:



If you want to use a negative range that is different from -1, the same principle applies: the start parameter of your string slice should be larger than the stop parameter (so that you can count down from start to stop) and then the absolute value will tell you how many elements you skip at a time. Take your time to work these results out:

```
>>> s = 'Slicing is easy!'
>>> s[15:2:-1]
```

```
'!ysae si gnic'  
>>> s[15:2:-2]  
'!ses nc'  
>>> s[15:2:-3]  
'!asgc'  
>>> s[15:2:-4]  
'!e c'
```

An important remark is due: while `range` accepts negative integers as the `start` and `end` arguments and interprets those as the *actual* negative numbers, remember that slicing also accepts negative numbers but those are interpreted in the context of the sequence you are slicing.

What is the implication of this?

It means that if `step` is negative in the slice `s[start:stop:step]`, then `start` needs to refer to an element that is *to the right of* the element referred to by `stop`.

I will give you an explicit example of the type of confusion that the above remark is trying to warn you about:

```
>>> s = 'Slicing is easy!'  
>>> list(range(2, -2, -1))  
[2, 1, 0, -1]  
>>> s[2:-2:-1]  
  
>>> s[2]  
'i'  

```

Notice how `range(2, -2, -1)` has four integers in it but `s[2:-2:-1]` is an empty slice. Why is that? Because `s[2]` is the first "i" in `s`, while `s[-2]` is the "y" close to the end of the string. Using a step of `-1` would have us go from the "i" to the "y", but going right to left... If you start at the "i" and go left, you reach the beginning of the string, not the "y".

Perhaps another way to help you look at this is if you recall that `s[-2]` is the same as `s[len(s)-2]`, which in this specific case is `s[14]`. If we take the piece of code above and replace all the `-2` with `14`, it should become clearer why the slice is empty:

```
>>> s = 'Slicing is easy!'  
>>> list(range(2, 14, -1))  
[]  
>>> s[2:14:-1]  
  
>>> s[2]  
'i'  

```

Reversing and then skipping

Another possible way to get you more comfortable with these negative steps is if you notice the relationship between slices with a step of the form `-n` and two consecutive slices with steps `-1` and `n`:

```
>>> s = 'Slicing is easy!'
>>> s[14:3:-2]
'ya igi'
>>> s[14:3:-1]
'ysae si gni'
>>> s[14:3:-1][::2]
'ya igi'
```

We can take this even further, and realise that the start and stop parameters are used to shorten the sequence, and that the step parameter is only then used to skip elements:

```
>>> s = 'Slicing is easy!'
>>> s[14:3:-2]
'ya igi'
>>> s[4:15]           # Swap `start` and `stop` and add 1...
'ing is easy'
>>> s[4:15][::-1]     # ...then reverse...
'ysae si gni'
>>> s[4:15][::-1][::2] # ...then pick every other element.
'ya igi'
```

Zero

For the sake of completeness, let's just briefly mention what happens if you use `0` as the step parameter, given that we have taken a look at strictly positive steps and strictly negative steps:

```
>>> s = "Slicing is easy!"
>>> s[::-0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

Using `0` as the step gives a `ValueError`, and that is all there is to it.

Recommendations

Did you know that the Python Standard Library (PSL) has around 6000 usages of sequence slicing, but less than 500 of those make use of the step parameter? That means that, in the PSL, only around 8.33% of the slicing operations make use of the step. (Rough figures for the PSL of Python 3.9.2 on my Windows machine.)

If I had to guess, I would say there are two main reasons that explain why only a “small” percentage of all the slices make use of the step parameter:

- using a step different from 1 is a very specific operation that only makes sense in few occasions and depends a lot on how you have structured your data; and
- step parameters other than 1 and -1 make your code much harder to read.

For those reasons, it is recommendable that you do not get overexcited about slices and force your data to be in such a way that slices are the best way to get to the data.

For example, *do not* store colour names and their hexadecimal values in an alternating fashion just so that you can use `[::-2]` and `[1::2]` to access them. However, if – for some reason – you receive data in this format, it is perfectly acceptable for you to split the data with two slices:

```
## Assume we got `colours` in this format from an API or some other place...
>>> colours = ["red", "#ff0000", "green", "#00ff00", "blue", "#0000ff"]
>>> names = colours[::-2]
>>> names
['red', 'green', 'blue']
>>> hexs = colours[1::2]
>>> hexs
['#ff0000', '#00ff00', '#0000ff']
```

Slices with three parameters tend to be dense and hard to parse with your eyes, given that they are enclosed in `[]` and then have `::` separating the parameters. If you write a slice of the form `s[a:b:c]`, you can expect the readers of your code to have to pause for a bit and understand what is going on. For that matter, when you write a long or complex slice, first consider reworking the code so that you don't have to write a long or complex slice. But if you do end up writing one, you should probably comment your slice explaining what is going on.

I had a look at how the Python Standard Library makes use of slicing with three parameters, and I found this nice example taken from the source code of the `dataclasses` module:

```
## From Lib/dataclasses.py, Python 3.9.2
def _process_class(cls, init, repr, eq, order, unsafe_hash, frozen):
    # [code deleted for brevity]

    # Find our base classes in reverse MRO order, and exclude
    # ourselves. In reversed order so that more derived classes
    # override earlier field definitions in base classes. As long as
    # we're iterating over them, see if any are frozen.
    any_frozen_base = False
    has_dataclass_bases = False
    for b in cls.__mro__[-1:0:-1]:
        # ...

    # [code deleted for brevity]
```

Notice that on top of that `for` loop using a slice, there are 4 lines of comments, and 3 of them are addressing what that slice is doing: why the step parameter is `-1`, because we want to “find our base classes in reverse order”, and then it explains why the start and stop parameters are `-1` and `0`, respectively, as it says “and exclude ourselves”. If we try that slice with a simpler sequence, we can see that `[-1:0:-1]` does in fact reverse a sequence while skipping the first element:

```
>>> s = "Slicing is easy!"  
>>> s[-1:0:-1]  
'!ysae si gnicil'
```

Sequence copying

Having taken a look at many different ways to slice and dice sequences, it is now time to mention a very important nuance about sequence slicing: when we create a slice, we are effectively creating a copy of the original sequence. This isn't necessarily a bad thing. For example, there is **one idiomatic slicing operation** that makes use of this behaviour.

I brought this up because it is important that you are aware of these subtleties, so that you can make informed decisions about the way you write your code.

An example of when this copying behaviour might be undesirable is when you have a really large list and you were considering using a slice to iterate over just a portion of that list. In this case, maybe using the slice will be a waste of resources because all you want is to iterate over a specific section of the list, and then you are done; you don't actually *need* to have that sublist later down the road.

In this case, what you might want to use is the `islice` function from the `itertools` module, that creates an *iterator* that allows you to iterate over the portion of the list that you care about.

Iterators are another awesome feature in Python, and I'll be exploring them in future Pydon'ts, so stay tuned for that!

A simple way for you to verify that slicing creates copies of the sliced sequences is as follows:

```
>>> l = [1, 2, 3, 4]  
>>> l2 = l  
>>> l.append(5)      # Append 5 to l...  
>>> l2              # ... notice that l2 also got the new 5,  
# so l2 = l did NOT copy l.  
[1, 2, 3, 4, 5]  
>>> l3 = l[2:5]      # Slice l into l3.  
>>> l3  
[3, 4, 5]  
>>> l[3] = 42        # Change a value of l...  
>>> l  
[1, 2, 3, 42, 5]    # ... the 4 was replaced by 42...  
>>> l3  
[3, 4, 5]            # ... but l3 still contains the original 4.
```

Manipulating mutable sequences

Let us continue down this journey of mastering sequence slicing. So far we have been using slices to extract parts of our sequences, but slices can also be used to manipulate the contents of our sequences!

Manipulating sequences with slices can only be done for certain types of sequences, namely *mutable* sequences; that is, sequences that we can alter. Prime examples of *mutable* sequences are lists, and prime

examples of *immutable* sequences are strings.

Slice assignment

Say that `l` is a list. We are used to “regular” assignment,

```
>>> l = [1, 2, 3, 4]
```

and we are used to assigning to specific indices:

```
>>> l[2] = 30
>>> l
[1, 2, 30, 4]
```

So how about assigning to slices as well? That is perfectly fine!

```
>>> l[:2] = [10, 20]      # Replace the first 2 elements of l.
>>> l
[10, 20, 30, 4]
>>> l[1::2] = [200, 400]    # Swap elements in odd positions.
>>> l
[10, 200, 30, 400]
```

The two short examples above showed how to replace some elements with the same number of elements. However, with simpler slices you can also change the size of the original slice:

```
>>> l = [1, 2, 3, 4]
>>> l[:2] = [0, 0, 0, 0, 0]
>>> l
[0, 0, 0, 0, 0, 3, 4]
```

When you have a slicing assignment like that, you should read it as “replace the slice on the left with the new sequence on the right”, so the example above reads “swap the first two elements of `l` with five zeroes”.

Notice that, if you use “extended slices” (slices with the step parameter), then the number of elements on the left and on the right *should* match:

```
>>> l = [1, 2, 3, 4]
>>> l[::2]    # This slice has two elements in it...
[1, 3]
>>> l[::2] = [0, 0, 0, 0, 0]  # ... and we try to replace those with 5 elements.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 5 to extended slice of size 2
```

The fact that you can assign to slices allows you to write some pretty beautiful things, if you ask me.

For example, as I was exploring the Python Standard Library, I came across a slicing assignment gem inside the `urljoin` function. `urljoin` from the `urllib.parse` module, takes a base path and a relative path, and tries to combine the two to create an absolute path. Here is an example:

```
>>> import urllib.parse
>>> urllib.parse.urljoin("https://mathspp.com/blog/", "pydnts/zip-up")
'https://mathspp.com/blog/pydnts/zip-up'
```

I'm using `urllib.parse.urljoin` to take the base URL for my blog and stitch that together with a relative link that takes me to one of the Pydnts I have published. Now let me show you part of the source code of that function:

```
## From Lib/urllib/parse.py in Python 3.9.2
def urljoin(base, url, allow_fragments=True):
    """Join a base URL and a possibly relative URL to form an absolute
    interpretation of the latter."""
    # [code deleted for brevity]

    # for rfc3986, ignore all base path should the first character be root.
    if path[:1] == '/':
        segments = path.split('/')
    else:
        segments = base_parts + path.split('/')
        # filter out elements that would cause redundant slashes on re-joining
        # the resolved_path
        segments[1:-1] = filter(None, segments[1:-1])
```

Notice the slice assignment to `segments[1:-1]`? That `segments` list contains the different portions of the two URLs I give the `urljoin` function, and then the `filter` function is used to filter out the parts of the URL that are empty. Let me edit the source code of `urljoin` to add two print statements to it:

```
## From Lib/urllib/parse.py in Python 3.9.2
def urljoin(base, url, allow_fragments=True):
    """Join a base URL and a possibly relative URL to form an absolute
    interpretation of the latter."""
    # [code deleted for brevity]

    # for rfc3986, ignore all base path should the first character be root.
    if path[:1] == '/':
        segments = path.split('/')
    else:
        segments = base_parts + path.split('/')
        # filter out elements that would cause redundant slashes on re-joining
        # the resolved_path
        print(segments)
        segments[1:-1] = filter(None, segments[1:-1])
        print(segments)
```

Now let me run the same example:

```
>>> import urllib.parse
```

```
>>> urllib.parse.urljoin("https://mathspp.com/blog/", "pydnts/zip-up")
['', 'blog', '', 'pydnts', 'zip-up']  # First `print(segments)`
['', 'blog', 'pydnts', 'zip-up']      # <----- segments has one less '' in it!
'https://mathspp.com/blog/pydnts/zip-up'
```

We can take the result of the first print and run the filter by hand:

```
>>> segments = ['', 'blog', '', 'pydnts', 'zip-up']
>>> segments[1:-1]
['blog', '', 'pydnts']
>>> list(filter(None, segments[1:-1]))
['blog', 'pydnts']
>>> segments[1:-1] = filter(None, segments[1:-1])
>>> segments
['', 'blog', 'pydnts', 'zip-up']
```

So this was a very interesting example usage of slice assignment. It is likely that you won't be doing something like this very frequently, but knowing about it means that when you do, you will be able to write that piece of code beautifully.

Slice deletion

If you can assign to slices, what happens if you assign the empty list [] to a slice?

```
>>> l = [1, 2, 3, 4]
>>> l[:2] = []          # Replace the first two elements with the empty list.
>>> l
[3, 4]
```

If you assign the empty list to a slice, you are effectively deleting those elements from the list. You can do this by assigning the empty list, but you can also use the `del` keyword for the same effect:

```
>>> l = [1, 2, 3, 4]
>>> del l[:2]
>>> l
[3, 4]
```

More idiomatic slicing

Now that we have learned some more cool features on sequence slicing, it is time to see how these features are better used and how they show up in their more idiomatic forms.

Idiomatic code is code that you can take a look at and read it for what it does, as a whole, without having to reason about each little piece individually... Think of it like “normal” reading: when you were learning, you had to read character by character, but now you grasp words as a whole. With enough practice, these idiomatic pieces of code become recognisable as a whole as well.

Even positions and odd positions

A simple slice that you may want to keep on the back of your mind is the slice that lets you access all the elements in the even positions of a sequence. That slice is `[:2]`:

```
>>> l = ["even", "odd", "even", "odd", "even"]
>>> l[:2]
['even', 'even']
```

Similarly, `l[1::2]` gives you the odd positions:

```
>>> l = ["even", "odd", "even", "odd", "even"]
>>> l[1::2]
['odd', 'odd']l = ["even", "odd", "even", "odd", "even"]
l[1::2]
```

`s[::-1]`

A slice with no start and stop parameters and a `-1` in the step is a very common slicing pattern. In fact, there are approximately 100 of these slices in the Python Standard Library, which is roughly one third of all the slices that make use of the step parameter.

`s[::-1]` should be read as “the sequence `s`, but reversed”. Here is a simple example:

```
>>> s = "Slicing is easy!"
>>> s[::-1]
'!ysae si gnicils'
```

What is noteworthy here, and related to the previous remark about slices creating copies, is that sometimes you don’t want to copy the whole thing to reverse your sequence; for example, if all you want to do is iterate over the sequence in reverse order. When that is the case, you might want to just use the `reversed` built-in function. This function takes a sequence and allows you to iterate over the sequence in *reverse* order, without paying the extra memory cost of actually copying the whole sequence.

`l[:] or l[::]`

If a slice makes a copy, that means that a slice is a very clean way to copy a sequence! The slices `[:]` and `[:]` select whole sequences, so those are prime ways to copy a sequence – for example, a list – when you really want to create copies.

Deep and shallow copies, the distinction between things that are passed by reference and things that are passed by value, etc, is a big discussion in itself.

It is easy to search the Python Standard Library for usage examples of this idiom (and for the ones before as well), so I will just leave you with one, from the `argparse` module, that contains a helper function named `_copy_items` (I deleted its comments):

```
## From Lib/argparse.py in Python 3.9.2
def _copy_items(items):
    if items is None:
        return []
```

```

if type(items) is list:
    return items[:]
import copy
return copy.copy(items)

```

Notice how the idiom fits in so nicely with the function name: the function says it copies the items. What does the function do? If the `items` argument is a list, then it returns a copy of it! So `l[:]` and `l[::]` should be read as “a copy of `l`”.

This idiom also explains the thumbnail image in the beginning of the article.

`del l[:]`

Another idiom that makes use of the slice `[:]`, but with something extra, is the idiom to delete the contents of a list.

Think of `l[:]` as “opening up `l`”, and then `del l[:]` reads “open up `l` to delete its *contents*”. This is the same as doing `l[:] = []` but it is *not* the same as doing `l = []` nor is it the same as doing `del l`.

It is easy to see why `del l` is different from the others: `del l` means that the name `l` is no longer in use:

```

>>> l = [1, 2, 3, 4]
>>> del l
>>> l
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l' is not defined

```

whereas the idiom just clears the list up:

```

>>> l = [1, 2, 3, 4]
>>> del l[:]
>>> l
[]

```

What might be trickier to understand is why `del l[:]` and `l[:] = []` are different from `l = []`. I’ll show you an example that shows they are *clearly* different, and then I will leave it up to you to decide whether or not you want to burn enough neurons to understand what is going on.

First, let me use `l[:] = ...`

```

>>> l = l_shallow = [1, 2, 3]
>>> l_shallow is l
True
>>> j = []
>>> l[:] = j
>>> l
[]
>>> l_shallow
[]

```

```
>>> l is j
False
>>> l_shallow is l
True
```

and now let me compare it with `l = ...`

```
>>> l = l_shallow = [1, 2, 3]
>>> l_shallow is l
True
>>> j = []
>>> l = j
>>> l
[]
>>> l_shallow
[1, 2, 3]
>>> l is j
True
>>> l_shallow is l
False
```

You can see above that the results of comparisons like `l is j` and `l_shallow is l`, as well as the contents of `l_shallow`, change in the two examples. Therefore, the two things cannot be the same. What is going on? Well, deep and shallow copies, and references to mutable objects, and the like, are at fault! I'll defer a more in-depth discussion of this for a later Pydon't, as this one has already become quite long.

Just remember, `l[:] = []` and `del l[:]` can be read as “delete the *contents* of `l`”.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Slices are really powerful and they are an essential tool to master for when you work with sequences, like strings and lists.”

This Pydon't showed you that:

- slices can have a step parameter that allows to skip elements of the sequence;
- the default value of the step parameter is 1;
- a negative step allows to pick elements from the end of the sequence to the start;
- when using a negative step, the start parameter should refer to an element of the sequence that is to the *right* of the element referred to by the stop parameter;
- there is a parallelism between slices (with negative steps) and the built-in `range` function;
- 0 is not a valid step parameter for a slice;
- slices are more common with just the start and stop parameters, in part because slices with `[start:stop:step]` can be really hard to read;
- slices create *copies* of the parts of the sequences we are looking at, so you have to be mindful of that when memory is constrained;
- you can assign to slices of mutable objects, like lists;

- when assigning to a slice, the final length of the sequence might change if we use a simple slice on the left (without the step parameter) and if the sequence on the right has a different number of elements;
- you can use the `del` keyword to delete slices of mutable sequences, or you can also assign the empty sequence to those slices for the same effect;
- there are some interesting idiomatic slices that you should be aware of:
 - `s[::-2]` and `s[1::2]` are “elements in even positions of `s`” and “elements in odd positions of `s`”, respectively;
 - `s[::-1]` is “`s`, but reversed”;
 - `l[:] and l[:] = None` are “a copy of `l`”; and
 - `del l[:] is “delete the contents of l” or “empty l”, which is not the same as doing l = [].`

References

- Python 3 Documentation, The Python Language Reference, Expressions – Slicings, <https://docs.python.org/3/reference/expressions.html#slicings> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – The Standard Type Hierarchy, <https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – Emulating Container Types, <https://docs.python.org/3/reference/datamodel.html#emulating-container-types> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, `slice`, <https://docs.python.org/3/library/functions.html#slice> [last accessed 20-04-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, `range`, <https://docs.python.org/3/library/functions.html#func-range> [last accessed 03-05-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, `filter`, <https://docs.python.org/3/library/functions.html#filter> [last accessed 11-05-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, `reversed`, <https://docs.python.org/3/library/functions.html#reversed> [last accessed 11-05-2021];
- Python 3 Documentation, The Python Standard Library, `dataclasses`, <https://docs.python.org/3/library/dataclasses.html> [11-05-2021];
- Python 3 Documentation, The Python Standard Library, `itertools`, `islice`, <https://docs.python.org/3/library/itertools.html#itertools.islice> [11-05-2021];
- Python 3 Documentation, The Python Standard Library, `urllib.parse`, `urljoin`, <https://docs.python.org/3/library/urllib.parse.html#urllib.parse.urljoin> [11-05-2021];
- “Effective Python – 90 Specific Ways to Write Better Python”; Slatkin, Brett; ISBN 9780134853987;
- Stack Overflow, “Why would I want to use `itertools.islice` instead of normal list slicing?”, <https://stackoverflow.com/q/32172612/2828287> [last accessed 10-05-2021].

Inner workings of sequence slicing

```
● ● ●  
>>> s = "Slicing is easy!"  
>>> s[2::3]  
'iniey'  
# Slicing is just syntactic sugar,  
# we can just use the slice() class.  
>>> s[slice(2, None, 3)]  
'iniey'
```

(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/inner-workings-of-sequence-slicing.>)

Introduction

We have written two Pydon'ts already on sequence slicing:

1. “[Idiomatic sequence slicing](#)”; and
2. “[Mastering sequence slicing](#)”.

Those two Pydon'ts taught you almost everything there is to know about sequence slicing, but there is something that we will only take a look at today:

- uncovering the two layers of syntactic sugar surrounding sequence slicing; and
- seeing how to implement slicing for your custom objects.

If you don't really know how sequence slicing works, you might want to take a look at the Pydon'ts I linked above. In particular, the Pydon't on [mastering sequence slicing](#) can really help you take your Python slicing skills to the next level.

Without further ado, let us begin!

The slice class

I don't know if you know this, but Python has, in its amazing documentation, a [section devoted to its built-in functions](#). In there, you can find things like `bool`, `enumerate`, or `len`. If you take a look at the built-in functions that start with “s”, you will find `slice` in there!

Taking a look at the docs about `slice`, we find it shows up in a way that is similar to `int` or `str`, which means that a `slice` defines a type of object we can have in our programs: much like `int(3)` creates an integer 3 or `str(3)` creates a string "3", `slice(3)` creates a slice:

```
>>> print(slice(3))
slice(None, 3, None)
```

This is the first level of syntactic sugar we are uncovering in this Pydon't: Python uses these `slice` objects when we write things like `s[2:3]`! But first, let us explore the `slice` objects a bit more.

Slicing parameters

If we read the docs, or if we play around with the `slice` built-in enough, we find out that this object stores the slicing parameters that we repeatedly talked about in the previous Pydon'ts. These parameters are the start, stop, and step, parameters of the slice, and the docs tell us that we can access them:

```
>>> sl = slice(1, 12, 3)
>>> sl.start
1
>>> sl.stop
12
>>> sl.step
3
```

However, we *cannot* modify them:

```

>>> sl = slice(None, 3, None)
>>> print(sl.start)
None
>>> sl.start = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: readonly attribute

```

Relationship with range

Another really important thing here lies in noting that this relationship that **I tried to make apparent**, between slicing and sets of indices specified by `range`, isn't just coincidental. No, the documentation specifically says that `slice(start, stop, step)` represents the indices specified by `range(start, stop, step)`. This is why it is so helpful to understand the relationship between doing `s[start:stop:step]` and something (much!) more verbose that makes use of a `for` loop and the corresponding `range`:

```

>>> s = "Slicing is easy!"
>>> print(s[1:15:3])
li s
>>> start, stop, step = 1, 15, 3
>>> result = ""
>>> for idx in range(start, stop, step):
...     result += s[idx]
...
>>> print(result)
li s

```

Explicit slices instead of colons

We have seen that we can create explicit `slice` objects, but can we use them..? Of course we can! I have been talking about syntactic sugar, and this is where it shows up: writing `s[start:stop:step]` or `s[sl]`, where `sl` is the appropriate slice, is the same thing!

Here are two examples of this:

```

>>> s = "Slicing is easy!"
>>> s[1:15:2]
'lcn ses'
>>> sl = slice(1, 15, 2)
>>> s[sl]
'lcn ses'
>>> s[2::3]
'iniey'
>>> sl = slice(2, None, 3)
>>> s[sl]
'iniey'

```

Notice how, in the example above, we use `None`, when creating a `slice` object, in order to specify an implicit slicing parameter, such as the omitted stop parameter in the slice `s[2::3]`, that would go between the two colons.

By the way, careful with naming your `slice` objects! The most obvious name is `slice`, but if you create a slice with that name then you will have a hard time creating other `slice` objects because you will overwrite the name of the built-in type. This is also why you shouldn't name your strings `str` or your integers `int`.

Getting items from sequences

We have seen that `slice` objects can be used to extract slices from sequences in the same way as when we use the syntactic sugar with the colons... But how, exactly, are these things used to extract elements from sequences? Tangent to this question, how would I implement slicing capabilities in my own objects?

The answer lies in the `__getitem__` dunder method.

Recall that “dunder” is short for “double underscore”, the common name that Python gives to methods that start and end with two underscores, which generally indicate that the method has to do with the inner workings of Python. We have seen other dunder methods in the Pydon'ts about `str` and `repr` and about `Truthy`, `Falsy`, and `bool`.

The `__getitem__` dunder method is the method that is called, behind the scenes, when you try to access indices or slices. An empirical verification of this is very easy to perform: we'll just create a new class, called `S`, that will be wrapping the built-in strings, and intercept the `__getitem__` call:

```
>>> class S(str):
...     def __getitem__(self, idx):
...         print("Inside __getitem__")
...         # Just let the built-in string handle indexing:
...         return super().__getitem__(idx)
...
>>> s = S("Slicing is easy!")
>>> s[3]
Inside __getitem__
'c'
>>> s[1::2]
Inside __getitem__
'lcn ses!'
```

This shows that the `__getitem__` method is the one that is responsible for indexing sequences.

The line that starts with `super()` is letting the built-in `str` class handle the indexing for us, given that our goal was just to verify that the `__getitem__` method is called.

Now, instead of just printing an irrelevant message, we could actually print the index (or slice!) that is about to be used:

```
>>> class S(str):
...     def __getitem__(self, idx):
...         print(f"The argument was: {idx}")
```

```

...
# Just let the built-in string handle indexing:
...
return super().__getitem__(idx)

...
>>> s = S("Slicing is easy!")
>>> s[3]
The argument was: 3
'c'
>>> s[1::2]
The argument was: slice(1, None, 2)
'lcn ses!'


```

As you can see above, we tried slicing the string with `s[1::2]` and that was converted to `slice(1, None, 2)` by the time it got to the `__getitem__` call!

This shows the two bits of syntactic sugar going on: using the colon syntax for slices, `start:stop:step`, is just syntactic sugar for creating an explicit `slice` object, and using brackets `[]` to index/slice is just syntactic sugar for a call to the `__getitem__` function:

```

>>> s = "Slicing is easy!"
>>> s[1::3]
'i s'
>>> s.__getitem__(slice(1, None, 3))
'i s'


```

This shows that you can use indexing/slicing in your own custom objects if you implement the `__getitem__` method for your own objects. I will show you an example of this below.

Setting items, deleting items, and container emulation

In the Pydon't about [mastering sequence slicing](#) we also saw how to do slicing assignment and how to delete slices of sequences. To do that in your own objects you have to deal with the `__setitem__` and `__delitem__` methods, whose signature is similar to `__getitem__`. Just take a look at [the docs](#) if you want to learn more about these methods or if you are looking at implementing custom classes that emulate built-in container types.

Comma-separated indices and slices

I would like to point out another cool thing that you can find if you dig “deep” enough in the documentation (see [here](#)), or that you probably already encountered if you use other modules like `numpy` or `pandas`. This “thing” is the fact that you can write several indices/slices if you separate them by commas.

Syntactically, that is perfectly valid. That is, you can write something like that and Python will accept it. However, Python’s built-in types do not support multiple indexing or slicing, so the built-in types do end up screaming at you:

```

>>> s = "Slicing is easy!"
>>> s[1, 2, 3, 4:16:2]


```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string indices must be integers
```

Python complained, but not about the syntax. It is strings that cannot handle the indices, and the extra slice, that you gave to the `__getitem__` setting. Compare this with an actual `SyntaxError`:

```
>>> for in range(10):
    File "<stdin>", line 1
        for in range(10):
    ^
SyntaxError: invalid syntax
```

I couldn't even change lines to continue my make-believe `for` loop, Python outright complained about the syntax being wrong.

However, in your *custom* objects, you can add support for multiple indexing/slicing:

```
>>> class Seq:
...     def __getitem__(self, idx):
...         print(idx)
...
>>> s = Seq()
>>> s[1, 2, 3, 4:16:2]
(1, 2, 3, slice(4, 16, 2))
```

As you can see, the multiple indices and slices get packed into a tuple, which is then passed in to `__getitem__`.

We have taken a look at how slices work under the hood, and also took a sneak peek at how regular indexing works, and now we will go through a couple of examples in code where these things could be helpful.

Examples in code

Bear in mind that it is likely that you won't be using explicit `slice` objects in your day-to-day code. The scarcity of usage examples of `slice` in the Python Standard Library backs my claim.

Most usages of `slice` I found were for testing other objects' implementations, and then I found a couple (literally two) usages in the `xml` module, but to be completely honest with you, I did not understand why they were being used! (Do let me know if you can explain to me what is happening there!)

`itertools.islice`

The first example we will be using is from the `itertools` module's `islice` function. The `islice` function can be used to slice into an iterator, much like regular slicing, with two key differences:

- `islice` does *not* work with negative parameters; and
- `islice` works with generic iterables, which is the main reason why `islice` is useful.

Iterables and generators are fascinating things in Python and *there will be* future Pydon'ts on this subject. Stay tuned for those.

Without going into too much detail about the iterables, let me show you a clear example of when regular slicing doesn't work but `islice` works:

```
>>> f = lambda x: x      # function that returns its input.
>>> f(3)
3
>>> f([1, 2, "Hey"])
[1, 2, 'Hey']
>>> s = "Slicing is easy!"
>>> s[2::3]
'iney'
>>> m = map(f, s)        # `m` is an iterable with the characters from `s`.
>>> m[2::3]               # regular slicing doesn't work...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'map' object is not subscriptable
>>> import itertools
>>> for char in itertools.islice(m, 2, None, 3):
...     print(char)
...
i
n
i
e
y
```

The example above just shows that `islice` works in some situations where regular slicing with `[start:stop:step]` doesn't. The documentation for `islice` provides an approximate Python implementation of `islice` (the actual function is written in C):

```
## From https://docs.python.org/3/library/itertools.html#itertools.islice,
## accessed on the 18th of May 2021
def islice(iterable, *args):
    # (Some comments removed for brevity...)
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    # (Code sliced for brevity, pun much intended.)
    # ...
```

In the example above, the `slice` object is being used just as an utility to map the arguments given to `islice` as the parameters that need to go into the `range` in the third code line of the example.

Another noteworthy thing is the line that assigns to `start`, `stop`, `step` with the `or` operators. The `or` is being used to assign default values to the parameters, in case the original argument as `None`:

```
>>> start = 4      # If `start` has a value,
```

```

>>> start or 0      # then we get that value.
4
>>> start = None    # However, if `start` is `None`,
>>> start or 0      # then we get the default value of `0`.
0
## Similarly for the `stop` and `step` parameters;
## here is another example with `stop`:
>>> import sys
>>> stop = 4
>>> stop or sys.maxsize
4
>>> stop = None
>>> stop or sys.maxsize
9223372036854775807

```

The short-circuiting capabilities of the `or` operator (and also of the `and`) will be discussed in detail in a later Pydon't, don't worry!

To conclude this example, we see that `slice` can be useful in the niche use-case of dispatching `range`-like arguments to their correct positions, because you can read the parameters off of a `slice` object.

Custom arithmetic and geometric sequences

In this example I will be showing you a simple example implementation of a custom object that supports slicing. For that, we will implement a class for the concept of geometric progression (see [Wikipedia](#)): a progression that is defined by two parameters:

- the starting number `s`; and
- the ratio `r`.

The first number of the progression is `s`, and each subsequent item is just `r` times the previous one. Here is how you would create the skeleton for such a concept:

```

class GeometricProgression:
    def __init__(self, start, ratio):
        self.start = start
        self.ratio = ratio

    def __str__(self):
        return f"GeometricProgression({self.start}, {self.ratio})"

gp = GeometricProgression(1, 3)
print(gp)    # prints GeometricProgression(1, 3)

```

Now, geometric progressions have infinite terms, so we cannot really just generate “all terms” of the progression and return them in a list or something like that, so if we want to support indexing and/or slicing, we need to do something else... We need to implement `__getitem__`!

Let us implement `__getitem__` in such a way that it returns a list with all the elements that the user tried to fetch:

```

import sys

class GeometricProgression:
    def __init__(self, start, ratio):
        self.start = start
        self.ratio = ratio

    def __str__(self):
        return f"GeometricProgression({self.start}, {self.ratio})"

    def nth(self, n):
        """Compute the n-th term of the progression, 0-indexed."""
        return self.start*pow(self.ratio, n)

    def __getitem__(self, idx):
        if isinstance(idx, int):
            return self.nth(idx)
        elif isinstance(idx, slice):
            start, stop, step = idx.start or 0, idx.stop or sys.maxsize, idx.step or 1
            return [self.nth(n) for n in range(start, stop, step)]
        else:
            raise TypeError("Geo. progression indices should be integers or slices.")

gp = GeometricProgression(1, 3)
print(gp[0])      # prints 1
print(gp[1])      # prints 3
print(gp[2])      # prints 9
print(gp[0:3])    # prints [1, 3, 9]
print(gp[1:10:3]) # prints [3, 81, 2187]

```

As you can see, our implementation already supports slicing and indexing, but we can take this just a little bit further, and add support for multiple indices/slices with ease:

```

import sys

class GeometricProgression:
    def __init__(self, start, ratio):
        self.start = start
        self.ratio = ratio

    def __str__(self):
        return f"GeometricProgression({self.start}, {self.ratio})"

    def nth(self, n):
        """Compute the n-th term of the progression, 0-indexed."""
        return self.start*pow(self.ratio, n)

```

```

def __getitem__(self, idx):
    if isinstance(idx, int):
        return self.nth(idx)
    elif isinstance(idx, slice):
        start, stop, step = idx.start or 0, idx.stop or sys.maxsize, idx.step or 1
        return [self.nth(n) for n in range(start, stop, step)]
    elif isinstance(idx, tuple):
        return [self.__getitem__(sub_idx) for sub_idx in idx]
    else:
        raise TypeError("Geo. progression indices should be integers or slices.")

gp = GeometricProgression(1, 3)
print(gp[0, 1, 4])          # prints [1, 3, 81]
print(gp[0:2, 0:2, 1, 0:2])  # prints [[1, 3], [1, 3], 3, [1, 3]]

```

And that is it, this shows you a (simple) working example of how you could define indexing and slicing into your own objects.

| You can find this simple implementation on [GitHub](#), in case you need it.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Sequence slicing hides two layers of syntactic sugar for you, but you do need to know about them if you want to write custom objects that support indexing and/or slicing.”

This Pydon't showed you that:

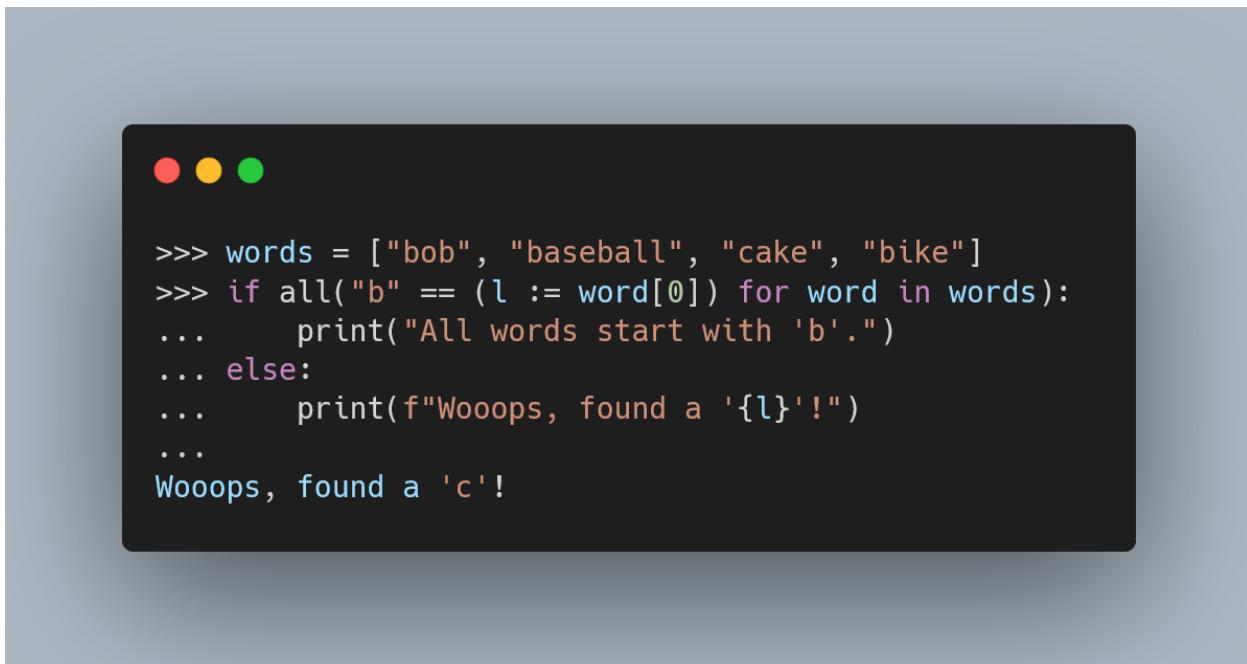
- there is a built-in slice type in Python;
- the syntax [start:stop:step] is just syntactic sugar for slice(start, stop, step);
- slice(start, stop, step) represents the indices of range(start, stop, step);
- when you use seq[] to index/slice into seq, you actually call the __getitem__ method of seq;
- __getitem__, __setitem__, and __delitem__, are the three methods that you would need in custom objects to emulate indexing, indexing assignment and indexing deletion;
- Python syntax *allows* for multiple indices/slices separated by commas;
- `itertools.islice` can be used with iterables, whereas plain slicing cannot; and
- it can be fairly straightforward to implement (multiple) indexing/slicing for your own objects.

References

- Python 3 Documentation, The Python Language Reference, Expressions – Slicings, <https://docs.python.org/3/reference/expressions.html#slicings> [last accessed 18-05-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – The Standard Type Hierarchy, <https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy> [last accessed 20-04-2021];

- Python 3 Documentation, The Python Language Reference, Data Model – Emulating Container Types, <https://docs.python.org/3/reference/datamodel.html#emulating-container-types> [last accessed 18-05-2021];
- Python 3 Documentation, The Python Language Reference, Data Model – Emulating Container Types, `__getitem__`, https://docs.python.org/3/reference/datamodel.html#object.__getitem__ [last accessed 18-05-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, `slice`, <https://docs.python.org/3/library/functions.html#slice> [last accessed 18-05-2021];
- Python 3 Documentation, The Python Standard Library, `itertools`, `islice`, <https://docs.python.org/3/library/itertools.html#itertools.islice> [18-05-2021];
- Stack Overflow, “Why would I want to use `itertools.islice` instead of normal list slicing?”, <https://stackoverflow.com/q/32172612/2828287> [last accessed 18-05-2021].

Boolean short-circuiting



(Thumbnail of the original article at <https://mathspp.com/blog/pydons/boolean-short-circuiting>.)

Introduction

In this Pydon't we will take a closer look at how `and` and `or` really work and at a couple of really neat things you can do because of the way they are defined. In particular, we will look at

- the fact that `and` and `or` return values from their operands, and not necessarily `True` or `False`;
- what “short-circuiting” is and how to make the best use of it;
- how short-circuiting in `and` and `or` extends to `all` and `any`; and
- some expressive use-cases of Boolean short-circuiting.

For this Pydon't, I will assume you are familiar with what “Truthy” and “Falsy” values are in Python. If you are

not familiar with this concept, or if you would like just a quick reminder of how this works, go ahead and read the “[Truthy, Falsy, and bool](#)” Pydon’t.

Return values of the and and or operators

If we [take a look at the docs](#), here is how or is defined:

“x or y returns y if x is false, otherwise it returns x.”

Equivalently, but written with an if expression,

```
(x or y) == (y if not x else x)
```

This may not seem like it is worth spending a thought on, but already right at this point we can see something very interesting: even though we look at the truthy or falsy value of x, what we return are the values associated with x/y, and *not* a Boolean value.

For example, look at the program below and think about what it outputs:

```
if 3 or 5:  
    print("Yeah.")  
else:  
    print("Nope.")
```

If you thought it should print “Yeah.”, you are right! Notice how 3 or 5 was the condition of the if statement and it evaluated to True, which is why the statement under if got executed.

Now, look at the program below and think about what it outputs:

```
print(3 or 5)
```

What do you think it outputs? If you think the output should be True, you are wrong! The program above outputs 3:

```
>>> 3 or 5  
3
```

Let’s go back to something I just said:

“*Notice how 3 or 5 was the condition of the if statement and it evaluated to True, which is why the statement under if got executed.*”

The wording of this statement is wrong, but the error in it is fairly subtle. If you spotted it before I pointed it out, give yourself a pat in the back, you deserve it. So, what did I say wrong?

3 or 5 does *not* evaluate to True! It evaluates to 3, which is *truthy* and therefore tells the if to execute its statements. Returning True or a truthy value is something significantly different.

A similar thing happens with and. [As per the docs](#), and can be defined as follows:

“x and y returns x if x is false, otherwise it returns y.”

We can also rewrite this as

```
(x and y) == (x if not x else y)
```

Take your time to explore this for a bit, just like we explored `x or y` above.

Short-circuiting

You might be asking why this distinction is relevant. It is mostly relevant because of the following property: `and` and `or` only evaluate the right operand if the left operand is not enough to determine the result of the operation. This is what short-circuiting is: not evaluating the whole expression (stopping short of evaluating it) if we already have enough information to determine the final outcome.

This short-circuiting feature, together with the fact that the boolean operators `and` and `or` return the values of the operands and not necessarily a Boolean, means we can do some really neat things with them.

`or`

`False or y`

`or` evaluates to `True` if any of its operands is `truthy`. If the left operand to `or` is `False` (or `falsy`, for that matter) then the `or` operator *has* to look to its right operand in order to determine the final result.

Therefore, we know that an expression like

```
val = False or y
```

will have the value of `y` in it, and in an `if` statement or in a `while` loop, it will evaluate the body of the construct only if `y` is `truthy`:

```
>>> y = 5    # truthy value.
>>> if False or y:
...     print("Got in!")
... else:
...     print("Didn't get in...")
...
Got in!
>>> y = []    # falsy value.
>>> if False or y:
...     print("Got in 2!")
... else:
...     print("Didn't get in 2...")
...
Didn't get in 2...
```

Let this sit with you: if the left operand to `or` is `False` or `falsy`, then we *need* to look at the right operand to determine the value of the `or`.

`True or y`

On the other hand, if the left operand to `or` is `True`, we do not need to take a look at `y` because we already know the final result is going to be `True`.

Let us create a simple function that returns its argument unchanged but that produces a side-effect of printing something to the screen:

```
def p(arg):
    print(f"Inside `p` with arg={arg}")
    return arg
```

Now we can use `p` to take a look at the things that Python evaluates when trying to determine the value of `x` or `y`:

```
>>> p(False) or p(3)
Inside `p` with arg=False
Inside `p` with arg=3
3
>>> p(True) or p(3)
Inside `p` with arg=True
True
```

Notice that, in the second example, `p` only did one print because it never reached the `p(3)`.

Short-circuiting of `or` expressions

Now we tie everything together. If the left operand to `or` is `False` or falsy, we know that `or` has to look at its right operand and will, therefore, return the value of its right operand after evaluating it. On the other hand, if the left operand is `True` or truthy, `or` will return the value of the left operand without even evaluating the right operand.

`and`

We now do a similar survey, but for `and`.

```
False and y
```

and gives `True` if *both* its operands are `True`. Therefore, if we have an expression like

```
val = False and y
```

do we need to know what `y` is in order to figure out what `val` is? No, we do not, because regardless of whether `y` is `True` or `False`, `val` is always `False`:

```
>>> False and True
False
>>> False and False
False
```

If we take the `False and y` expressions from this example and compare them with the `if` expression we wrote earlier, which was

```
(x and y) == (x if not x else y)
```

we see that, in this case, `x` was substituted by `False`, and, therefore, we have

```
(False and y) == (False if not False else y)
```

Now, the condition inside that `if` expression reads

```
not False
```

which we know evaluates to `True`, meaning that the `if` expression never returns `y`.

If we consider any left operand that can be `False` or falsy, we see that `and` will never look at the right operand:

```
>>> p([]) and True # [] is falsy
Inside `p` with arg=[]
[]
>>> p(0) and 3242 # 0 is falsy
Inside `p` with arg=0
0
>>> p({}) and 242 # {} is falsy
Inside `p` with arg={}
{}
>>> p(0) and p(0) # both are falsy, but only the left matters
Inside `p` with arg=0
0
```

`True and y`

Now, I invite you to take a moment to work through the same reasoning, but with expressions of the form `True and y`. In doing so, you should figure out that the result of such an expression is always the value of `y`, because the left operand being `True`, or any other truthy value, doesn't give `and` enough information.

Short-circuiting of `and` expressions

Now we tie everything together. If the left operand to `and` is `False` or falsy, we know the expression returns the value of the left operand regardless of the right operand, and therefore we do not even evaluate the right operand. On the other hand, if the left operand to `and` is `True`, then `and` will evaluate the right operand and return its value.

Short-circuiting in plain English

Instead of memorising rules about what sides get evaluated when, just remember that both `and` and `or` will evaluate as many operands as needed to determine the overall Boolean result, and will then return the value of the last side that they evaluated.

As an immediate conclusion, the left operand is *always* evaluated, as you might imagine.

If you understand that, then it is just a matter of you knowing how `and` and `or` work from the Boolean perspective.

all and any

The built-in functions `all` and `any` *also* short-circuit, as they are simple extensions of the behaviours provided by `and` and `or`, respectively.

`all` wants to make sure that *all* the values of its argument are truthy, so as soon as it finds a falsy value, it knows it's game over. That's why [the docs](#) say `all` is equivalent to the following code:

```
def all(it):
    for elem in it:
        if not elem:
            return False
    return True
```

Similarly, `any` is going to do its best to look for *some* value that is truthy. Therefore, as soon as it finds one, `any` knows it has achieved its purpose and does not need to evaluate the other values.

Can you write an implementation of `any` that is similar to the above implementation of `all` and that also short-circuits?

Short-circuiting in chained comparisons

A [previous Pydon't](#) has shown you that comparison operators can be chained arbitrarily, and those are almost equivalent to a series of comparisons separated with `and`, except that the subexpressions are only evaluated once, to prevent wasting resources. Therefore, because we are also using an `and` in the background, chained comparisons can also short-circuit:

```
## 1 > 2 is False, so there's no need to look at p(2) < p(3)
>>> p(1) > p(2) < p(3)
Inside `p` with arg=1
Inside `p` with arg=2
False
```

Examples in code

Now that we have taken a look at how all of these things work, we will see how to put them to good use in actual code.

Short-circuit to save time

One of the most basic usages of short-circuiting is to save time. When you have a `while` loop or an `if` statement with multiple statements, you may want to include the faster expressions before the slower ones, as that might save you some time if the result of the first expression ends up short-circuiting.

Conditionally creating a text file

Consider this example that should help me get my point across: imagine you are writing a function that creates a helper .txt file but only if it is a .txt file and if it does not exist yet.

With this preamble, your function needs to do two things: - check the suffix of the file is .txt; - check if the file exists in the filesystem.

What do you feel is faster? Checking if the file ends in .txt or looking for it in the whole filesystem? I would guess checking for the .txt ending is simpler, so that's the expression I would put first in the code:

```
import pathlib

def create_txt_file(filename):
    path = pathlib.Path(filename)
    if filename.suffix == ".txt" and not path.exists():
        # Create the file but leave it empty.
        with path.open():
            pass
```

This means that, whenever `filename` does not respect the .txt format, the function can exist right away and doesn't even need to bother the operating system with asking if the file exists or not.

Conditionally checking if a string matches a regular expression

Now let me show you a real example of an `if` statement that uses short-circuiting in this way, saving some time. For this, let us take a look at a function from the `base64` module, that we take from the Python Standard Library:

```
## From Lib/base64.py in Python 3.9.2
def b64decode(s, altchars=None, validate=False):
    """Decode the Base64 encoded bytes-like object or ASCII string s.
    [docstring cut for brevity]
    """
    s = _bytes_from_decode_data(s)
    if altchars is not None:
        altchars = _bytes_from_decode_data(altchars)
        assert len(altchars) == 2, repr(altchars)
        s = s.translate(bytes.maketrans(altchars, b'+/'))
    if validate and not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s):    # <-
        raise binascii.Error('Non-base64 digit found')
    return binascii.a2b_base64(s)
```

This `b64decode` function takes a string (or a bytes-like object) that is assumed to be in base 64 and decodes it.

Here is a quick demo of that:

```
>>> import base64
>>> s = b"Base 64 encoding and decoding."
```

```

>>> enc = base64.b64encode(s)
>>> enc
b'QmFzZSA2NCB1bmNvZGluZyBhbmqgZGVjb2Rpbmcu'
>>> base64.b64decode(enc)
b'Base 64 encoding and decoding.'

```

Now, look at the `if` statement that I marked with a comment:

```

if validate and not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s):
    pass

```

`validate` is an argument to `b64decode` that tells the function if we should validate the string that we want to decode or not, and then the `re.fullmatch()` function call does that validation, ensuring that the string to decode only contains valid base 64 characters. In case we want to validate the string and the validation fails, we enter the `if` statement and raise an error.

Notice how we *first* check if the user wants to validate the string and only then we run the regular expression match. We would obtain the exact same result if we changed the order of the operands to `and`, but we would be spending much more time than needed.

To show that, let us try both cases! Let's build a string with 1001 characters, where only the last one is invalid. Let us compare how much time it takes to run the boolean expression with the regex validation before and after the Boolean `validate`.

```

import timeit

## Code that sets up the variables we need to evaluate the expression that we
## DO NOT want to be taken into account for the timing.
setup = """
import re
s = b"a"*1000 + b"@"
validate = False
"""

## with short-circuiting: 0.01561140s on my machine.
print(timeit.timeit("validate and not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s)", setup))
## without short-circuiting: 27.4744187s on my machine.
print(timeit.timeit("not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s) and validate", setup))

```

Notice that short-circuiting speeds up these comparisons by a factor of ~1750.

The `timeit` module is great and I recommend you take a peek at its docs. Here, we use it to run that Boolean expression repeatedly (one million times, to be more specific).

Of course we could try longer or shorter strings, we could try strings that pass the validation and we could also try strings that fail the validation at an earlier stage, but this is just a small example that shows how short-circuiting can be helpful.

Short-circuit to flatten `if` statements

Short-circuiting can, and should, be used to keep `if` statements as flat as possible.

Conditional validation

A typical usage pattern is when we want to do some validation if certain conditions are met.

Keeping the previous b64decode example in mind, that previous if statement could've been written like so:

```
## Modified from Lib/base64.py in Python 3.9.2
def b64decode(s, altchars=None, validate=False):
    """Decode the Base64 encoded bytes-like object or ASCII string s.
    [docstring cut for brevity]
    """
    s = _bytes_from_decode_data(s)
    if altchars is not None:
        altchars = _bytes_from_decode_data(altchars)
        assert len(altchars) == 2, repr(altchars)
        s = s.translate(bytes.maketrans(altchars, b'+/'))
    # Do we want to validate the string?
    if validate:                                     # <-
        # Is the string valid?
        if not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s):    # <-
            raise binascii.Error('Non-base64 digit found')
    return binascii.a2b_base64(s)
```

Now we took the actual validation and nested it, so that we have two separate checks: one tests if we need to do validation and the other one does the actual validation. What is the problem with this? From a fundamentalist's point of view, you are clearly going against [the Zen of Python](#), that says

“Flat is better than nested.”

But from a practical point of view, you are also increasing the vertical space that your function takes up by having a ridiculous if statement hang there. What if you have multiple conditions that you need to check for? Will you have a nested if statement for each one of those?

This is exactly what short-circuiting is useful for! Only running the second part of a Boolean expression if it is relevant!

Checking preconditions before expression

Another typical usage pattern shows up when you have something you need to check, for example you need to check if a variable `names` is a list containing strings or you need to check if a given argument `term` is smaller than zero. It may happen that, in that context, it is not a good idea to do those checks immediately:

- the variable `names` might not be a list or might be empty; or
- the argument `term` might be of a different type and, therefore, might be incomparable to zero.

Here is a concrete example of what I mean:

```
## From Lib/asynchat in Python 3.9.2
def set_terminator(self, term):
    """Set the input delimiter.
```

```

Can be a fixed string of any length, an integer, or None.

"""
if isinstance(term, str) and self.use_encoding:
    term = bytes(term, self.encoding)
elif isinstance(term, int) and term < 0:
    raise ValueError('the number of received bytes must be positive')
self.terminator = term

```

This is a helper function from within the `asynchat` module. We don't need to know what is happening outside of this function to understand the role that short-circuiting has in the `elif` statement. If the `term` variable is smaller than 0, then we want to raise a `ValueError` to complain, but the previous `if` statement shows that `term` might also be a string. If `term` is a string, then comparing it with 0 raises another `ValueError`, so what we do is start by checking a necessary precondition to `term < 0`: `term < 0` only makes sense if `term` is an integer, so we start by evaluating `isinstance(term, int)` and only then running the comparison.

Let me show you another example from the `enum` module:

```

## From Lib/enum.py in Python 3.9.2
def _create_(cls, class_name, names, *, module=None, qualname=None, type=None, start=1):
    """
    Convenience method to create a new Enum class.
    """
    # [cut for brevity]

    # special processing needed for names?
    if isinstance(names, str):
        names = names.replace(',', ' ').split()
    if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
        original_names, names = names, []
        last_values = []
        for count, name in enumerate(original_names):
            value = first_enum._generate_next_value_(name, start, count, last_values[:])
            last_values.append(value)
            names.append((name, value))

    # [cut for brevity]

```

The longer `if` statement contains three expressions separated by `ands`, and the first two expressions are there to make sure that the final one,

`isinstance(names[0], str)`

makes sense. You can read along the statement and think about what it means if execution reaches that point:

```

if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
    ##^ lets start checking this `if` statement.

if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
    ##

```

```

## we only need to take a look at the right-hand side of this `and` if `names` 
## is either a tuple or a list.

if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
## 
## at this point, I've checked if `names` is a list or a tuple and I have 
## checked if it is truthy or falsy (i.e., checked if it is empty or not).
## I only need to look at the right-hand side of this `and` if `names` 
## is NOT empty.

if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
## 
## If I'm evaluating this expression, it is because `names` is either a 
## list or a tuple AND it is not empty, therefore I can index safely into it 
## with `names[0]`.

```

This flat if statement is much better than the completely nested version:

```

if isinstance(names, (tuple, list)):
    if names:
        if isinstance(names[0], str):
            pass

```

Of course, you might *need* the nested version if, at different points, you might need to do different things depending on what happens. For example, suppose you want to raise an error if the list/tuple is empty. In that case, you would need the nested version:

```

if isinstance(names, (tuple, list)):
    if names:
        if isinstance(names[0], str):
            pass
    else:
        raise ValueError("Empty names :(")

```

Can you understand why this if statement I just wrote is different from the two following alternatives?

```

## Can I put `and names` together with the first check?
if isinstance(names, (tuple, list)) and names:
    if isinstance(names[0], str):
        pass
else:
    raise ValueError("Empty names..? :(")

## What if I put it together with the second `isinstance` check?
if isinstance(names, (tuple, list)):
    if names and isinstance(names[0], str):
        pass
    else:

```

```
    raise ValueError("Empty names..? :(")
```

If this is a silly exercise for you, sorry about that! I just want you to be aware of the fact that when you have many Boolean conditions, you need to be careful when checking specific configurations of what is `True` and what is `False`.

Define default values

How it works

If you've been skimming this article, just pay attention to this section right here. This, right here, is my favourite use of short-circuiting. Short-circuiting with the Boolean operator `or` can be used to assign default values to variables.

How does this work? This uses `or` and its short-circuiting functionality to assign a default value to a variable if the current value is falsy. Here is an example:

```
greet = input("Type your name >> ") or "there"
print(f"Hello, {greet}!")
```

Try running this example and press Enter without typing anything. If you do that, `input` returns an empty string "", which is falsy. Therefore, the operator `or` sees the falsy value on its left and needs to evaluate the right operand to determine the final value of the expression. Because it evaluates the right operand, it is the right value that is returned, and "there" is assigned to `greet`.

Ensuring a list is not empty

Now that we've seen how this mechanism to assign default values works, let us take a look at a couple of usage examples from the Python Standard Library.

We start with a simple example from the `collections` module, specifically from the implementation of the `ChainMap` object:

```
## From Lib/collections/__init__.py in Python 3.9.2
class ChainMap(_collections_abc.MutableMapping):
    '''A ChainMap groups multiple dicts (or other mappings) together
    [docstring cut for brevity]
    '''

    def __init__(self, *maps):
        '''Initialize a ChainMap by setting *maps* to the given mappings.
        If no mappings are provided, a single empty dictionary is used.

        ...
        self.maps = list(maps) or [{}]
            # always at least one map
```

This `ChainMap` object allows you to combine multiple mappings (for example, dictionaries) into a single mapping that combines all the keys and values.

```

>>> import collections
>>> a = {"A": 1}
>>> b = {"B": 2, "A": 3}
>>> cm = collections.ChainMap(a, b)
>>> cm["A"]
1
>>> cm["B"]
2

```

The assignment that we see in the source code ensures that `self.maps` is a list of, at least, one empty mapping. If we give no mapping at all to `ChainMap`, then `list(maps)` evaluates to `[]`, which is falsy, and forces the `or` to look at its right operand, returning `[{}]`: this produces a list with a single dictionary that has nothing inside.

Default value for a mutable argument

I'll share another example with you, now. This example might look like the same as the one above, but there is a nice subtlety here.

First, the code:

```

## From Lib/cgitb.py in Python 3.9.2
class Hook:
    """A hook to replace sys.excepthook that shows tracebacks in HTML."""

    def __init__(self, display=1, logdir=None, context=5, file=None,
                 format="html"):
        self.display = display      # send tracebacks to browser if true
        self.logdir = logdir        # log tracebacks to files if not None
        self.context = context      # number of source code lines per frame
        self.file = file or sys.stdout # place to send the output
        self.format = format

```

This code comes from the `cgitb` module and defines `sys.stdout` to be the default value for the `self.file` variable. The definition of the `__init__` function has `file=None` as a keyword argument *also* with a default value of `None`, so why don't we just write `file=sys.stdout` in the first place?

The problem is that `sys.stdout` can be a mutable object, and therefore, using `file=sys.stdout` as a keyword argument with a default value is not going to work as you expect. This is easier to demonstrate with a list as the default argument, although the principle is the same:

```

>>> def append(val, l=[]):
...     l.append(val)
...     print(l)
...
>>> append(3, [1, 2])
[1, 2, 3]
>>> append(5)
[5]

```

```
>>> append(5)
[5, 5]
>>> append(5)
[5, 5, 5]
```

Notice the three consecutive calls `append(5)`. We would expect the three calls to behave the same way, but because a list is a mutable object, the three consecutive calls to `append` add the values to the default value itself, that started out as an empty list but keeps growing.

I'll write about mutability in more detail in future Pydon'ts, so be sure to [subscribe](#) to not miss that future Pydon't.

Find witnesses in a sequence of items

As the final usage example of short-circuiting, I'll share something really neat with you.

If you use [assignment expressions and the walrus operator](#) `:=` together with generator expressions, we can use the fact that `all` and `any` also short-circuit in order to look for "witnesses" in a sequence of elements.

If we have a predicate function `predicate` (a function that returns a Boolean value) and if we have a sequence of values, `items`, we could use

```
any(predicate(item) for item in items)
```

to check if any element(s) in `items` satisfy the predicate function.

If we modify that to be

```
any(predicate(witness := item) for item in items)
```

Then, in case any `item` satisfies the predicate function, `witness` will hold its value!

For example, if `items` contains many integers, how do we figure out if there are any odd numbers in there and how do we print the first one?

```
items = [14, 16, 18, 20, 35, 41, 100]
any_found = False
for item in items:
    any_found = item % 2
    if any_found:
        print(f"Found odd number {item}.")
        break

## Prints 'Found odd number 35.'
```

This is one alternative. What other alternatives can you come up with?

Now, compare all those with the following:

```
items = [14, 16, 18, 20, 35, 41, 100]
is_odd = lambda x: x % 2
if any(is_odd(witness := item) for item in items):
    print(f"Found odd number {witness}.")
```

```
## Prints 'Found odd number 35.'
```

Isn't this neat?

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Be mindful when you order the left and right operands to the and and or expressions, so that you can make the most out of short-circuiting.”

This Pydon't showed you that:

- and and or return the value of one of its operands, and not necessarily a Boolean value;
- both Boolean operators short-circuit:
 - and only evaluates the right operand if the left operand is truthy;
 - or only evaluates the right operand if the left operand is falsy;
- the built-in functions all and any also short-circuit;
- short-circuiting also happens in chained comparisons, because those contain an implicit and operator;
- using short-circuiting can save you a lot of computational time;
- nested structures of if statements can, sometimes, be flattened and simplified if we use short-circuiting with the correct ordering of the conditions;
- it is customary to use short-circuiting to test some preconditions before applying a test to a variable;
- another great use-case for short-circuiting is to assign default values to variables and function arguments, especially if the default value is a mutable value; and
- short-circuiting, together with the walrus operator :=, can be used to find a witness value with respect to a predicate function.

References

- Python 3 Documentation, The Python Standard Library, Built-in Types, Boolean Operations – and, or, not, <https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not> [last accessed 31-05-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, all, <https://docs.python.org/3/library/functions.html#all> [last accessed 26-05-2021];
- Python 3 Documentation, The Python Language Reference, Built-in Functions, any, <https://docs.python.org/3/library/functions.html#any> [last accessed 26-05-2021];
- Stack Overflow, “Does Python support short-circuiting”, <https://stackoverflow.com/a/14892812/2828287> [last accessed 31-05-2021];
- Python 3 Documentation, The Python Standard Library, base64, <https://docs.python.org/3/library/base64.html> [last accessed 01-06-2021];
- Python 3 Documentation, The Python Standard Library, asynchat, <https://docs.python.org/3/library/asynchat.html> [last accessed 01-06-2021];
- Python 3 Documentation, The Python Standard Library, enum, <https://docs.python.org/3/library/enum.html> [last accessed 01-06-2021];

- Python 3 Documentation, The Python Standard Library, `collections.ChainMap`, <https://docs.python.org/3/library/collections.html#collections.ChainMap> [last accessed 01-06-2021];
- Python 3 Documentation, The Python Standard Library, `cgitb`, <https://docs.python.org/3/library/cgitb.html> [last accessed 01-06-2021];
- Real Python, “How to Use the Python or Operator”, <https://realpython.com/python-or-operator/> [last accessed 01-06-2021];

The power of reduce



(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/the-power-of-reduce>.)

Introduction

In this Pydon't I'll talk about `reduce`, a function that used to be a built-in function and that was moved to the `functools` module with Python 3.

Throughout all of the Pydon'ts I have been focusing only on Python features that you can use without having to import anything, so in that regard this Pydon't will be a little bit different.

In this Pydon't, you will:

- see how `reduce` works;
- learn about the relationship between `reduce` and `for` loops;

- notice that `reduce` hides in a handful of other built-in functions we all know and love;
- learn about a neat use-case for `reduce`;

How `reduce` works

`reduce` is a tool that is typically associated with functional programming, which is a programming paradigm that I feel sometimes is underappreciated. In a short sentence, `reduce` takes an iterable and a binary function (a function that takes two arguments), and then uses that binary function to boil the iterable down to a single value.

This might sound weird or complicated, so the best thing I can do is to show you a simplified implementation of `reduce`:

```
def reduce(function, iterable, initial_value):
    result = initial_value
    for value in iterable:
        result = function(result, value)
    return result
```

If you look at it, there really isn't much going on inside that `for` loop: we just keep updating the `result` variable with the argument `function` and the consecutive values in the `iterable` argument.

But I can make this even easier to understand for you. And, in order to do that, I just have to point out a bunch of `reduce` use cases that you use all the time! Perhaps the simplest one, and the one that shows up more often, is the `sum` built-in:

```
>>> sum(range(10))
45
>>> from functools import reduce; import operator
>>> reduce(operator.add, range(10))
45
```

The `operator.add` there is just a way to programmatically refer to the built-in addition with `+` in Python.

From the [documentation on `operator`](#),

“The `operator` module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`.”

You probably have seen `sum` before, right? It just adds up all the elements in an iterable. That's basically what `reduce` does, if the function we give it is the addition. To make the connection clearer, let's implement our own `sum` function:

```
def sum(iterable):
    acc = 0
    for elem in iterable:
        acc += elem
    return acc
```

Are you comfortable with the implementation above? Now let me rejig it a little bit:

```
def sum(iterable, start=0):
    acc = start
    for elem in iterable:
        acc = acc + elem
    return acc
```

Now, our `sum` function can start adding up at a different value and we use the operator `.add` function instead of using `+` or modified assignment `+=`. Let us now stack this alternative implementation side by side with the original `reduce` implementation:

```
def sum(iterable, start=0):      # def reduce(function, iterable, initial_value):
    acc = start                  #     result = initial_value
    for elem in iterable:         #     for value in iterable:
        acc = acc + elem          #         result = function(result, value)
    return acc                   #     return result
```

Can you see how they are the same thing?

The rabbit hole of the built-in reductions

Some built-ins

Now that we have seen that `sum` is a reduction, what other built-in functions are reductions? Well, part of what a reduction does is taking an iterable and *reducing* it to a single value. What built-in functions do that?

Going through the [list of built-in functions](#) in the docs, here are some functions that catch my attention:

- `all` – expects an iterable of truthy/falsy values and returns a Boolean;
- `any` – expects an iterable of truthy/falsy values and returns a Boolean;
- `max` – accepts an iterable of numbers and returns a single number;
- `min` – accepts an iterable of numbers and returns a single number;
- `sum` – we've seen this one already;

Can you implement all of these with a `for` loop? Can you write all of these as reductions?

I'll give you a hand with the reductions:

```
>>> all = lambda iterable: reduce(operator.and_, iterable)
>>> any = lambda iterable: reduce(operator.or_, iterable)
>>> # Define `max` on iterables at the expense of just the binary max.
>>> max_ = lambda a, b: a if a >= b else b
>>> max = lambda iterable: reduce(max_, iterable)
>>> # Define `min` on iterables at the expense of just the binary min.
>>> min_ = lambda a, b: a if a <= b else b
>>> min = lambda iterable: reduce(min_, iterable)
>>> sum = lambda iterable: reduce(operator.add, iterable)
```

I just find it very interesting that there are so many reductions amongst the built-in functions! That makes you think that `reduce` really is a powerful tool, right? Given that it is worth adding *five* specialised reductions to the built-ins...

Other common reductions

And there is more, of course. If we use `operator.mul` (for multiplication), then we get the `math.prod` function that we can use to multiply all the numbers in an iterable:

```
>>> from math import prod
>>> prod(range(1, 11))  # 10!
3628800
>>> reduce(operator.mul, range(1, 11))
3628800
```

What if you have a bunch of strings that you want to piece together? For example, what if you have a list of words that you want to put back together, separated by spaces?

```
>>> words = ["Do", "I", "like", "reductions?"]
>>> " ".join(words)
'Do I like reductions?'
```

If we define “string addition” to be the concatenation of the two strings, but with a space in the middle, then we get the same thing:

```
>>> reduce(lambda s1, s2: s1 + " " + s2, words)
'Do I like reductions?'
```

Now, please don’t get me wrong. I am *not* suggesting you start using `reduce` when you need to join strings. I am *just* trying to show you how these patterns are so common and appear in so many places, even if you don’t notice them.

Why bother?

Why should you bother with knowing that `reduce` exists, and how it works? Because that is what “learning Python” means: you need to be exposed to the library, to the built-ins, you need to learn new algorithms, new ways of doing things, new tools.

`reduce` is another tool you now have in your toolbelt. Maybe it is not something you will use every day. Maybe it is something you will use once a year. Or even less. But when the time comes, you *can* use it, and your code will be better for that: because you know how to use the right tool for the job.

People learn a lot by building knowledge on top of the things that they already learned elsewhere... And the more you learn elsewhere, the more connections with different things you can make, and the more things you can discover. Maybe this article does nothing for you, but maybe this article was the final push you needed to help something else click. Or maybe it feels irrelevant now, but in 1 week, 1 month, or 1 year, something else will click *because* you took the time to learn about `reduce` and to understand how it relates to all these other built-in functions.

Far-fetched reductions

The reductions above were reductions that are more “normal”, but we can do all kinds of interesting things with `reduce`! Skip this section altogether if you are starting to feel confused or repulsed by reductions, I

don't want to damage your relationship with `reduce` beyond repair. This section contains some reductions that are – well, how to put this nicely..? – that are not necessarily suitable for production.

First and last

Here's is a little amusing exercise for you. Can you write a reduction that, given an iterable, returns its first element? Similarly, can you write a reduction that, given an iterable, returns its last element?

Give it some thought, really.

Ok, here are my proposed solutions:

```
>>> left = lambda l, r: l
>>> reduce(left, range(10))
0
>>> right = lambda l, r: r
>>> reduce(right, range(10))
9
```

[I wrote the text above, a couple of hours went by, and then I came back.]

I have to be honest with you: I started out thinking these are crazy, but in all honesty, how do you write a function to retrieve the *last* element of an iterable? Mind you, iterables are not necessarily indexable, so something like `iterable[-1]` isn't guaranteed to work. How would you do it? You could write a `for` loop:

```
def get_last(iterable):
    for elem in iterable:
        last = elem
    return last
```

But why is that any better than the alternative below?

```
from functools import reduce
def get_last(iterable):
    return reduce(lambda l, r: r, iterable)
```

Feel free to leave your opinions in the comments below. *I* actually like the `reduce` alternative.

Creating built-in types

Another couple of reductions I wouldn't recommend for production are the replacements for `dict`, `list`, `set`, and `tuple`. For example, if you have an iterable, how do you build the corresponding tuple? How do you write that as a reduction? Well, for this one you need to remember that `reduce` accepts a third argument that is the initial value that we are modifying...

Do you get it? The third argument needs to be an empty tuple:

```
>>> reduce(lambda t, v: t + (v,), range(10), ())
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

With similar workarounds, you can redefine `dict`, `list`, and `set`, as reductions. Again, not that I recommend that.

The identity element...

...or lack thereof

We have seen some reductions already and, if you were brave enough, you even took a sneak peek at some crazy reductions in the previous section. However, up until now, I have been (purposefully) not giving much attention to the third argument to `reduce`. Let us discuss it briefly.

First, why do we need a third argument to `reduce`? Well... because we like things to work:

```
>>> from functools import reduce
>>> import operator
>>> sum([1, 2])
3
>>> reduce(operator.add, [1, 2])
3
>>> sum([1])
1
>>> reduce(operator.add, [1])
1
>>> sum([])
0
>>> reduce(operator.add, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

From a strictly practical point of view, the third argument to `reduce` exists so that `reduce` can know what to return in case the given iterable is empty. This means that, in general, you don't need to worry about that argument if you know your iterables are never going to be empty...

The documentation is quite clear with regards to how it uses this third argument, to which they refer as `initializer`:

"If the optional `initializer` is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If `initializer` is not given and iterable contains only one item, the first item is returned." [functools.reduce Python 3 docs, 8th June 2021].

So, in practical terms, you only *really* need the `initializer` when the iterable is empty, and therefore you should use it when it might happen that you pass an empty iterable into `reduce`.

What is the identity element

So, if you cannot be 101% sure your iterable is not going to be empty, how do you decide what value to use in the third argument to `reduce`? How do you pick the `initializer` argument? Well, the value that `initializer` should have depends on the function you are using in your reduction and, in particular, the `initializer` should be an identity element for that function. What does that mean?

Again, from a very practical perspective, the identity element is a special element with a very special behaviour: the identity element is such that, if the iterable is not empty, having the identity element or not should be exactly the same thing. In other words, when in the presence of other values, the identity element should have no effect at all.

For example, if we are multiplying a list of numbers, what is the identity element that we should feed `reduce` with? What is the number that, when multiplied by some other numbers, does exactly nothing? It is 1:

```
>>> from functools import reduce
>>> reduce(operator.mul, range(4, 10))
60480
>>> reduce(operator.mul, range(4, 10), 1)
60480
```

For the built-in reductions, you can generally figure out what the identity element is by trying to call the reduction with an empty iterable:

```
>>> sum([])
0
>>> import math
>>> math.prod([])
1
>>> all([])
True
>>> any([])
False
>>> max([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
```

`max` and `min` are interesting reductions because, from the mathematical point of view, they have suitable identity elements:

- for `max`, the identity element is $-\infty$; and
- for `min`, the identity element is ∞ .

Why is that? Again, because these are the values that will not impact the final result when mixed in with other numbers.

Take a look at the following excerpt from my session:

```
>>> max(float("-inf"), 10)
10
>>> max(float("-inf"), -132515632534250)
-132515632534250
>>> max(float("-inf"), 67357321)
67357321
```

These six lines of the session show three instances of how calling `max` with minus infinity as one of the arguments always returns the other one, because no number is smaller than minus infinity.

However, `max` and `min` will throw an error if you call them with empty iterables, *even though* there is an identity element that you could use.

```
>>> max([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
```

Maybe they do this so that people don't have to deal with infinities in their programs? I honestly don't know!

Edit: I went online and [asked people](#), and the answer that made the most sense to me is that `max` and `min` can be used with any comparable objects, and for other objects, the infinities might make absolutely no sense.

For example, `max("abc", "da")` returns "da", and when comparing strings it really makes no sense to add `float("-inf")` to the mix.

The identity element doesn't always exist

There are some operations that look like sensible reductions but that just *don't* have an identity element. Even if you skipped the section on "scary" reductions, I showed you a reduction that does not have an identity element. Can you spot it? (Drop a comment below with your guess).

If you have an operation for which you cannot find an identity element, then you are either going down the wrong road – and you really shouldn't use a reduction – or you need to wrap your reduction with an `if`-statement or a `try` statement (take a look at [this](#) article to help you understand which one to choose).

Why some people dislike `reduce`

If `reduce` is such a powerful tool, why was it moved from the built-ins into `functools`? More importantly, why do people dislike `reduce`?

There is [some information](#) online about why `reduce` was moved into `functools`, but I can only speak about my experience with `reduce` and how I have seen people around me react to it.

One of the things I have seen is that people look at `reduce` as if it were a tool that people only use when they are trying to be smart, but I think that is just prejudice against `reduce`. Sometimes, it may be difficult to draw the line between what is code that is worth having people think about for a bit, versus code that isn't.

Furthermore, sometimes functions like `reduce` are used in convoluted academic exercises, or in brain-teasers, that are meant just to jog your brain. People then forget those are not indications of how `reduce` should be used in the wild, and build these bitter feelings for such wonderful tools.

Examples in code

I looked for usages of `reduce` in the Python Standard Library and I didn't find many, but I found one usage pattern (in two different places) and I just found it to be really elegant, and that's what I am sharing with you here.

Other than that, even if you are not explicitly using `reduce`, just remember that functions like `sum`, `math.prod`, `max`, `min`, `all`, `any`, etc, are pervasive in our code and, whether you like it or not, you are using reductions in your own code.

Reaching inside nested dictionaries

In case you want to take a look at the original pattern, you can find it in the `importlib.metadata.EntryPoint.load` function, but I'll change it a little bit to make it simpler.

Say you have a series of nested dictionaries:

```
>>> d = {"one": {2: {"c": {4: 42}}}}
```

Now, say that you want to access the nested 42 through a series of successive key accesses that you have in a list:

```
>>> keys = ["one", 2, "c", 4]
```

How do you reach the inner 42? Well, you can write a loop:

```
>>> d = {"one": {2: {"c": {4: 42}}}}
>>> val = d
>>> for key in keys:
...     val = val[key]
...
>>> val
42
```

But we can, once more, compare that `for` loop with the definition of the reduction:

```
# def reduce(function, iterable, initial_value):
val = d                      #     result = initial_value
for key in keys:              #     for value in iterable:
    val = val[key]           #         result = function(result, value)
val                           #     return result
```

So we can see the structure is very similar! We just have to figure out what is the correct function to use, and that is `dict.get`:

```
>>> reduce(dict.get, keys, d)
42
```

Isn't this neat?

Reaching inside nested classes

Similarly, we can use this pattern to programmatically access class attributes that are deeply nested.

Let me define a class with nothing in it:

```
>>> class C:  
...     pass  
...
```

Now, let me create a couple of instances and nest them:

```
>>> c = C()  
>>> c.one = C()  
>>> c.one._2 = C()  
>>> c.one._2.c = C()  
>>> c.one._2.c._4 = 42
```

If I have the base instance `c`, and if I have the names of the successive attributes that lead to 42, how do I get there? Well, instead of using `dict.get`, we can use `getattr`:

```
>>> attrs = ["one", "_2", "c", "_4"]  
>>> reduce(getattr, attrs, c)  
42
```

I'll be writing about `getattr` soon, so be sure to [subscribe](#) to stay tuned.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Reductions are classical techniques that you use frequently, even if you do not realise you are doing so!”

This Pydon't showed you that:

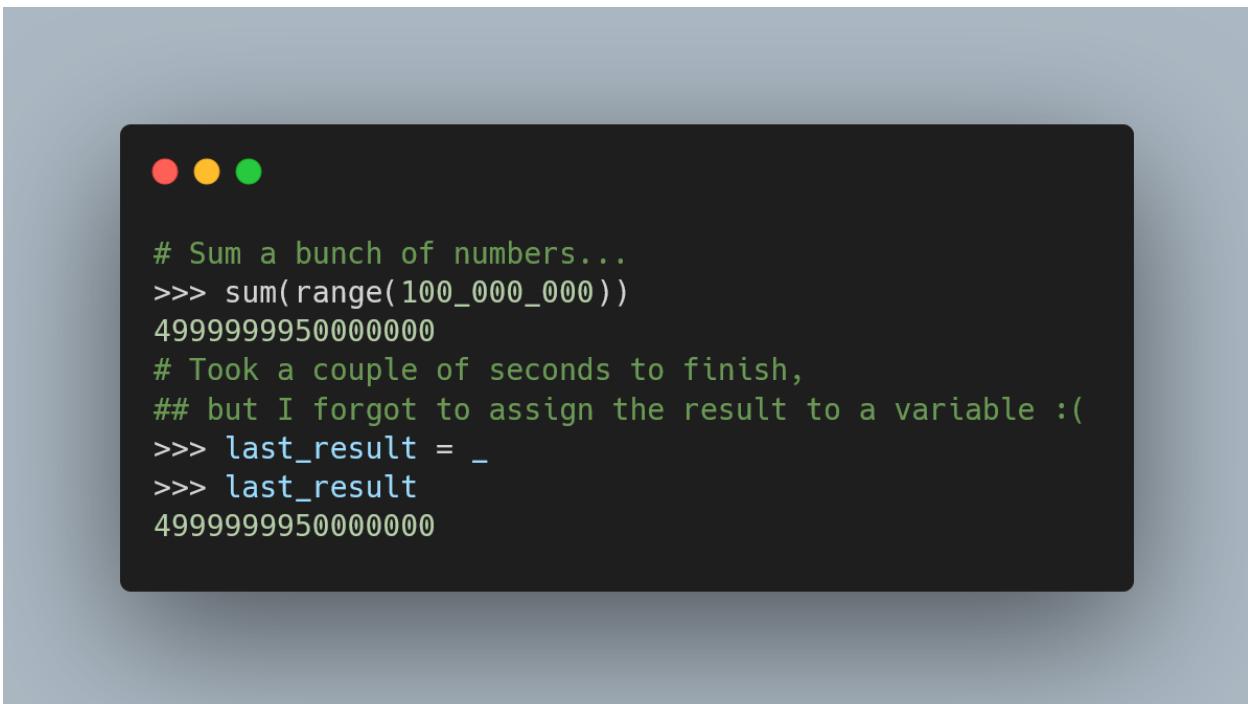
- `reduce` takes an iterable of objects and applies a function successively, to build a single final object;
- `reduce` was a built-in function in Python 2 and in Python 3 it lives in the `functools` module;
- reductions can be converted to `for` loops and back following a very well-defined pattern;
- built-in functions like `sum`, `max`, `min`, `any`, and `all`, are reductions;
- a reduction can work with an optional third argument, to initialise the process, and that element is supposed to be the identity element of the function you are using;
- not all functions have identity elements;
- the `operator` module allows you to access built-in operations, like addition and subtraction, and pass them around your code; and
- `reduce` can be used to reach programmatically inside nested dictionaries or class attributes.

References

- Python 3 Documentation, The Python Standard Library, Built-in Functions, <https://docs.python.org/3/library/functions.html> [last accessed 07-06-2021];
- Python 2 Documentation, The Python Standard Library, Built-in Functions, `reduce`, <https://docs.python.org/2.7/library/functions.html#reduce> [last accessed 06-06-2021];
- Python 3 Documentation, The Python Standard Library, `functools.reduce`, <https://docs.python.org/3/library/functools.html#functools.reduce> [last accessed 06-06-2021];

- Python 3 Documentation, The Python Standard Library, `operator`, <https://docs.python.org/3/library/operator.html> [last accessed 07-06-2021];
- Artima Weblogs, “The fate of `reduce()` in Python 3000” by Guido van Rossum, <https://www.artima.com/weblogs/viewpost.jsp?thread=98196> [last accessed 06-06-2021];
- Real Python, “Python’s `reduce()`: From Functional to Pythonic Style”, <https://realpython.com/python-reduce-function/> [last accessed 06-06-2021];
- Stack Overflow, “Why don’t `max` and `min` return the appropriate infinities when called with empty iterables?”, <https://stackoverflow.com/q/67894680/2828287> [last accessed 08-06-2021];

Usages of underscore



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/usages-of-underscore>.)

Introduction

In this Pydon't we will take a look at all the use cases there are for `_` in Python. There are a couple of places where `_` has a very special role syntactically, and we will talk about those places. We will also talk about the uses of `_` that are just conventions people follow, and that allow one to write more idiomatic code.

In this Pydon't, you will:

- learn about the utility of `_` in the Python REPL;
- learn what `_` does when used as a prefix and/or suffix of a variable name;

- a single underscore used as a suffix;
- a single underscore used as a prefix;
- double underscore used as a prefix;
- double underscore used as a prefix and suffix;
- see the idiomatic usage of `_` as a “sink” in assignments;
- and understand how that was extended to `_`’s role in the new `match` statement;
- see the idiomatic usage of `_` in localising strings; and
- learn how to use `_` to make your numbers more readable.

Recovering last result in the session

Have you ever called a slow function in the Python session and then lost the return value because you forgot to assign it to a variable? I know I have done that countless times! Because of people like (you and) me, someone made the *best* decision ever, and decided that `_` can be used in the Python session to refer to the last return result:

```
>>> 1 + 1
2
>>> _
2
>>> sum(range(100_000_000))      # Takes a couple of seconds to finish.
499999950000000
>>> _
499999950000000
>>> save_for_later = _
>>> save_for_later
499999950000000
```

This prevents you from having to re-run the previous line of code, which is especially helpful if the previous line of code takes some time to finish, if it had side-effects that you don’t want to trigger again, or even if it can’t be re-run (e.g. because you deleted a file or because you exhausted an iterable).

So, next time you are playing around in the interpreter session and forget to assign the result of a function call, or some other piece of code, remember to use `_` to refer back to it.

Notice that if you explicitly assign to `_`, then the value you assign will stay there until you explicitly delete it. When you delete it, then `_` will go back to referring to the last returned result:

```
>>> _ = "hey"
>>> "_ was explicitly assigned."
'_ was explicitly assigned.'
>>> _
'hey'
>>> del _
>>> "_ is no longer explicitly assigned."
'_ is no longer explicitly assigned.'
>>> _
'_ is no longer explicitly assigned.'
```

Prefixes and suffixes for variable names

Single underscore as a suffix

As you know, some words have a special meaning in Python, and are therefore dubbed as keywords. This means we cannot use those names for our variables. Similarly, Python defines a series of built-in functions that are generally very useful and ideally we would like to avoid using variable names that match those built-in names.

However, there are occasions in which the perfect variable name is either one of those keywords or one of those built-in functions. In those cases, it is common to use a single `_` as a suffix to prevent clashes.

For example, in statistics, there is a random distribution called the “exponential distribution” that depends on a numeric parameter, and that parameter is typically called “lambda” in the mathematical literature. So, when `random` decided to implement that distribution in `random.expovariate`, they would ideally like to use the word `lambda` as the parameter to `random.expovariate`, but `lambda` is a reserved keyword and that would throw an error:

```
>>> def expovariate(lambda):
    File "<stdin>", line 1
        def expovariate(lambda):
            ^
SyntaxError: invalid syntax
```

Instead, they could have named the parameter `lambda_`. (The implementers ended up going with `lambda`, however.)

There are many examples in the Python Standard Library where the implementers opted for the trailing underscore. For example, in the code for IDLE (the IDE that comes by default with Python and that is implemented fully in Python) you can find this function:

```
## From Lib/idlelib/help.py in Python 3.9.2
def handle_starttag(self, tag, attrs):
    "Handle starttags in help.html."
    class_ = ''
    for a, v in attrs:
        if a == 'class':
            class_ = v
    # Truncated for brevity...
```

Notice the `class_` variable that is defined and updated inside the loop. “class” would be the obvious variable name here because we are dealing with HTML classes, but `class` is a reserved keyword that we use to define, well, classes... And that's why we use `class_` here!

Single underscore as prefix

While the usage of a single underscore as a suffix was more or less a convention, the usage of a single underscore as a prefix is both a convention and something that affects some Python programs.

Let me start by explaining the convention: when you define a name that starts with a single underscore, you are letting other programmers know that such a name refers to something that is for internal use only, and that outside users shouldn't mess around with.

For example, suppose that you are implementing a framework for online shops, and you are now writing the part of the code that will fetch the price of an item. You could write a little function like so:

```
prices = {  
    "jeans": 20,  
    "tshirt": 10,  
    "dress": 30,  
}  
  
def get_price(item):  
    return prices.get(item, None)
```

Now, shops nowadays can't do business without having sales from time to time, so you add a parameter to your function so that you can apply discounts:

```
def get_price(item, discount=0):  
    p = prices.get(item, None)  
    if p is not None:  
        return (1 - discount)*p  
    else:  
        return p
```

Now all is good, except you think it might be a good idea to validate the discount that the function is trying to apply, so that discounts are never negative or greater than 100%. You could do that in the main function, or you can devise a helper function to do that for you, probably because you will need to verify that discount amounts are correct in a variety of places.

So, you write your helper function:

```
def valid_discount(discount):  
    return 0 <= discount <= 1
```

By the way, if you want to learn more about the fact that Python allows the chaining of comparisons, like what you see above, you can read [this Pydon't](#) on the subject.

Now you have a way to validate discounts and you can use that:

```
def get_price(item, discount=0):  
    if not valid_discount(discount):  
        raise ValueError(f"Trying to apply an illegal discount on {item}.")  
    p = prices.get(item, None)  
    if p is not None:  
        return (1 - discount)*p  
    else:  
        return p
```

Perfect! The codebase for your online shop management framework is well on its way.

Now imagine, for a second, that you are a user of your framework, and not an implementer. You will probably install the framework from PyPI, with pip, or maybe directly from GitHub. But when you do, and when you import the code to start using it, you will import the `get_price` and the `valid_discount` functions. Now, you need the `get_price` function but you don't need the `valid_discount` because the whole framework already protects the user from illegal discounts and negative prices and whatnot! In other words, the `valid_discount` function is more relevant to the internals of the framework than to users of the framework. Except the user probably doesn't know that, because the user sees the `valid_discount` function and it is fair to assume that the user will think they have to use that function to validate discounts for themselves... How could they know they don't need to?

One solution would be for you to follow the convention we just started discussing! If you name your function just a tad differently:

```
def _valid_discount(discount):
    return 0 <= discount <= 1
```

The user of the framework immediately understands "oh, I don't have to worry about this function because its name starts with a single underscore". Not only that, but Python even helps users not worry about those functions with leading underscores.

Go ahead and write the following in your `onlineshop.py` file:

```
## onlineshop.py
def _valid_discount(discount):
    return 0 <= discount <= 1

prices = {
    "jeans": 20,
    "tshirt": 10,
    "dress": 30,
}

def get_price(item, discount=0):
    if not _valid_discount(discount):
        raise ValueError(f"Trying to apply an illegal discount on {item}.")
    p = prices.get(item, None)
    if p is not None:
        return (1 - discount)*p
    else:
        return p
```

After you do that, open your Python REPL, import everything from `onlineshop` and try getting some prices and discounts:

```
>>> from onlineshop import *
>>> get_price("jeans")
20
>>> get_price("jeans", discount=0.5)
10.0
>>> get_price("jeans", discount=1.3)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\rodri\Documents\mathspp\onlineshop.py", line 13, in get_price
    raise ValueError(f"Trying to apply an illegal discount on {item}.")
ValueError: Trying to apply an illegal discount on jeans.
```

Notice how both functions appear to be working just fine, and notice that we got an error on the last call because 1.3 is too big of a discount, so the `_valid_discount` function said it wasn't valid.

Let us check it for ourselves:

```
>>> _valid_discount(1.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_valid_discount' is not defined
```

We get a `NameError` because the `_valid_discount` function isn't defined... Because it was never imported! The function was not imported into your code, even though the original code can still use it internally. If you really need to access `_valid_discount`, then you either import it explicitly, or you just import the module name and then access it with its dotted name:

```
>>> from onlineshop import _valid_discount
>>> _valid_discount(0.5)
True
>>> import onlineshop
>>> onlineshop._valid_discount(1.3)
False
```

This mechanism also works with the variables, as long as their name starts with a leading underscore. Go ahead and rename the `prices` variable to `_prices`, close the REPL, open it again, and run `from onlineshop import *`. `_prices` will not be defined!

So, on the one hand, notice that a leading underscore really is an *indication* of what things you should and shouldn't be concerned with when using code written by others. On the other hand, the leading underscore is *just* an indication, and it won't prevent others from accessing the names that you write with a leading underscore.

Finally, there is one other way of controlling what gets imported when someone uses the `*` to import everything from your module: you can use the `__all__` variable to specify the names that should be imported on that occasion.

Go ahead and add the following line to the top of your `onlineshop.py` file:

```
__all__ = ("get_price", "_valid_discount")
```

After you do that, close your REPL and reopen it:

```
>>> from onlineshop import *
>>> get_price
<function get_price at 0x0000029410907430>
>>> _valid_discount
<function _valid_discount at 0x0000029410907280>
```

```
>>> prices
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'prices' is not defined
```

Notice that all the names inside `__all__` were imported, regardless of them starting with a single underscore or not, and the names that were *not* listed did not get included. In my example, my variable was named `prices` (so it didn't even have a leading underscore!) and it was not imported.

This `__all__` variable is the perfect segue into the next subsection:

Leading and trailing double underscores

In Python, a name that starts and ends with double underscores is a name that has internal relevance to Python. For example, many functions like `__str__`, `__repr__`, `__bool__`, and `__init__`, are sometimes referred to as “magic” functions because they interact, in some way, with Python’s “internal” functioning.

A better name for these magic functions and variables is “dunder function”, or “dunder variable”, or “dunder method”, depending on the context. (The word “dunder” – a common word in the Python world – is short for “double underscore”!)

However, these dunder names are not really magical: they are *just* functions. (Or variables, just like `__all__`.) What you can know is that when you find a name that starts and ends with a double underscore, chances are, it is a name that interacts with Python’s syntax in some way.

For example, what calling the `str` built-in function with some argument do is exactly the same as calling the `__str__` function of that same argument:

```
>>> n = 3
>>> str(n)
'3'
>>> n.__str__()
'3'
```

Of course writing `str(n)` looks much nicer than `n.__str__()`, but this just tells you that if you define your own objects, you need to implement the `__str__` method so that your objects can be given as arguments to the `str` built-in. (I wrote about `str`, `__str__`, `repr`, and `__repr__` in more detail [here](#), so give that Pydon’t a read if you need.)

So, in conclusion, double leading and trailing underscores are used for functions and variables with some “special” meaning that often has to do with the default Python behaviour.

Don’t use (create) dunder names in your own programs, so that you don’t trip on something unexpected and to avoid collisions with future changes/additions to the Python language!

Double leading underscore

In this subsection we will take a look at what happens when you use a double underscore in the beginning of a name. A double underscore, in the beginning of a name, has a special use case: you use it for variables and methods that you would wish to “protect” with the leading underscore (so that users know to leave it alone) but that have such common names that you are afraid others might overwrite them.

What does this mean?

First, let us see this in action. Modify the `onlineshop.py` file so that our code now belongs to a class called `OnlineShop`:

```
## onlineshop.py
class OnlineShop:
    __prices = {
        "jeans": 20,
        "tshirt": 10,
        "dress": 30,
    }

    def _valid_discount(self, discount):
        return 0 <= discount <= 1

    def get_price(self, item, discount=0):
        if not self._valid_discount(discount):
            raise ValueError(f"Trying to apply an illegal discount on {item}.")
        p = self.__prices.get(item, None)
        if p is not None:
            return (1 - discount)*p
        else:
            return p
```

Notice that the `prices` variable now is `__prices`. Let us take this little class for a spin:

```
>>> from onlineshop import OnlineShop as OS
>>> shop = OS()
>>> shop.get_price("jeans")
20
```

The code appears to be working, so now let us take a look at the `__prices` variable:

```
>>> shop.__prices
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'OnlineShop' object has no attribute '__prices'
```

Uh oh, an error again! We can't reach the `__prices` variable, even though the `get_price` method clearly makes (successful!) use of it. Why can't we reach the `__prices` variable? Well, we can use the `dir()` built-in to list all the attributes of our `shop` object:

```
>>> dir(shop)
['__OnlineShop__prices', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_valid_discount', 'get_price']
```

Go ahead and look for the names of the things we defined. Can you find the `_valid_discount` and `get_price` functions? What about `__prices`? You won't be able to find `__prices` in that list, but the very first item of the list is `_OnlineShop__prices`, which looks awfully related.

Remember when I said that a double leading underscore is used to avoid name collisions? Well, there's a high chance that people might want to create a variable named `prices` if they extend your online shop framework, and you might still need your original `prices` variable, so you have two options:

- give a huge, very complicated, name to your `prices` variable, so that it becomes highly unlikely that others will create a variable with the same name; or
- you use `__prices` to ask Python to *mangle* the variable name, to avoid future collisions.

Going with the second option meant that Python took the original variable name, which was `__prices`, and prepended the class name to it, plus an additional leading underscore, so that users still know they should leave that name alone. That is the explicit name you can use to reach that variable from outside the class:

```
>>> shop._OnlineShop__prices
{'jeans': 20, 'tshirt': 10, 'dress': 30}
```

This name mangling facility works for both variables and functions, so you could have a `__valid_discount` method that would look like `_OnlineShop__valid_discount` from outside of the class, for example.

It is highly likely that you won't have the need to use double leading underscores in your code, but I couldn't just ignore this use case!

Underscore as a sink

One of my favourite use cases for the underscore is when we use the underscore as the target for an assignment. I am talking about the times we use `_` as a variable name in an assignment.

It is a widely-spread convention that using `_` as a variable name means "I don't care about this value". Having said this, you should be asking yourself this: If I don't care about a value, why would I assign it in the first place? Excellent question!

Doing something like

```
_ = 3      # I don't care about this 3.
```

is silly. Using the underscore as a sink (that is, as the name of a variable that will hold a value that I do *not* care about) is useful in *other* situations.

Unpacking

I have written at length about unpacking in other Python's:

- ["Unpacking with starred assignments"](#)
- ["Deep unpacking"](#)

Unpacking is a feature that lets you, well, unpack multiple values into multiple names at once. For example, here is how you would split a list into its first and last items, as well as into the middle part:

```
>>> first, *mid, last = range(0, 10)
>>> first
0
>>> mid
[1, 2, 3, 4, 5, 6, 7, 8]
>>> last
9
```

Isn't this neat? Well, it is! But what if you only cared about the first and last items? There are various options, naturally, but I argue that the most elegant one uses `_` as a sink for the middle part:

```
>>> first, *_, last = range(0, 10)
>>> first
0
>>> last
9
```

Why is this better than the alternative below?

```
>>> sequence = range(0, 10)
>>> first, last = sequence[0], sequence[-1]
```

Obviously, `sequence = range(0, 10)` is just an example of a sequence. If I knew in advance this were the sequence I'd be using, then I would assign `first = 0` and `last = 9` directly. But for generic sequences, the two use cases behave differently.

Can you figure out when? I talk about that in [this Pydon't](#).

The behaviour is different when `sequence` has only one element. Because they behave differently, there might be cases where you *have* to use one of the two alternatives, but when you are given the choice, the unpacking looks more elegant and conveys the intent to split the sequence in its parts better.

Of course `_` is a valid variable name and you can ask for its value:

```
>>> first, *_, last = range(0, 10)
>>> _
[1, 2, 3, 4, 5, 6, 7, 8]
```

But when I see the `*_` in the assignment, I immediately understand the semantics of that assignment as "ignore the middle part of the range".

This can also be used when you are unpacking some structure, and only care about specific portions of the structure. You could use indexing to access the specific information you want:

```
>>> colour_info = ("lightyellow", (255, 255, 224))
>>> blue_channel = colour_info[1][2]
>>> blue_channel
224
```

But if the `colour_info` variable is malformed, you will have a hard time figuring that out. Instead, using unpacking, you can assert that the structure is correct and at the same time only access the value(s) that matter:

```
>>> colour_info = ("lightyellow", (255, 255, 224))
>>> _, (_, _, blue_channel) = colour_info
>>> blue_channel
224
```

Iterating independently of the iteration number

Another similar use case shows up when you need to iterate with a `for` loop, but you really do *not* care about the iteration number you are in. For example, say that you want to generate 5 random integers between 0 and 20. How would you write that? I would write it as such:

```
>>> import random
>>> nums = [random.randint(0, 20) for _ in range(5)]
[16, 1, 17, 3, 1]
```

Why did I use `_` in front of `for`? Because the expression I am running repeatedly does not depend on the iteration count, it is independent of that count. So, in order to convey that meaning more clearly, I use the `_` as a sink for the iterator variable.

Again, `_` is a perfectly valid variable name and I could use it in the expression itself:

```
>>> [_ + 2 for _ in range(5)]
[2, 3, 4, 5, 6]
```

But the point is that using `_` as a sink is a *convention* to make the semantics of your programs more clear.

Matching everything in the new match statement

The new `match` statement is coming in Python 3.10, and [there is much to look forward to](#). Following the spirit of the common use case of using `_` as a sink in assignments, the underscore will also be used in the new `match` statement as the wildcard that matches “anything else”:

```
## Needs Python 3.10 to run
>>> v = 10
>>> match v:
...     case 0:
...         print("null")
...     case 1:
...         print("uno")
...     case 2:
...         print("two")
...     case _:
...         print("whatever")
...
whatever
```

And in the case of the `match` statement, it is a true sink: you cannot use the `_` to refer to the original value, so in the `match` statement, `_` really means “I don’t care”! Take a look:

```

>>> v = 10
>>> match v:
...     case _:
...         print(_)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name '_' is not defined

```

If you want to match anything else *and* be able to refer to the original value, then you need to use a valid target name:

```

>>> v = 10
>>> match v:
...     case wtv:
...         print(wtv)
...
10

```

String localisation

Another niche use case for the underscore, but that I find absolutely lovely, is for when you need to localise your programs. Localising a program means making it suitable for different regions/countries. When you do that, one of the things that you have to do is translate the strings in your program, so that they can be read in many different languages.

How would you implement a mechanism to enable your program to output in (arbitrarily many) different languages? Do think about that for a second, it is a nice challenge! Assume you can't use modules built specifically for localisation.

Whatever you do, for example a function call or accessing a dictionary, is going to happen in various places and is going to generate too much noise. If your program has plenty of strings, going from

```

print("Hello, world!")
to
print(translate("Hello, world!"))

```

may look harmful, but in a program with many strings, all the `translate` calls will add a lot of visual clutter. So, it is common practice to create an alias to a function like the `translate` function and call it `_`. Then, localising a string doesn't add much visual clutter:

```
print_("Hello, World!")
```

This is just a *convention*, but it is so common that it is even mentioned in the [gettext docs](#), the documentation for a module designed specifically to help your programs handle multiple (natural) languages.

When I first found this usage of `_` I was very confused. I found it when looking at the source code for the `argparse` module. Because `argparse` deals with command-line interfaces, it makes sense that its inner-

workings are localised, so that its command-line messages match the language of the command-line itself. I still remember the very first time I saw it; I was looking at these two lines:

```
if prefix is None:  
    prefix = _('usage: ')
```

I was very confused with the `_('usage: ')` part of the assignment, but eventually I found the import statement in that file:

```
from gettext import gettext as _, ngettext
```

And I realised they were setting `_` as an alias for `gettext`.

Improve number readability

The final use case for underscores that we will discuss has to do with improving the readability of numbers.

Quick.

How much is `n` below?

```
>>> n = 99999999
```

If you thought/said “99 million, 999 thousand and 999”, you got it right.

Now, how much is `n` now?

```
>>> n = 100_000_000
```

Is there any doubt that we are talking about 100 million? Using `_` as a thousands separator really makes a difference here, and you shouldn’t need any more convincing! But I’ll just show you a little example from the Python Standard Library. Take a look at the two conditions below, and let me know which one is easier to read.

Without separator:

```
if not 1000 <= rounds <= 999999999:  
    raise ValueError('rounds out of the range 1000 to 999999999')
```

With separator:

```
if not 1000 <= rounds <= 999_999_999:  
    raise ValueError('rounds out of the range 1000 to 999_999_999')
```

If you tell me you prefer the first one, go away. I don’t want you here any more!

The underscore doesn’t have to be the *thousands* separator, you can use it between any digits you may want. But most importantly, it works with any other bases.

For example, use `_` to group bits in binary digits:

```
>>> thirty_five = 0b0010_0011  
>>> forty_seven = 0b0010_1111
```

Or maybe to separate the R, G, and B channels of the hexadecimal value of a colour:

```
>>> lightyellow = 0xff_ff_e0
>>> peachpuff    = 0xff_da_b9    # I didn't invent this name!
```

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Coding conventions exist to make our lives easier, so it is worth learning them to make our code more expressive and idiomatic.”

This Pydon't showed you that:

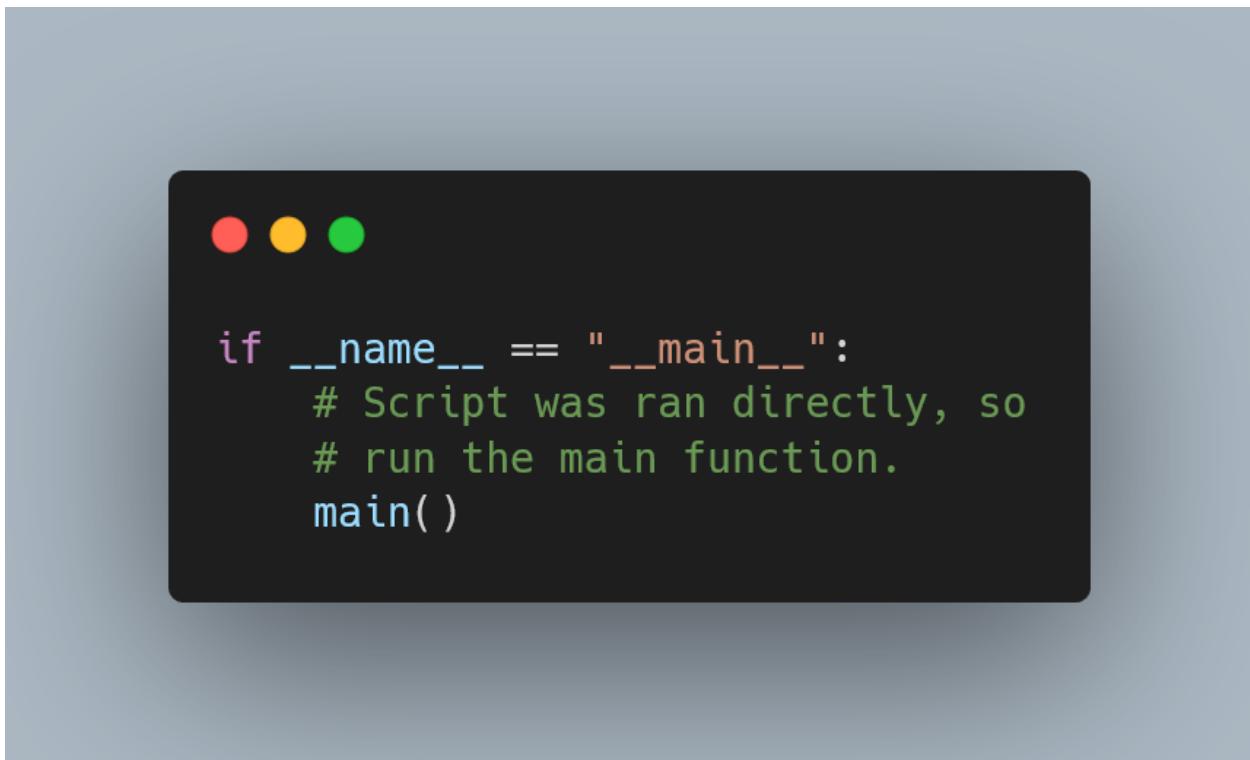
- you can recover the last value of an expression in the Python REPL with `_`;
- `_` has quite an impact on names when used as a prefix/suffix:
 - `name_` is a common choice for when `name` is a reserved keyword;
 - `_name` is a *convention* to signal that `name` is an internal name and that users probably shouldn't mess with it;
 - * `_name` won't be imported if someone uses a `from mymodule import *` wildcard import; and
 - * this can be overridden if `_name` is added to the `__all__` list in `mymodule`.
 - dunder names (that start and end with double underscore) refer to Python's internals and allow you to interact with Python's syntax;
 - `__name__` is used inside classes to prevent name collisions, when you want to use an internal variable with a name that you are afraid users might override by mistake;
- `_` is used in an idiomatic fashion as a sink in assignments, especially
 - when unpacking several values, when only some are of interest;
 - when iterating in a `for` loop where we don't care about the iteration number;
- the new `match` statement uses `_` as the “match all” case and makes it a true sink because `_` can't be used to access the original value;
- `_` is often used as an alias for localisation functions because of its low visual impact;
- numbers in different bases (decimal, binary, ...) can have their digits split by underscores to improve readability. For example, compare `99999999` with `999_999_999` with `999999999`.

References

- Python 3 Documentation, The Python Tutorial, Modules, “Importing * From a Package”, <https://docs.python.org/3/tutorial/modules.html#importing-from-a-package> [last accessed 14-06-2021];
- Python 3 Documentation, The Python Standard Library, `gettext`, <https://docs.python.org/3/library/gettext.html> [last accessed 14-06-2021];
- Python 3 Documentation, The Python Standard Library, `random.expovariate`, <https://docs.python.org/3/library/random.html#random.expovariate> [last accessed 14-06-2021];
- Weisstein, Eric W. “Exponential Distribution.” From MathWorld – A Wolfram Web Resource. <https://mathworld.wolfram.com/ExponentialDistribution.html> [last accessed 14-06-2021];
- Bader, Dan “The Meaning of Underscores in Python”, <https://dbader.org/blog/meaning-of-underscores-in-python> [last accessed 14-06-2021];
- Datacamp, “Role of Underscore(`_`) in Python”, <https://www.datacamp.com/community/tutorials/role-underscore-python> [last accessed 14-06-2021];

- Hackernoon, “Understanding the Underscore(_) of Python”, <https://hackernoon.com/understanding-the-underscore-of-python-309d1a029edc> [last accessed 14-06-2021];

name dunder attribute



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/name-dunder-attribute>.)

Introduction

In this Pydon't we will take a look at the `__name__` attribute. If you Google it, you will find a ton of results explaining *one* use case of the `__name__` attribute, so in this Pydon't I'll try to tell you about another couple of use cases so that you learn to use `__name__` effectively in your Python programs.

In this Pydon't, you will:

- learn about the idiomatic usage of `__name__` to create “main” functions in Python;
- learn about the read-only attribute `__name__` that many built-in objects get;
- see how `__name__` is used in a convention involving logging; and
- see some code examples of the things I will be teaching.

What is `__name__`?

`__name__` is a special attribute in Python. It is special because it is a dunder attribute, which is just the name that we give, in Python, to attributes whose names start and end with a double underscore. (I explain in greater detail what a dunder attribute/method is in [a previous Pydon’t](#).)

You can [look `__name__` up in the Python documentation](#), and you will find two main results that we will cover here. One of the results talks about `__main__` as a module attribute, while the other result talks about `__main__` as an attribute to built-in object types.

The module attribute `__name__`

The most commonly known use case for `__name__` is as a module attribute, when using `__name__` to create “main” functions in Python. What this means is that you can use `__name__` to determine programmatically if your code is being ran directly as a script or if it is being imported from another module.

How can we do this? Simple!

Go ahead and write the following line in your `print_name.py` file:

```
print(__name__)
```

Now open your command line and run the script:

```
> python print_name.py
__main__
```

What this is showing you is that the attribute `__name__` was automatically set to “`__main__`” when you ran your code as a script. This is relevant because that is *not* what happens when you import your code from elsewhere.

As an example, go ahead and write the following line into your `importer.py` file:

```
import print_name
```

Then go ahead and run this new Python script:

```
> python importer.py
print_name
```

Where is that “`print_name`” being printed from? Well, the *only* `print` statement you have is in the `print_name.py` file, so that was definitely the place from where the printed value came out. Notice that some code got executed (and some things were printed to the console) just by importing code from another module.

Also, notice that the value printed matches the name of the file it came from. Here, we see that `__name__` was automatically set to the name of the file it was in (`print_name`) when the code from `print_name` was imported from `importer`.

So, we see that `__name__` takes on different values depending on whether the code is ran directly as a script or imported from elsewhere.

When you write code, you often write a couple of functions that help you solve your problem, and then you apply those functions to the problem you have at hands.

For example, when I wrote some Python code to count valid passwords in an efficient manner, I wrote a class to represent an automaton in a file called `automaton.py`:

```
## automaton.py
```

```
class Automaton:  
    # ...
```

That class was problem-agnostic, it just implemented some basic behaviour related to automatons. It just so happened that that behaviour was helpful for me to solve the problem of counting passwords efficiently, so I imported that `Automaton` class in another file and wrote a little program to solve my problem. Thus, we can say that the majority of the times that I will use the code in my `automaton.py` file will be to import it from elsewhere and to use it.

However, I also added a little demo of the functionality of the `Automaton` class in the `automaton.py` file. Now, the problem is that I don't want this little demo to run every time the `Automaton` class is imported by another program, so I have to figure out a way to only run the demo if the `automaton.py` file is ran directly as a script... The reason is that my demo code has some `print` statements that wouldn't make sense to a user that just did `import automaton` from within another script... Imagine importing a module into your program and suddenly having a bunch of prints in your console!

Now, we can use `__name__` to avoid that! We have seen that `__name__` is set to `"__main__"` when a script is ran directly, so we just have to check that:

```
## automaton.py
```

```
class Automaton:  
    # ...  
  
if __name__ == "__main__":  
    print("Demo code.")
```

This is the most well-known use case of `__name__`. This is why you will commonly see snippets like

```
if __name__ == "__main__":  
    main()
```

It is just the Pythonic way of separating the functions and classes and other definitions, that might be useful for you to import later on, from the code that you only want to run if your program is the main piece of code being executed.

By the way, this global variable `__name__` really is a variable that just gets initialised without you having to do anything. But you can assign to it, even though it is unlikely that you might need to do that. Hence, this code is perfectly valid:

```
__name__ = "My name!"  
if __name__ == "__main__":  
    # This will never run:  
    print("Inside the __main__ if.")
```

`__name__` as an object type attribute

There is another Pythonic use case for the `__name__` attribute, another common usage pattern that employs `__name__`. This pattern I will teach you about now doesn't have to do with module attributes, but with object type attributes.

As you may be aware, all objects have a type that tells you "what" that object is.

Here are a couple of common types:

```
>>> type(0.5)  
<class 'float'>  
>>> type("hello")  
<class 'str'>  
>>> type(sum)  
<class 'builtin_function_or_method'>
```

Notice how the built-in type tells you what is the class of which the object is an instance. For example, "hello" is a string and that is why `type("hello")` returns `<class 'str'>`, which clearly contains the name of the class in there: it is the 'str' in there, between the single quotes.

With this information, how would you implement a `get_type_name` function that only returns the string with the type name, without the extra `<class>` fluff?

Give it some thought.

Here is a possibility:

```
>>> def get_type_name(obj):  
...     return str(type(obj)).split(" ") [1]  
...  
>>> get_type_name("hello")  
'str'  
>>> get_type_name(sum)  
'builtin_function_or_method'
```

Is this a good solution? I think we can definitely do better.

The documentation tells us that many built-in object types and definitions come with its `__name__` attribute, which is "the name of the class, function, method, descriptor, or generator instance".

Therefore, a better implementation of the `get_type_name` function would be

```
>>> def get_type_name(obj):
...     return type(obj).__name__
...
>>> get_type_name("hello")
'str'
>>> get_type_name(sum)
'builtin_function_or_method'
```

This is much shorter, much cleaner (doesn't have nested function calls, for example), and much easier to read, as the code says what it is doing. The name we picked for our function is good already, because it is easy to make an educated guess about what the function does, but it is much better if the body of the function itself makes it absolutely clear that we are getting what we want!

This ability of reaching out for the `__name__` of things is useful, for example, when you want to print an error message because you expected an argument of some type and, instead, you got something else. Using `__name__` you can get prettier error messages.

You can query the `__name__` of things other than built-in types. You can also query the name of functions, for example:

```
>>> sum.__name__
'sum'
>>> get_type_name.__name__
'get_type_name'
```

This might be relevant if you get ahold of a function in a programmatic way and need to figure out what function it is:

```
>>> import random
>>> fn = random.choice([sum, get_type_name])
>>> fn.__name__
'sum'
```

I don't think you are likely to receive a function from a `random.choice` call, but this just shows how you can use `__name__` to figure out what function you are looking at.

Another great thing that already comes with a `__name__` is your custom classes. If you define a class, `__name__` will be a very clean way of accessing the pretty class name without having to jump through too many hoops or doing hacky string processing:

```
>>> class A():
...     pass
...
>>> A
<class '__main__.A'>
>>> A.__name__
'A'
>>> a = A()
>>> a
>>> type(a)
<class '__main__.A'>
```

```
>>> type(a).__name__  
'A'
```

Similarly to the module `__name__`, the `__name__` attribute of types, functions, etc, can be assigned directly:

```
>>> type(a).__name__  
'A'  
>>> A.__name__ = "name..?"  
>>> type(a).__name__  
'name..?'
```

Sometimes this is useful, for example when you need to copy some metadata from one object to another.

Examples in code

I showed you what is the meaning that the `__name__` attribute has, both as a module attribute and as an attribute of type objects, and now I will show you how this knowledge can be put to practice. I will be drawing my examples from the Python Standard Library, as per usual.

Defining a command line interface to the module code

If you take a look at the `calendar` module, you will find functions to deal with calendars in various ways. If you inspect the source code, you will see that the implementation of the module ends with the following two lines:

```
## From Lib/calendar.py in Python 3.9.2  
if __name__ == "__main__":  
    main(sys.argv)
```

In short, the module defines a series of functions that people might want to import from elsewhere, but if the program is run directly, then it will call the function `main`, passing it in whatever arguments the program received from the terminal.

As an example, try running the following on your command line:

```
> python -m calendar 2021 6  
       June 2021  
Mo Tu We Th Fr Sa Su  
 1  2  3  4  5  6  
 7  8  9 10 11 12 13  
14 15 16 17 18 19 20  
21 22 23 24 25 26 27  
28 29 30
```

This shows a great example of how a module might have something interesting/useful to run if it is started as the main program. The function `main` in the `calendar` module implements a simple command line interface with the `argparse` module, and thus the `main` function is *simply* an entry point to the code that the module already defined.

Just out of curiosity, the Python Standard Library for my installation of Python 3.9.2 has 2280 .py files, and if you look for it, you can find the line `if __name__ == "__main__":` in 469 files, a little over a fifth of the files... So this really is a common pattern in Python!

Pretty error messages

Deleting from an enum

Like I mentioned before, if you want to get the name of the type of an object, `type(obj).__name__` is likely to be the way to go about doing that.

An example of where this shows up is in the `enum` module. The `enum` module gives support to enumerations in Python, which are sets of symbolic names (members) bound to unique, constant values.

Here is an example enumeration:

```
>>> import enum
>>> class Colour(enum.Enum):
...     RED = "RED"
...     GREEN = "GREEN"
...     BLUE = "BLUE"
...
```

Now we have a way of talking about specific colours:

```
>>> Colour.RED
<Colour.RED: 'RED'>
>>> Colour.BLUE
<Colour.BLUE: 'BLUE'>
```

Now, what if we want to delete one of the colours? For example, GREEN?

```
>>> del Colour.GREEN
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "C:\Program Files\Python39\lib\enum.py", line 382, in __delattr__
      raise AttributeError("%s: cannot delete Enum member." % cls.__name__)
AttributeError: Colour: cannot delete Enum member.
```

We can't delete GREEN from the enum because the GREEN attribute isn't just like any other class attribute, it is actually an integral part of the enumeration structure, and so the implementers of this module guarded the class against this type of deletions.

The question is, how did they get the pretty name of my Colour class? I don't even have to show you the code, you can just look at the error message above, that says we got an `AttributeError` inside the Colour enum. What was the line of code that produced this pretty error message? It was the following line:

```
raise AttributeError("%s: cannot delete Enum member." % cls.__name__)
```

In this line, `cls` is already a class, something like this:

```
>>> Colour          # <--  
<enum 'Colour'>    # <-- this is what `cls` is...  
                      # By the way, this is *not* a string.
```

So we could get its `__name__` directly and produce a pretty error message, or at least as pretty as error messages go.

Doing argument validation

Another similar use case can be found in the `fractions` module. This module provides a function, called `from_decimal`, to convert a `Decimal` number (from the `decimal` module) into an exact fraction. From my description of the function maybe you understood it, but this `from_decimal` function expects a `decimal.Decimal` instance, and errors out if the argument given is not such a thing:

```
>>> import fractions  
>>> fractions.Fraction.from_decimal("3")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "C:\Program Files\Python39\lib\fractions.py", line 188, in from_decimal  
    raise TypeError()  
TypeError: Fraction.from_decimal() only takes Decimals, not '3' (str)
```

Now the error message isn't enough because the code spans a couple of different lines, but here is the type validation that the `from_decimal` function performs:

```
## From Lib/fractions.py in Python 3.9.2  
class Fraction(numbers.Rational):  
    # ...  
  
    @classmethod  
    def from_decimal(cls, dec):  
        """Converts a finite Decimal instance to a rational number, exactly."""  
        from decimal import Decimal  
        if isinstance(dec, numbers.Integral):  
            dec = Decimal(int(dec))  
        elif not isinstance(dec, Decimal):  
            raise TypeError()  
            "%s.from_decimal() only takes Decimals, not %r (%s)" %  
            (cls.__name__, dec, type(dec).__name__))  
        return cls(*dec.as_integer_ratio())
```

Notice how the function takes a `dec` and tries to convert it to a `Decimal` if the argument isn't a `Decimal` but is easy to treat as one. That is why giving 3 to the function doesn't give an error:

```
>>> fractions.Fraction.from_decimal(3)  
Fraction(3, 1)
```

However, "3" is not a `numbers.Integral` and it is also *not* a `Decimal`, so `dec` fails the tests and we end up with

```
raise TypeError(  
    "%s.from_decimal() only takes Decimals, not %r (%s)" %  
    (cls.__name__, dec, type(dec).__name__))
```

Notice how we even have two `__name__` usages here. The first one is similar the example above with `Enum`, and we take our `cls` (that is already a class) and simply ask for its name. That is the part of the code that built the beginning of the message:

```
TypeError: Fraction.from_decimal() ...  
~~~~~
```

Then we print the value that actually got us into trouble, and that is what the `dec` is doing there:

```
TypeError: Fraction.from_decimal() only takes Decimals, not '3' ...  
~~~
```

Finally, we want to tell the user what it is that the user passed in, just in case it isn't clear from the beginning of the error message. To do that, we figure out the type of `dec` and then ask for its `__name__`, hence the `type(dec).__name__` in the code above. This is what produces the end of the error message:

```
TypeError: Fraction.from_decimal() only takes Decimals, not '3' (str)  
~~~
```

The `%s` and `%r` in the string above have to do with string formatting, a topic that is yet to be covered in these Pydon'ts. [Stay tuned](#) to be the first to know when those Pydon'ts are released.

This `type(obj).__name__` pattern is also very common. In my 3.9.2 installation of the Python Standard Library, it appeared 138 times in 74 different .py files. The specific `cls.__name__` pattern also showed up a handful of times.

Logging convention

For the final code example I will be showing you a common convention that is practised when using the `logging` module to log your programs.

The `logging` module provides a `getLogger` function to the users, and that `getLogger` function accepts a `name` string argument. This is so that `getLogger` can return a logger with the specified name.

On the one hand, you want to name your loggers so that, inside huge applications, you can tell what logging messages came from where. On the other hand, the `getLogger` function always returns the same logger if you give it the same name, so that inside a single module or file, you don't need to pass the logger around, you can just call `getLogger` always with the same name.

Now, you want to get your logger by using always the same name and you also want the name to identify clearly and unequivocally the module that the logging happened from. This shows that hand-picking something like "logger" is a bad idea, as I am likely to pick the same logger name as other developers picked in their code, and so our logging will become a huge mess if our code interacts.

The other obvious alternative is to name it something specific to the module we are in, like the file name. However, if I set the logger name to the file name by hand, I *know* I will forget to update it if I end up changing the file name, so I am in a bit of a pickle here...

Thankfully, this type of situation is a textbook example of when the `__name__` attribute might come in handy!

The `__name__` attribute gives you a readable name that clearly identifies the module it is from, and using `__name__` even means that your logging facilities are likely to behave well if your code interacts with other code that also does some logging.

This is why using `getLogger(__name__)` is the recommended convention in the documentation and that is why this pattern is used approximately 84% of the times! (It is used in 103 .py files out of the 123 .py files that call the `getLogger` function in the Python Standard Library.)

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“The `__name__` attribute is a dynamic attribute that tells you the name of the module you are in, or the name of the type of your variables.”

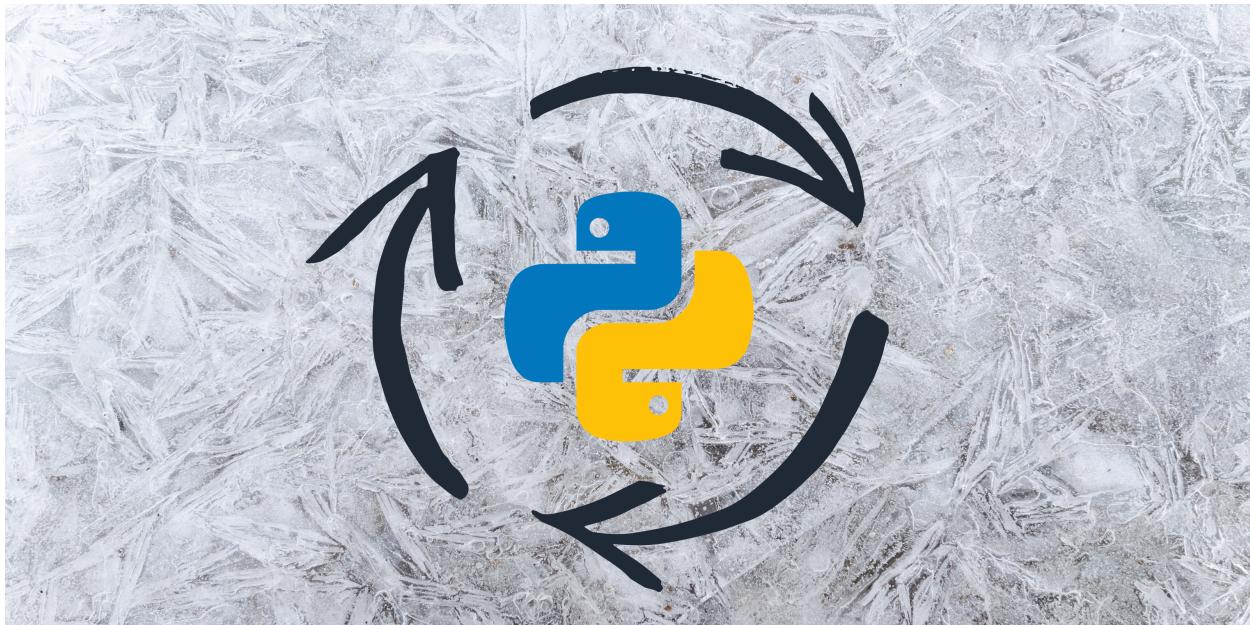
This Pydon't showed you that:

- `__name__` is a module attribute that tells you the name of the module you are in;
- `__name__` can be used to tell if a program is being ran directly by checking if `__name__` is `__main__`;
- you can and should use `__name__` to access the pretty name of the types of your objects;
- `__name__` is an attribute that can be assigned to without any problem;
- the if statement `if __name__ == "__main__":` is a very Pythonic way of making sure some code only runs if the program is ran directly;
- the pattern `type(obj).__name__` is a simple way of accessing the type name of an object; and
- there is a well-established convention that uses `__name__` to set the name of loggers when using the logging module.

References

- Python 3 Documentation, The Python Standard Library, Built-in Types, Special Attributes, https://docs.python.org/3/library/stdtypes.html?highlight=name#definition.__name__ [last accessed 29-06-2021];
- Python 3 Documentation, The Python Standard Library, Top-level script environment, <https://docs.python.org/3/library/main> [last accessed 29-06-2021];
- Python 3 Documentation, The Python Language Reference, The import system, <https://docs.python.org/3/reference/import.html> [last accessed 29-06-2021];
- Python 3 Documentation, Python HOWTOs, Logging HOWTO, Advanced Logging Tutorial, <https://docs.python.org/3/howto/logging.html#advanced-logging-tutorial> [last accessed 29-06-2021];
- Python 3 Documentation, The Python Standard Library, calendar, <https://docs.python.org/3/library/calendar.html> [last accessed 29-06-2021];
- Python 3 Documentation, The Python Standard Library, enum, <https://docs.python.org/3/library/enum.html> [last accessed 29-06-2021];
- Python 3 Documentation, Search results for the query “`__name__`”, https://docs.python.org/3/search.html?q=name&check_keywords=0&area=default [last accessed 29-06-2021];

Bite-sized refactoring



(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/bite-sized-refactoring>.)

Introduction

Refactoring code is the act of going through your code and changing bits and pieces, generally with the objective of making your code shorter, faster, or better any metric you set.

In this Pydon't I share my thoughts on the importance of refactoring and I share some tips for when you need to refactor your code, as I walk you through a refactoring example.

In this Pydon't, you will:

- understand the importance of refactoring;
- walk through a real refactoring example with me; and
- learn tips to employ when refactoring your own code.

Refactoring

REFACTOR – verb

“restructure (the source code of an application or piece of software) so as to improve operation without altering functionality.”

As you can see from the definition above, the act of refactoring your code is an attempt at making your code better. Making your code better might mean different things, depending on your context:

- it might mean it is easier to maintain;
- it might mean it is easier to explain to beginners;
- it might mean it is faster;
- ...

Regardless of the metric(s) you choose to improve, everyone can benefit from learning to refactor code.

Why is that?

When you are refactoring code you are training a series of skills that are helpful to you as a developer, like your ability to read code and really comprehend it, pattern recognition skills, critical thinking, amongst others.

Ability to read code and really comprehend it

If you change a piece of code without understanding it, you are much more likely to break it. Therefore, when you want to refactor a piece of code, you should do your best to try and *really* comprehend what the code is doing and how it does it.

Pattern recognition skills

One of the things that you should be looking out for, when refactoring code, is redundancies and repetitions. If you see code that looks like it was copied and pasted, or if you find code that has a very similar structure, then it probably is a good target for refactoring.

Sometimes, spotting these things is very simple, because there will be lines of code that are *identical*. However, finding *structural* similarities between different parts of your code is harder than finding identical lines, so in trying to spot these you will be training your pattern recognition skills. Beware that this becomes much easier to do *after* you have really understood the code.

Critical thinking

When reading code you wish to refactor, you will invariably find pieces of code that look like they shouldn't be there.

This can have many meanings.

It might be a piece of code that is in the wrong file. A piece of code that is in the wrong function. Sometimes, even, a piece of code that looks like it could/should be deleted. At these points in time, the only thing you can do is use your brain to figure out what are the implications of moving things around. You shouldn't be

afraid to move things around, *after* you have considered what are the implications of leaving things as-is versus changing them.

Remember, [you should strive to write elegant code](#), and part of that entails writing code in a way that makes it as easy as possible to refactor later on. Code is a mutable thing, so make sure to facilitate the life of your future self by writing elegant code that is easy to read.

What to refactor?

I am sure that people with different life experiences will answer differently to this question, the only thing I can do is share my point of view on the subject.

Refactor often...

... or at least create the conditions for that.

If you have the possibility to refactor a piece of code and you *know* there are things that can be improved upon, go ahead and do it. As you mature as a developer and gain experience, you keep learning new things; on top of that, the technologies you are using are probably also evolving over time. This means that code naturally goes into a state where it could benefit from refactoring.

This is a never-ending cycle: you should write code that is elegant and easy to read; that means that, in the future, refactoring the code is easier and faster; refactoring makes the code easier to read and even more elegant; which makes it easier to refactor in the future; that will make it easier to read and more elegant; and so on and so forth.

Code refactoring shouldn't be a daunting task because there is much to gain from it, so make sure to write your code in a way that will allow you, or someone else, to refactor it later.

Refactor little by little

Of course there should be a balance between refactoring code that already exists and writing new code for new features, etc.

Refactoring often makes it a very manageable task that you can actually learn to appreciate. If you don't refactor often, you let all these sub-optimal structures, bad design choices, etc, pile up, and those will be much more difficult to fix all at the same time.

Refactor little by little, at your own scale. If you are a very fresh beginner, this might mean that you want to refactor a line of code at a time, or maybe a couple of lines. If you are much more experienced, this might mean you are refactoring one or more files at the same time. Just refactor "a little", regardless of what that means to you.

Case study

Now I will go in-depth into a short Python function that was written by a beginner and [shared to Reddit](#) I will walk you through the process that happened in my brain when I tried refactoring that piece of code, and I

will share little tips as we go along.

First, let me tell you the task that the code is supposed to solve.

Write a function that changes the casing of its letters:

- letters in even positions should become uppercase; and
- letters in odd positions should become lowercase.

Go ahead and try solving this task.

Starting point

The piece of code that was shared on the Internet was the following:

```
def myfunc(a):  
    empty=[]  
    for i in range(len(a)):  
        if i%2==0:  
            empty.append(a[i].upper())  
        else:  
            empty.append(a[i].lower())  
  
    return "".join(empty)
```

Automatic style formatting

The very first step you can take towards writing code that is as elegant as possible is running an auto formatter. If you use `black`, for example, you can fix many style issues and inconsistencies right from the get-go.

In this case, running `black` on the code above produces the following:

```
def myfunc(a):  
    empty = []  
    for i in range(len(a)):  
        if i % 2 == 0:  
            empty.append(a[i].upper())  
        else:  
            empty.append(a[i].lower())  
  
    return "".join(empty)
```

The only difference here was the spacing in `empty = []` and in `if i % 2 == 0`. Spacing around operators is very important because it gives your code room to breathe. Making sure that your code has a consistent style goes a great length in making it readable to yourself and to others, so do try and build the habit of following a certain style.

[PEP 8](#) proposes a Python style and many follow that style, so it might be a good idea to take your time to review that style guide. After you figure out how that style works, remember that you don't need to start doing

everything at the same time. You can pick that style up gradually. Also, recall that critical thinking is very important. Sometimes it is best to ignore the style guide completely.

Naming

Names are very important, and naming your functions and variables correctly is crucial. Names can make or break a program. Good names aid the reader of the code, whereas bad names make you spend hours analysing otherwise simple code.

Names should reflect the intent, or a very important property, of the thing they refer to. This is the opposite of using very generic names, like `myfunc` for a function or `num` for a number, when that function has a specific role or that number contains some specific information.

A notable exception is the usage of `i` in for loops, for example, although personally I tend to prefer the slightly more verbose `idx`.

So, looking at the code we currently have, I can identify three names that could be improved upon. Can you figure out what those are? Have a go at changing them to something better.

Now, your suggestion doesn't have to match mine, but here is what I came up with:

```
def alternate_casing(text):
    letters = []
    for idx in range(len(text)):
        if idx % 2 == 0:
            letters.append(text[idx].upper())
        else:
            letters.append(text[idx].lower())

    return "".join(letters)
```

Here are the changes that I made:

- `myfunc` -> `alternate_casing`;
- `a` -> `text`;
- `empty` -> `letters`; and
- `i` -> `idx` (because of my personal preference).

Now, in and of itself, `empty = []` seems to be a pretty good name. However, right after we initialise `empty` with the empty list, we start filling it in, and so the name doesn't reflect a property of the object that holds throughout the program or that is important. Instead, by naming it `letters`, we specify what will be stored in there.

Traversing data and indices

Python has some really good capabilities to deal with for loops, and one of the tools that we are given is the `enumerate` built-in.

I [wrote at length about `enumerate` before](#), so just be advised that `enumerate` is the tool to reach for when you write a for loop where you need to work with the indices *and* the data at the same time.

In our function we need the indices *and* the data, because we need the index to determine the operation to do, and then we need the data (the actual letter) to change its casing. Using `enumerate`, here is how that loop would end up:

```
def alternate_casing(text):
    letters = []
    for idx, letter in enumerate(text):
        if idx % 2 == 0:
            letters.append(letter.upper())
        else:
            letters.append(letter.lower())

    return "".join(letters)
```

Not only we were able to remove the explicit indexing, therefore cutting down on one operation, but we also express our intent more clearly: when someone finds an `enumerate`, they should immediately understand that to mean “in this loop I need both the indices and the data I’m traversing”.

Nest only what is needed

In Python, indentation indicates code nesting, which indicates dependence. If a line of code is nested inside a `for` loop, it means it depends on the `for` loop. If it is further nested inside an `if` statement, it means it only applies when certain conditions are met. If it is further nested inside a `try` statement, we may expect it to raise an error, etc.

Nesting code means we need to keep track of many contexts in our head while we read the code, and even though you might not notice it, that’s exhausting. Going in and out of all those indented structures, making all those context switches, consumes brain power. Flatter code places less strain on our brains and makes it easier to keep up with the code.

To make it simpler to keep up with the context, we should try and nest as little code as possible. We should only nest the pieces of code that are absolutely necessary to be nested.

For `for` loops, that’s generally things that depend on the iterator variables between `for` and `in`, and for `if-else` statements, that’s the pieces of code that are *unique* to each statement.

Now, in the `if-else` statement above, can you spot something that is not *unique* to a single branch? Here is the code:

```
if idx % 2 == 0:
    letters.append(letter.upper())
else:
    letters.append(letter.lower())
```

Notice that we are doing a `letters.append` *regardless* of the branch we are in, which makes it less clear that the thing that is changing from one branch to the other is the choice of method that we call on `letter`. It is even *less* clear because `.upper()` and `.lower()` take up exactly the same number of characters, so the two lines are aligned and make it harder to notice the `.upper()` vs `.lower()` going on.

Now, if we work on factoring out that `.append()`, because that's independent of the value of `idx % 2`, we could get something like

```
def alternate_casing(text):
    letters = []
    for idx, letter in enumerate(text):
        if idx % 2 == 0:
            capitalised = letter.upper()
        else:
            capitalised = letter.lower()
        letters.append(capitalised)

    return "".join(letters)
```

You may feel strongly about the fact that I just added a line of code, making the code longer instead of shorter, but sometimes better code takes up more space. However...

Conditional assignment and conditional expressions

Having factored out the `.append()` to outside of the `if` makes it blatantly clear that the `if` statement is only there to decide on what to assign to `capitalised`. This opens the door for another simplification, that will come in the form of a conditional expression.

Conditional expressions are like condensed `if-else` blocks that are great for conditional assignment.

Using a conditional expression, we rewrite the `if-else` as

```
capitalised = letter.upper() if idx % 2 == 0 else letter.lower()
```

All in all, the intermediate variable is not needed and we can write the whole thing as

```
def alternate_casing(text):
    letters = []
    for idx, letter in enumerate(text):
        letters.append(letter.upper() if idx % 2 == 0 else letter.lower())

    return "".join(letters)
```

Truthy and Falsy

The next step concerns itself with simplifying the condition of the `if` statement. In Python, we have this wonderful thing which allows us to interpret many objects as Booleans, even if they are not Booleans themselves. This is often referred to as the **Truthy/Falsy** value of an object in Python, and you can learn all about this in a [previous Pydon't](#).

For our case, what matters is that the number 0 is treated as `False` and any other integer is treated as `True`. Therefore, the condition `if idx % 2: ...` reads as “if `idx` has a remainder when divided by 2”, which is equivalent to “if `idx` is odd”. Now, if the index is odd, we want the letter to be lowercased, so we can simplify the conditional expression if we simplify the condition and then switch the `.upper()` and `.lower()` calls:

```
def alternate_casing(text):
    letters = []
    for idx, letter in enumerate(text):
        letters.append(letter.lower() if idx % 2 else letter.upper())
    return "".join(letters)
```

At this point, the function is getting so short that there's no point in having an extra blank line separating the return statement, so I decided to put everything together.

List comprehensions versus appending

One thing that you can also learn to spot is when you are building a list by calling `.append()` on it successively. When that is the case, look for an opportunity to use a list comprehension. List comprehensions are very Pythonic when used well, and they allow you to initialise a variable with the correct contents right from the start, instead of having to initialise a variable to change it right away.

Using a list comprehension, you can rewrite your loop into something like

```
def alternate_casing(text):
    letters = [letter.lower() if idx % 2 else letter.upper() for idx, letter in enumerate(text)]
    return "".join(letters)
```

Avoid long lines

The problem with the list comprehension above is that now we have a really long line of code. Long lines of code are things to be avoided whenever possible, because they make it harder to read the code and make it harder to work with the code when you have it side-by-side with a debugger, or another file, or a Zoom call, or whatever. Horizontal scrolling in code is to be avoided at all costs, and that means lines shouldn't get too long.

There are a couple of ways in which we could fix that long list comprehension. Something that is always an option is *not* doing it. Just because an idea looks good under a certain angle, doesn't mean it is clearly superior.

However, we have something else up our sleeves. The names inside the list comprehension only live inside the list comprehension, so they are very short-lived and have a very specific role. Because of that, if the structure of what is happening is clear enough, we can use shorter variable names inside the list comprehension:

```
def alternate_casing(text):
    letters = [l.lower() if i % 2 else l.upper() for i, l in enumerate(text)]
    return "".join(letters)
```

Now, bear in mind that we can only get away with this because the target variable is well-named (`letters`) and so is the variable we are iterating over (`text`). I think there are several sensible alternatives for the list comprehension above, for example using `c` or `char` instead of `l`.

If you prefer, you could've left the long names and split the list comprehension instead:

```
def alternate_casing(text):
    letters = [
```

```

        letter.lower() if idx % 2 else letter.upper()
        for idx, letter in enumerate(text)
    ]
    return "".join(letters)

```

Auxiliary variables

Once again, auxiliary variables aren't always needed. Whether you have the broken up list comprehension or the one with the short names, you can just get rid of the auxiliary variable and call `.join()` on those letters directly:

```
def alternate_casing(text):
    return "".join([l.lower() if i % 2 else l.upper() for i, l in enumerate(text)])
```

or

```
def alternate_casing(text):
    return "".join([
        letter.lower() if idx % 2 else letter.upper()
        for idx, letter in enumerate(text)
    ])
```

Redundant list comprehensions

We have come so far, but there is one final thing we can do, and that is related to how we can get rid of the `[]` of the list comprehension. I mean we can literally delete them, so that we end up with the following:

```
def alternate_casing(text):
    return "".join(l.lower() if i % 2 else l.upper() for i, l in enumerate(text))
```

or

```
def alternate_casing(text):
    return "".join(
        letter.lower() if idx % 2 else letter.upper()
        for idx, letter in enumerate(text)
    )
```

What is happening? Now, instead of a list comprehension, we have a generator expression. Generator expressions are amazing, in my opinion, and they come with memory and speed benefits, so try to use them when you can. In practice, when you are calling a function with a list comprehension, you can often omit the `[]` altogether to switch to a generator expression.

I will devote a single Pydon't to generator expressions, so be sure to [subscribe](#) so you don't miss it!

Final comparison

For your reference, here is the code we started with:

```
def myfunc(a):
    empty=[]
```

```

for i in range(len(a)):
    if i%2==0:
        empty.append(a[i].upper())
    else:
        empty.append(a[i].lower())

return "".join(empty)

```

and here are two possible end products:

```

def alternate_casing(text):
    return "".join(l.lower() if i % 2 else l.upper() for i, l in enumerate(text))

```

and

```

def alternate_casing(text):
    return "".join(
        letter.lower() if idx % 2 else letter.upper()
        for idx, letter in enumerate(text)
    )

```

Notice how the end products look so different from the starting point, but notice that we did everything one small change at a time. Take your time to understand the small steps separately, and then appreciate how they all fit together in this refactor.

One of the main takeaways is really that refactoring doesn't need to happen in one fell swoop. It is ok to do incremental changes, and maybe even preferable: incremental changes are easier to manage and easier to reason about.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Elegant code is easier to refactor, and when you refactor your code, you should strive to make it more elegant.”

This Pydon't showed you that:

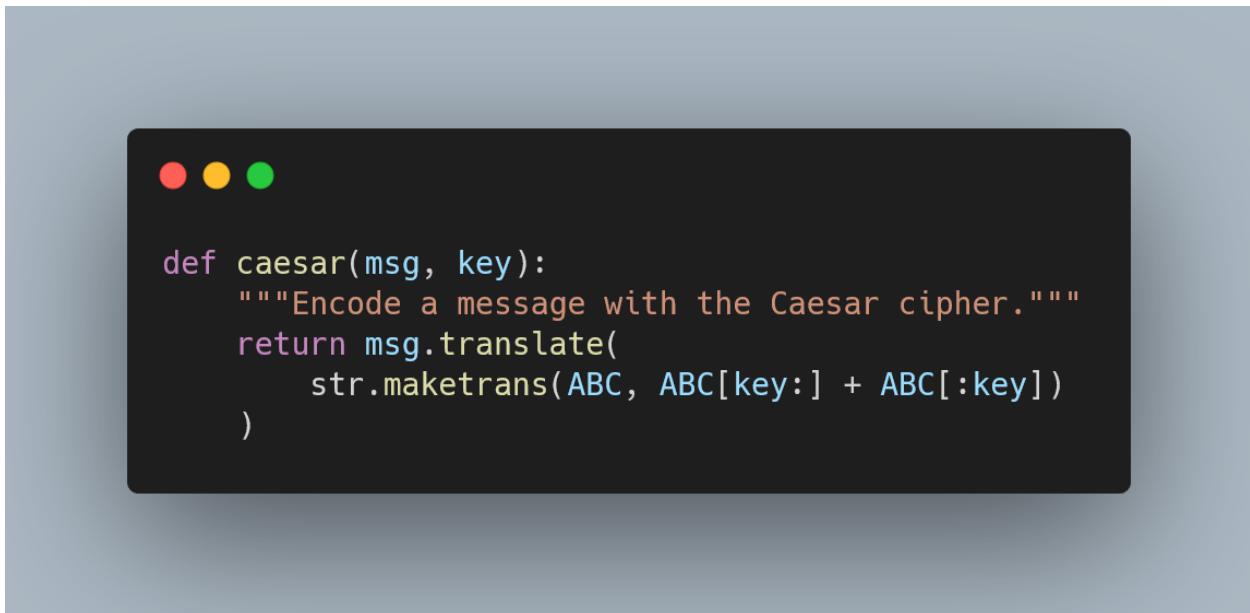
- the ability to refactor code is important;
- the ability to refactor code is something you train;
- code refactoring can (and maybe should!) happen in small steps;
- consistent style increases code readability;
- auto-formatters can help enforce a fixed style upon our code;
- naming is important and should reflect
 - the purpose of an object; or
 - an important characteristic that is invariant;
- enumerate is your best friend when traversing data *and* indices;
- repeated code under an if-else block can be factored out;
- conditional expressions excel at conditional assignments;
- if conditions can be simplified with Truthy and Falsy values;

- list comprehensions are good alternatives to simple `for` loops with `.append()` operations; and
- list comprehensions can be turned into generator expressions.

References

- Reddit /r/Python post “I get zero output even though there’s nothing wrong with this code according to pycharm. What can be the reason? I would appreciate any help.”, https://www.reddit.com/r/learnpython/comments/o2ko8l/i_get_zero_output_even_though_theres_nothing [last accessed 12-07-2021];
- Hoekstra, Conor; “Beautiful Python Refactoring” talk at PyCon US 2020, <https://www.youtube.com/watch?v=W-IZtZhsUY>;
- PEP 8 – Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008/> [last accessed 12-07-2021];

String translate and maketrans methods



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/string-translate-and-maketrans-methods.>)

Introduction

The strings methods `str.translate` and `str.maketrans` might be some of the lesser known string methods in Python.

Sadly, most online resources that cover this topic make a really poor job of explaining how the two methods work, so hopefully this Pydon't will serve you and let you know about two really cool string methods.

In this Pydon't, you will:

- be introduced to the string method `str.translate`;
- learn the available formats for the method `translate`;
- see that *all* characters (even emojis!) have a corresponding integer value;
- review the behaviour of the built-in functions `ord` and `char`;
- learn about the complementary string method `str.maketrans`;
- see good use cases for both `str.translate` and `str.maketrans`.

`str.translate`

The `str.translate` method is much unknown, but not because it is difficult to understand. It's just underappreciated, which means it doesn't get used much, which means it gets less attention than it deserves, which means people don't learn it, which means it doesn't get used much, ... Do you see where this is going?

I won't pretend like this method will completely revolutionise every single piece of Python code you will write in your life, *but* it is a nice tool to have in your tool belt.

As per the documentation, the `str.translate(table)` method returns

"a copy of the string in which each character has been mapped through the given translation table."

The translation table being mentioned here is the only argument that the method `str.translate` accepts.

In its simplest form, the method `str.translate` is *similar* to the method `str.replace`.

In case you don't know it, here is what `str.replace` looks like:

```
>>> s = "Hello, world!"
>>> s.replace("l", "L")
'HeLLo, worLd!'
```

Character code points

Computers work with zeroes and ones, binary – that's something we've all heard someone say at some point in our lives. But if that's the case, then how can we work with characters and text? How do we encode text information as zeroes and ones?

I'm not going to pretend like I know *really well* how the internals of these things work, but essentially we just need to attribute a number to every single character, and then the computer can take a look at that number and say "Oh, that's a 65? Alright, then I'll show an"A".

For that to work, computer programs must agree on what integers represent what characters. For example, who said 65 is "A"? Did I just invent that?

Much like the scientific community agreed that the metre would be the standard unit to describe distances, computer people have standards that specify how to map numbers to characters. The most well-known such standard is the [Unicode](#) standard, and that's the standard that Python uses.

All this talk has a single purpose: make you comfortable with the idea that characters can be turned into integers and back. Python even provides two useful built-in functions to do these conversions, the built-in functions `chr` and `ord`:

```
>>> ord("A")
65
>>> ord("a")
97
>>> ord(" ")
32
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(32)
' '
>>> chr(128013)
' '
```

Notice that even emoji have an integer that represents them!

`chr` takes an integer and returns the character that that integer represents, whereas `ord` takes a character and returns the integer corresponding to its Unicode code point.

The “code point” of a character is the integer that corresponds to it in the standard being used – which is the Unicode standard in the case of Python.

Translation dictionaries

Now that we know about the code points of characters, we can learn how to use the method `str.translate`, because now we can build dictionaries that can be passed in as translation tables.

The translation dict that is fed as the argument to `str.translate` specifies the substitutions that are going to take place in the target string.

The dictionary needs to map Unicode code points (i.e., characters) to other Unicode code points, to other strings, or to `None`.

Let's see if you can infer how each case works:

```
>>> ord("a"), ord("b"), ord("c")
(97, 98, 99)
>>> ord("A")
65
>>> "aaa bbb ccc".translate(
...     {97: 65, 98: "BBB", 99: None}
... )
'AAA BBBBBBBB '
```

Notice that the method `str.translate` above received a dictionary with 3 keys:

- 97 (the code point for "a") mapped to 65 (the code point for "A");
- 98 (the code point for "b") mapped to "BBB"; and
- 99 (the code point for "c") mapped to `None`.

In the final result, we see that all lower case "A"s were replaced with upper case "A"s, the lower case "B"s were replaced with triple "BBB" (so much so that we started with three "B"s and the final string has nine "B"s), and the lower case "C"s were removed.

This is subtle, but notice that the empty spaces were left intact. What happens if the string contains other characters?

```
>>> "Hey, aaa bbb ccc, how are you?".translate(  
...     {97: 65, 98: "BBB", 99: None}  
... )  
'Hey, AAA BBBBBBBBB , how Are you?'
```

We can see that the characters that were not keys of the dictionary were left as-is.

Hence, the translation works as follows:

- characters that do not show up in the translation table are left untouched;
- all other characters are replaced with their values in the mapping; and
- characters that are mapped to None are removed.

Non-equivalence to `str.replace`

Some of you might be thinking that I'm just being silly, making a huge fuss about `str.translate`, when all I need is a simple `for` loop and the method `str.replace`. Are you right?

Let me rewrite the example above with a `for` loop and the string method `str.replace`:

```
>>> s = "Hey, aaa bbb ccc, how are you?"  
>>> from_ = "abc"  
>>> to_ = ["A", "BBB", ""]  
>>> for f, t in zip(from_, to_):  
...     s = s.replace(f, t)  
...  
>>> s  
'Hey, AAA BBBBBBBBB , how Are you?'
```

As we can see, the result seems to be exactly the same, and we didn't have to introduce a new string method.

If you are not comfortable with the `zip` in that `for` loop above, I got you: take a look at the [Pydon't about zip](#).

Of course, we are forgetting the fact that the `for` loop technique using successive `str.replace` calls is doing more work than the `str.translate` method. What do I mean by this?

For every loop iteration, the `str.replace` method has to go over the whole string looking for the character we want to replace, and that's because consecutive `str.replace` calls are *independent* of one another.

But wait, if the successive calls are *independent* from one another, does that mean that..? Yes!

What if we wanted to take a string of zeroes and ones and replace all zeroes with ones, and vice-versa? Here is the solution using the successive `str.replace` calls:

```

>>> s = "001011010101001"
>>> from_ = "01"
>>> to_ = "10"
>>> for f, t in zip(from_, to_):
...     s = s.replace(f, t)
...
>>> s
'0000000000000000'

```

It didn't work! Why not? After the first iteration is done, all zeroes have been turned into ones, and `s` looks like this:

```

>>> s = "001011010101001"
>>> s.replace("0", "1")
'1111111111111111'

```

The second iteration of the `for` loop has no way to know what ones are original and which ones used to be zeroes that were just converted, so the call `s.replace("1", "0")` just replaces everything with zeroes.

In order to achieve the correct effect, we need `str.translate`:

```

>>> "001011010101001".translate(
...     {ord("0"): "1", ord("1"): "0"}
... )
'110100101010110'

```

Therefore, we have shown that `str.translate` is not equivalent to making a series of successive calls to `str.replace`, because `str.replace` might jumble the successive transformations.

Generic translation tables

The method `str.translate` accepts a “translation table”, but that table does *not* need to be a dictionary. That table can be any object that supports indexing with square brackets. In general, people use mappings (like dictionaries) or sequences (like lists), but you can even use your own custom objects.

I really enjoy using dictionaries (and other similar objects) because it is really easy to specify what maps to what, but for the sake of learning, let me show you an example where a list is used.

For that, let's write a translation table (as a list) that maps each of the 26 upper case letters of the alphabet to two times the same letter, but lower case.

The upper case letters range from the code point 65 to 90, so first we need to create a list with 90 elements, where each index should map to itself, so that other characters are left intact:

```
>>> translation_table = [i for i in range(91)]
```

Then, for the upper case letters, we need to update the corresponding positions so that they map to the values we want:

```

>>> for l in "ABCDEFGHIJKLMNPQRSTUVWXYZ":
...     translation_table[ord(l)] = 2 * l.lower()

```

```
...
>>> translation_table[60:70]
[60, 61, 62, 63, 64, 'aa', 'bb', 'cc', 'dd', 'ee']
```

Now, we just need to call the method `str.translate`:

```
>>> "Hey, what's UP?".translate(translation_table)
"hey, what's upp?"
```

Here is all of the code from this little example, also making use of the `string` module, so that I don't have to type all of the alphabet again:

```
>>> from string import ascii_uppercase
>>> ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> translation_table = [i for i in range(91)]
>>> for l in ascii_uppercase
...     translation_table[ord(l)] = 2 * l.lower()
...
>>> translation_table[60:70]
[60, 61, 62, 63, 64, 'aa', 'bb', 'cc', 'dd', 'ee']
>>> "Hey, what's UP?".translate(translation_table)
"hey, what's upp?"
```

`str.maketrans`

Having seen the generic form of translation tables, it is time to introduce `str.translate`'s best friend, `str.maketrans`.

The method `str.maketrans` is a utility method that provides for a convenient way of creating translation tables that can be used with `str.translate`.

`str.maketrans` accepts up to 3 arguments, so let's break them down for you.

Single argument

The version of `str.maketrans` that only accepts one argument has the purpose of making it simpler for us, users, to define dictionaries that can be used with `str.translate`.

Why would that be useful?

As we have seen above, when using dictionaries as translation tables we need to make sure that the keys of the dictionary are the code points of the characters we want to replace.

This generally introduces some boilerplate, because in the most common cases we *know* the characters we want to replace, not their code points, so we need to do the conversion by hand previously, or when defining the dictionary with `ord`.

This is ugly, just take a look at the example we used before:

```
>>> "001011010101001".translate(  
...     {ord("0"): "1", ord("1"): "0"}  
... )  
'110100101010110'
```

It would be lovely if we could just write the dictionary in its most natural form:

```
trans_table = {"0": "1", "1": "0"}
```

For this to work, we need to use `str.maketrans`:

```
>>> "001011010101001".translate(  
...     str.maketrans({"0": "1", "1": "0"})  
... )  
'110100101010110'
```

Two arguments

If you look at the example I just showed, we see that we did a very specific type of translation: we replaced some characters with some other single characters.

This is so common, that the method `str.maketrans` can be used to create translation tables of this sort. For that, the first argument to `str.maketrans` should be a string consisting of the characters to be replaced, and the second argument is the string with the corresponding new characters.

Redoing the example above:

```
>>> "001011010101001".translate(  
...     str.maketrans("01", "10")  
... )  
'110100101010110'
```

Here is another example where the two strings have different characters, just for the sake of diversity:

```
>>> "#0F45cd".translate(  
...     str.maketrans("abcdef", "ABCDEF")  
... )  
'#0F45CD'
```

In this example we took a hexadecimal value representing a colour and made sure all the letters were upper case.

(Of course we could have, and maybe should have, done that with the method `str.upper`.)

The third argument

Finally, the third argument to `str.translate` is simply a string of all the characters that should be mapped to `None` or, in other words, that should be removed altogether from the string.

Here is a little example:

```
>>> "# 0F45cd".translate(  
...     str.maketrans("abcdef", "ABCDEF", "# ")  
... )  
'0F45CD'
```

Examples in code

Now that you have been introduced to the string methods `str.translate` and `str.maketrans`, I will share a couple of interesting use cases for these methods.

I will start with a personal use case, and then include three use cases from the Python Standard Library. These code examples should help you understand how the two methods are used in the real world.

Caesar cipher

I wrote on [Twitter](#), asking people for their most Pythonic implementation of the Caesar cipher.

I defined the Caesar cipher as a function that takes two arguments. The first, a string, specifies some text. The second, an integer, specifies an integer key. Then, the upper case letters of the argument string should be shifted, along the alphabet, by the amount specified by the key. All other characters should be left as-is:

```
>>> caesar("ABC", 1)  
'BCD'  
>>> caesar("ABC", 13)  
'NOP'  
>>> caesar("ABC", 25)  
'ZAB'  
>>> caesar("HELLO, WORLD", 7)  
'OLSSV, DVYSK'
```

Some time later, I went to Twitter again to comment on some straightforward solutions and to also share the most elegant solution ever.

Can you guess what my Caesar implementation leverages? If you said/thought `str.translate` and `str.maketrans`, you are absolutely right!

Here is the nicest implementation of the Caesar cipher you will ever see:

```
def caesar(msg, key):  
    return msg.translate(  
        str.maketrans(ABC, ABC[key:] + ABC[:key]))
```

In the code above, `ABC` is a global constant that contains the alphabet that is subject to change. If we set `ABC = string.ascii_uppercase`, then we match exactly the Caesar cipher that I defined in the beginning:

```
>>> from string import ascii_uppercase  
>>> ABC = ascii_uppercase  
>>> def caesar(msg, key):  
...     return msg.translate(
```

```

...           str.maketrans(ABC, ABC[key:] + ABC[:key])
...
...
>>> caesar("HELLO, WORLD", 7)
'OLSSV, DVYSK'

```

Sanitising file names

The Python Standard Library provides a module to work with ZIP archives, and that module is called [zipfile](#).

This module can be used, for example, to extract ZIP archives programmatically.

When you use `zipfile` on Windows and `zipfile` extracts an archive, the module will look at the files that are being extracted and make sure that those files have names that are illegal on Windows. Can you guess what's the code that does this? It's a piece of code using `str.maketrans` and `str.translate`!

Here it is:

```

## In Lib/zipfile.py in Python 3.9.2

class ZipFile:
    # ...

    @classmethod
    def _sanitize_windows_name(cls, arcname, pathsep):
        """Replace bad characters and remove trailing dots from parts."""
        table = cls._windows_illegal_name_trans_table
        if not table:
            illegal = ':<>|?*'
            table = str.maketrans(illegal, '_' * len(illegal))
            cls._windows_illegal_name_trans_table = table
        arcname = arcname.translate(table)
        # ...

```

The `arcname` is the name of the archive. The first thing we do is fetch the `table` and see if it has been set. If it has *not* been set, then we set it for ourselves!

We define a series of illegal characters, and then use `str.maketrans` to create a translation table that translates them to underscores `_`:

```

>>> illegal = ':<>|?*'
>>> table = str.maketrans(illegal, '_' * len(illegal))
>>> table
{58: 95, 60: 95, 62: 95, 124: 95, 34: 95, 63: 95, 42: 95}

```

Then, we save this computed table for later and proceed to translating the name of the archive, `arcname`.

This shows a straightforward usage of both `str.maketrans` and `str.translate`.

Whitespace munging

(I didn't know, so I Googled it: "to munge" means to manipulate data.)

Along the same spirit, Python's `textwrap` module (used to wrap text along multiple lines and to do other related string manipulations) uses `str.translate` to munge whitespace in the given text.

As a preprocessing step to wrapping a string, we replace all sorts of funky whitespace characters with a simple blank space.

Here is how this is done:

```
## In Lib/textwrap.py from Python 3.9.2

_whitespace = '\t\n\x0b\x0c\r '

class TextWrapper:
    # ...

    unicode_whitespace_trans = {}
    uspace = ord(' ')
    for x in _whitespace:
        unicode_whitespace_trans[ord(x)] = uspace

    # ...

    def _munge_whitespace(self, text):
        # ...
        if self.replace_whitespace:
            text = text.translate(self.unicode_whitespace_trans)
        return text
```

Notice that we start by hardcoding a series of whitespace characters. Comments (that I omitted) explain why we do that. Then, inside the `TextWrapper` class, we define the translation table by hand. Later, in `_munge_whitespace`, we use that table to replace the funky whitespace characters with the blank space.

You might be wondering why `str.maketrans` was not used here, and I wouldn't know! Can you rewrite the lines of code that define the translation table, so that it uses `str.maketrans` instead? It should be similar to the code from the previous example.

Default replacement

If we peek at the source code for `IDLE`, the IDE that ships with Python, we can also find a usage of the method `str.translate`, and this one in particular defines a custom object for the translation table.

Before showing you the code, let me tell you what it should do: we want to create a translation table that

- preserves the whitespace characters " \t\n\r";
- maps "(", "[", and "{" to "(";
- maps ")", "]", and "}" to ")"; and

- maps *everything* else to “x”.

The point here is that we need to parse some Python code and we are only interested in the structure of the lines, while not so much in the actual code that is written.

By replacing code elements with “x”, those “x”s can then be deduplicated. When the “x”s are deduplicated the string becomes (much!) smaller and the processing that follows becomes significantly faster. At least that’s what the comments around the code say!

To help in this endeavour, we will implement a class called ParseMap that will be *very* similar to a vanilla dict, with one exception: when we try to access a ParseMap with a key it doesn’t know, instead of raising a `KeyError`, we return 120. Why 120? Because:

```
>>> ord("x")
120
```

Assuming ParseMap is already defined, here is what using it could look like:

```
>>> pm = ParseMap()
>>> pm
{}
>>> pm[0] = 343
>>> pm["hey"] = (1, 4)
>>> pm
{0: 343, 'hey': (1, 4)}
>>> pm[999]
120
```

By implementing this behaviour of returning 120 by default, we know that our translation table will map any character to “x” by default.

Now that the idea was introduced, here is the code:

```
## In Lib/idlelib/pyparse.py from Python 3.9.2

class ParseMap(dict):
    # [comments omitted for brevity]

    def __missing__(self, key):
        return 120  # ord('x')

trans = ParseMap.fromkeys(range(128), 120)
trans.update((ord(c), ord('(')) for c in "([{")  # open brackets => '(';
trans.update((ord(c), ord(')')) for c in "})"])  # close brackets => ')';
trans.update((ord(c), ord(c)) for c in "\\""\n#")  # Keep these.
```

In order to implement the “return 120 by default” behaviour, all that was needed was to say that ParseMap inherits from dict, and then we implement the `__missing__` dunder method.

Then, we initialise a translation table that already maps a bunch of characters to 120. We do that with the `dict.fromkeys` method:

```
>>> dict.fromkeys("abc", 42)
{'a': 42, 'b': 42, 'c': 42}
>>> dict.fromkeys(range(3), "Hello, world!")
{0: 'Hello, world!', 1: 'Hello, world!', 2: 'Hello, world!'}
```

The line

```
trans = ParseMap.fromkeys(range(128), 120)
```

is there to explicitly map many common characters to "x", which is supposed to speed up the translation process itself.

Then, the three lines that follow update the translation table in such a way that the parenthesis, brackets, and braces, are mapped like I said they would.

In the end, the translation behaves like this:

```
>>> s = "(This [is]\tsome\n\ttext.)"
>>> print(s)
(This [is]      some
text.)
>>> print(s.translate(trans))
(XXXXXX(xx)XXXXX
XXXXXX)
```

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"When you need to replace several characters with other characters or strings, the method str.translate is your best friend."

This Pydon't showed you that:

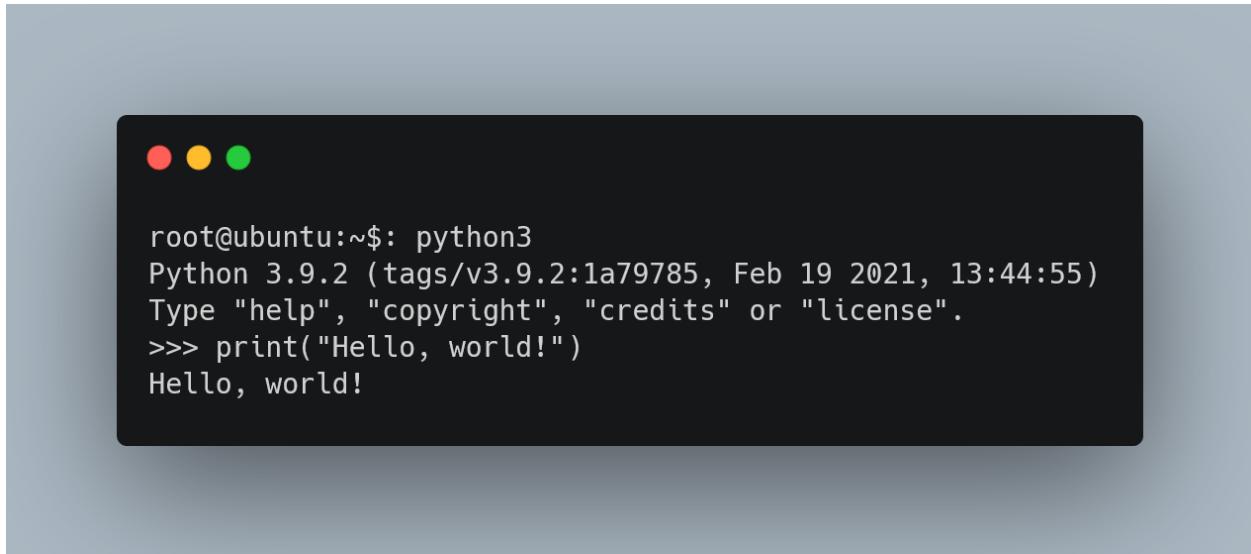
- the `str.translate` method replaces characters from an origin string with new characters or substrings;
- the character translation is controlled by a translation table that can be any object that supports indexing by integers;
- all characters (even emojis!) can be converted to a unique integer, and back, through the use of the built-in functions `ord` and `chr`;
- the “code point” of a character is the integer that represents it;
- Python uses the code points set by the [Unicode](#) standard, the most widely-used in the world;
- the translation tables make use of the code points of characters to decide what is replaced by what;
- in general, `str.translate` cannot be replaced with a series of calls to `str.replace`;
- Python provides a utility method (`str.maketrans`) to help us create translation tables:
 - with a single argument, it can process dictionaries to have the correct format;
 - with two arguments, it builds a translation table that maps single characters to single characters; and
 - the third argument indicates characters that should be removed from the string. And
- the `__missing__` dunder method controls how custom `dict` subclasses work when indexed with missing keys;

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [don't forget to subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 3 Documentation, The Python Standard Library, Built-in Types, `str.maketrans`, <https://docs.python.org/3/library/stdtypes.html#str.maketrans> [last accessed 16-08-2021];
- Python 3 Documentation, The Python Standard Library, Built-in Types, `str.translate`, <https://docs.python.org/3/library/stdtypes.html#str.translate> [last accessed 16-08-2021];
- Python 3 Documentation, The Python Standard Library, `zipfile`, <https://docs.python.org/3/library/zipfile.html> [last accessed 17-08-2021];
- Python 3 Documentation, The Python Standard Library, `textwrap`, <https://docs.python.org/3/library/textwrap.html> [last accessed 17-08-2021];
- Python 3 Documentation, The Python Standard Library, IDLE, <https://docs.python.org/3/library/idle.html> [last accessed 17-08-2021];
- Unicode, <https://home.unicode.org> [last accessed 17-08-2021];

Boost your productivity with the REPL



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/boost-your-productivity-with-the-repl>.)

Introduction

The REPL is an amazing tool that every Python programmer should really know and appreciate! Not only that, but you stand to gain a lot if you get used to using it and if you learn to make the most out of it ☺

In this Pydon't, you will:

- learn what “REPL” stands for;
- understand how important the REPL is for your learning;
- understand the mechanism that “prints” results in the REPL;
- see how to recover the previous result in the REPL, in case you forgot to assign it;
- learn about the built-in help system;
- learn some tips for when you’re quickly hacking something together in the REPL;

- be told about two amazing tools to complement your usage of the REPL.

REPL

Read. Evaluate. Print. Loop.

That's what "REPL" stands for, and it is often referred to as "read-eval-print-loop". The REPL is the program that takes your input code (i.e., reads your code), evaluates it, prints the result, and then repeats (i.e., loops).

The REPL, sometimes also referred to as the "interactive session", or the "interpreter session", is what you get when you open your computer's command line and type `python` or `python3`.

That should result in something like the following being printed:

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Of course, the exact things that are printed (especially the first line) are likely to differ from what I show here, but it's still the REPL.

(By the way, if you ever need to leave the REPL, just call the `exit()` function.)

Just fire up the REPL

The REPL is, hands-down, one of your best friends when you are writing Python code. Having a REPL to play around with just makes it much easier to learn the language.

Can't remember the argument order to a built-in function? Just fire up the REPL.

Need to do a quick computation that is just a bit too much for the conventional desktop calculator? Just fire up the REPL.

Can't remember how to spell that module you want to import? Just fire up the REPL.

You get the idea.

I *cannot* stress this enough. Get used to the REPL. Play with it. Write code in it. As soon as you become familiar with it, you'll love it and thank me for that.

REPL mechanics

Basic input and output

The REPL generally contains a `>>>` in the beginning of the line, to the left of your cursor. You can type code in front of that prompt and press Enter. When you press Enter, the code is evaluated and you are presented with the result:

```
>>> 3 + 3
6
```

Multiline input

The REPL also accepts code that spans multiple lines, like `if` statements, `for` loops, function definitions with `def`, etc.

In order to do those, just start typing your Python code regularly:

```
>>> if True:
```

When you press Enter after the colon, Python realises the body of the `if` statement is missing, and thus starts a new line containing a `...` on the left. The `...` tells you that this is the *continuation* of what you started above.

In order to tell Python you are done with the multiline code blocks is by pressing Enter on an empty line with the continuation prompt `...`:

```
>>> if True:  
...     print("Hello, world!")  
...  
Hello, world!  
>>>
```

Pasting into the REPL

Pasting into the REPL should work without any problem.

For example, the function below returns the double of the input. Try copying it into your REPL and then using it.

```
def double(x):  
    return 2 * x
```

However, if you try to copy and paste a multiline block that contains empty lines in the middle, then the REPL will break your definition.

For example, if you try pasting the following, you get an error:

```
def double(x):  
  
    return 2 * x
```

Copying the code above and pasting it into the session, you will end up with a session log like this:

```
>>> def double(x):  
...  
File "<stdin>", line 2  
  
^  
IndentationError: expected an indented block  
>>>     return 2 * x  
File "<stdin>", line 1
```

```
    return 2 * x
IndentationError: unexpected indent
```

This happens because the REPL finds a blank line and thinks we tried to conclude the definition of the function.

Implicit printing of results

One last thing you should know about the REPL is that it implicitly “prints” the results of the expressions you type.

I wrote “prints” in quotes because the REPL doesn’t really print the result, it just shows its *representation*. The *representation* of an object is what you get when you call `repr` on the object. If you explicitly print something, then what you get is the result of calling `str` on it.

I wrote a [very detailed Pydon’t](#) explaining the differences between the two, so let me just *show* you how things are different:

```
## Define a string.
>>> s = "Hello\nworld!"
## Print its `str` and `repr` values:
>>> print(str(s))
Hello
world!
>>> print(repr(s))
'Hello\nworld!'
## Print the string explicitly and evaluate it in the REPL.
>>> print(s)
Hello
world!
>>> s
'Hello\nworld!'
```

As you can see, printing `s` or just typing it in the REPL gives two different results. Just be mindful of that.

No printing, or `None`

In particular, if the expression you wrote evaluates to `None`, then nothing gets printed.

The easiest way to see this is if you just type `None` in the REPL. Nothing gets displayed; contrast that with what happens if you just type `3`:

```
>>> None
>>> 3
3
```

If you call a function that doesn’t have an explicit return value, or that returns `None` explicitly, then those functions will not show anything in the REPL:

```

>>> def explicit_None_return():
...     # Return None explicitly.
...     return None
...
>>> explicit_None_return()      # <- nothing gets displayed.

>>> def implicit_None_return():
...     # Ending without a `return` returns `None` implicitly.
...     pass
...
>>> implicit_None_return()      # <- nothing gets displayed.

```

Repeated imports

Sometimes it is useful to use the REPL to quickly import a function you just defined. Then you test the function out and then proceed to changing it in the source file. Then you'll want to import the function again and test it again, except that won't work.

You need to understand how the REPL handles imports, because you can't import repeatedly to "update" what's in the session.

To show you this, go ahead and create a file `hello.py`:

```

## In `hello.py`:
print("Being imported.")

```

Just that.

Now open the REPL:

```

>>> import hello
Being imported!

```

Now try modifying the string inside the `print`, and re-import the module:

```

>>> import hello
Being imported!
## Modify the file, then import again:
>>> import hello
>>>

```

Nothing happens! That's because Python already went through your file and knows what's in there, so it doesn't need to parse and run the file again. It can just give you the functions/variables you need.

In short, if you modify variables, functions, code; and you need those changes to be reflected in the REPL, then you need to leave the REPL with `exit()`, start it again, and import things again.

That's why some of the tips for quick hacks I'll share below are so helpful.

Edit: Another alternative – brought to my attention by a kind reader – is to use `importlib.reload(module)` in Python 3.4+. In our example, you could use `importlib.reload(hello)`:

```
>>> import hello
Being imported
>>> import importlib           # Use `imp` from Python 3.0 to Python 3.3
>>> importlib.reload(hello)
Being imported
<module 'hello' from 'C:\\tmp\\hello.py'>
```

We get that final line because `importlib.reload` returns the module it reloaded.

You can take a look at [this StackOverflow question and answers](#) to learn a bit more about this approach.

Be mindful that it may not work as you expect when you have multiple imports. Exiting the REPL and opening it again may be the cleanest way to reload your imports in those situations.

REPL history

I'll be honest with you, I'm not entirely sure if what I'm about to describe is a feature of the Python REPL or of all the command lines I have worked with in my entire life, but here it goes:

You can use the up and down arrow keys to go over the history of expressions you already entered. That's pretty standard.

What's super cool is that the REPL remembers this history of expressions, even if you exit the REPL, *as long as you don't close the terminal*.

The last result

If you read my [Pydon't about the usages of underscore](#) you might know this already, but you can use the underscore `_` to retrieve the result of the last expression if you want to use it and forgot to assign.

Here is a silly example:

```
>>> 3 + 6
9
>>> _ + 10
19
```

This might come in handy when you call a function or run some code that takes a long time. For example, downloading something from the Internet.

It can also be helpful if you just ran an expression with side-effects and you don't want to run that again because you don't want to trigger the side-effects twice. For example, if you just made a call to an API.

Of course `_` is a valid variable name in and out of itself, so you can still use it as a variable name. If you do, however, then `_` will stop reflecting the result of the last expression:

```
>>> _ = 0
>>> _
0
>>> 3 + 9
12
```

```
>>> _  
0           # <- it still evaluates to 0!
```

If you want to get back the magical behaviour of `_` holding the result of the last expression, just delete `_` with `del _`.

Getting help from within the REPL

Another great feature that is often underappreciated is the built-in help system. If you need to take a look at a quick reference for a built-in function, for example, because you forgot what the arguments are, just use `help`!

```
>>> help(sum)  
Help on built-in function sum in module builtins:  
  
sum(iterable, /, start=0)  
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers  
  
    When the iterable is empty, return the start value.  
    This function is intended specifically for use with numeric values and may  
    reject non-numeric types.
```

```
>>>
```

What is great about this `help` built-in is that it can even provide help about *your own code*, provided you document it well enough.

Here is the result of calling `help` on a function defined by you:

```
>>> def my_function(a, b=3, c=4):  
...     return a + b + c  
...  
>>> help(my_function)  
Help on function my_function in module __main__:  
  
my_function(a, b=3, c=4)
```

```
>>>
```

You can see that `help` tells you the module where your function was defined and it also provides you with the signature of the function, default values and all!

To get more information from `help` you need to document your function with a docstring:

```
>>> def my_function(a, b=3, c=4):  
...     """Return the sum of the three arguments."""  
...     return a + b + c  
...  
>>> help(my_function)
```

```
Help on function my_function in module __main__:
```

```
my_function(a, b=3, c=4)
    Return the sum of the three arguments.
```

```
>>>
```

Now you can see that the `help` function also gives you the information stored in the docstring.

I'll be writing a Pydon't about docstrings soon. Be sure to [subscribe to my newsletter](#) so you don't miss it!

Tips for quick hacks

The Python REPL is amazing when you need to flesh an idea out, as it allows you to quickly test some code, tweak it, and iterate over that repeatedly with instant feedback.

It goes without saying, but the REPL is not a replacement for your IDE! However, sometimes it helps to know about a couple of little tricks that you can employ to help you make the most out of your REPL.

Semicolons

Yes, **really**.

Python supports semicolons to separate statements:

```
>>> a = 3; b = a + 56; print(a * b)
177
```

However, this feature is something that often does **not** belong in your code, so refrain from using it.

Despite being generally inadequate for production code, the semicolons are your best friends when in the REPL. I'll explain it to you, and you'll agree.

In the command line you can usually use the up and down arrows to cycle through the most recently typed commands. You can do that in the REPL as well. Just try evaluating a random expression, then press the up arrow and Enter again. That should run the exact same expression again.

Sometimes you will be working in the REPL testing out a solution or algorithm incrementally. However, if you make a mistake, you must reset everything.

At this point, you just press the arrows up and down, furiously trying to figure out all the code you have ran already, trying to remember which were the correct expressions and which ones were wrong...

Semicolons can prevent that! You can use semicolons to keep track of your whole "progress" as you go: whenever you figure out the next step, you can use the arrows to go up to the point where you last "saved your progress" and then you can add the correct step at the end of your sequence of statements.

Here is an example of an interactive REPL session of me trying to order a list of names according to a list of ages.

Instead of two separate assignments, I put them on the same line with `:`

```
>>> names = ["John", "Anna", "Bill"]; ages = [20, 40, 30]
```

I could have written

```
>>> names, ages = ["John", "Anna", "Bill"], [20, 40, 30]
```

but using the semicolon expresses the intent of having the two assignments in separate lines when it comes time to write the real code down.

Then, I will try to see how to put the ages and names together in pairs:

```
>>> [(age, name) for name, age in zip(names, ages)]
[(20, 'John'), (40, 'Anna'), (30, 'Bill')]
```

However, at this point I realise I'm being redundant and I can just use `zip` if I reverse the order of the arguments:

```
>>> list(zip(ages, names))
[(20, 'John'), (40, 'Anna'), (30, 'Bill')]
```

Now that I'm happy with how I've paired names and ages together, I use the arrow keys to go back to the line with the assignment. Then, I use a semicolon to add the new piece of code I worked out:

```
>>> names = ["John", "Anna", "Bill"]; ages = [20, 40, 30]; info_pairs = zip(ages, names)
```

`zip` is an amazing tool in Python and is one of my favourite built-in functions. You can learn how to wield its power with [this Pydon't](#).

Now I can move on to the next step, knowing that a mistake now won't be costly: I can reset everything by going up to the line with all the intermediate steps and run that single line.

Not changing lines

When you want to define a simple multiline block, you can often get away with inlining what comes after the colon.

For example, instead of

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

you can write

```
>>> for i in range(3): print(i)
...
0
1
2
```

While this is style that is *not* recommended for production code, it makes it more convenient to go up and down the REPL history.

If you really want to push the boundaries, you can even combine this with semicolons:

```
>>> i = 1
>>> while i < 30: print(i); i *= 2
...
1
2
4
8
16
```

Import, test, loop

If you are writing some code and want to take it for a spin – just to make sure it makes sense – fire up the REPL, import the code, and play with it! That’s the magic of the REPL.

Be sure to do any setup for the “tests” in a single line separated with semicolons, together with the import statements. That way, when you tweak the code you just wrote, you can type `exit()` to leave the REPL, enter it again, and then with a couple of up-arrow presses you get your setup code intact and are ready to play with it again.

Other tools

I try to stick to vanilla Python as much as possible when writing these Pydon’ts, for one simple reason: the world of vanilla Python is huge and, for most developers, has lots of untapped potential.

However, I believe I would be doing you a disservice if I didn’t mention two tools that can really improve your experience in/with the REPL.

Rich

“Rich is a Python library for rich text and beautiful formatting in the terminal.”

[Rich](#) is an open source library that I absolutely love. You can read the documentation and the examples to get up to speed with Rich’s capabilities, but I want to focus on a very specific one, in particular:

```
>>> from rich import pretty
>>> pretty.install()
```

Running this in your REPL will change your life. With these two lines, Rich will pretty-print your variables and even include highlighting.

IPython

[IPython](#) is a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language. IPython offers introspection, rich media, shell syntax, tab

completion, and history, among other features.

In short, it is a Python REPL with more bells and whistles.

It is beyond the scope of this Pydon't to tell you all about IPython, but it is something I had to mention (even though I personally don't use it).

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"Get comfortable with using the REPL because that will make you a more efficient Python programmer."

This Pydon't showed you that:

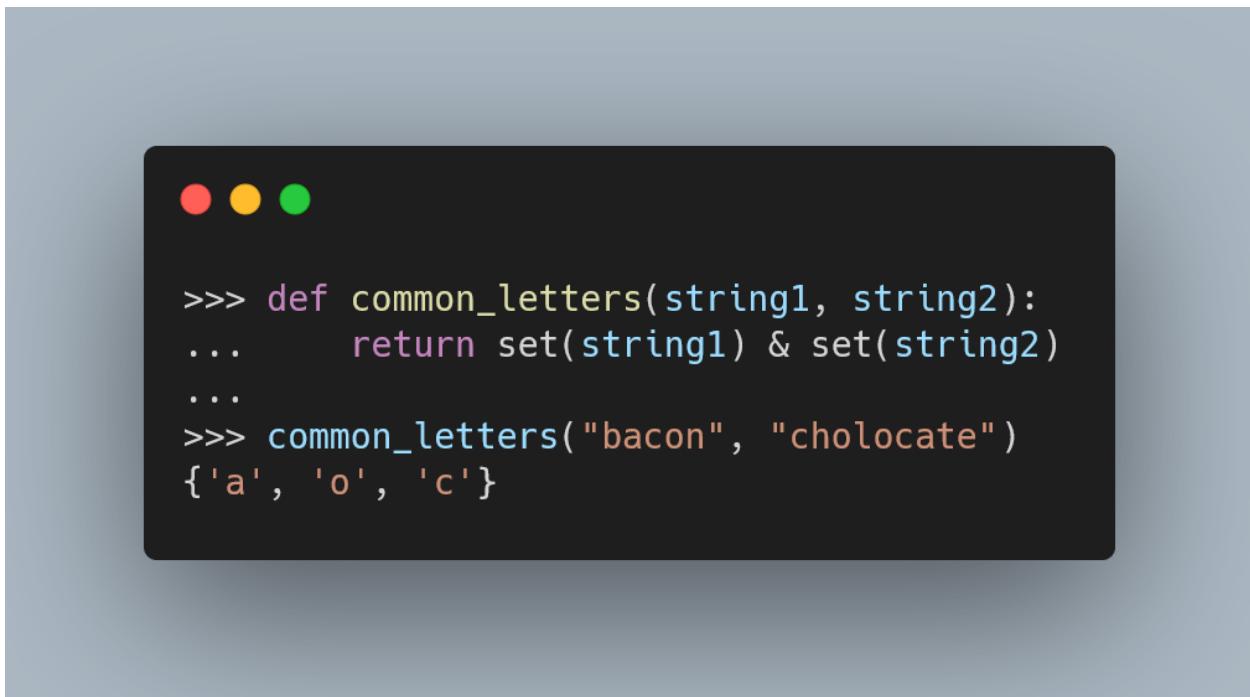
- the REPL is a great tool to help prototype small ideas and solutions;
- the REPL supports multiline input, and breaks it after an empty line;
- the REPL implicitly shows the result of the expressions you type, with the caveat that what is shown is an objects representation (`repr`), not its string value (`str`);
- you can use the arrows to navigate the history of the code you typed in the REPL;
- history of typed code is preserved after you exit the REPL, as long as you don't close the terminal window;
- None results don't get displayed implicitly;
- repeatedly importing the same module(s) does not update their contents;
- you can access the result of the previous line using `_`;
- the `help` built-in can give you basic documentation about the functions, and other objects, you have "lying around"; it even works on user-defined objects;
- by using docstrings, you improve the utility of the built-in `help` when used on custom objects;
- although not recommended best practices, the usage of semicolons and in-line multiline statements can save you time when navigating the history of the REPL;
- Rich is a tool that you can use in your REPL to automatically pretty-print results with highlighting;
- IPython is an alternative Python REPL that comes with even more bells and whistles.

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Rich, <https://github.com/willmcgugan/rich> [last accessed 25-08-2021]
- IPython, <https://ipython.org/> [last accessed 25-08-2021]
- Feedback with suggestions for improvements, Reddit comment, https://www.reddit.com/r/Python/comments/pbkq3z/boost_your_productivity_with_the_repl_pydon13/ [last accessed 26-08-2021];
- Stack Overflow question, "How to re import an updated package while in Python Interpreter?", <https://stackoverflow.com/q/684171/2828287> [last accessed 26-08-2021];

set and frozenset



(Thumbnail of the original article at <https://mathspp.com/blog/pydnts/set-and-frozenset>.)

Introduction

Python contains a handful of built-in types, among which you can find integers, lists, strings, etc...

Python also provides two built-in types to handle sets, the `set` and the `frozenset`.

In this Pydon't, you will:

- understand the relationship between the `set` built-in and the mathematical concept of “set”;
- learn what the `set` and `frozenset` built-ins are;
- see what the differences between `set` and `frozenset` are;

- learn how to create sets and frozen sets;
- understand how sets fit in with the other built-in types, namely lists;
- establish a parallel between lists and tuples, and sets and frozen sets;
- see good example usages of set (and frozenset) in Python code;

(Mathematical) sets

A set is simply a collection of unique items where order doesn't matter. Whenever I have to think of sets, I think of shopping carts.

No ordering

If you go shopping, and you take a shopping cart with you, the order in which you put the items in the shopping cart doesn't matter. The only thing that *actually* matters is the items that are in the shopping cart.

If you buy milk, chocolate, and cheese, it doesn't matter the order in which those items are registered. What matters is that you bought milk, chocolate, and cheese.

In that sense, you could say that the groceries you bought form a set: the set containing milk, chocolate, and cheese. Both in maths and in Python, we use {} to denote a set, so here's how you would define the groceries set in Python:

```
>>> groceries = {"milk", "cheese", "chocolate"}
>>> groceries
{'cheese', 'milk', 'chocolate'}
>>> type(groceries).__name__
'set'
```

We can check that we created a set indeed by checking the __name__ of the type of groceries.

If you don't understand why we typed type(groceries).__name__ instead of just doing type(groceries), then I advise you to skim through [the Pydon't about the dunder attribute __name__](#). (P.S. doing isinstance(groceries, set) would also work here!)

To make sure that order really doesn't matter in sets, we can try comparing this set with other sets containing the same elements, but written in a different order:

```
>>> groceries = {"milk", "cheese", "chocolate"}
>>> groceries == {"cheese", "milk", "chocolate"}
True
>>> groceries == {"chocolate", "milk", "cheese"}
True
```

Uniqueness

Another key property of (mathematical) sets is that there are no duplicate elements. It's more or less as if someone told you to go buy cheese, and when you get back home, that person screams from another room:

"Did you buy cheese?"

This is a yes/no question: you either bought cheese or you didn't.

For sets, the same thing happens: the element is either in the set or it isn't. We don't care about element count. We don't even consider it.

Here's proof that Python does the same:

```
>>> {"milk", "cheese", "milk", "chocolate", "milk"}  
{'cheese', 'milk', 'chocolate'}
```

(Common) Operations on sets

Sets define many methods, like [the docs](#) will tell you.

Creation

There are three main ways to create a set.

Explicit {} notation

Using the {} notation, you write out the elements of the set inside braces in a comma-separated list:

```
>>> {1, 2, 3}  
{1, 2, 3}  
>>> {"cheese", "ham"}  
{'cheese', 'ham'}  
>>> {"a", "b", "c"}  
{'c', 'a', 'b'}
```

By the way, you *cannot* use {} to create an empty set! {} by itself will create an empty dictionary. To create empty sets, you need the next method.

Calling set on an iterable

You can call the built-in function `set` on any iterable to create a set out of the elements of that iterable. Notable examples include ranges, strings, and lists.

```
>>> set(range(3))  
{0, 1, 2}  
>>> set([73, "water", 42])  
{73, 'water', 42}
```

Notice that calling `set` on a string produces a set with the characters of the string, not a set containing the whole string:

```
>>> {"mississippi"}  
{'mississippi'}  
  
## ↑ different ↓
```

```
>>> set("mississippi")
{'s', 'i', 'p', 'm'}
```

Calling `set()` by itself will produce an empty set.

Set comprehensions

Using `{}`, one can also write what's called a set comprehension. Set comprehensions are very similar to list comprehensions, so learning about list comprehensions will be helpful here.

I'll just show a couple of brief examples.

First, one using filtering some of the elements we want to include:

```
>>> veggies = ["broccoli", "carrot", "tomato", "pepper", "lettuce"]
>>> {veggie for veggie in veggies if "c" in veggie}
{'lettuce', 'carrot', 'broccoli'}
```

And secondly, a set comprehension with two nested `for` loops:

```
>>> veggies = ["broccoli", "carrot", "tomato", "pepper", "lettuce"]
>>> {char for veggie in veggies for char in veggie}
{'c', 'u', 't', 'o', 'p', 'b', 'l', 'i', 'a', 'e', 'm', 'r'}
```

I'll be writing a thorough Pydon't about all types of comprehensions that Python supports, so be sure to [subscribe to the newsletter](#) in order to not miss that upcoming Pydon't!

Operations on a single set

Many common operations are done on/with a single set, namely:

- membership testing:

```
>>> "milk" in groceries
True
>>> "broccoli" in groceries
False
```

- computing the size of the set:

```
>>> len(groceries)
3
```

- popping a random element from the set:

```
>>> groceries.pop()
'cheese'
>>> groceries
{'milk', 'chocolate'}
```

- adding an element to the set:

```
>>> groceries.add("cheese")
>>> groceries
{'milk', 'cheese', 'chocolate'}
```

Iteration

I often relate sets with lists (and tuples). Sets are similar to lists with unique elements, but lists are ordered: a list can be traversed from the beginning to the end, and a list can be indexed.

While sets can also be iterated over (in an order you can't rely on),

```
>>> for item in groceries:
...     print(item)
...
cheese
milk
chocolate
```

sets cannot be indexed directly:

```
>>> groceries[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Computations with multiple sets

When you have multiple sets (two, or more) you may need to do other sorts of operations.

Let's define just a couple of sets to use here:

```
>>> groceries = {"milk", "cheese", "chocolate"}
>>> treats = {"chocolate", "popcorn", "cookie"}
```

Here are some of the more common operations

- check for overlap between two sets:

```
>>> groceries & treats
{'chocolate'}
```

- join the two sets:

```
>>> groceries | treats
{'cheese', 'milk', 'popcorn', 'chocolate', 'cookie'}
```

Notice that the usage of the pipe | here is akin to the usage of | to merge dictionaries in Python 3.9+.

- find the difference between the two sets (what's on the left set but not on the right set):

```
>>> groceries - treats
{'cheese', 'milk'}
```

- check for containment using <, <=, >=, and >:

```

>>> {"cheese", "milk"} < groceries
True
>>> groceries < groceries
False
>>> {"cheese", "milk"} <= groceries
True
>>> groceries <= groceries
True
>>> treats > {"chocolate"}
True
>>> treats >= {"chocolate", "cheese"}
False

```

Notice that most of the operator-based operations have corresponding method calls. The corresponding method calls can accept an arbitrary iterator, whereas the operator-based versions expect sets.

Differences between set and frozenset

Creation

While you can create a set with the built-in `set` or through the `{}` notation, frozensets can only be created through their respective built-in.

frozensets can be created out of other sets or out of any iterable, much like sets.

When printed, frozensets display the indication that they are frozen:

```

>>> groceries = {'cheese', 'milk', 'chocolate'}
>>> frozenset(groceries)
frozenset({'cheese', 'milk', 'chocolate'})
>>> frozenset(['cheese', 'milk', 'chocolate'])
frozenset({'cheese', 'milk', 'chocolate'})

```

Mutability

Sets are mutable. Sets are said to be mutable because they can change, that's what "mutable" means in English.

As I showed you above, the contents of sets can change, for example through calls to the methods `.add` and `.pop`.

However, if you need to create an object that behaves like a set, (i.e. where order doesn't matter and where uniqueness is guaranteed) but that you don't want to be changed, then you want to create a frozenset.

An instance of a frozenset is pretty much like a set, except that frozenset isn't mutable. In other words, a frozenset is immutable, it can't be mutated, it was frozen.

To create a frozenset, you just call the appropriate class:

```

>>> groceries_ = frozenset(groceries)
>>> # Can't add items:
>>> groceries_.add("beans")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> # Can't pop items:
>>> groceries_.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'pop'

```

There's a very similar pair of built-in types that have this same dichotomy: lists and tuples. Lists are mutable (they have the methods `.append` and `.pop`, for example) whereas tuples are immutable (they don't have the methods `.append` or `.pop`, nor can you assign directly to indices):

```

## Lists are mutable:
>>> l = [0, 1, 2]
>>> l[0] = 73
>>> l.pop()
2
>>> l.append(42)
>>> l
[73, 1, 42]

## Tuples are immutable:
>>> t = (0, 1, 2)
>>> t[0] = 73
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
>>> t.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'pop'
>>> t.append(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'

```

To be (hashable) or not to be

An object that is hashable is an object for which a hash can be computed, hence, hash-able.

A hash is an integer that the built-in function `hash` computes to help with fast operations with dictionaries, e.g. key lookups.

The built-in function knows how to work with some types of objects, and not with others. The built-in function `hash` dictates what can and cannot be a dictionary key: if it is hashable, it can be a dictionary key; if it isn't

hashable, it cannot be a dictionary key.

For example, lists are mutable and unhashable, and hence they cannot be dictionary keys. Attempting to use a list as a dictionary key raises an error:

```
>>> d = []
>>> d[[1, 2, 3]] = 73
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

However, the tuple – list’s sibling – is immutable, and immutable objects can generally be made hashable. A tuple can be used as a dictionary key:

```
>>> d = {}
>>> d[(1, 2, 3)] = 73
>>> d
{(1, 2, 3): 73}
```

Similarly, because sets are mutable, they cannot be hashable. However, frozensets are not mutable, and they are also hashable! A set cannot be a dictionary key, but a frozenset can:

```
>>> d = {}
>>> d[groceries] = 73
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> d[frozenset(groceries)] = 73
>>> d
{frozenset({'cheese', 'milk', 'chocolate'}): 73}
```

What are sets used for?

Quoting [directly from the docs](#),

“Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.”

In short, sets are useful when the problems at hand would benefit from the properties that are inherent to mathematical sets (element uniqueness and lack of order) and other benefits we inherit from those properties.

The example of fast membership checking is a good one.

In a recent tweet, I showed how 10-element sets already outperform 10-element lists when doing membership checking:

Did you know that Python’s sets are very appropriate for fast membership checking? If you look, you can see that checking if something is in a set is MUCH faster than in the list as the element goes further down the list...And the example below only has 10 elements! pic.twitter.com/iJWwqeNIt0

— Rodrigo ☒ (@mathsppblog) September 1, 2021

These properties will be the main rationale followed by the programmers that wrote the pieces of code I will be showing you, showcasing good usages of set.

Examples in code

The examples that follow are my attempts at showing you good usages of the built-in types set and frozenset.

Fast membership checking with set

The module `argparse` is a built-in module that allows you to create command line interfaces.

The main class, the argument parser `ArgumentParser`, contains the following snippet of code:

```
## In Lib/argparse.py from Python 3.9.2
class ArgumentParser(_AttributeHolder, _ActionsContainer):
    ...
    def _parse_known_args(self, arg_strings, namespace):
        ...
        # map all mutually exclusive arguments to the other arguments
        # they can't occur with
        action_conflicts = {}
        ...

        seen_actions = set()
        seen_non_default_actions = set()

        def take_action(action, argument_strings, option_string=None):
            seen_actions.add(action)
            argument_values = self._get_values(action, argument_strings)

            # error if this argument is not allowed with other previously
            # seen arguments, assuming that actions that use the default
            # value don't really count as "present"
            if argument_values is not action.default:
                seen_non_default_actions.add(action)
                for conflict_action in action_conflicts.get(action, []):
                    if conflict_action in seen_non_default_actions:
                        msg = _('not allowed with argument %s')
                        action_name = _get_action_name(conflict_action)
                        raise ArgumentError(action, msg % action_name)
```

TL;DR: the sets `seen_actions` and `seen_non_default_actions` are being used precisely for fast membership checking.

Now follows a lengthier explanation.

When you create a command line application with argparse, you have to specify the options that your command takes. For example, `-v` for verbose output or `-h` to display the help message.

Sometimes, there may be conflicting options. For example, if you provide `-v` for verbose output, and also `-q` for quiet output, then it won't make sense to specify both at the same time.

The `action_conflicts` dictionary will keep track of what things conflict with what.

Later, we initialise two empty sets, `seen_actions` and `seen_non_default_actions`. Now, every time we see an action, we add it to the set that contains all actions that have been seen.

Then, if that action was really specified by the user, we add it to the set of actions that didn't have the default value.

Finally, we access the `action_conflicts` to get a list of all the actions that are incompatible with the action we are parsing now. If any conflicting action shows up in the set of actions we already saw previously, then we throw an error!

Later down the road, we can also find the following:

```
## In Lib/argparse.py from Python 3.9.2
class ArgumentParser(_AttributeHolder, _ActionsContainer):
    #
    def _parse_known_args(self, arg_strings, namespace):
        #

        seen_actions = set()
        seen_non_default_actions = set()

        def take_action(action, argument_strings, option_string=None):
            #

            #
            # make sure all required actions were present and also convert
            # action defaults which were not given as arguments
            required_actions = []
            for action in self._actions:
                if action not in seen_actions:
                    if action.required:
                        required_actions.append(_get_action_name(action))
                #

            if required_actions:
                self.error(_( 'the following arguments are required: %s' ) %
                           ', '.join(required_actions))
```

Once more, we are using the set `seen_actions` for fast membership checking: we traverse all the actions that the command line interface knows about, and we keep track of all the required actions that the user didn't specify/mention.

After that, if there are any actions in the list `required_actions`, then we let the user know that they forgot some things.

Unconditional set addition

There is one other neat detail about the previous example, that I'd like to highlight.

Let me show you the snippet that matters:

```
## In Lib/argparse.py from Python 3.9.2
class ArgumentParser(_AttributeHolder, _ActionsContainer):
    #
    def _parse_known_args(self, arg_strings, namespace):
        #
        def take_action(action, argument_strings, option_string=None):
            seen_actions.add(action)
            #
Focus on the very last line of code: seen_actions.add(action).
```

This might not seem obvious at first, but `action` might already be in the set `seen_actions`.

To make this clear, modify `take_action` to include a `print`:

```
## In Lib/argparse.py from Python 3.9.2
class ArgumentParser(_AttributeHolder, _ActionsContainer):
    #
    def _parse_known_args(self, arg_strings, namespace):
        #
        def take_action(action, argument_strings, option_string=None):
            print(action)
            seen_actions.add(action)
            #
Now, go ahead and paste the following code into a file foo.py:
```

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-t", action="store_true")
args = parser.parse_args()
```

Now open your terminal in the directory where `foo.py` lives:

```
> python foo.py -ttt
_StoreTrueAction(option_strings=['-t'], dest='t', nargs=0, const=True, default=False, type=None, choices=None)
_StoreTrueAction(option_strings=['-t'], dest='t', nargs=0, const=True, default=False, type=None, choices=None)
_StoreTrueAction(option_strings=['-t'], dest='t', nargs=0, const=True, default=False, type=None, choices=None)
```

You get three lines of identical output, one per each time you typed a `t` in the command.

So, we see that we have duplicate actions showing up... Shouldn't we check if an action has been added before adding it? Something like

```
## In Lib/argparse.py from Python 3.9.2
class ArgumentParser(_AttributeHolder, _ActionsContainer):
    ...
    def _parse_known_args(self, arg_strings, namespace):
        ...
        def take_action(action, argument_strings, option_string=None):
            if action not in seen_actions:
                seen_actions.add(action)
        ...

```

No! Don't do that! This is an anti-pattern and is repeating unnecessary work! Checking if an element is inside a set or adding it unconditionally is almost the same work, so checking if it is there and *then* adding it is going to double the work you do for all new actions!

The set already handles uniqueness for you, so you don't have to be worried enforcing it. In that sense, this is a great example usage of sets.

Fast membership checking with frozenset

In the example above, we saw that the sets we were working with would grow as the program progressed. Therefore, we needed mutability and used set.

The example I'm about to show is such that the set we care about is fixed, it always has the same elements. Henceforth, we will use a frozenset instead of a plain set. Why? Because using frozenset makes it crystal clear that the set is fixed.

```
## In Lib/idlelib/hyperparser.py from Python 3.9.2

import string

## ...

## all ASCII chars that may be in an identifier
_ASCII_ID_CHARS = frozenset(string.ascii_letters + string.digits + "_")
## all ASCII chars that may be the first char of an identifier
_ASCII_ID_FIRST_CHARS = frozenset(string.ascii_letters + "_")

## lookup table for whether 7-bit ASCII chars are valid in a Python identifier
_IS_ASCII_ID_CHAR = [(chr(x) in _ASCII_ID_CHARS) for x in range(128)]
## lookup table for whether 7-bit ASCII chars are valid as the first
## char in a Python identifier
_IS_ASCII_ID_FIRST_CHAR = \
    [(chr(x) in _ASCII_ID_FIRST_CHARS) for x in range(128)]
```

Granted, the snippet above does not tell you what the variables `_IS_ASCII_ID_CHAR` and `_IS_ASCII_ID_FIRST_CHAR` are for, but it is quite clear that those two are being built through a list comprehension that does membership

checking on `_ASCII_ID_CHARS` and `_ASCII_ID_FIRST_CHARS`. In turn, these two variables are frozensets of characters!

So there you have it! One more usage of sets for fast membership checking.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Use (frozen) sets when you are dealing with collections and where what matters is (fast) membership checking.”

This Pydon't showed you that:

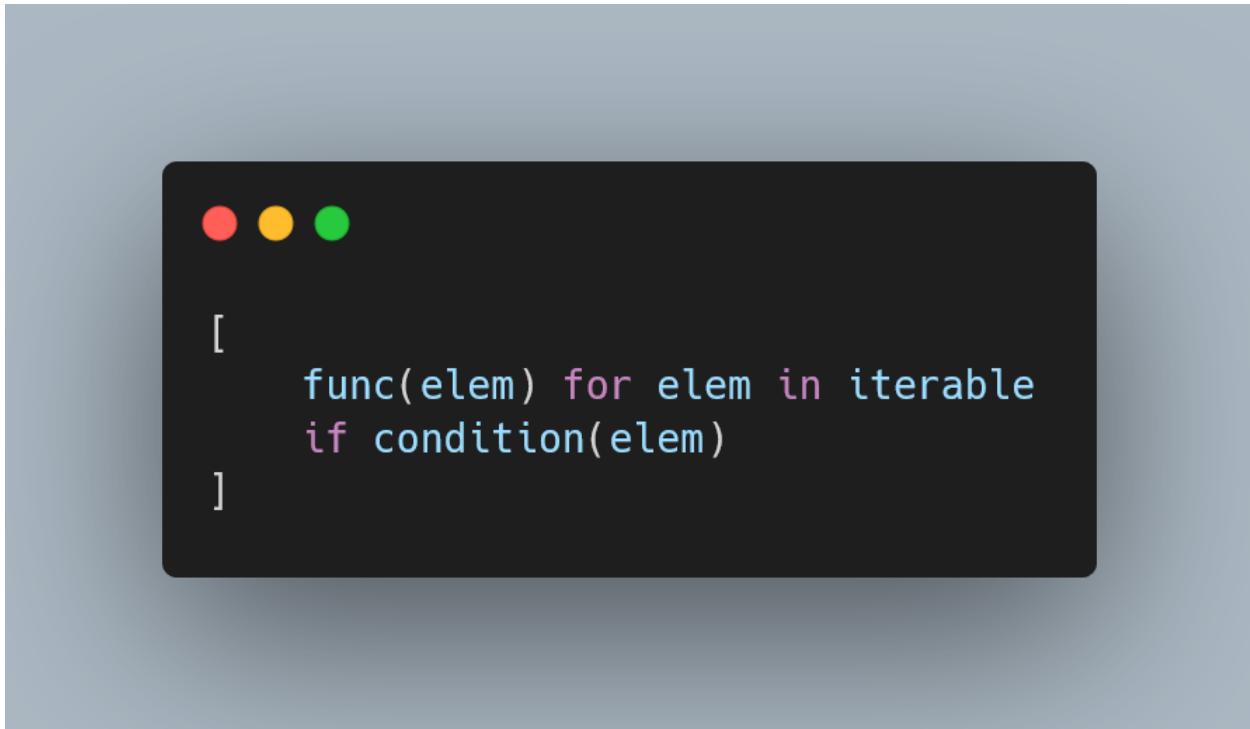
- sets are (mathematical) objects that contain elements;
 - the elements are unique; and
 - their ordering doesn't matter.
- the built-in type `set` provides an implementation for the mathematical concept of set;
- the `frozenset` is an immutable and hashable version of `set`;
- tuples are to lists like frozen sets are to sets;
- you can create sets with
 - `{}` enclosing a comma-separated list of items;
 - `set()` and an iterable; and
 - set comprehensions.
- sets have operations that allow to mutate them (like `.add` and `.append`), among many others;
- you can combine sets in many different ways, with operators like `&` and `|`;
- you can check for set containment with `<`, `<=`, `>=`, `>`;
- you should use `frozenset` if you know the collection of objects won't change;
- (frozen) sets are often used for fast membership checking; and
- unconditionally adding to a set is faster than checking for membership first and adding latter.

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 3 Docs, The Python Standard Library, Built-in Types, Set Types \otimes `set`, `frozenset`, <https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset> [last accessed 14-09-2021];
- Python 3 Docs, The Python Standard Library, Built-in Functions, `hash`, <https://docs.python.org/3/library/functions.html#hash> [last accessed 14-09-2021];
- Python 3 Docs, The Python Language Reference, Special method names, `object.__hash__`, https://docs.python.org/3/reference/datamodel.html#object.__hash__ [last accessed 14-09-2021];
- Python 3 Docs, The Python Standard Library, `argparse`, <https://docs.python.org/3/library/argparse.html> [last accessed 14-09-2021];

List comprehensions 101



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/list-comprehensions-101>.)

Introduction

List comprehensions are, hands down, one of my favourite Python features.

It's not THE favourite feature, but that's because Python has a lot of things I really like! List comprehensions being one of those.

This Pydon't (the first in a short series) will cover the basics of list comprehensions.

In this Pydon't, you will:

- learn the anatomy of a list comprehension;
 - learn how to create list comprehensions; and
 - understand the building blocks of list comprehensions;
- see the parallel that exists between some `for` loops and list comprehensions;
- establish a correspondence between `map` and `filter`, and list comprehensions;
- understand the main use-case for this feature; and
- see good usages of list comprehensions in real code written by real people.

I also summarised the contents of this Pydon't in a cheatsheet that you can get for free [from here](#).

What is a list comprehension?

A list comprehension is a Python expression that returns a list.

List comprehensions are great because they provide a very convenient syntax to generate simple lists. This often is a good alternative to using a full `for` loop with a series of calls to the `.append` method of the list you want to build.

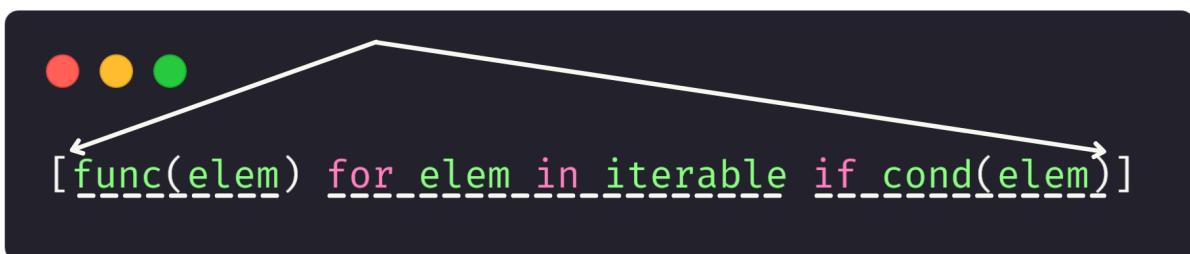
However, list comprehensions have a structure that is very similar to the equivalent `for` loops, so list comprehensions should be easy to use if you understand the relationship between a list comprehension and the corresponding loop.

The key idea behind list comprehensions is that many lists can be built out of other, simpler iterables (lists, tuples, strings, ...).

For example, if you want to build a list with some square numbers, you will probably start off by using a `range` to create a series of consecutive integers.

Anatomy of a list comprehension

A list comprehension has four parts, one of which is optional. Here is a diagram representing those four parts:



The first part is a set of opening and closing brackets, that delimit the list comprehension. The brackets, by themselves, do not automatically indicate a list comprehension, because they can also be used to create list literal, like [1, 2, 3].

The second part is the expression that you apply to each element of the initial seed data you are using. This is often a function call or another expression, like an arithmetic expression, that transforms each element into a new one. In the diagram above, this is represented by `func(elem)`.

The third part is the `for` component that establishes what the initial data is, and where we are going to draw our elements from. This is akin to the `for ... in ...` of a standard `for` loop. In fact, it looks exactly the same as the initial statement of a `for` loop, and it is represented by `for elem in iterable` in the diagram above.

The fourth part, which is optional, is an `if` statement. This `if` statement is used to filter elements from the initial seed data, in case we want to ignore some of it/only use part of the data. This is represented by the `cond(elem)` above.

Enough of theoretical gibberish, let's look at some actual list comprehensions.

Example list comprehensions without filtering

1. First square numbers:

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. Uppercasing a series of words:

```
>>> words = "This is Sparta!".split()
>>> [word.upper() for word in words]
['THIS', 'IS', 'SPARTA!']
```

3. Find the length of each word in a sentence:

```
>>> words = "To be or not to be, that is the question.".split()
>>> [len(word) for word in words]
[2, 2, 2, 3, 2, 3, 4, 2, 3, 9]
```

Example list comprehensions with filtering

1. First square numbers for even `n`:

```
>>> [n ** 2 for n in range(10) if n % 2 == 0]
[0, 4, 16, 36, 64]
```

This is also equivalent to,

```
>>> [n ** 2 for n in range(0, 10, 2)]
[0, 4, 16, 36, 64]
```

which is better because it uses less initial data.

2. Uppercase a series of words if they are all lower case:

```
>>> words = "This is Sparta!".split()  
>>> [word.upper() for word in words if word == word.lower()]  
['IS']
```

The result only contains the word “is” because that’s the only word that was entirely lower case in the original sentence “This is Sparta!”.

3. Find the length of each word that does not have punctuation next to it:

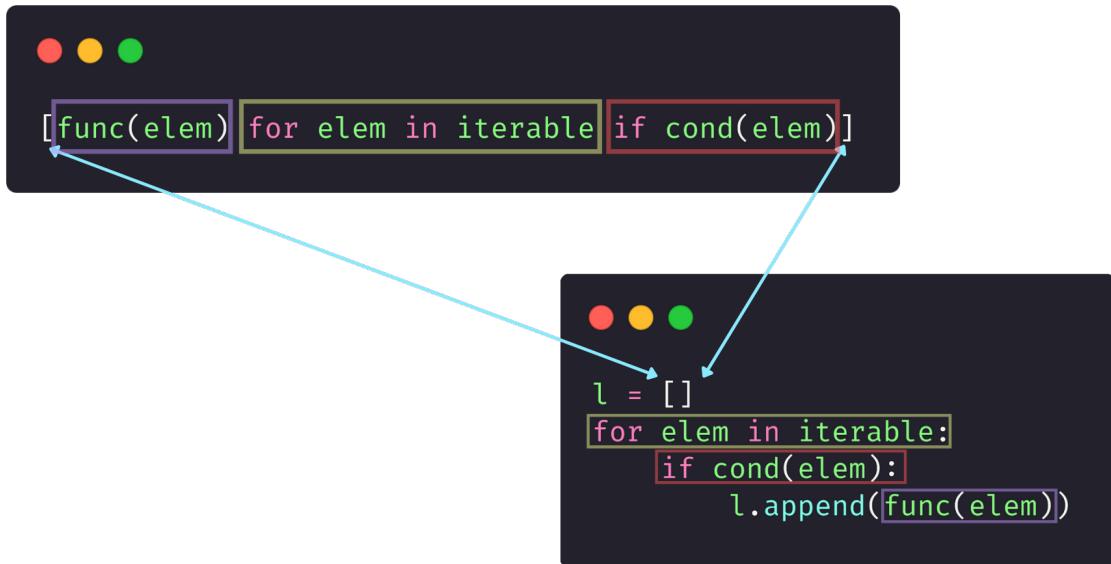
```
>>> words = "To be or not to be, that is the question.".split()  
>>> words  
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question.'][  
>>> [len(word) for word in words if word.isalpha()]  
[2, 2, 2, 3, 2, 4, 2, 3]
```

The final result only contains 8 numbers (while the original sentence contains 10 words) because the words “be,” and “question.” had punctuation next to them.

Equivalence with for loops

Now that you looked at some list comprehensions, it is time to create an analogy with for loops. If you know your for loops well enough, and if you study this equivalence, you will master list comprehensions.

A list comprehension is equivalent to a for loop that consecutively calls the .append method of an (initially) empty list:



This is, in fact, one of the most common patterns that list comprehensions are useful for.

If you find a piece of code that initialises an empty list, and then uses a `for` loop to populate it with data, that's probably a good use case for a list comprehension.

Of course this isn't always doable in a sensible way, list comprehensions are not meant to replace *all* `for` loops. But if you have a short loop exhibiting the structure above, then that could probably be replaced by a list comprehension.

I challenge you to do just that. Go through some code of yours and look for that pattern. Then, try replacing it with a list comprehension.

Here are the list comprehensions from before, with the equivalent `for` loops:

1. Squaring:

```

even_squares = [n ** 2 for n in range(10) if n % 2 == 0]
## ↑
## ↓
even_squares = []
for n in range(10):
    if n % 2 == 0:
        even_squares.append(n ** 2)

```

2. Upper casing words:

```
words = "This is Sparta!".split()
```

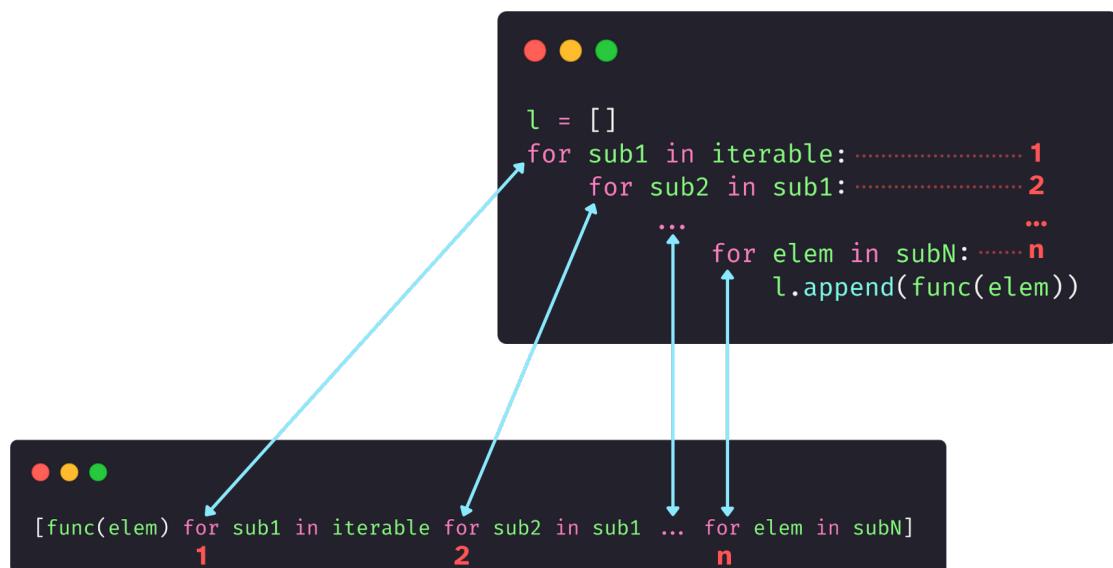
```
upper_cased = [word.upper() for word in words if word == word.lower()]
## ↑
## ↓
upper_cased = []
for word in words:
    if word == word.lower():
        upper_cased.append(word.upper())
```

3. Finding length of words:

```
words = "To be or not to be, that is the question.".split()
lengths = [len(word) for word in words if word.isalpha()]
## ↑
## ↓
lengths = []
for word in words:
    if word.isalpha():
        lengths.append(len(word))
```

Nesting for loops

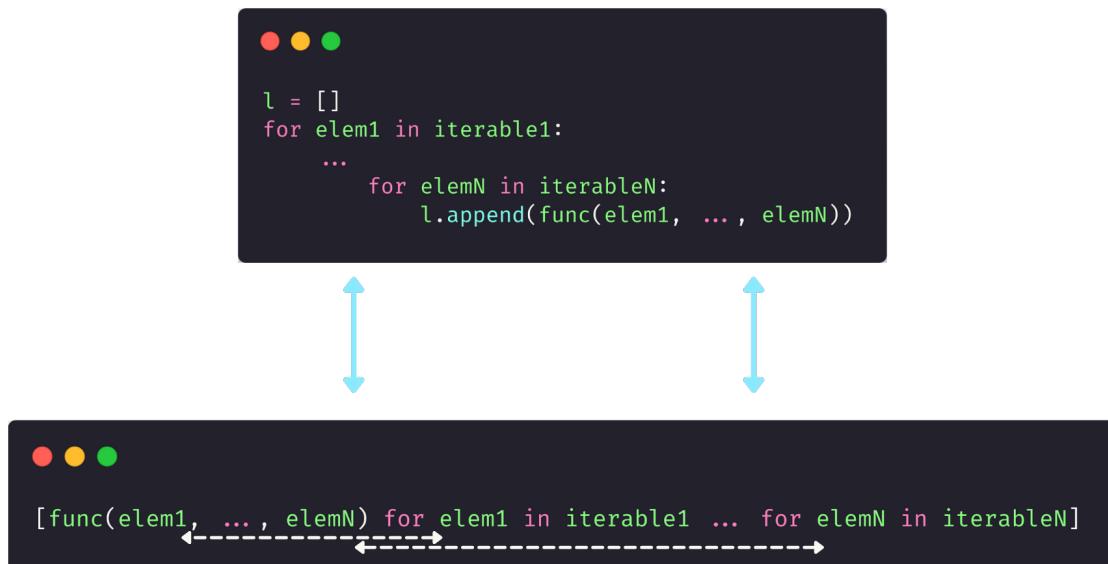
Much like in standard for loops, list comprehensions can also have nested loops:



A prime usage example of this is to flatten a list of lists:

```
>>> lists = [[1, 2, 3], [4, 5, 6, 7], [8, 9]]  
>>> [elem for sublist in lists for elem in sublist]  
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The second loop doesn't need to depend explicitly on the first one; it can iterate over another iterable, to create another variable. When you do so, all the temporary variables that are going through iterables become available to be used on the left.



This pattern arises naturally when you want to *combine* information from two or more data sources:

```
>>> colours = ["red", "green", "blue"]  
>>> clothes = ["t-shirt", "shirt"]  
>>> [f"{colour} {clothing}" for colour in colours for clothing in clothes]  
['red t-shirt', 'red shirt', 'green t-shirt', 'green shirt', 'blue t-shirt', 'blue shirt']
```

Notice that, in here, to “combine” the information means to create all different pairings with the data from one and the other iterable. If you want to create pairings by traversing two iterables in parallel, then you should read up on `zip`.

Nesting if statements

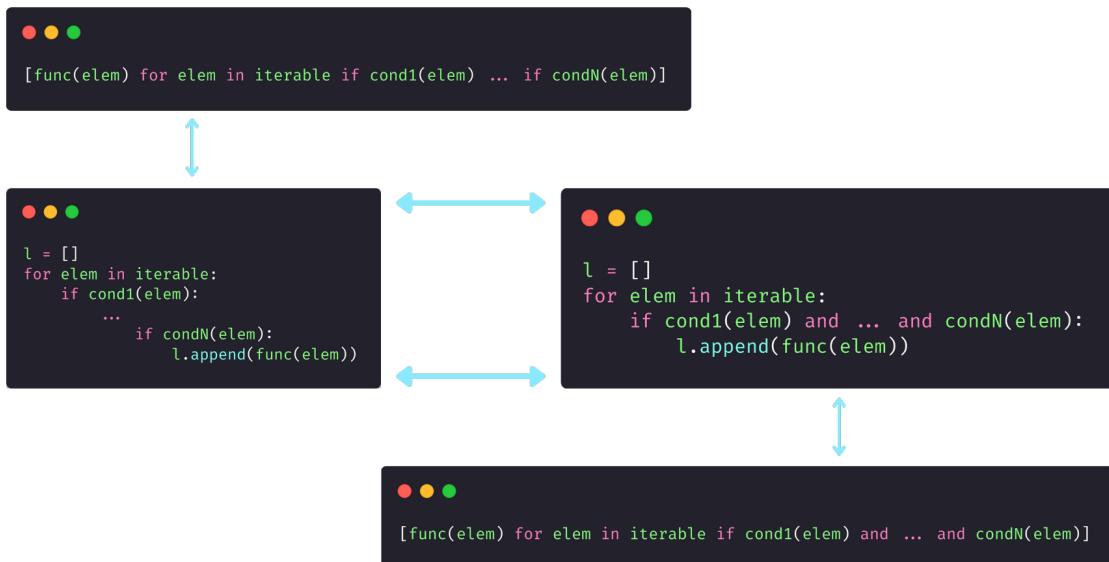
Much like you can nest `for` loops to iterate over more iterables, you can nest `if` statements to create stricter filters.

When you have a series of `if` statements, the second condition only runs if the first one passed; the third

condition only runs if the second one passed; and so on.

However, with the `if` statements, this is the same as combining the successive conditions with `ands`. That's because **Boolean short-circuiting** makes sure that later conditions only get evaluated if the earlier ones evaluated to `True`.

This means there is a series of equivalences when we think about list comprehensions with nested `if` statements:



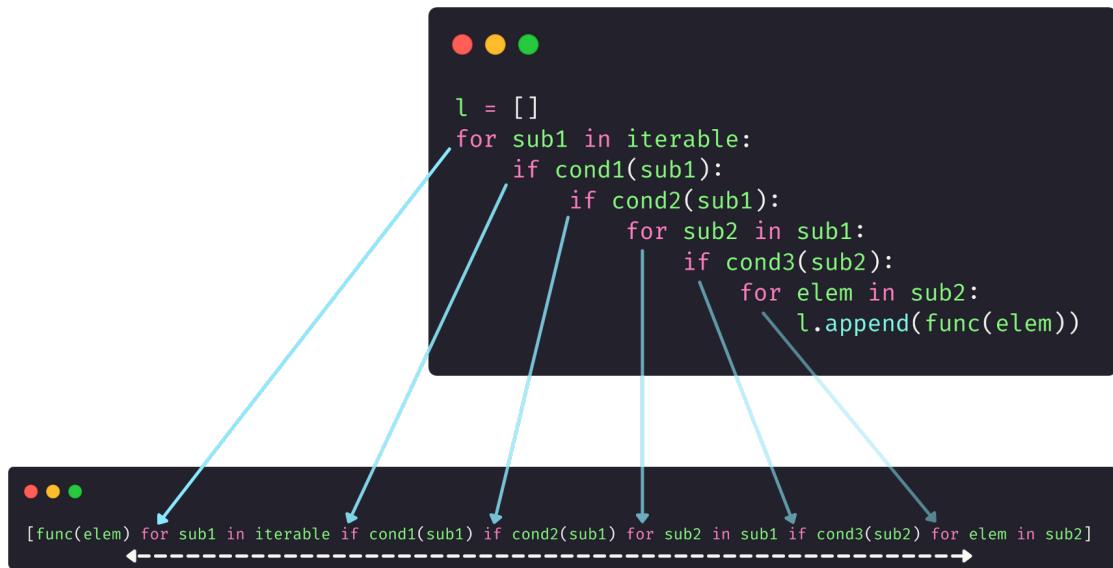
Arbitrary nesting

The two sections above showed you that you can nest multiple `for` loops, and also multiple `if` statements.

Now, the only thing left for you to know is that these can be mixed and nested arbitrarily. Of course, you should not nest things too much, because a long list comprehension is harder to read than the equivalent nested formulation.

The diagram below helps you in understanding the correspondence between the order of things in the nested formulation and the left-to-right ordering of things in the list comprehension.

The further you are to the right in a list comprehension, the deeper you are in the equivalent nested formulation:



List comprehensions instead of map and filter

List comprehensions are often deemed a more Pythonic replacement for calls to the built-in functions `map` and `filter`.

`map` takes a function and applies it to all elements of an iterable, and that's straightforward to do with a list comprehension:



Similarly, the built-in `filter` can often be replaced with a more Pythonic list comprehension.



Please, bear in mind that the list comprehension versions of `map` and `filter` are **not** equivalent to using `map` and `filter`. The underlying data is, but the containers themselves are slightly different.

Not only that, but I'm also not saying that `map` and `filter` are useless. A later Pydon't will be devoted to understanding when to use `map` and `filter`, so make sure to [subscribe to the newsletter](#) to not miss that Pydon't.

Examples in code

Random data

A neat little example of where a list comprehension is *the way to go*, is when generating some random data.

For example, to generate three integers to represent an RGB colour,

```
>>> from random import randint
>>> r, g, b = [randint(0, 255) for _ in range(3)]
>>> r
180
>>> g
148
>>> b
188
```

or when generating a random string:

```
>>> from string import ascii_lowercase, ascii_uppercase
>>> from random import choice
>>> "".join([choice(ascii_lowercase + ascii_uppercase) for _ in range(16)])
'qMQLkhvKJfdZGBEZ'
```

Getting AWS prefixes

While browsing Twitter, I found someone writing a little Python script to interact with Amazon Web Services to get IP prefixes for different services. (Whatever that means.)

At some point, they had a simple `for` loop that was iterating through a bunch of prefixes and storing them in a list, provided that that prefix had to do with a specific Amazon service.

This person is a self-proclaimed Python beginner, and so I thought this was a good opportunity to [show how list comprehensions can be useful](#).

The relevant excerpt of the original code is as follows:

```
def get_service_prefixes(amazon_service):
    aws_prefixes = get_aws_prefixes()
    count = 0
    service_prefixes = []
    for prefix in aws_prefixes["prefixes"]:
        if amazon_service in prefix["service"]:
            count += 1
            service_prefixes.append(prefix["ip_prefix"])

    # ...
```

Looking at the code above, we can see that the list `service_prefixes` is being created and then appended to in the `for` loop; also, that's the *only* purpose of that `for` loop.

This is the generic pattern that indicates a list comprehension might be useful!

Therefore, we can replace the loop with a list comprehension. The variable `count` is superfluous because it keeps track of the length of the resulting list, something we can find out easily with the function `len`.

Here is a possible alternative using a list comprehension:

```
def get_service_prefixes(amazon_service):
    service_prefixes = [
        prefix for prefix in get_aws_prefixes()
        if amazon_service in prefix["service"]
    ]
    count = len(service_prefixes)

    # ...
```

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"List comprehensions are a powerful Python feature that is useful for building lists."

This Pydon't was also summarised in a [free cheatsheet](#):

Python list comprehensions 101  [@mathspb](#)

Anatomy

Brackets used to build lists start and end the list comp.

(Optional) Condition to filter each element. Only those elements which the condition is True will be processed and included.

[`func(elem)` for `elem` in `iterable` if `cond(elem)`]

Process each element. The function call symbolises any valid Python expression.

Temporary variable holding each element of the iterable.

Iterable (list, tuple, string...) that we want to traverse. Often, where we draw the iterable from.

Examples

```
words = "This is Sports".split()
words = [word.upper() for word in words]
words = [word.upper() for word in words if word == word.lower()]
words = [word]
```

```
words = "To be or not to be, that is the question.".split()
words = [word for word in words if word[0] == 'T' or word[-1] == 't' or word == 'that' or word == 'the' or word == 'question']
words = [word for word in words if word.isalpha()]
words = [word for word in words if word[0] == 'T' or word[-1] == 't' or word == 'that' or word == 'the' or word == 'question']
```

Multiple loops

You can nest loops inside one another, arbitrarily.

However, you shouldn't go too far. (Neither in list comprehensions, nor in "regular" loops.)

```
l = []
for sub1 in iterable:
    for sub2 in sub1:
        ...
        for elem in sub2:
            l.append(func(elem))
```

The loops can be independent. This comes in handy, for example, when 'func' accepts 2+ arguments.

```
[func(elem1, ..., elemN) for elem1 in iterable1 ... for elemN in iterableN]
```

Names defined on the right can be used on the left.

Multiple conditions

You can nest a series of 'if' statements...

```
[func(elem) for elem in iterable if cond1(elem) ... if condN(elem)]
```

Nesting 'if' statements = composing with 'and'.

```
l = []
for elem in iterable:
    if cond1(elem):
        if cond2(elem):
            l.append(func(elem))
```

... but you should opt for composing with 'and'.

```
[func(elem) for elem in iterable if cond1(elem) and ... and condN(elem)]
```

'map' and 'filter'

List comprehension instead of 'map':

```
[func(elem) for elem in iterable] ← list(map(func, iterable))
```

Special case when the 'filter' argument is 'None':

```
[elem for elem in iterable if elem] ← list(filter(None, iterable))
```

Arbitrary nesting

```
(func1(elem) for sub1 in iterable1 cond1(sub1) if cond2(sub1) for sub2 in sub1 cond2(sub2) for sub3 in sub2 cond3(sub3) for elem in sub3)
```

This Pydon't showed you that:

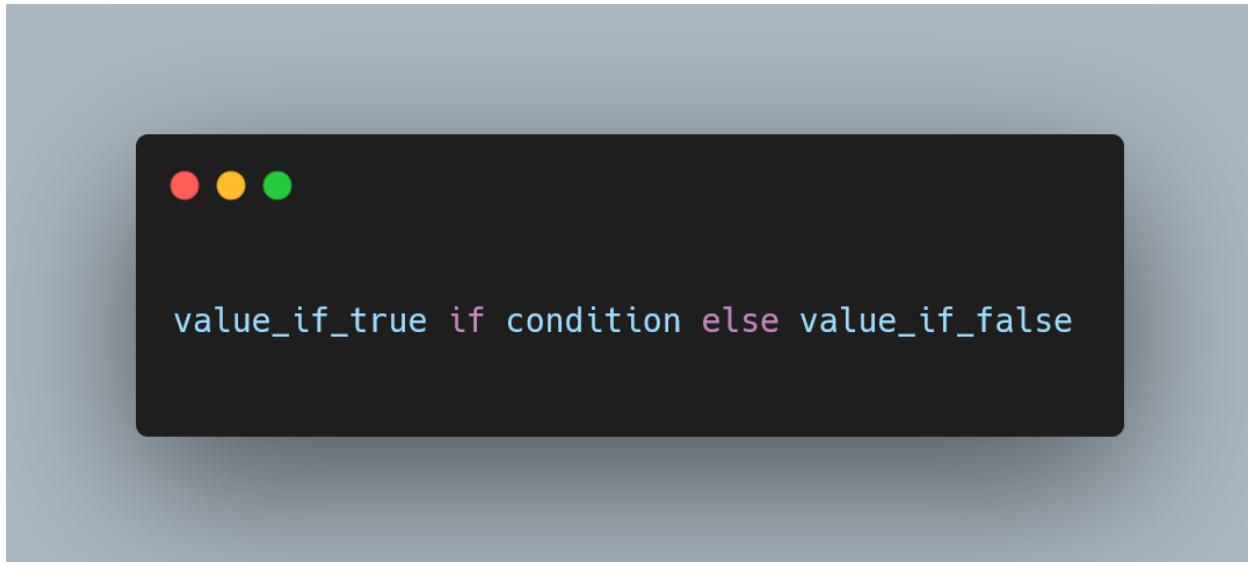
- a list comprehension has 4 parts, one of which is optional;
- list comprehensions can transform data drawn from another iterable;
- list comprehensions can filter the data they transform;
- each list comprehension is equivalent to a for loop that successively calls .append on a list that is initialised empty;
- list comprehensions can nest arbitrarily many for loops;
- list comprehensions can nest arbitrarily many if statements;
- nesting if statements is equivalent to combining conditions with and;
- map and filter can often be replaced with list comprehensions;
- simple loops whose only job is to append to a list can often be replaced with list comprehensions.

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 3 Docs, The Python Tutorial, Data Structures, More on Lists, List Comprehensions <https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions> [last accessed on 24-09-2021];
- Python 3 Docs, The Python Standard Library, Built-in Functions, filter, <https://docs.python.org/3/library/functions.html#filter> [last accessed 22-09-2021];
- Python 3 Docs, The Python Standard Library, Built-in Functions, map, <https://docs.python.org/3/library/functions.html#map> [last accessed 22-09-2021];

Conditional expressions



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/conditional-expressions>.)

Introduction

Conditional expressions are what Python has closest to what is called a “ternary operator” in other languages.

In this Pydon’t, you will:

- learn about the syntax of conditional expressions;
- understand the rationale behind conditional expressions;
- learn about the precedence of conditional expressions;
- see how to nest conditional expressions;
- understand the relationship between `if: ... elif: ... else:` statements and conditional expressions;
- see good and bad example usages of conditional expressions;

What is a conditional expression?

A conditional expression in Python is an expression (in other words, a piece of code that evaluates to a result) whose value depends on a condition.

Expressions and statements

To make it clearer, here is an example of a Python expression:

```
>>> 3 + 4 * 5
23
```

The code `3 + 4 * 5` is an expression, and that expression evaluates to 23.

Some pieces of code are not expressions. For example, `pass` is not an expression because it does not evaluate to a result. `pass` is just a statement, it does not “have” or “evaluate to” any result.

This might be odd (or not!) but to help you figure out if something is an expression or not, try sticking it inside a `print` function. Expressions can be used inside other expressions, and function calls are expressions. Therefore, if it can go inside a `print` call, it is an expression:

```
>>> print(3 + 4 * 5)
23
>>> print(pass)
File "<stdin>", line 1
    print(pass)
^
SyntaxError: invalid syntax
```

The syntactic error here is that the statement `pass` cannot go inside the `print` function, because the `print` function wants to print *something*, and `pass` gives nothing.

Conditions

We are very used to using `if` statements to run pieces of code when certain *conditions* are met. Rewording that, a condition can dictate what piece(s) of code run.

In conditional expressions, we will use a condition to change the value to which the expression evaluates. Wait, isn’t this the same as an `if` statement? No! Statements and expressions are *not* the same thing.

Syntax

Instead of beating around the bush, let me just show you the anatomy of a conditional expression:

```
expr_if_true if condition else expr_if_false
```

A conditional expression is composed of three sub-expressions and the keywords `if` and `else`. None of these components are optional. All of them have to be present.

How does this work?

First, condition is evaluated. Then, depending on whether condition evaluates to **Truthy or Falsy**, the expression evaluates `expr_if_true` or `expr_if_false`, respectively.

As you may be guessing from the names, `expr_if_true` and `expr_if_false` can themselves be expressions. This means they can be simple literal values like 42 or "spam", or other "complicated" expressions.

(Heck, the expressions in conditional expressions can even be *other* conditional expressions! Keep reading for that ☺)

Examples of conditional expressions

Here are a couple of simple examples, broken down according to the `expr_if_true`, condition, and `expr_if_false` anatomy presented above.

1.

```
>>> 42 if True else 0
42
```

expr_if_true	condition	expr_if_false
42	True	0

2.

```
>>> 42 if False else 0
0
```

expr_if_true	condition	expr_if_false
42	False	0

3.

```
>>> "Mathspp".lower() if pow(3, 27, 10) > 5 else "Oh boy."
'mathsp'p'
```

expr_if_true	condition	expr_if_false
"Mathspp".lower()	pow(3, 27, 10) > 5	"Oh boy."

For reference:

```
>>> pow(3, 27, 10)
7
```

Reading a conditional expression

While the conditional expression presents the operands in an order that may throw some of you off, it is easy to read it as an English sentence.

Take this reference conditional expression:

```
value if condition else other_value
```

Here are two possible English “translations” of the conditional expression:

“Evaluate to value if condition is true, otherwise evaluate to other_value.”

or

“Give value if condition is true and other_value otherwise.”

With this out of the way, ...

Does Python have a ternary operator?

Many languages have a ternary operator that looks like `condition ? expr_if_true : expr_if_false`. Python does not have such a ternary operator, but conditional expressions are similar.

Conditional expressions are similar in that they evaluate one of two values, but they are syntactically different because they use keywords (instead of ? and :) and because the order of the operands is different.

Rationale

The rationale behind conditional expressions is simple to understand: programmers are often faced with a situation where they have to pick one of two values.

That's just it.

Whenever you find yourself having to choose between one value or another, typically inside an `if: ... else: ...` block, that might be a good use-case for a conditional expression.

Examples with if statements

Here are some simple functions that show that:

1. computing the parity of an integer:

```
def parity(n):  
    if n % 2:  
        return "odd"  
    else:  
        return "even"  
  
>>> parity(15)  
"odd"
```

```
>>> parity(42)
"even"
```

2. computing the absolute value of a number (this already exists as a built-in function):

```
def abs(x):
    if x > 0:
        return x
    else:
        return -x

>>> abs(10)
10
>>> abs(-42)
42
```

These two functions have a structure that is very similar: they check a condition and return a given value if the condition evaluates to `True`. If it doesn't, they return a different value.

Refactored examples

Can you refactor the functions above to use conditional expressions? Here is one possible refactoring for each:

```
def parity(n):
    return "odd" if n % 2 else "even"
```

This function now reads as

“return “odd” if `n` leaves remainder when divided by 2 and “even” otherwise.”

As for the absolute value function,

```
def abs(n):
    return x if x > 0 else -x
```

it now reads as

“return `x` if `x` is positive, otherwise return `-x`.”

Short-circuiting

You may be familiar with **Boolean short-circuiting**, in which case you might be pleased to know that conditional expressions also short-circuit.

For those of you who don't know Boolean short-circuiting yet, I can recommend [my thorough Pydon't article on the subject](#). Either way, it's something to understand for our conditional expressions: a conditional expression will only evaluate what it really has to.

In other words, if your conditional expression looks like

```
expr_if_true if condition else expr_if_false
```

then only one of `expr_if_true` and `expr_if_false` is ever evaluated. This might look silly to point out, but is actually quite important.

Some times, we might want to do something (`expr_if_true`) that *only* works if a certain condition is met.

For example, say we want to implement the quad-UCS function from [APL](#). That function is simple to explain: it converts integers into characters and characters into integers. In Python-speak, it just uses `chr` and `ord`, whatever makes sense on the input.

Here is an example implementation:

```
def ucs(x):
    if isinstance(x, int):
        return chr(x)
    else:
        return ord(x)

>>> ucs("A")
65
>>> ucs(65)
'A
>>> ucs(102)
'f'
>>> ucs("f")
102
```

What isn't clear from this piece of code is that `ord` throws an error when called on integers, and `chr` fails when called on characters:

```
>>> ord(65)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ord() expected string of length 1, but int found

>>> chr("f")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: an integer is required (got type str)
```

Thankfully, this is not a problem for conditional expressions, and therefore `ucs` can be implemented with one:

```
def ucs(x):
    return chr(x) if isinstance(x, int) else ord(x)

>>> ucs("A")
65
>>> ucs(65)
'A
>>> ucs(102)
```

```
'f'  
>>> ucs("f")  
102
```

Therefore, we see that when `x` is an integer, `ord(x)` never runs. On the flip side, when `x` is *not* an integer, `chr(x)` never runs. This is a very useful subtlety!

Conditional expressions and if statements

Equivalence to if

This has been implicit throughout the article, but I'll write it down explicitly now for the sake of clarity.

(And also because “**Explicit is better than implicit.**” ☺!)

There is a close relationship between the conditional expression

```
name = expr_if_true if condition else expr_if_false
```

and the if statement

```
if condition:  
    name = expr_if_true  
else:  
    name = expr_if_false
```

And that close relationship is that of equivalence. The two pieces of code are exactly equivalent.

Equivalence to if-elif-else blocks

Given the equivalence between conditional expressions and if: ... else: ... blocks, it is natural to wonder whether there is some equivalent to the `elif` statement in conditional expressions as well.

For example, can we rewrite the following function to use a conditional expression?

```
def sign(x):  
    if x == 0:  
        return 0  
    elif x > 0:  
        return 1  
    else:  
        return -1
```

```
>>> sign(-73)  
-1  
>>> sign(0)  
0  
>>> sign(42)  
1
```

How can we write this as a conditional expression? Conditional expressions do not allow the usage of the `elif` keyword so, instead, we start by reworking the `if` block itself:

```
def sign(x):
    if x == 0:
        return 0
    else:
        if x > 0:
            return 1
        else:
            return -1
```

This isn't a great implementation, but this intermediate representation makes it clearer that the bottom of the `if` block can be replaced with a conditional expression:

```
def sign(x):
    if x == 0:
        return 0
    else:
        return 1 if x > 0 else -1
```

Now, if we abstract away from the fact that the second return value is a conditional expression itself, we can rewrite the existing `if` block as a conditional expression:

```
def sign(x):
    return 0 if x == 0 else (1 if x > 0 else -1)

>>> sign(-73)
-1
>>> sign(0)
0
>>> sign(42)
1
```

This shows that conditional expressions can be nested, naturally. Now it is just a matter of checking whether the parenthesis are needed or not.

In other words, if we write

`A if B else C if D else E`

does Python interpret it as

`(A if B else C) if D else E`

or does it interpret it as

`A if B else (C if D else E)`

As it turns out, it's the latter. So, the `sign` function above can be rewritten as

```
def sign(x):
    return 0 if x == 0 else 1 if x > 0 else -1
```

It's this chain of `if ... else ... if ... else ...` – that can be arbitrarily long – that emulates `elifs`.

To convert from a long `if` block (with or without `elifs`) to a conditional expression, go from top to bottom and interleave values and conditions, alternating between the keyword `if` and the keyword `else`.

When reading this aloud in English, the word “otherwise” helps clarify what the longer conditional expressions mean:

```
return 0 if x == 0 else 1 if x > 0 else -1
```

reads as

“return 0 if x is 0, otherwise, return 1 if x is positive otherwise return -1.”

The repetition of the word “otherwise” becomes cumbersome, a good indicator that it is generally not a good idea to get carried away and chaining several conditional expressions.

For reference, here's a “side-by-side” comparison of the first conditional block and the final conditional expression:

```
## Compare
if x == 0:
    return 0
elif x > 0:
    return 1
else:
    return -1

## to:
return 0 if x == 0 else 1 if x > 0 else -1
```

Non-equivalence to function wrapper

Because of the equivalence I just showed, many people may then believe that conditional expressions could be implemented as a function enclosing the previous `if: ... else: ...` block:

```
def cond(condition, value_if_true, value_if_false):
    if condition:
        return value_if_true
    else:
        return value_if_false
```

With this definition, we might *think* we have implemented conditional expressions:

```
>>> cond(pow(3, 27, 10) > 5, "Mathspp".lower(), "Oh boy.")
'mathspp'
>>> "Mathspp".lower() if pow(3, 27, 10) > 5 else "Oh boy."
'mathspp'
```

In fact, we haven't! That's because the function call to `cond` only happens after we have evaluated all the arguments. This is different from what conditional expressions really do: as I showed [above](#), conditional expressions only evaluate the expression they need.

Hence, we can't use this `cond` to implement `ucs`:

```
def ucs(x):
    return cond(isinstance(x, int), chr(x), ord(x))
```

This code looks sane, but it won't behave like we would like:

```
>>> ucs(65)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ucs
TypeError: ord() expected string of length 1, but int found
```

When given 65, the first argument evaluates to `True`, and the second argument evaluates to "A", but the third argument raises an error!

Precedence

Conditional expressions are the expressions with lowest precedence, [according to the documentation](#).

This means that sometimes you may need to parenthesise a conditional expression if you are using it *inside* another expression.

For example, take a look at this function:

```
def foo(n, b):
    if b:
        to_add = 10
    else:
        to_add = -10
    return n + to_add
```

```
>>> foo(42, True)
52
>>> foo(42, False)
32
```

You might spot the pattern of assigning one of two values, and decide to use a conditional expression:

```
def foo(n, b):
    to_add = 10 if b else -10
    return n + to_add

>>> foo(42, True)
52
>>> foo(42, False)
32
```

But then, you decide there is no need to waste a line here, and you decide to inline the conditional expression (that is, you put the conditional expression *inside* the arithmetic expression with `n +`):

```
def foo(n, b):
    return n + 10 if b else -10
```

By doing this, you suddenly break the function when `b` is `False`:

```
>>> foo(42, False)
-10
```

That's because the expression

```
n + 10 if b else -10
```

is seen by Python as

```
(n + 10) if b else -10
```

while you meant for it to mean

```
n + (10 if b else -10)
```

In other words, and in not-so-rigorous terms, the `+` “pulled” the neighbouring `10` and it's the whole `n + 10` that is seen as the expression to evaluate if the condition evaluates to `Truthy`.

Conditional expressions that evaluate to Booleans

Before showing good usage examples of conditional expressions, let me just go ahead and show you something you should *avoid* when using conditional expressions

Conditional expressions are suboptimal when they evaluate to Boolean values.

Here is a silly example:

```
def is_huge(n):
    return True if n > pow(10, 10) else False
```

Can you see what is wrong with this implementation of `is_huge`?

This function might look really good, because it is short and readable, and its behaviour is clear:

```
>>> is_huge(3.1415)
False
>>> is_huge(999)
False
>>> is_huge(73_324_634_325_242)
True
```

However... The conditional expression isn't doing anything relevant! The conditional expression just evaluates to the same value as the condition itself!

Take a close look at the function. If `n > pow(10, 10)` evaluates to `True`, then we return `True`. If `n > pow(10, 10)` evaluates to `False`, then we return `False`.

Here is a short table summarising this information:

n > pow(10, 10) evaluates to...		We return...
True		True
False		False

So, if the value of `n > pow(10, 10)` is the same as the thing we return, why don't we *just* return `n > pow(10, 10)`? In fact, that's what we should do:

```
def is_huge(n):
    return n > pow(10, 10)
```

Take this with you: never use `if: ... else: ...` or conditional expressions to evaluate to/return Boolean values. Often, it suffices to work with the condition alone.

A related use case where conditional expressions shouldn't be used is when assigning default values to variables. Some of these default values can be assigned with **Boolean short-circuiting**, using the `or` operator.

Examples in code

Here are a couple of examples where conditional expressions shine.

You will notice that these examples aren't particularly complicated or require much context to understand the mechanics of what is happening.

That's because the rationale behind conditional expressions is simple: pick between two values.

The dictionary `.get` method

The `collections` has a `ChainMap` class. This can be used to chain several dictionaries together, as I've shown in a tweet in the past:

In #Python, you can use `collections.ChainMap` to create a larger mapping out of several other maps. Useful, for example, when you want to juxtapose user configurations with default configurations. Follow for more #tips about Python [#learnpython #learncode #100daysofcode](#) pic.twitter.com/ip9IInItYG

— Rodrigo  (@mathsppblog) June 4, 2021

What's interesting is that `ChainMap` also defines a `.get` method, much like a dictionary. The `.get` method tries to retrieve a key and returns a default value if it finds it:

```
>>> from collections import ChainMap
>>> user_config = {"name": "mathspp"}
>>> default_config = {"name": "<noname>", "fullscreen": True}
## Access a key directly:
>>> config["fullscreen"]
True
## config["darkmode"] would've failed with a KeyError.
>>> config.get("darkmode", False)
False
```

Here is the full implementation of the `.get` method:

```
## From Lib/collections/__init__.py in Python 3.9.2

class ChainMap(_collections_abc.MutableMapping):
    # ...

    def get(self, key, default=None):
        return self[key] if key in self else default
```

Simple! Return the value associated with the key if key is in the dictionary, otherwise return the default value! Just that.

Resolving paths

The module `pathlib` is great when you need to deal with paths. One of the functionalities provided is the `.resolve` method, that takes a path and makes it absolute, getting rid of symlinks along the way:

```
## Running this from C:/Users/rodri:
>>> Path("../").resolve()
WindowsPath('C:/Users')
## The current working directory is irrelevant here:
>>> Path("C:/Users").resolve()
WindowsPath('C:/Users')
```

Here is part of the code that resolves paths:

```
## In Lib/pathlib.py from Python 3.9.2

class _PosixFlavour(_Flavour):
    # ...

    def resolve(self, path, strict=False):
        # ...

        base = '' if path.is_absolute() else os.getcwd()
        return _resolve(base, str(path)) or sep
```

As you can see, before calling the auxiliary function `_resolve` and returning, the function figures out if there is a need to add a base to the path.

If the path I enter is relative, like the `..` path above, then the base is set to be the current working directory (`os.getcwd()`). If the path is absolute, then there is no need for a base, because it is already there.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Conditional expressions excel at evaluating to one of two distinct values, depending on the value of a condition.”

This Pydon't showed you that:

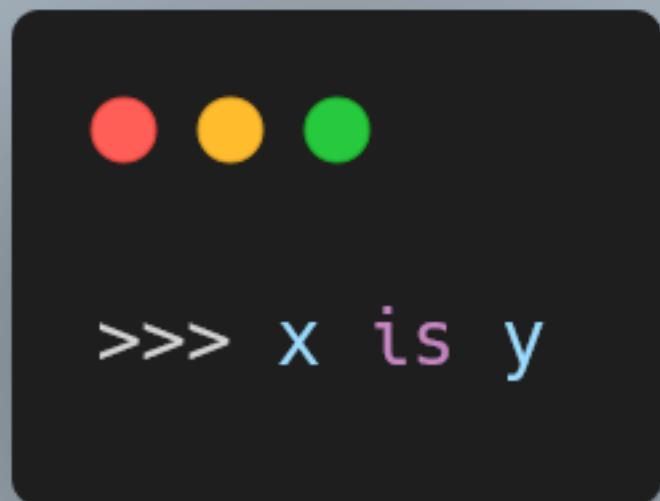
- conditional expressions are composed of three sub-expressions interleaved with the `if` and `else` keywords;
- conditional expressions were created with the intent of providing a convenient way of choosing between two values depending on a condition;
- conditional expressions can be easily read out as English statements;
- conditional expressions have the lowest precedence of all Python expressions;
- short-circuiting ensures conditional expressions only evaluate one of the two “value expressions”;
- conditional expressions can be chained together to emulate `if: ... elif: ... else: ...` blocks;
- it is impossible to emulate a conditional expression with a function; and
- if your conditional expression evaluates to a Boolean, then you should only be working with the condition.

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 3 Docs, The Python Language Reference, Conditional Expressions, <https://docs.python.org/3/reference/expressions.html#conditional-expressions> [last accessed 28-09-2021];
- PEP 308 – Conditional Expressions, <https://www.python.org/dev/peps/pep-0308/> [last accessed 28-09-2021];
- Python 3 Docs, The Python Standard Library, `collections.ChainMap`, <https://docs.python.org/3/library/collections.html#chainmap-objects> [last accessed 28-09-2021];
- Python 3 Docs, The Python Standard Library, `pathlib.Path.resolve`, <https://docs.python.org/3/library/pathlib.html#pathlib.Path.resolve> [last accessed 28-09-2021];
- “Does Python have a ternary conditional operator?”, Stack Overflow question and answers, <https://stackoverflow.com/questions/394809/does-python-have-a-ternary-conditional-operator> [last accessed 28-09-2021];
- “Conditional Expressions in Python”, note.nkmk.me, <https://note.nkmk.me/en/python-if-conditional-expressions/> [last accessed 28-09-2021];
- “Conditional Statements in Python, Conditional Expressions (Python’s Ternary Operator)”, Real Python, <https://realpython.com/python-conditional-statements/#conditional-expressions-pythons-ternary-operator> [last accessed 28-09-2021];

Pass-by-value, reference, and assignment



assignment.)

Introduction

Many traditional programming languages employ either one of two models when passing arguments to functions:

- some languages use the pass-by-value model; and
- most of the others use the pass-by-reference model.

Having said that, it is important to know the model that Python uses, because that influences the way your code behaves.

In this Pydon't, you will:

- see that Python doesn't use the pass-by-value nor the pass-by-reference models;
- understand that Python uses a pass-by-assignment model;
- learn about the built-in function `id`;
- create a better understanding for the Python object model;
- realise that every object has 3 very important properties that define it;
- understand the difference between mutable and immutable objects;
- learn the difference between shallow and deep copies; and
- learn how to use the module `copy` to do both types of object copies.

Is Python pass-by-value?

In the pass-by-value model, when you call a function with a set of arguments, the data is copied into the function. This means that you can modify the arguments however you please and that you won't be able to alter the state of the program outside the function. This is not what Python does, Python does not use the pass-by-value model.

Looking at the snippet of code that follows, it might look like Python uses pass-by-value:

```
def foo(x):  
    x = 4  
  
a = 3  
foo(a)  
print(a)  
## 3
```

This looks like the pass-by-value model because we gave it a 3, changed it to a 4, and the change wasn't reflected on the outside (a is still 3).

But, in fact, Python is not *copying* the data into the function.

To prove this, I'll show you a different function:

```

def clearly_not_pass_by_value(my_list):
    my_list[0] = 42

l = [1, 2, 3]
clearly_not_pass_by_value(l)
print(l)
## [42, 2, 3]

```

As we can see, the list `l`, that was defined outside of the function, changed after calling the function `clearly_not_pass_by_value`. Hence, Python does not use a pass-by-value model.

Is Python pass-by-reference?

In a true pass-by-reference model, the called function gets access to the variables of the callee! Sometimes, it can *look* like that's what Python does, but Python does not use the pass-by-reference model.

I'll do my best to explain why that's not what Python does:

```

def not_pass_by_reference(my_list):
    my_list = [42, 73, 0]

l = [1, 2, 3]
not_pass_by_reference(l)
print(l)
## [1, 2, 3]

```

If Python used a pass-by-reference model, the function would've managed to completely change the value of `l` outside the function, but that's not what happened, as we can see.

Let me show you an actual pass-by-reference situation.

Here's some Pascal code:

```

program callByReference;
var
    x: integer;

procedure foo(var a: integer);
{ create a procedure called `foo` }
begin
    a := 6           { assign 6 to `a` }
end;

begin
    x := 2;          { assign 2 to `x` }
    writeln(x);     { print `x` }
    foo(x);         { call `foo` with `x` }
    writeln(x);     { print `x` }
end.

```

Look at the last lines of that code:

- we assign 2 to `x` with `x := 2`;
- we print `x`;
- we call `foo` with `x` as argument; and
- we print `x` again.

What's the output of this program?

I imagine that most of you won't have a Pascal interpreter lying around, so you can just go to [tio.run](#) and [run this code online](#)

If you run this, you'll see that the output is

```
2
6
```

which can be rather surprising, if the majority of your programming experience is in Python!

The procedure `foo` effectively received the variable `x` and changed the value that it contained. After `foo` was done, the variable `x` (that lives outside `foo`) had a different value. You can't do anything like this in Python.

Python object model

To really understand the way Python behaves when calling functions, it's best if we first understand what Python objects are, and how to characterise them.

The three characteristics of objects

In Python, *everything* is an object, and each object is characterised by three things:

- its identity (an integer that uniquely identifies the object, much like social security numbers identify people);
- a type (that identifies the operations you can do with your object); and
- the object's content.

Here is an object and its three characteristics:

```
>>> id(obj)
2698212637504      # the identity of `obj`
>>> type(obj)
<class 'list'>      # the type of `obj`
>>> obj
[1, 2, 3]            # the contents of `obj`
```

As we can see above, `id` is the built-in function you use to query the identity of an object, and `type` is the built-in function you use to query the type of an object.

(Im)mutability

The (im)mutability of an object depends on its type. In other words, (im)mutability is a characteristic of types, not of specific objects!

But what *exactly* does it mean for an object to be mutable? Or for an object to be immutable?

Recall that an object is characterised by its identity, its type, and its contents. A type is mutable if you can change the contents of its objects without changing its identity and its type.

Lists are a great example of a mutable data type. Why? Because lists are *containers*: you can put things inside lists and you can remove stuff from inside those same lists.

Below, you can see how the contents of the list `obj` change as we make method calls, but the identity of the list remains the same:

```
>>> obj = []
>>> id(obj)
2287844221184
>>> obj.append(0); obj.extend([1, 2, 3]); obj
[42, 0, 1, 2, 3]
>>> id(obj)
2287844221184
>>> obj.pop(0); obj.pop(0); obj.pop(); obj
42
0
3
[1, 2]
>>> id(obj)
2287844221184
```

However, when dealing with immutable objects, it's a completely different story. If we check an English dictionary, this is what we get for the definition of "immutable":

adjective: immutable – unchanging over time or unable to be changed.

Immutable objects' contents never change. Take a string as an example:

```
>>> obj = "Hello, world!"
```

Strings are a good example for this discussion because, sometimes, they can *look* mutable. But they are not!

A very good indicator that an object is immutable is when all its methods return something. This is unlike list's `.append` method, for example! If you use `.append` on a list, you get no return value. On the other hand, whatever method you use on a string, the result is returned to you:

```
>>> [].append(0)      # No return.
>>> obj.upper()      # A string is returned.
'HELLO, WORLD!'
```

Notice how `obj` wasn't updated automatically to "HELLO, WORLD!". Instead, the new string was created and returned to you.

Another great hint at the fact that strings are immutable is that you cannot assign to its indices:

```
>>> obj[0]
'H'
>>> obj[0] = "h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

This shows that, when a string is created, it remains the same. It can be used to build *other* strings, but the string itself always stays unchanged.

As a reference, `int`, `float`, `bool`, `str`, `tuple`, and `complex` are the most common types of immutable objects; `list`, `set`, and `dict` are the most common types of mutable objects.

Variable names as labels

Another important thing to understand is that a variable name has very little to do with the object itself.

In fact, the name `obj` was just a label that I decided to attach to the object that has identity 2698212637504, has the list type, and contents 1, 2, 3.

Just like I attached the label `obj` to that object, I can attach many more names to it:

```
>>> foo = bar = baz = obj
```

Again, these names are just labels. Labels that I decided to stick to the *same* object. How can we know it's the same object? Well, all their "social security numbers" (the ids) match, so they must be the same object:

```
>>> id(foo)
2698212637504
>>> id(bar)
2698212637504
>>> id(baz)
2698212637504
>>> id(obj)
2698212637504
```

Therefore, we conclude that `foo`, `bar`, `baz`, and `obj`, are variable names that all refer to the same object.

The operator `is`

This is exactly what the operator `is` does: it checks if the two objects are the *same*.

For two objects to be the same, they must have the same identity:

```
>>> foo is obj
True
>>> bar is foo
True
```

```
>>> obj is foo
True
```

It is *not* enough to have the same type and contents! We can create a new list with contents [1, 2, 3] and that will *not* be the same *object* as obj:

```
>>> obj is [1, 2, 3]
False
```

Think of it in terms of perfect twins. When two siblings are perfect twins, they look identical. However, they are different people!

is not

Just as a side note, but an important one, you should be aware of the operator `is not`.

Generally speaking, when you want to negate a condition, you put a `not` in front of it:

```
n = 5
if not isinstance(n, str):
    print("n is not a string.")
## n is not a string.
```

So, if you wanted to check if two variables point to different objects, you could be tempted to write

```
if not a is b:
    print(`a` and `b` are different objects.)
```

However, Python has the operator `is not`, which is much more similar to a proper English sentence, which I think is really cool!

Therefore, the example above should actually be written

```
if a is not b:
    print(`a` and `b` are different objects.)
```

Python does a similar thing for the `in` operator, providing a `not in` operator as well... How cool is that?!

Assignment as nicknaming

If we keep pushing this metaphor forward, assigning variables is just like giving a new nickname to someone.

My friends from middle school call me “Rojer”. My friends from college call me “Girão”. People I am not close to call me by my first name – “Rodrigo”. However, regardless of what they call me, *I* am still *me*, right?

If one day I decide to change my haircut, everyone will see the new haircut, regardless of what they call me!

In a similar fashion, if I modify the *contents* of an object, I can use whatever nickname I prefer to see that those changes happened. For example, we can change the middle element of the list we have been playing around with:

```
>>> foo[1] = 42
>>> bar
```

```
[1, 42, 3]
>>> baz
[1, 42, 3]
>>> obj
[1, 42, 3]
```

We used the nickname `foo` to modify the middle element, but that change is visible from all other nicknames as well.

Why?

Because they all pointed at the *same* list object.

Python is pass-by-assignment

Having laid out all of this, we are now ready to understand how Python passes arguments to functions.

When we call a function, each of the parameters of the function is assigned to the object they were passed in. In essence, each parameter now becomes a *new* nickname to the objects that were given in.

Immutable arguments

If we pass in immutable arguments, then we have *no* way of modifying the arguments themselves. After all, that's what immutable means: "doesn't change".

That is why it can look like Python uses the pass-by-value model. Because the only way in which we can have the parameter hold something else is by assigning it to a completely different thing. When we do that, we are reusing the same nickname for a different object:

```
def foo(bar):
    bar = 3
    return bar

foo(5)
```

In the example above, when we call `foo` with the argument 5, it's as if we were doing `bar = 5` at the beginning of the function.

Immediately after that, we have `bar = 3`. This means "take the nickname"bar" and point it at the integer 3". Python doesn't care that `bar`, as a nickname (as a variable name) had already been used. It is now pointing at that 3!

Mutable arguments

On the other hand, mutable arguments *can* be changed. We can modify their internal contents. A prime example of a mutable object is a list: its elements can change (and so can its length).

That is why it can look like Python uses a pass-by-reference model. However, when we change the *contents* of an object, we didn't change the identity of the object itself. Similarly, when you change your haircut or your clothes, your social security number does *not* change:

```
>>> l = [42, 73, 0]
>>> id(l)
3098903876352
>>> l[0] = -1
>>> l.append(37)
>>> id(l)
3098903876352
```

Do you understand what I'm trying to say? If not, drop a comment below and I'll try to help.

Beware when calling functions

This goes to show you should be careful when defining your functions. If your function expects mutable arguments, you should do one of the two:

- do not mutate the argument in any way whatsoever; or
- document explicitly that the argument may be mutated.

Personally, I prefer to go with the first approach: to not mutate the argument; but there are times and places for the second approach.

Sometimes, you do need to take the argument as the basis for some kind of transformation, which would mean you would want to mutate the argument. In those cases, you might think about doing a copy of the argument (discussed in the next section), but making that copy can be resource intensive. In those cases, mutating the argument might be the only sensible choice.

Making copies

Shallow vs deep copies

“Copying an object” means creating a second object that has a different identity (therefore, is a *different* object) but that has the same contents. Generally speaking, we copy one object so that we can work with it and mutate it, while also preserving the first object.

When copying objects, there are a couple of nuances that should be discussed.

Copying immutable objects

The first thing that needs to be said is that, for immutable objects, it does not make sense to talk about copies.

“Copies” only make sense for mutable objects. If your object is immutable, and if you want to preserve a reference to it, you can just do a second assignment and work on it:

```
string = "Hello, world!"
string_ = string
## Do stuff with `string_` now...
```

Or, sometimes, you can just call methods and other functions directly on the original, because the original is not going anywhere:

```
string = "Hello, world!"  
print(string.lower())  
## After calling `lower`, `string` is still "Hello, world!"
```

So, we only need to worry about mutable objects.

Shallow copy

Many mutable objects can contain, themselves, mutable objects. Because of that, two types of copies exist:

- shallow copies; and
- deep copies.

The difference lies in what happens to the mutable objects inside the mutable objects.

Lists and dictionaries have a method `.copy` that returns a shallow copy of the corresponding object.

Let's look at an example with a list:

```
>>> sublist = []  
>>> outer_list = [42, 73, sublist]  
>>> copy_list = outer_list.copy()
```

First, we create a list inside a list, and we copy the outer list. Now, because it is a *copy*, the copied list isn't the same object as the original outer list:

```
>>> copy_list is outer_list  
False
```

But if they are not the same object, then we can modify the contents of one of the lists, and the other won't reflect the change:

```
>>> copy_list[0] = 0  
>>> outer_list  
[42, 73, []]
```

That's what we saw: we changed the first element of the `copy_list`, and the `outer_list` remained unchanged.

Now, we try to modify the contents of `sublist`, and that's when the fun begins!

```
>>> sublist.append(999)  
>>> copy_list  
[0, 73, [999]]  
>>> outer_list  
[42, 73, [999]]
```

When we modify the contents of `sublist`, both the `outer_list` and the `copy_list` reflect those changes...

But wasn't the `copy` supposed to give me a second list that I could change without affecting the first one? Yes! And that is what happened!

In fact, modifying the contents of `sublist` doesn't *really* modify the contents of neither `copy_list` nor `outer_list`: after all, the third element of both was pointing at a list object, and it still is! It's the (inner) contents of the object to which we are pointing that changed.

Sometimes, we don't want this to happen: sometimes, we don't want mutable objects to share inner mutable objects.

Common shallow copy techniques

When working with lists, it is common to use slicing to produce a shallow copy of a list:

```
>>> outer_list = [42, 73, []]
>>> shallow_copy = outer_list[::]
>>> outer_list[2].append(999)
>>> shallow_copy
[42, 73, [999]]
```

Using the built-in function for the respective type, on the object itself, also builds shallow copies. This works for lists and dictionaries, and is likely to work for other mutable types.

Here is an example with a list inside a list:

```
>>> outer_list = [42, 73, []]
>>> shallow_copy = list(outer_list)
>>> shallow_copy[2].append(999)
>>> outer_list
[42, 73, [999]]
```

And here is an example with a list inside a dictionary:

```
>>> outer_dict = {42: 73, "list": []}
>>> shallow_copy = dict(outer_dict)
>>> outer_dict["list"].append(999)
>>> shallow_copy
{42: 73, 'list': [999]}
```

Deep copy

When you want to copy an object "thoroughly", and you don't want the copy to share references to inner objects, you need to do a "deep copy" of your object. You can think of a deep copy as a recursive algorithm.

You copy the elements of the first level and, whenever you find a mutable element on the first level, you recurse down and copy the contents of those elements.

To show this idea, here is a simple recursive implementation of a deep copy for lists that contain other lists:

```
def list_deepcopy(l):
    return [
        elem if not isinstance(elem, list) else list_deepcopy(elem)
        for elem in l
    ]
```

We can use this function to copy the previous `outer_list` and see what happens:

```
>>> sublist = []
>>> outer_list = [42, 73, sublist]
>>> copy_list = list_deepcopy(outer_list)
>>> sublist.append(73)
>>> copy_list
[42, 73, []]
>>> outer_list
[42, 73, [73]]
```

As you can see here, modifying the contents of `sublist` only affected `outer_list` indirectly; it didn't affect `copy_list`.

Sadly, the `list_deepcopy` method I implemented isn't very robust, nor versatile, but the Python Standard Library has got us covered!

The module `copy` and the method `deepcopy`

The module `copy` is exactly what we need. The module provides two useful functions:

- `copy.copy` for shallow copies; and
- `copy.deepcopy` for deep copies.

And that's it! And, what is more, the method `copy.deepcopy` is smart enough to handle issues that might arise with circular definitions, for example! That is, when an object contains another that contains the first one: a naïve recursive implementation of a deep copy algorithm would enter an infinite loop!

If you write your own custom objects and you want to specify how shallow and deep copies of those should be made, you only need to implement `__copy__` and `__deepcopy__`, respectively!

It's a great module, in my opinion.

Examples in code

Now that we have gone deep into the theory – pun intended –, it is time to show you some actual code that plays with these concepts.

Mutable default arguments

Let's start with a Twitter favourite:

Python is an incredible language but sometimes appears to have quirks. For example, one thing that often confuses beginners is why you shouldn't use lists as default values. Here is a thread that will help you understand this. pic.twitter.com/HVhPjS2PSH

— Rodrigo (@mathsppblog) October 5, 2021

Apparently, it's a bad idea to use mutable objects as default arguments. Here is a snippet showing you why:

```
def my_append(elem, l=[]):
    l.append(elem)
    return l
```

The function above appends an element to a list and, if no list is given, appends it to an empty list by default.

Great, let's put this function to good use:

```
>>> my_append(1)
[1]
>>> my_append(1, [42, 73])
[42, 73, 1]
>>> my_append(3)
[1, 3]
```

We use it once with 1, and we get a list with the 1 inside. Then, we use it to append a 1 to another list we had. And finally, we use it to append a 3 to an empty list... Except that's not what happens!

As it turns out, when we define a function, the default arguments are created and stored in a special place:

```
>>> my_append.__defaults__
([1, 3],)
```

What this means is that the default argument is *always* the same object. Therefore, because it is a mutable object, its contents can change over time. That is why, in the code above, `__defaults__` shows a list with two items already.

If we redefine the function, then its `__defaults__` shows an empty list:

```
>>> def my_append(elem, l=[]):
...     l.append(elem)
...     return l
...
>>> my_append.__defaults__
([],)
```

This is why, in general, mutable objects shouldn't be used as default arguments.

The standard practice, in these cases, is to use `None` and then [use Boolean short-circuiting](#) to assign the default value:

```
def my_append(elem, l=None):
    lst = l or []
    lst.append(elem)
    return lst
```

With this implementation, the function now works as expected:

```
>>> my_append(1)
[1]
>>> my_append(3)
[3]
```

```
>>> my_append(3, [42, 73])
[42, 73, 3]
```

`is not None`

Searching through the Python Standard Library shows that the `is not` operator is used a bit over 5,000 times. That's a lot.

And, by far and large, that operator is almost always followed by `None`. In fact, `is not None` appears 3169 times in the standard library!

`x is not None` does exactly what it's written: it checks if `x` is `None` or not.

Here is a simple example usage of that, from the `argparse` module to create command line interfaces:

```
## From Lib/argparse.py from Python 3.9
class HelpFormatter(object):
    # ...

    class _Section(object):
        # ...

        def format_help(self):
            # format the indented section
            if self.parent is not None:
                self.formatter._indent()
            # ...
```

Even without a great deal of context, we can see what is happening: when displaying command help for a given section, we may want to indent it (or not) to show hierarchical dependencies.

If a section's parent is `None`, then that section has no parent, and there is no need to indent. In other words, if a section's parent is `not None`, then we want to indent it. Notice how my English matches the code exactly!

Deep copy of the system environment

The method `copy.deepcopy` is used a couple of times in the standard library, and here I'd like to show an example usage where a dictionary is copied.

The module `os` provides the attribute `environ`, similar to a dictionary, that contains the environment variables that are defined.

Here are a couple of examples from my (Windows) machine:

```
>>> os.environ["lang"]
'en_US.UTF-8'
>>> os.environ["appdata"]
'C:\\\\Users\\\\rodri\\\\AppData\\\\Roaming'
>>> os.environ["systemdrive"]
```

```
'C: '
## Use list(os.environ.keys()) for a list of your environment variables.
```

The module `http.server` provides some classes for basic HTTP servers.

One of those classes, `CGIHTTPRequestHandler`, implements a HTTP server that can also run CGI scripts and, in its `run_cgi` method, it needs to set a bunch of environment variables.

These environment variables are set to give the necessary context for the CGI script that is going to be ran. However, we don't want to actually modify the current environment!

So, what we do is create a deep copy of the environment, and then we modify it to our heart's content! After we are done, we tell Python to execute the CGI script, and we provide the altered environment as an argument.

The exact way in which this is done may not be trivial to understand. I, for one, don't think I could explain it to you. But that doesn't mean we can't infer parts of it:

Here is the code:

```
## From Lib/http/server.py in Python 3.9
class CGIHTTPRequestHandler(SimpleHTTPRequestHandler):
    # ...

    def run_cgi(self):
        # ...
        env = copy.deepcopy(os.environ)
        env['SERVER_SOFTWARE'] = self.version_string()
        env['SERVER_NAME'] = self.server.server_name
        env['GATEWAY_INTERFACE'] = 'CGI/1.1'
        # and many more `env` assignments!

        # ...

    else:
        # Non-Unix -- use subprocess
        # ...
        p = subprocess.Popen(cmdline,
                            stdin=subprocess.PIPE,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            env = env
                            )
```

As we can see, we copied the environment and defined some variables. Finally, we created a new subprocess that gets the modified environment.

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

“Python uses a pass-by-assignment model, and understanding it requires you to realise all objects are characterised by an identity number, their type, and their contents.”

This Pydon't showed you that:

- Python doesn't use the pass-by-value model, nor the pass-by-reference one;
- Python uses a pass-by-assignment model (using “nicknames”);
- each object is characterised by
 - its identity;
 - its type; and
 - its contents.
- the `id` function is used to query an object's identifier;
- the `type` function is used to query an object's type;
- the type of an object determines whether it is mutable or immutable;
- shallow copies copy the reference of nested mutable objects;
- deep copies perform copies that allow one object, and its inner elements, to be changed without ever affecting the copy;
- `copy.copy` and `copy.deepcopy` can be used to perform shallow/deep copies; and
- you can implement `__copy__` and `__deepcopy__` if you want your own objects to be copyable.

See also

If you prefer video content, you can check this [YouTube video](#), which was inspired by this Pydon't.

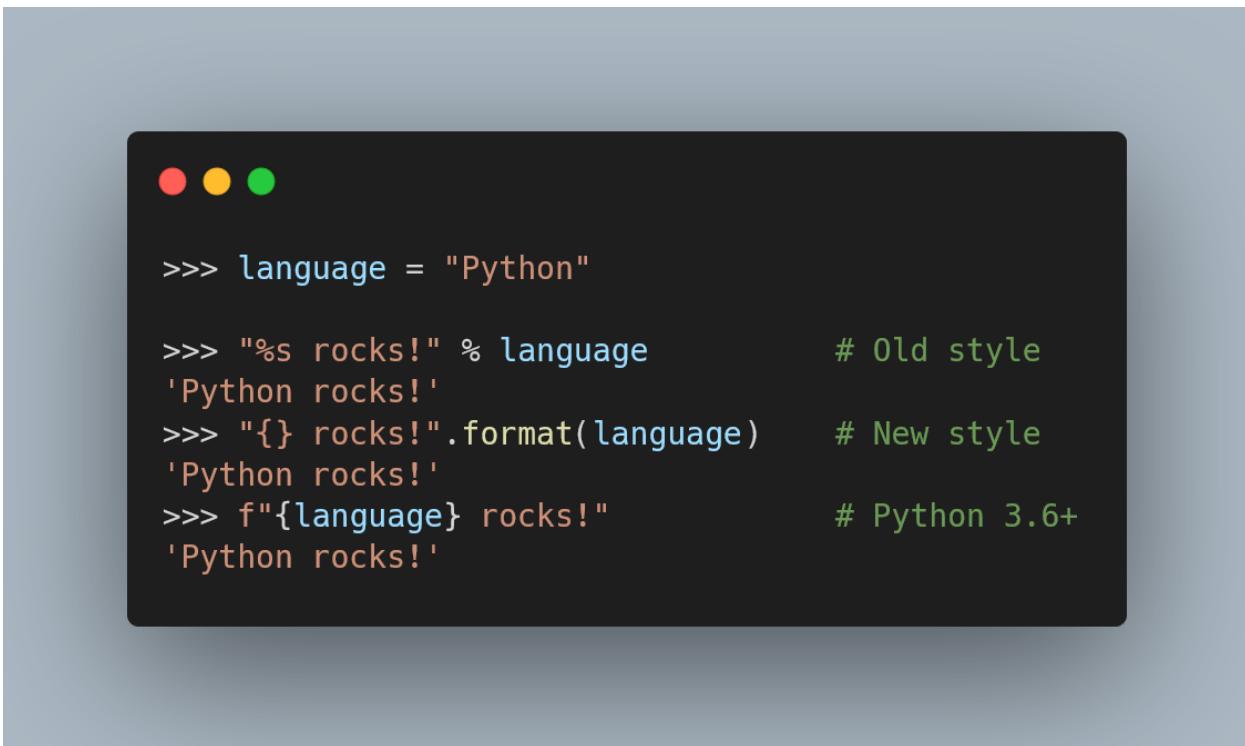
If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 3 Docs, Programming FAQ, “How do I write a function with output parameters (call by reference)?”, <https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference> [last accessed 04-10-2021];
- Python 3 Docs, The Python Standard Library, `copy`, <https://docs.python.org/3/library/copy.html> [last accessed 05-10-2021];
- effbot.org, “Call by Object” (via “arquivo.pt”), <https://arquivo.pt/wayback/20160516131553/http://effbot.org/zone/call-by-object.htm> [last accessed 04-10-2021];
- effbot.org, “Python Objects” (via “arquivo.pt”), <https://arquivo.pt/wayback/20191115002033/http://effbot.org/zone/python-objects.htm> [last accessed 04-10-2021];
- Robert Heaton, “Is Python pass-by-reference or pass-by-value”, <https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/> [last accessed 04-10-2021];
- StackOverflow question and answers, “How do I pass a variable by reference?”, <https://stackoverflow.com/q/986006/2828287> [last accessed 04-10-2021];

- StackOverflow question and answers, “Passing values in Python [duplicate]”, <https://stackoverflow.com/q/534375/2828287> [last accessed 04-10-2021];
- Twitter thread by [@mathsppblog](https://twitter.com/mathspblog), <https://twitter.com/mathspblog/status/1445148566721335298> [last accessed 20-10-2021];

String formatting comparison



(Thumbnail of the original article at <https://mathspp.com/blog/pydonts/string-formatting-comparison>.)

Introduction

The [Zen of Python](#) says that

“There should be one – and preferably only one – obvious way to do it.”

And yet, there are three main ways of doing string formatting in Python. This Pydon’t will settle the score, comparing these three methods and helping you decide which one is the obvious one to use in each situation.

In this Pydon't, you will:

- learn about the old C-style formatting with %;
- learn about the string method .format;
- learn about the Python 3.6+ feature of literal string interpolation and f-strings;
- understand the key differences between each type of string formatting; and
- see where each type of string formatting really shines.

String formatting rationale

Let's pretend, for a second, that Python had zero ways of doing string formatting.

Now, I have a task for you: write a function that accepts a programming language name and returns a string saying that said programming language rocks. Can you do it? Again, *without* any string formatting whatsoever!

Here is a possible solution:

```
def language_rocks(language):
    return language + " rocks!"

## ---
>>> language_rocks("Python")
'Python rocks!'
```

Great job!

Now, write a function that accepts a programming language name and its (estimated) number of users, and returns a string saying something along the lines of "<insert language> rocks! Did you know that <insert language> has around <insert number> users?".

Can you do it? Recall that you are *not* supposed to use any string formatting facilities, whatsoever!

Here is a possible solution:

```
def language_info(language, users_estimate):
    return (
        language + " rocks! Did you know that " + language +
        " has around " + str(users_estimate) + " users?!"
    )

## ---
>>> language_info("Python", 10)
'Python rocks! Did you know that Python has around 10 users?!"
```

Notice how that escalated quite quickly: the purpose of our function is still very simple, and yet we have a bunch of string concatenations happening all over the place, just because we have some pieces of information that we want to merge into the string.

This is what string formatting is for: it's meant to make your life easier when you need to put information inside strings.

Three string formatting methods

Now that we've established that string formatting is useful, let's take a look at the three main ways of doing string formatting in Python.

First, here is how you would refactor the function above:

```
## Using C-style string formatting:
def language_info_cstyle(language, users_estimate):
    return (
        "%s rocks! Did you know that %s has around %d users?!" %
        (language, language, users_estimate)
    )

## Using the Python 3 `format` method from strings:
def language_info_format(language, users_estimate):
    return "{} rocks! Did you know that {} has around {} users?!" .format(
        language, language, users_estimate
    )

## Using f-strings, from Python 3.6+:
def language_info_fstring(language, users_estimate):
    return (
        f"{language} rocks! Did you know that {language}" +
        f" has around {users_estimate} users?!"
    )
```

All three functions above behave in the same way.

1. `language_info_cstyle` uses old-style string formatting, borrowed from the similar C syntax that does the same thing;
2. `language_info_format` uses the string method `.format`, that was introduced in [PEP 3101](#); and
3. `language_info_fstring` uses the new f-strings, which were introduced in [PEP 498](#) for Python 3.6+.

C-style formatting

The C-style formatting, which is the one that has been around the longer, is characterised by a series of percent signs ("%) that show up in the template strings.

(By “template strings”, I mean the strings in which we want to fill in the gaps.)

These percent signs indicate the places where the bits of information should go, and the character that comes next (above, we've seen "%s" and "%d") determine how the information being passed in is treated.

Additionally, the way in which you apply the formatting is through the binary operator `%`: on the left you put the template string, and on the right you put all the pieces of information you need to pass in.

String method .format

The string method `.format` is, like the name suggests, a *method of the string type*. This means that you typically have a format string and, when you get access to the missing pieces of information, you just call the `.format` method on that string.

Strings that use the method `.format` for formatting are typically characterised by the occurrence of a series of curly braces "`{}`" within the string. It is also common to find that the method `.format` is called where/when the string literal is defined.

Literal string interpolation, or f-strings

Literal string interpolation is the process through which you interpolate values into strings. Notice the definition of the word "interpolate":

verb: interpolate – insert (something of a different nature) into something else.

That's exactly what this technique does: it directly inserts the additional values into the template string.

When people talk about "f-strings" they are also talking about this technique. That's because you need to prepend an `f` to your string to use literal string interpolation.

Literal string interpolation is (clearly) characterised by the `f` prefix on the string literals, and also the curly braces "`{}`" inside the string. Unlike with the string method `.format`, the braces *always* have something inside them.

In case you are wondering, using a letter as a prefix to a string literal isn't an idea introduced with literal string interpolation. Two common examples include `r` (raw) strings, and `b` (binary) strings:

```
>>> b"This is a bytes object!"  
b'This is a bytes object!'  
>>> type(_)           # Use _ to refer to the previous string.  
<class 'bytes'>  
>>> r"This is a \nstring"  
'This is a \\nstring'
```

Now that we have taken a look at the three string formatting methods, we will show a series of different (simple) scenarios and how formatting would work with the three options.

As we will see, the C-style formatting will almost always look clunkier and less elegant, which should help you realise that f-strings and the string method `.format` are the way to go.

After this series of comparisons, we will give some suggestions as to what type of formatting to use, and when.

Value conversion

When we do string formatting, the objects that we want to format into the template string need to be converted to a string.

This is typically done by calling `str` on the objects, which in turn calls the dunder method `__str__` of those objects. However, sometimes it is beneficial to have the object be represented with the result from calling

`repr`, and not `str`. (I wrote about why you would want this before, so [read this Pydon't](#) if you are not familiar with how `__str__`/`__repr__` works.)

There are special ways to determine which type of string conversion happens.

Take this dummy class:

```
class Data:
    def __str__(self):
        return "str"
    def __repr__(self):
        return "repr"
```

With that class defined, the three following strings are the same:

```
"%s %r" % (Data(), Data())
"{!s} {!r}".format(Data(), Data())
f"[Data()!s] [{Data()!r}]"
## Result is 'str repr'
```

With C-style formatting we use "%s" and "%r" to distinguish from the regular string version of the object or its representation. The two more modern methods do the distinction with the `!s` and `!r` flags.

Alignment

When we need to format many values across many lines, for example to display a table-like piece of output, we may want to align all values and pad them accordingly. This is one of the great use cases where string formatting shines.

```
lang = "Python"
"%-10s" % lang
":<10".format(lang)
f"[lang:<10]"
## Result is 'Python      '
```

The C-style aligns on the right, by default, whereas `.format` and f-strings align on the left. Hence, above we could have written

":>10".format(lang)
f"[lang:>10]"

and we would still get the same result. However, for the sake of comparison, I decided to include the `<` for left alignment.

C-style formatting can't do it, but the two modern methods can use `^` to align the output in the centre:

```
"{:^10}".format(lang)
f"[lang:^10]"

## Result is ' Python '
```

To right align, use `>` for the modern methods, or use nothing at all for the C-style formatting.

Remember, the modern methods use `<^>` for alignment, and the tip of the arrow points to the alignment direction:

Named placeholders

For longer strings, or strings with many slots to be filled in, it may be helpful to include placeholder strings, instead of just the symbol to denote string formatting. With f-strings, this happens more or less automatically, but C-style formatting and `.format` also support that:

```
name, age = "John", 73

"%(name)s is %(age)d years old." % {"name": name, "age": age}

"{name} is {age} years old.".format(name=name, age=age)

f"{name} is {age} years old."

## Result is 'John is 73 years old.'
```

Accessing nested data structures

Let's look at the example above again, but let's imagine that the name and age were actually stored in a dictionary.

In this case, the old-style formatting and the string method `.format` are particularly handy:

```
data = {"name": "John", "age": 73}

"%(name)s is %(age)d years old." % data

"{data[name]} is {data[age]} years old.".format(data=data)
## or
"{name} is {age} years old.".format(**data)

f"[data['name']] is [data['age']] years old."

## Result is 'John is 73 years old.'
```

The first usage of the string method `.format` shows an interesting feature that formatting with `.format` allows: the formatted objects can be indexed and they can also have their attributes accessed.

Here is a very convoluted example:

```
class ConvolvedExample:
    values = [{"name": "Charles"}, {"name": "William"}]

ce = ConvolvedExample()

"Name is: {ce.values[0][name]}".format(ce=ce)

f"Name is: {ce.values[0]['name']}"

## Result is 'Name is: Charles'
```

Parametrised formatting

Sometimes, you want to do some string formatting, but the exact formatting you do is dynamic: for example, you might want to print something with variable width, and you'd like for the width to adapt to the longest element in a sequence.

For example, say you have a list of companies and their countries of origin, and you want that to be aligned:

```
data = [("Toyota", "Japanese"), ("Ford", "USA")]

for brand, country in data:
    print(f"{brand:>7}, {country:>9}")

"""
Result is
Toyota, Japanese
Ford, USA
"""
```

The thing is, what if we now include a company with a longer name?

```
data = [("Toyota", "Japanese"), ("Ford", "USA"), ("Lamborghini", "Italy")]

for brand, country in data:
    print(f"{brand:>7}, {country:>9}")

"""
Result is
Toyota, Japanese
Ford, USA
Lamborghini, Italy
"""
```

The output is no longer aligned because the word “Lamborghini” does not fit within the specified width of 7. Therefore, we need to dynamically compute the maximum lengths and use them to create the correct format specification. This is where parametrising the format specification comes in handy:

```
data = [("Toyota", "Japanese"), ("Ford", "USA"), ("Lamborghini", "Italy")]
## Compute brand width and country width needed for formatting.
bw = 1 + max(len(brand) for brand, _ in data)
cw = 1 + max(len(country) for _, country in data)

for brand, country in data:
    print(f"{brand:>{bw}}, {country:>{cw}}")

"""
Result is
Toyota, Japanese
Ford, USA
Lamborghini, Italy
"""
```

Old style formatting only allows parametrisation of the width of the field and the precision used. For the string method `.format` and for f-strings, parametrisation can be used with all the format specifier options.

```
month = "November"
prec = 3
value = 2.7182

"%.*s = %.*f" % (prec, month, prec, value)

"{:.{prec}} = {:.{prec}f}".format(month, value, prec=prec)

f"{month:.{prec}} = {value:.{prec}f}"

## Result is 'Nov = 2.718'
```

Custom formatting

Finally, the string method `.format` and f-strings allow you to define how your own custom objects should be formatted, and that happens through the dunder method `__format__`.

The dunder method `__format__` accepts a string (the format specification) and it returns the corresponding string.

Here is a (silly) example:

```
class YN:
    def __format__(self, format_spec):
        return "N" if "n" in format_spec else "Y"
```

```
"{:aaabbbccc}".format(YN()) # Result is 'Y'  
f"[YN():nope]" # Result is 'N'
```

Of course, when possible, you would want to implement a format specification that matches the built-in `format` spec.

Examples in code

As the little snippets of code above have shown you, there is hardly any reason to be using the old string formatting style. Of course, remember that consistency is important, so it might still make sense if you are maintaining an old code base that uses old-style formatting everywhere.

Otherwise, you are better off using the string method `.format` and/or f-strings. Now, I will show you some usage patterns and I will help you figure out what type of string formatting works best in those cases.

Plain formatting

F-strings are very, very good. They are short to type, they have good locality properties (it is easy to see what is being used to format that specific portion of the string), and they are fast.

For all your plain formatting needs, prefer f-strings over the method `.format`:

```
## Some random variables:  
name, age, w = "John", 73, 10  
  
## Prefer...  
f"{name!s} {name!r}"  
f"{name:<10}"  
f"{name} is {age} years old."  
f"{name:~{w}}"  
  
## ... over `format`  
"{!s} {!r}".format(name, name)  
":<10)".format(name)  
"{name} is {age} years old.".format(name=name, age=age)  
":~{w}}".format(name, w=w)
```

Data in a dictionary

If all your formatting data is already in a dictionary, then using the string method `.format` might be the best way to go.

This is especially true if the keys of said dictionary are strings. When that is the case, using the string method `.format` almost looks like using f-strings! Except that, when the data is in a dictionary, using f-strings is much more verbose when compared to the usage of `**` in `.format`:

```

data = {"name": "John", "age": 73}

## This is nice:
"{} is {} years old.".format(**data)

## This is cumbersome:
f"{{data['name']}} is {{data['age']}} years old."

```

In the example above, we see that the `.format` example exhibits the usual locality that f-strings tend to benefit from!

Deferred formatting

If you need to create your formatting string first, and only format it later, then you cannot use f-strings.

When that is the case, using the method `.format` is probably the best way to go.

This type of scenario might arise, for example, from programs that run in (many) different languages:

```

def get_greeting(language):
    if language == "pt":
        return "Olá, {}!"
    else:
        return "Hello, {}!"

lang = input(" [en/pt] >> ")
name = input(" your name >> ")
get_greeting(lang).format(name)

```

Conclusion

Here's the main takeaway of this Pydon't, for you, on a silver platter:

"Don't use old-style string formatting: use f-strings whenever possible, and then `.format` in the other occasions."

This Pydon't showed you that:

- Python has three built-in types of string formatting;
- using `.format` and/or f-strings is preferred over %-formatting;
- you can use `!s` and `!r` to specify which type of string representation to use;
- alignment can be done with the `<^>` specifiers;
- format specifications can be parametrised with an extra level of `{}`;
- custom formatting can be implemented via the dunder method `__format__`;
- f-strings are very suitable for most standard formatting tasks;
- the method `.format` is useful when the formatting data is inside a dictionary; and
- for deferred string formatting, f-strings don't work, meaning `.format` is the recommended string formatting method.

If you liked this Pydon't be sure to leave a reaction below and share this with your friends and fellow Pythonistas. Also, [subscribe to the newsletter](#) so you don't miss a single Pydon't!

References

- Python 2 Docs, String Formatting Operations, <https://docs.python.org/2/library/stdtypes.html#string-formatting> [last accessed 10-11-2021]
- Python 3 Docs, The Python Tutorial, Fancier Output Formatting, <https://docs.python.org/3/tutorial/inputoutput.html> [last accessed 10-11-2021]
- Python 3 Docs, The Python Standard Library, string, <https://docs.python.org/3/library/string.html#string-formatting> [last accessed 10-11-2021]
- PEP 3101 – Advanced String Formatting, <https://www.python.org/dev/peps/pep-3101/> [last accessed 10-11-2021]
- PEP 498 – Literal String Interpolation, <https://www.python.org/dev/peps/pep-0498/> [last accessed 10-11-2021]
- PyFormat, <https://pyformat.info> [last accessed 17-11-2021]

Closing thoughts

I would like to thank you for investing your time improving your Python knowledge through this book and I invite you to let me know of your feedback.

All criticism that you might have, positive and negative, is welcome and will be read by me. Just drop me a line at rodrigo@mathspp.com or reach out to me on Twitter where I go by the name [mathsppblog](#).

I hope to talk to you soon!

— Rodrigo, <https://mathspp.com>