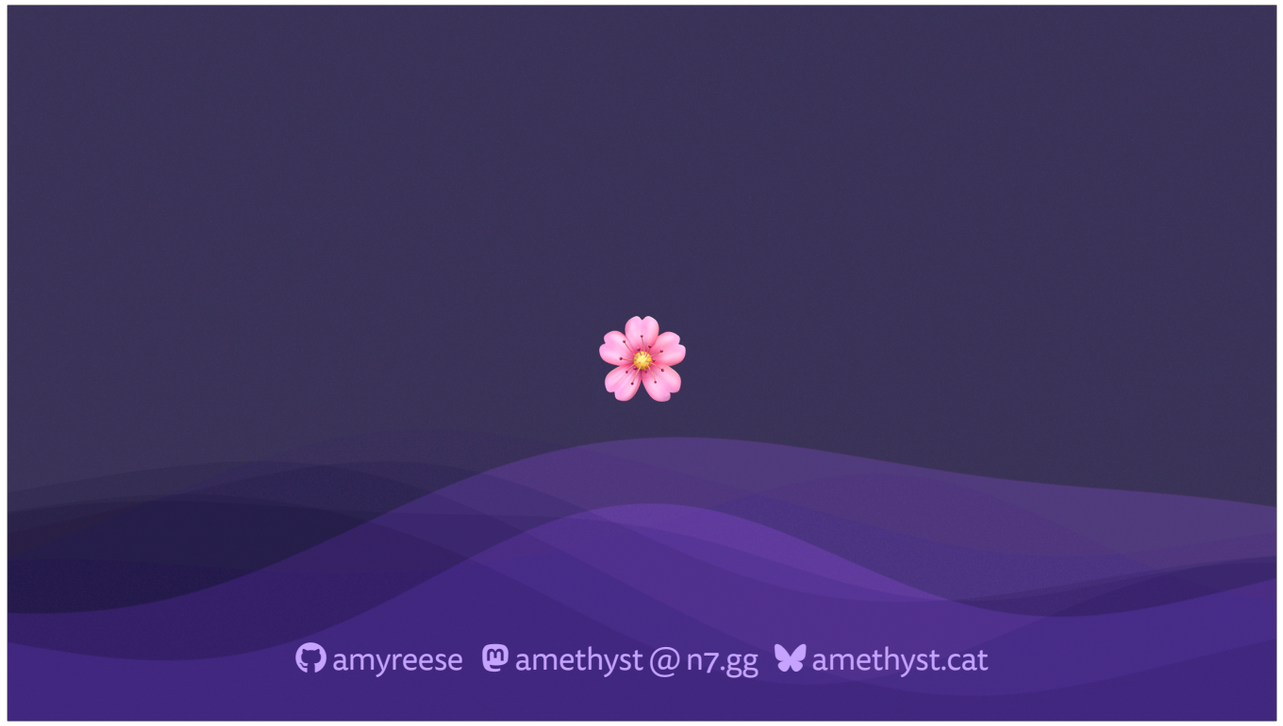


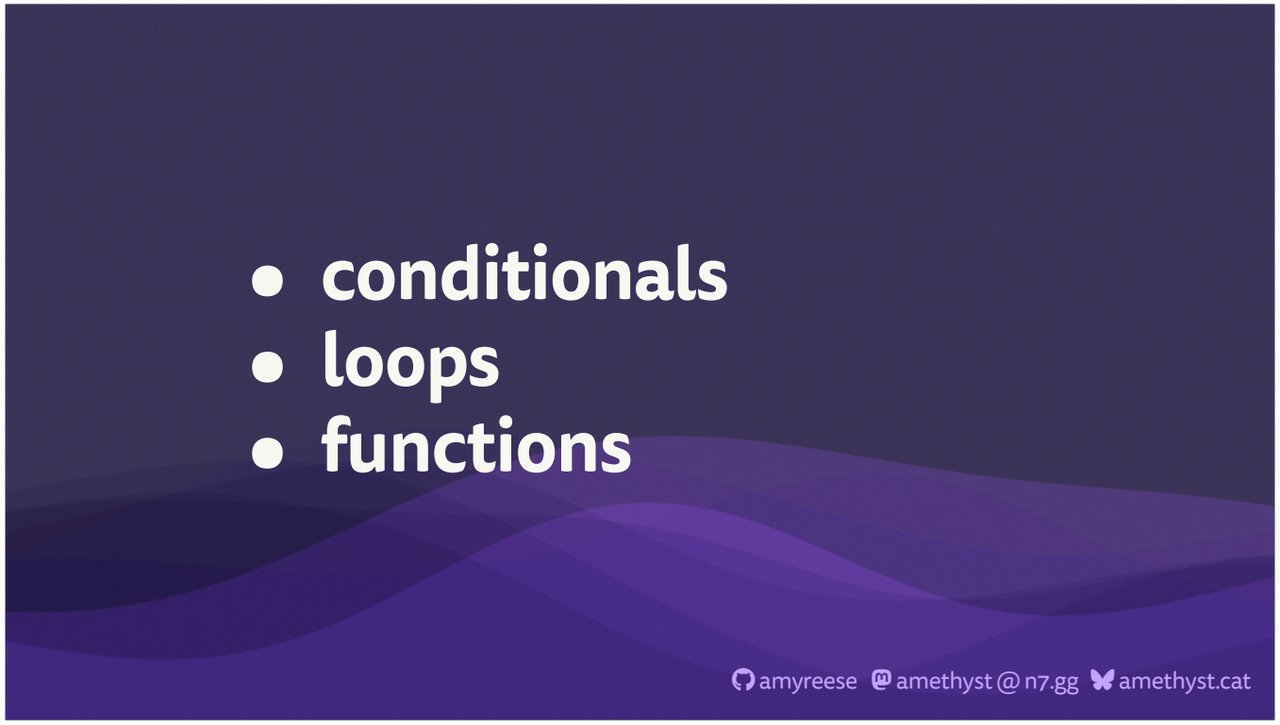
Hi folks, here we are in the final talk slot of the day. You could be off getting food already, but for some reason you chose to be here. I'm sorry.

Before we begin, I need to give a bit of a content warning for this talk. I don’t know how else to say this, but it’s going to show code...

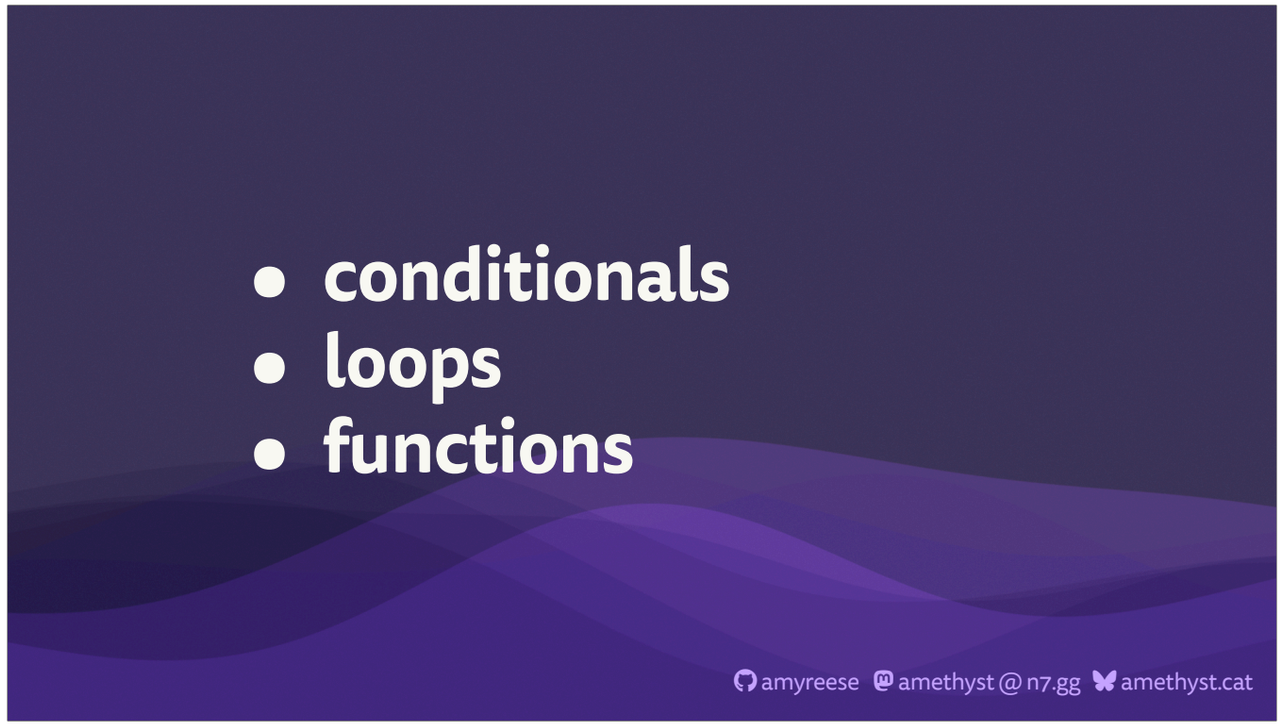
... lots of code — and I'm going to move quick. Most of the code will be small snippets with large fonts, so it should be easy to read and follow. But don't worry about catching key moments with your phone. I'll share a URL at the end with everything, including the code, the slides, and my script. Now...

I was planning to give you all an ultimatum to go with this talk title. But I think we've had enough threats in the world lately, and it's hard to imagine a punishment worse than what we're already going through, so instead, I'm just going to try and ignore all that and have fun with the topic.

So let’s start with defining terms: what do we mean by “control flow”? This refers to any part of our code that tells the Python runtime how to alter the “flow” of execution from line of code to the next.



Hopefully, you’re already familiar with the fundamental building blocks of control flow: conditionals, loops, and functions. Every other form of control flow can be expressed as a combination of one or more of these three basic mechanisms.

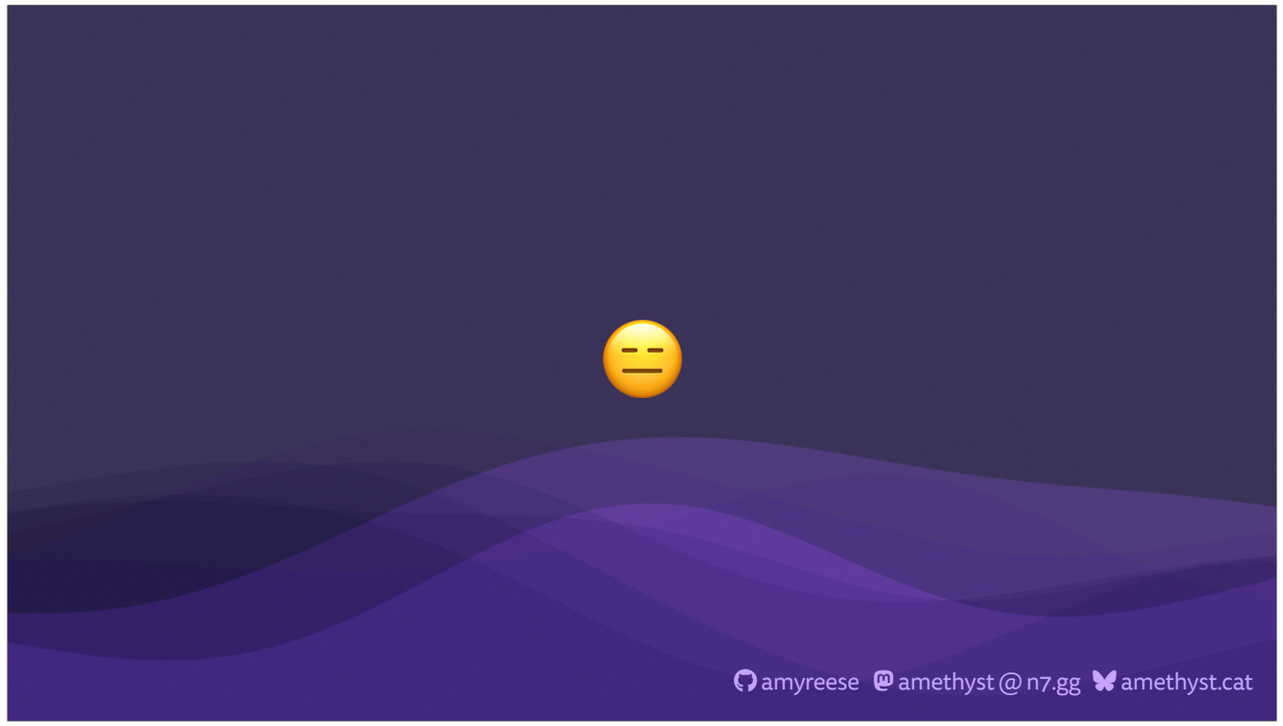


Of course, Python has many different types of control flow, all of which are defined as part of the language grammar. We can even look at Python's grammar specification to see all of the possibilities...

cpython - python.gram

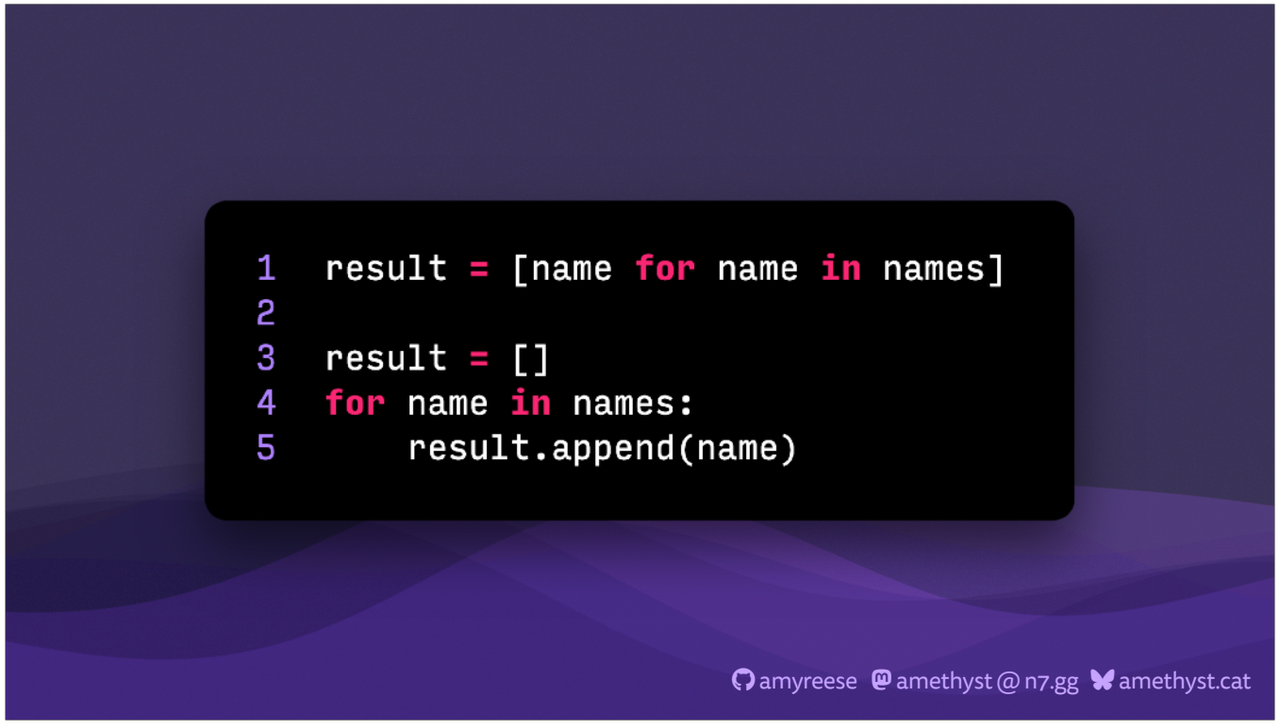
```
# STARTING RULES
# ==============

file[mod_ty]: a=[statements] ENDMARKER { _PyPegen_make_module(p, a) }
interactive[mod_ty]: a=statement_newline { _PyAST_Interactive(a, p->arena) }
eval[mod_ty]: a=expressions NEWLINE* ENDMARKER { _PyAST_Expression(a, p->arena) }
func_type[mod_ty]: '(' a=[type_expressions] ')' '->' b=expression NEWLINE* ENDMARKER { _PyAST_FunctionType(a, b, p->arena) }

# GENERAL STATEMENTS
# ==================

statements[asdl_stmt_seq*]: a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }

statement[asdl_stmt_seq*]: a=compound_stmt { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) } | a[asdl_stmt_seq*]=simple_stmts { a }

statement_newline[asdl_stmt_seq*]:
    | a=compound_stmt NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
    | simple_stmts
    | NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, CHECK(stmt_ty, _PyAST_Pass(EXTRA))) }
    | ENDMARKER { _PyPegen_interactive_exit(p) }

simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) } # Not needed, there for speedup
    | a[asdl_stmt_seq*]=';'.simple_stmt+ [';'] NEWLINE { a }

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt[stmt_ty] (memo):
    | assignment
    | &"type" type_alias
    | e=star_expressions { _PyAST_Expr(e, EXTRA) }
    | &'return' return_stmt
    | &('import' | 'from') import_stmt
    | &'raise' raise_stmt
    | &'pass' pass_stmt
    | &'del' del_stmt
    | &'yield' yield_stmt
    | &'assert' assert_stmt
    | &'break' break_stmt
    | &'continue' continue_stmt
    | &'global' global_stmt
    | &'nonlocal' nonlocal_stmt

compound_stmt[stmt_ty]:
    | &('def' | '@' | 'async') function_def
```

Oh. Oh..... Hmm... Did you catch all of that?

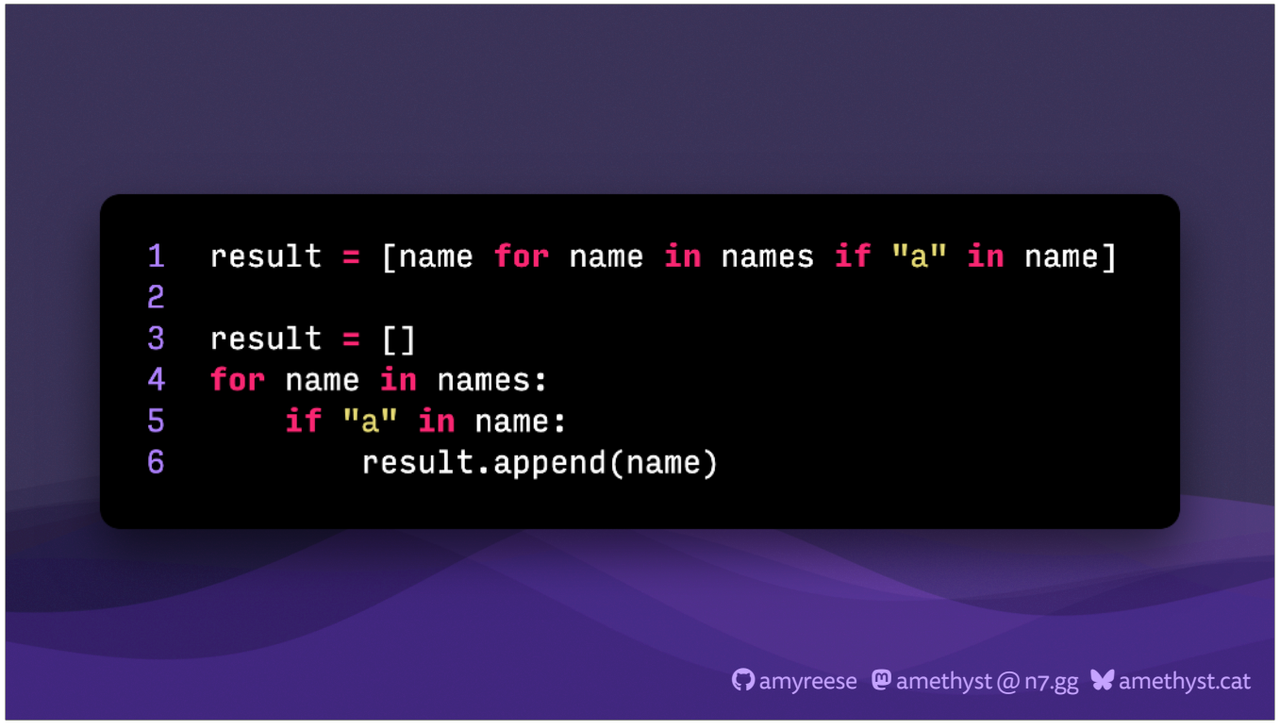


Now obviously, that's a lot, and thankfully for you, I'm not here today to teach you about language grammar. I also assume everyone here is familiar with the basics, so instead let's start to look at some of the less obvious options that Python gives us to manage control flow in our projects.

Comprehensions are relatively unique to Python among mainstream programming languages. At their core, comprehensions are syntactic sugar around for loops and if blocks, with the primary purpose of building collections of data.



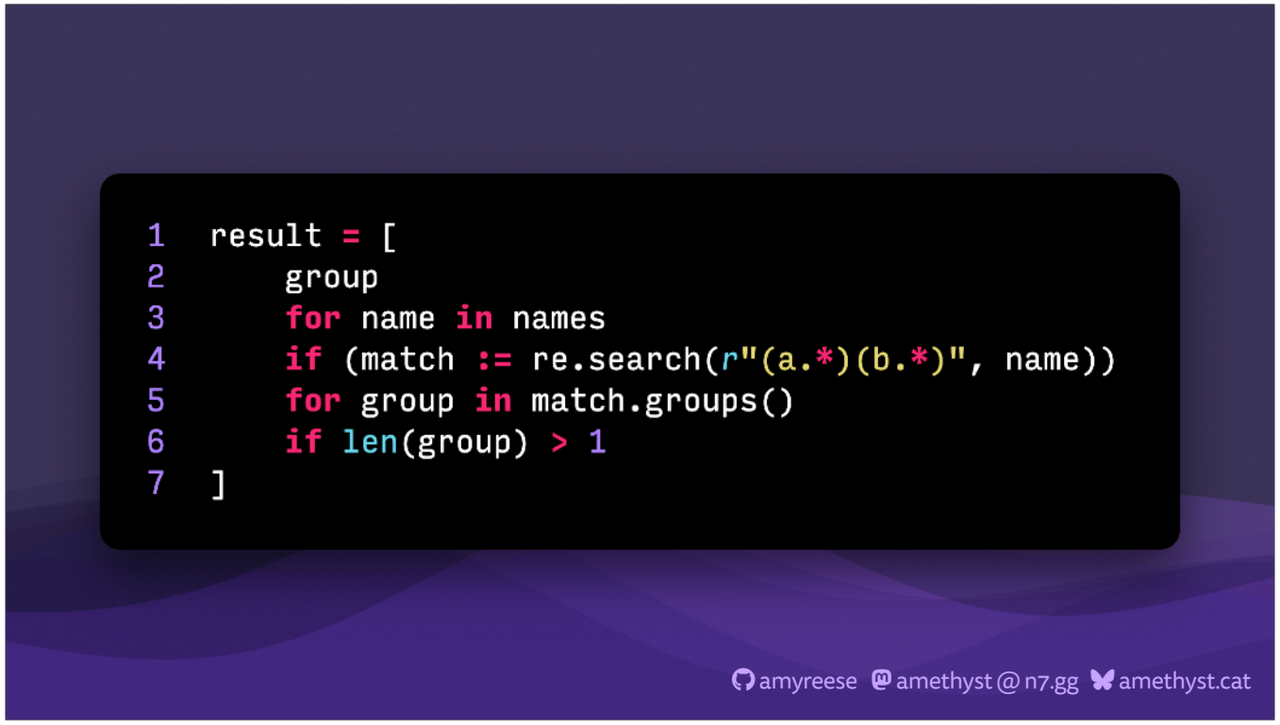
Now ignoring performance, these two pieces of code produce identical results. But the key difference is in readability. Now, slightly ironically, opinions on *which* is more readable will vary, depending on familiarity with Python vs other languages.



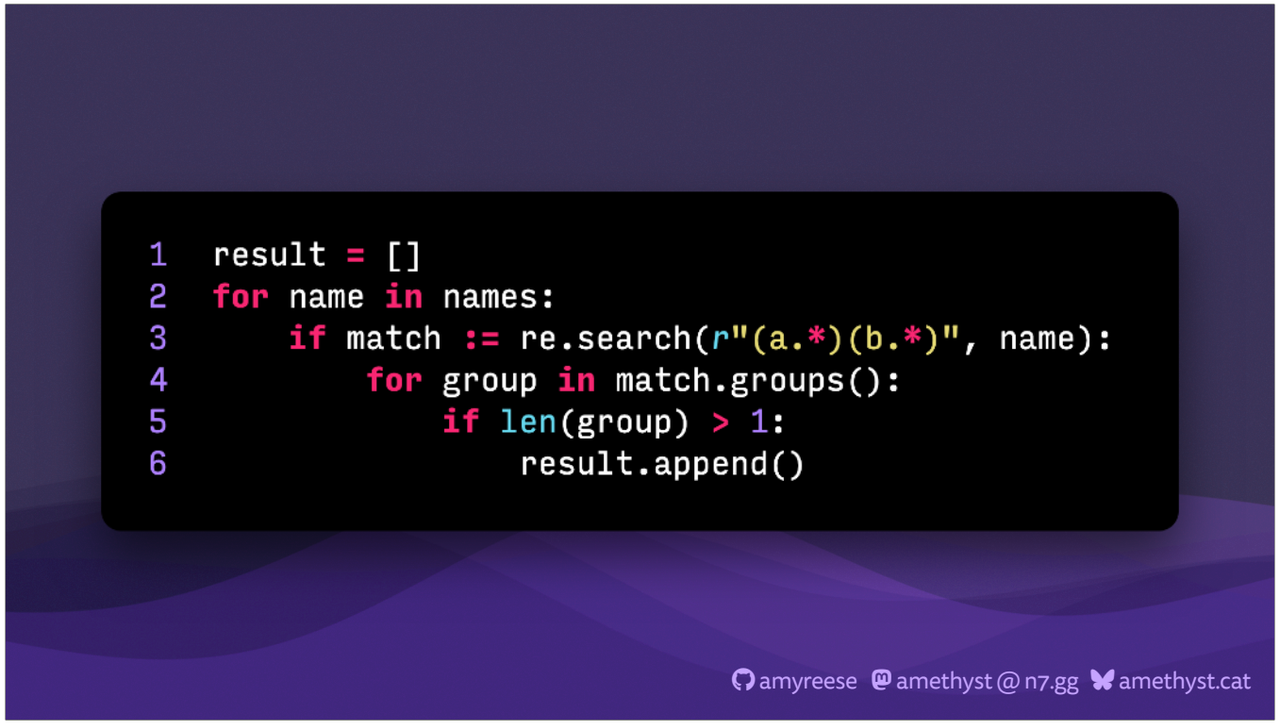
As we make the comprehension more complicated, we should consider the benefits and tradeoffs of performance and compactness against readability and maintainability of the code in question. I don’t think many would argue against using a comprehension for simple operations.

But what about now? Here we're adding a second for loop to the comprehension. But because of the way comprehensions work, we're forced to order the clauses out of contextual order, obfuscating the intention of the code.

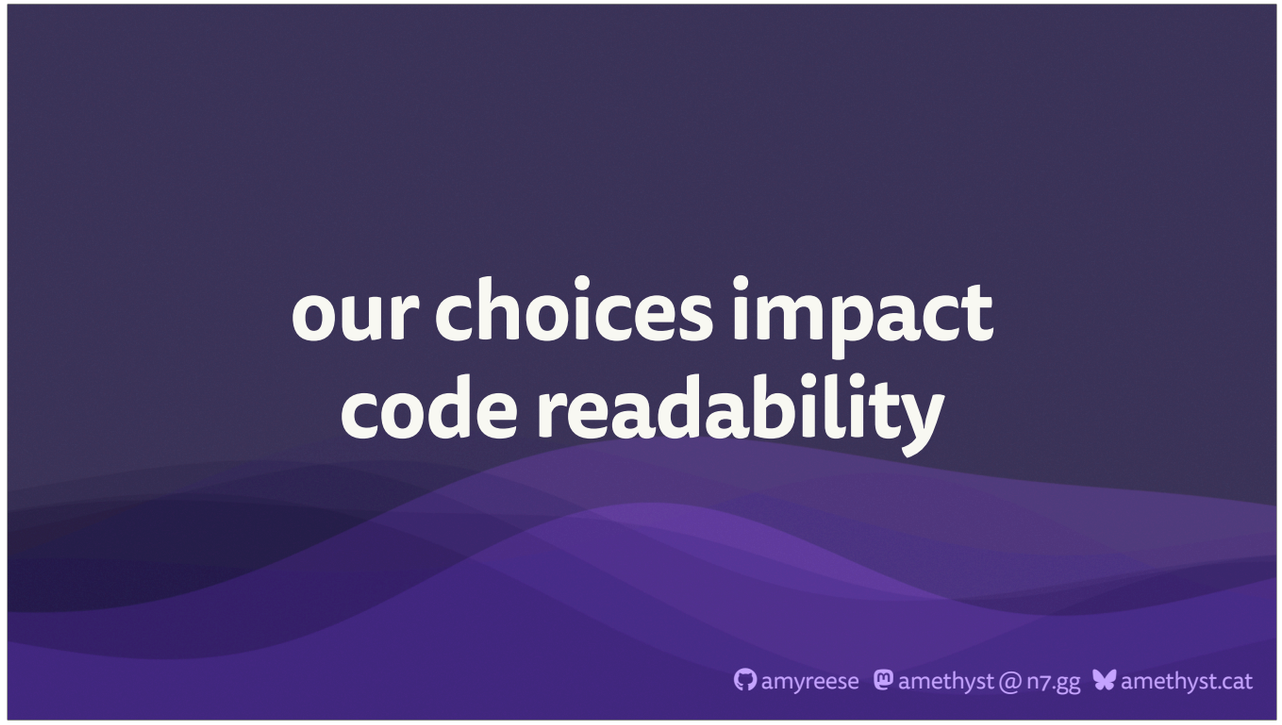
Some manual reformatting can help, and I do think this improves readability. But at this point, it's becoming harder to argue which of these two methods is easier to read and understand.



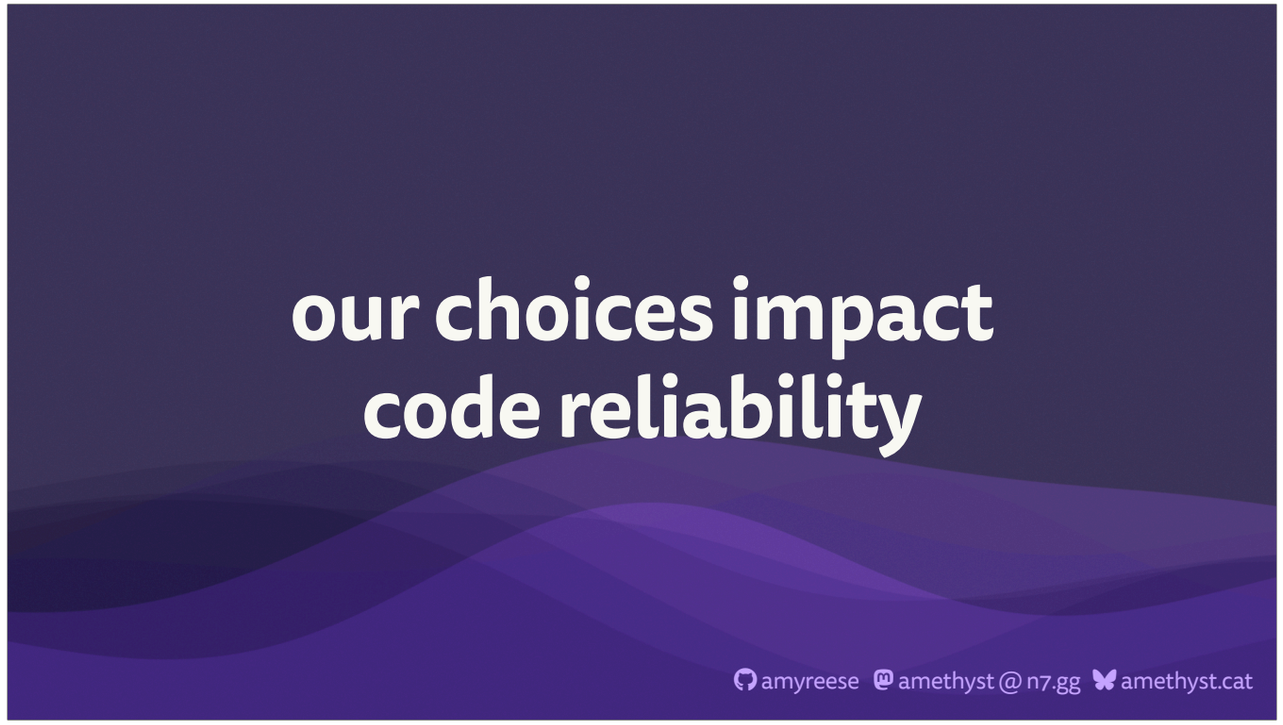
So where do we draw the line? One thing to consider is that multiline comprehensions like this hide away the inherent cyclomatic complexity of their code. It’s a single comprehension of course, so it must be fast!



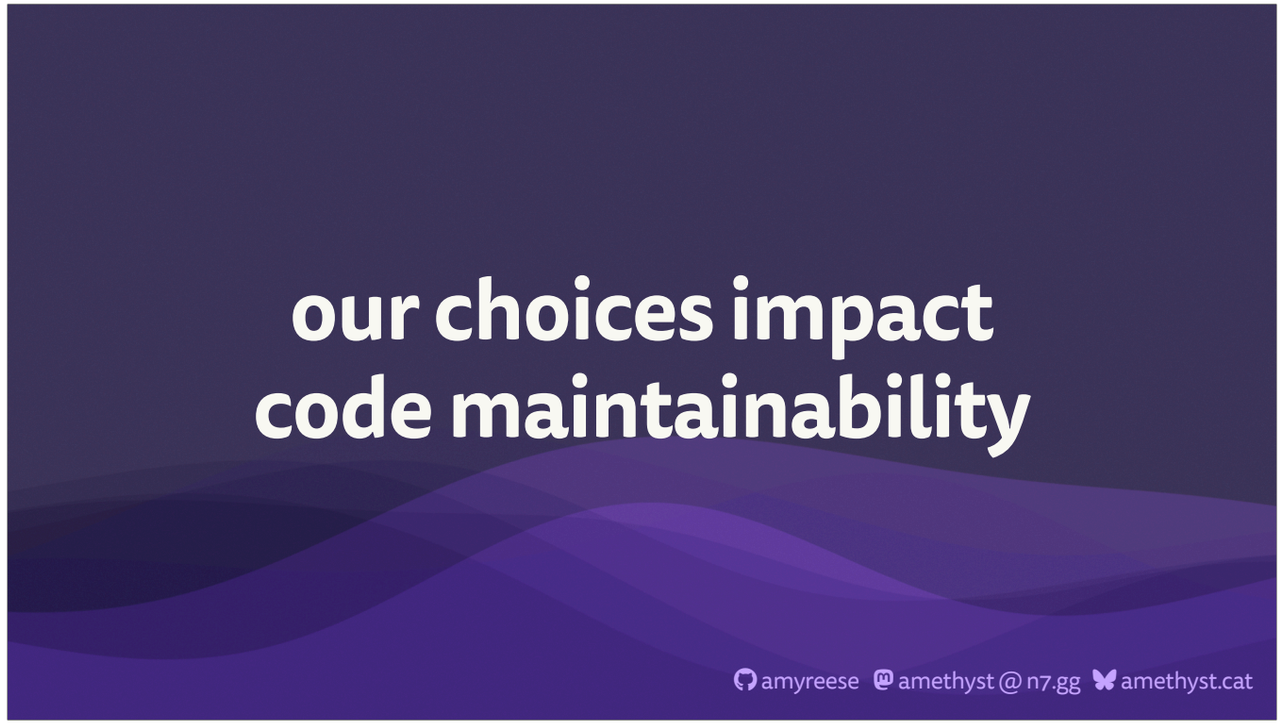
Whereas the traditional form of that code requires multiple levels of nesting, making it much more obvious to the reader that this will likely result in slow performance on large data sets.



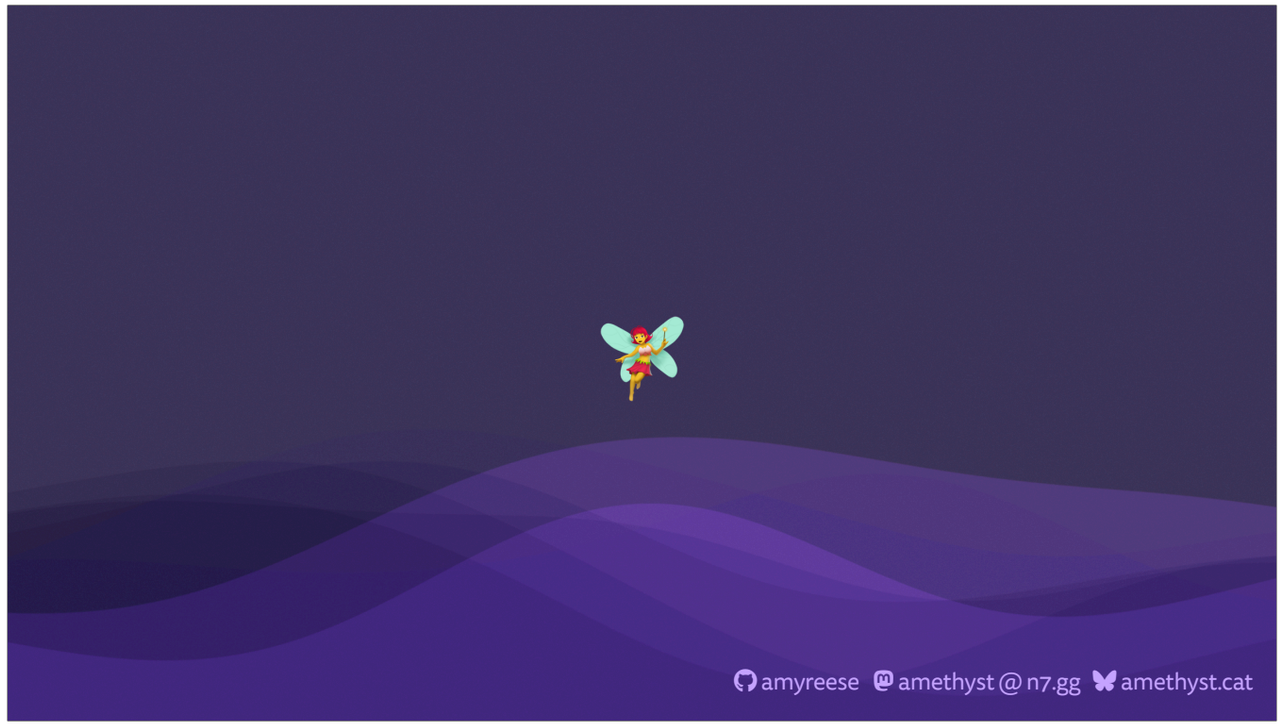
Our choices as developers impact code readability, and thereby impact the maintainability and reliability of our software.



We must weigh the advantages and disadvantages of various approaches, and consider not just the performance of our code, but our confidence that future changes won’t cause bugs...



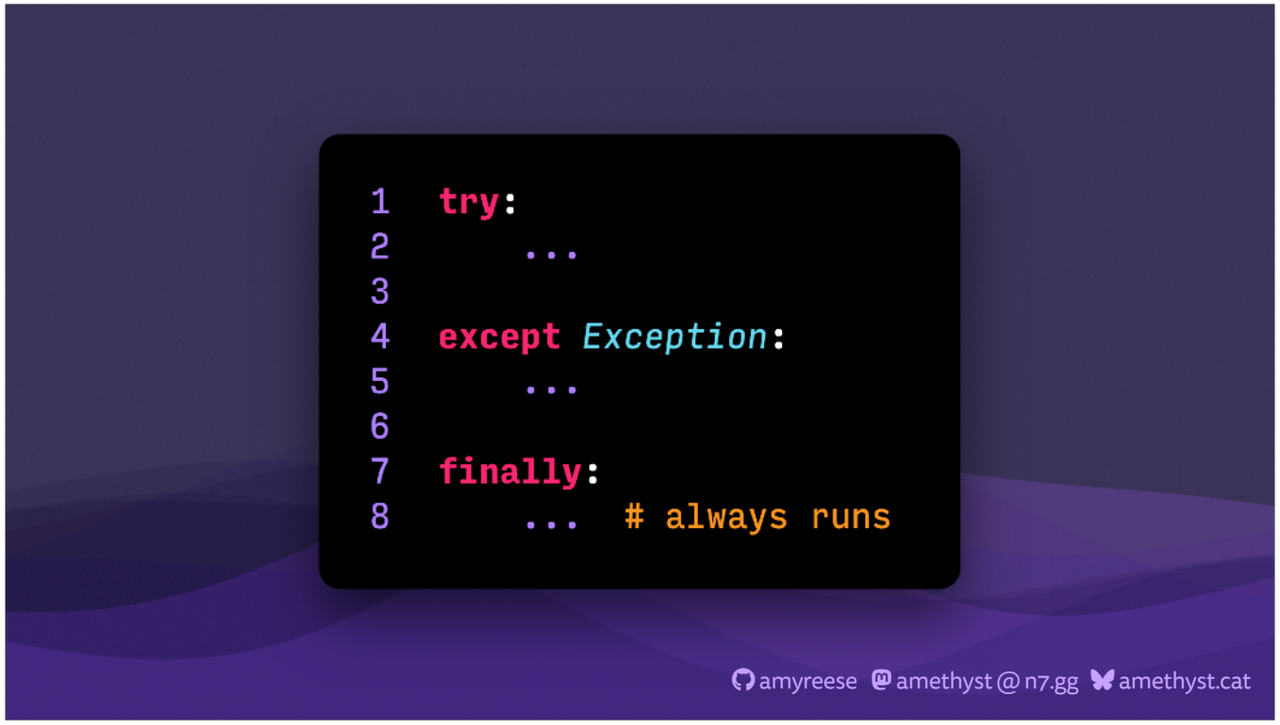
... or even just the cost in time and experience needed to understand and modify our code after it's already been shipped to production. We need to consider what the right choices and tradeoffs are when writing code.



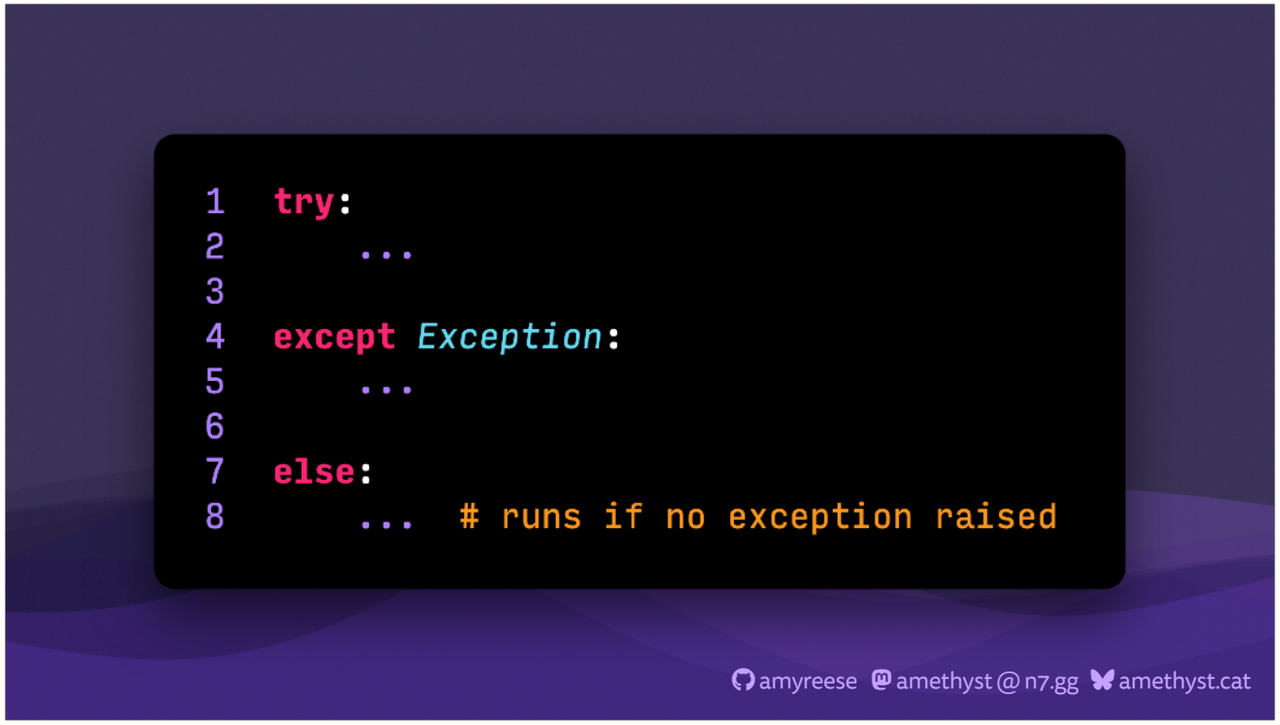
<break> With that said, let’s look at the some of the less obvious parts of the language, and see if we can’t find ways to improve readability of our code in the process.

In particular, let’s look at the “else” block. Mostly known for its use in conjunction with an “if” statement. But it can be paired with other code blocks as well.

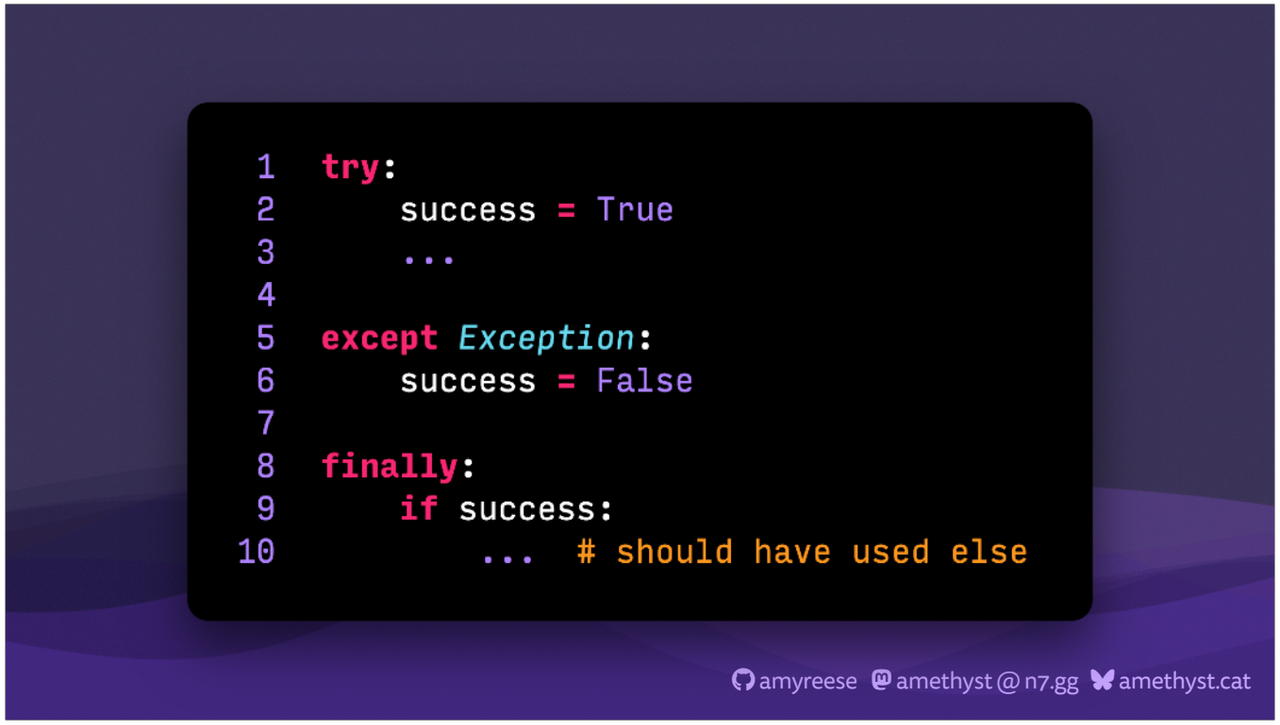
Like exception handling, where we can use the else block for some common code patterns.



As many of you are likely aware, adding a finally block to try-except gives us a way to run some code afterward, regardless of whether an exception is raised or not.

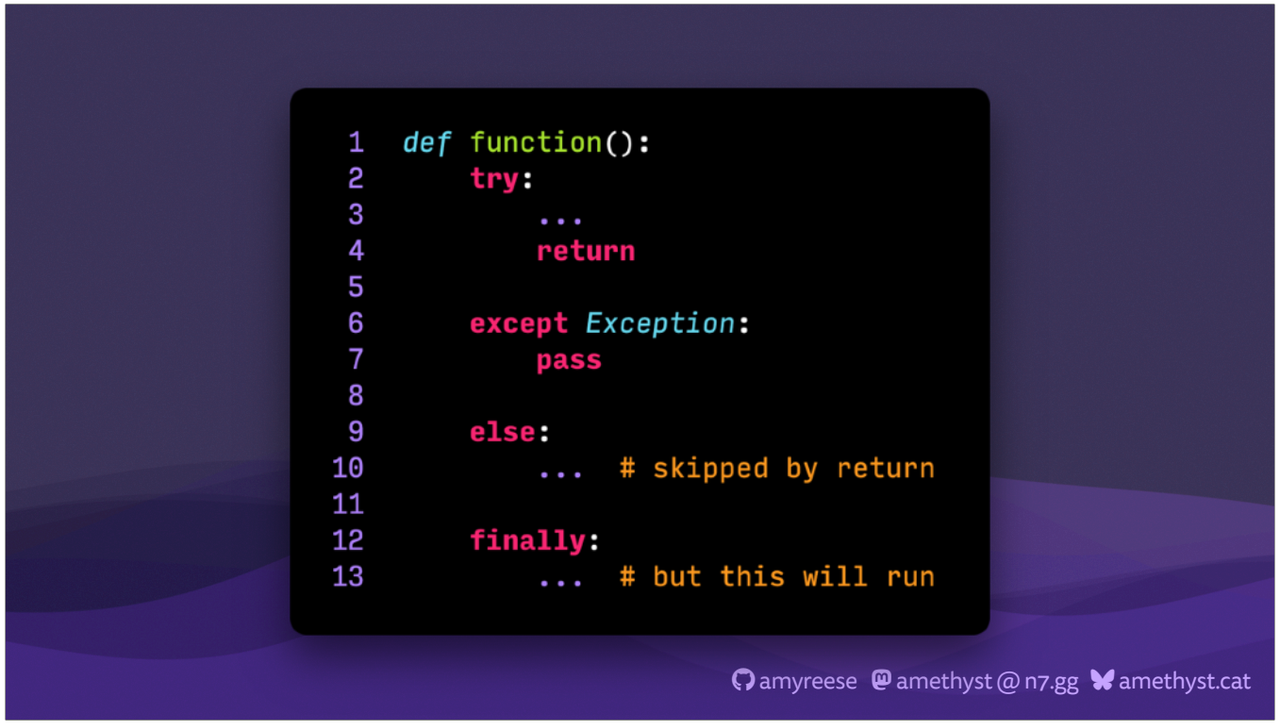


By comparison, the else block is only run if there was no exception raised.



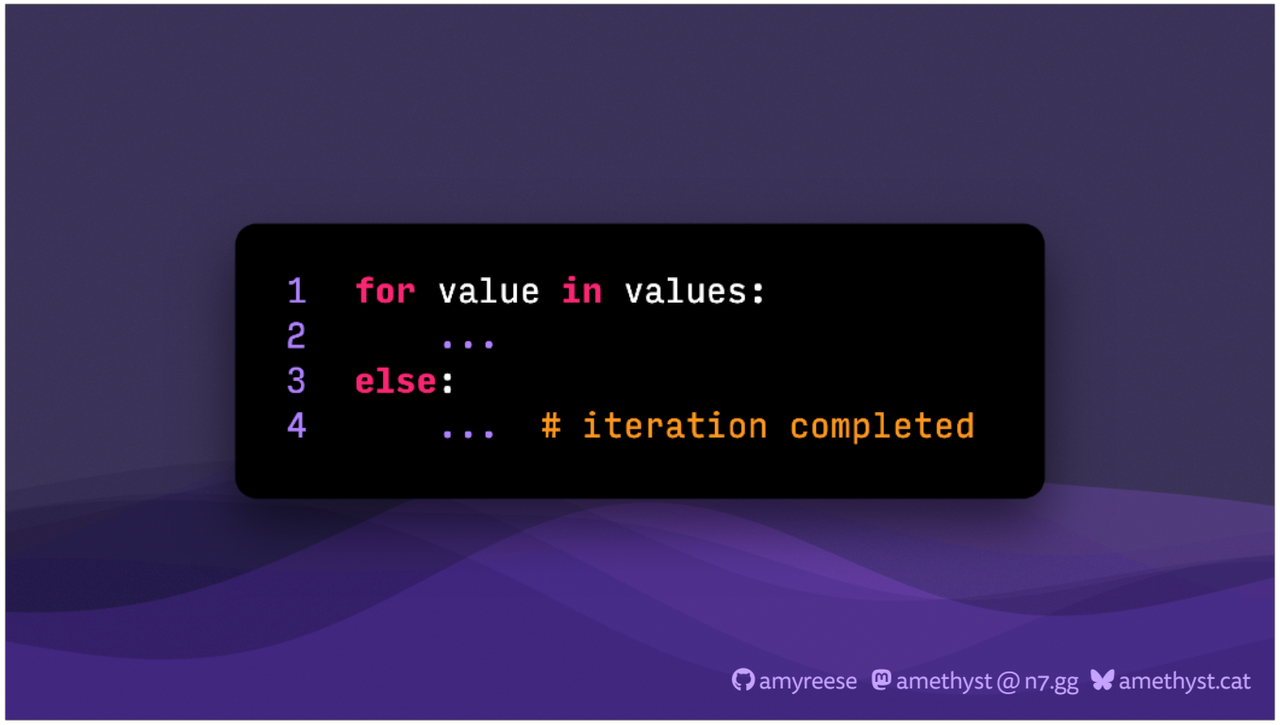
Think of it like setting a “success” variable, and then checking that value later, either in a finally block or outside of the try-except entirely. I’m sure many of us have seen code patterns like this in the past. Maybe that code should have just been an else block — Python gives us that logic for free, and it’s fewer lines of code in the process!

This lets us maintain the best practice of minimizing the amount of code inside the try block itself, while also minimizing effort needed to track whether our code was successful.

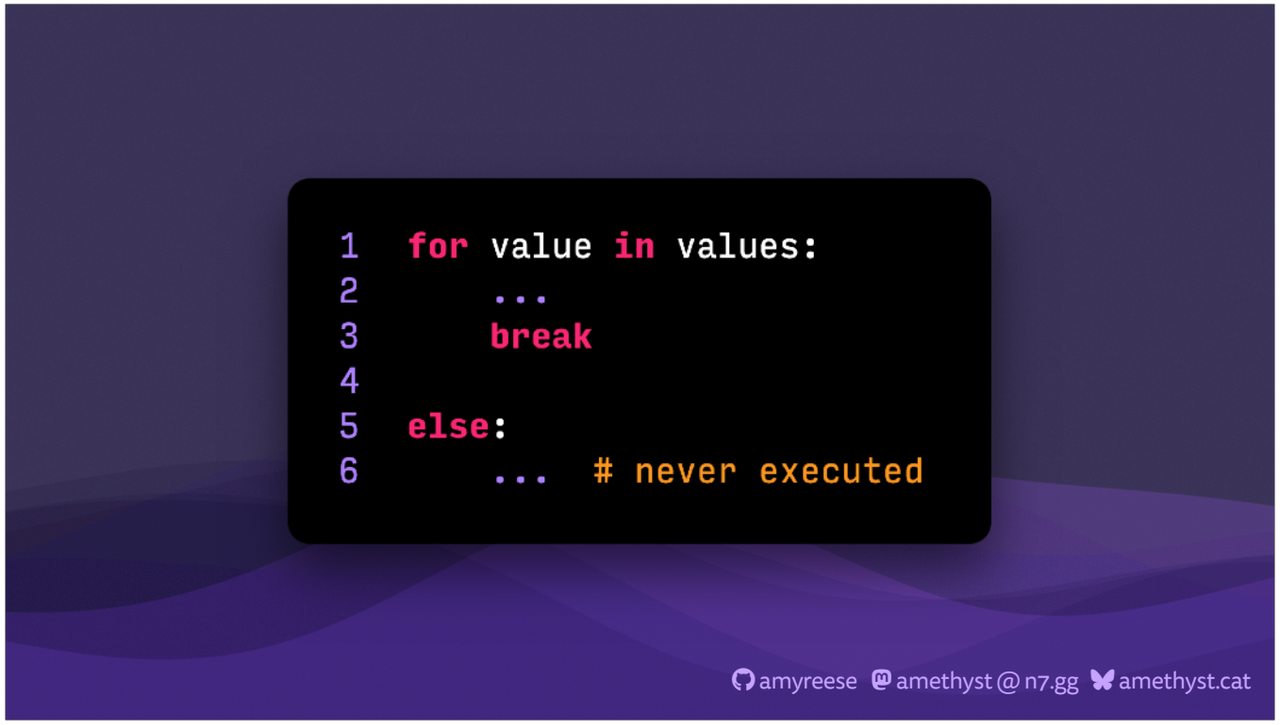


But beware! Unlike finally, a return statement here will *not* execute the else block. So either keep the return statement outside of the try block, or be sure that critical code remains in the finally block instead.

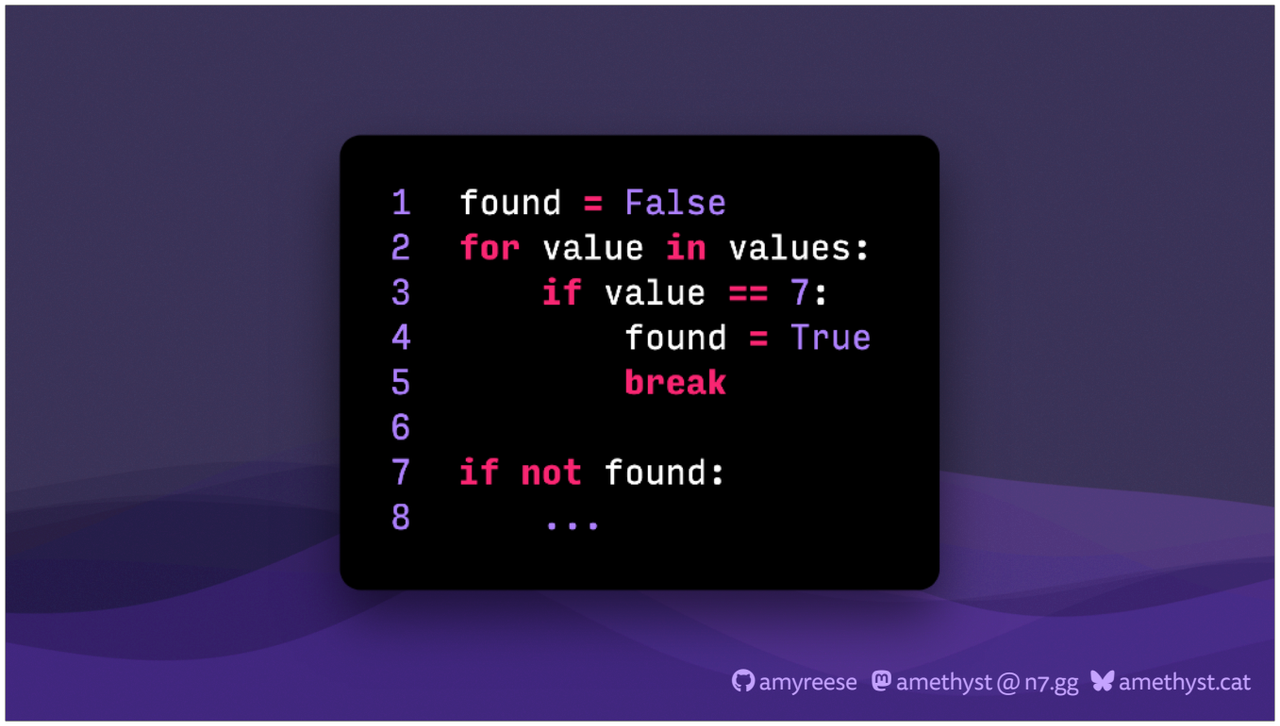
Now we can also use the else block in loops, somewhat analogous to its use with try-except.



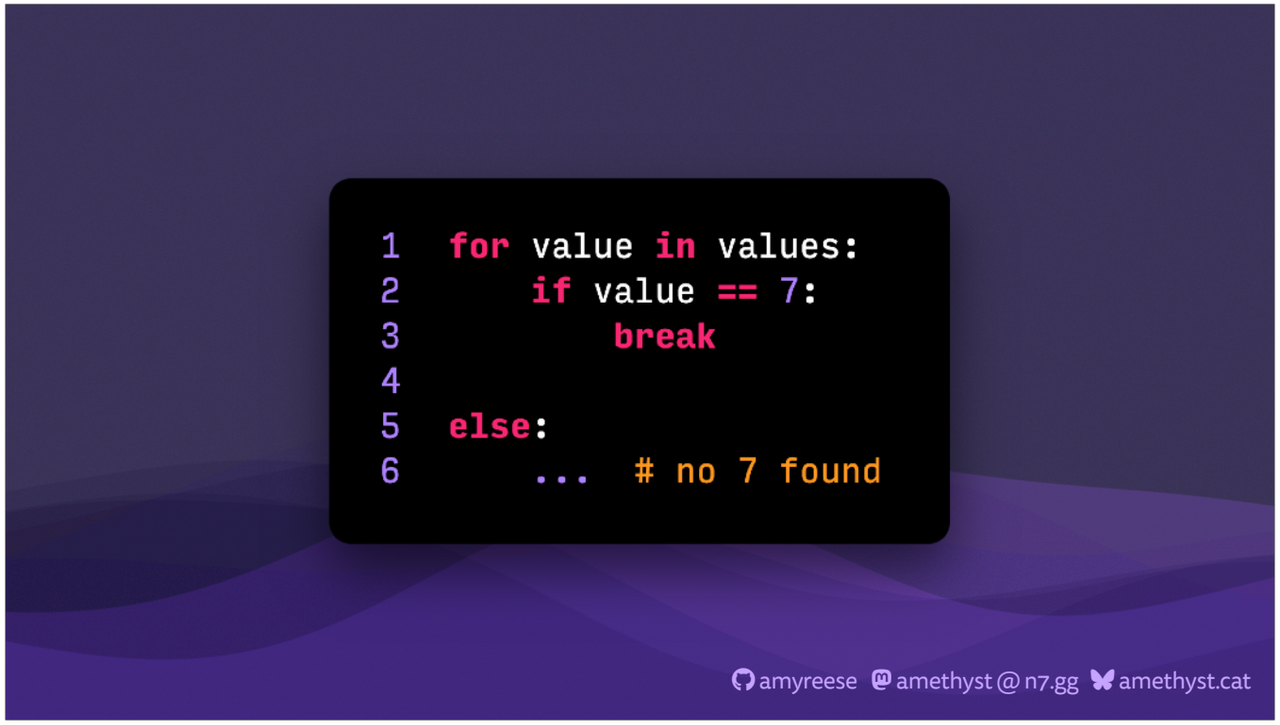
Here the else block is executed if and only if the for loop exhausts its iterable.



But if the for loop reaches a break statement, then the else block is skipped. So you can think of this like try-except-else, but instead of executing if no exceptions are raised, it executes if no breaks are raised.

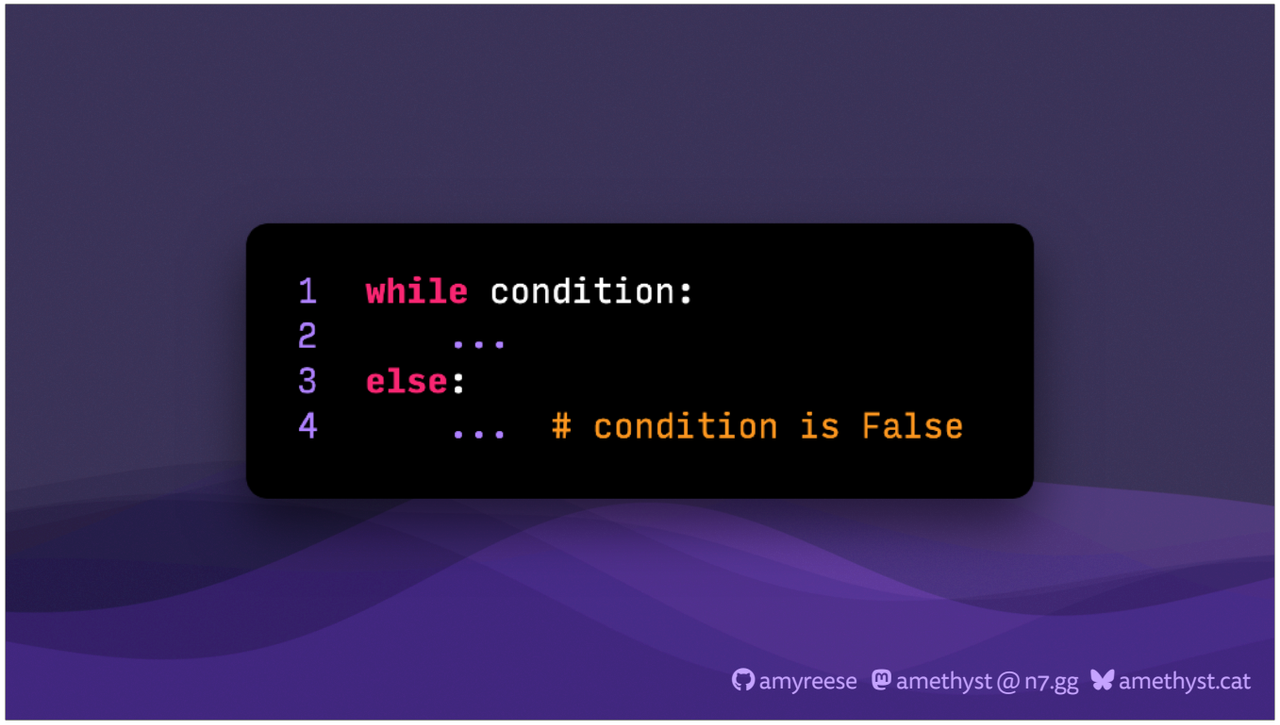


Imagine this code pattern. We're iterating over some data, and break on the first time we find a match. Now, instead of maintaining new variables to handle cases with no match...

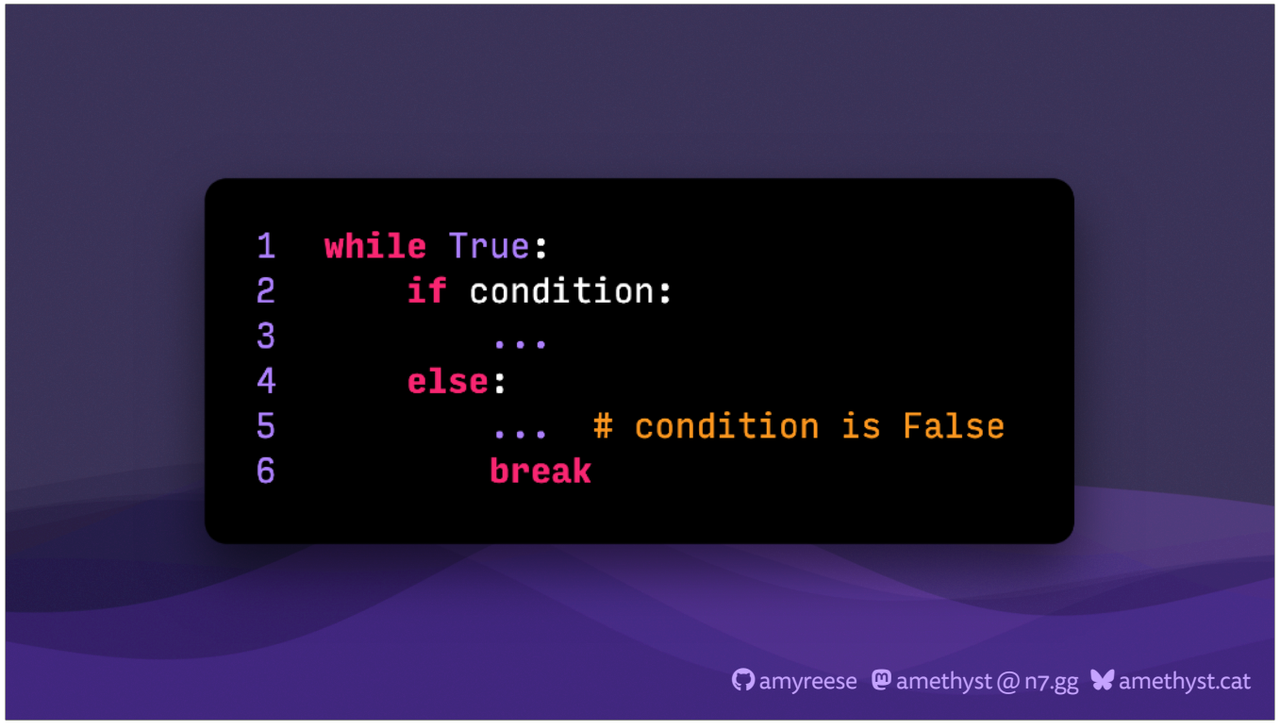


... we can replace that logic with an else block, simplifying our code, and hopefully making it more readable in the process.

Our last use case is the while loop. Like for loops, the else block here executes if no break statement was reached. But we can also think of it in a better way.

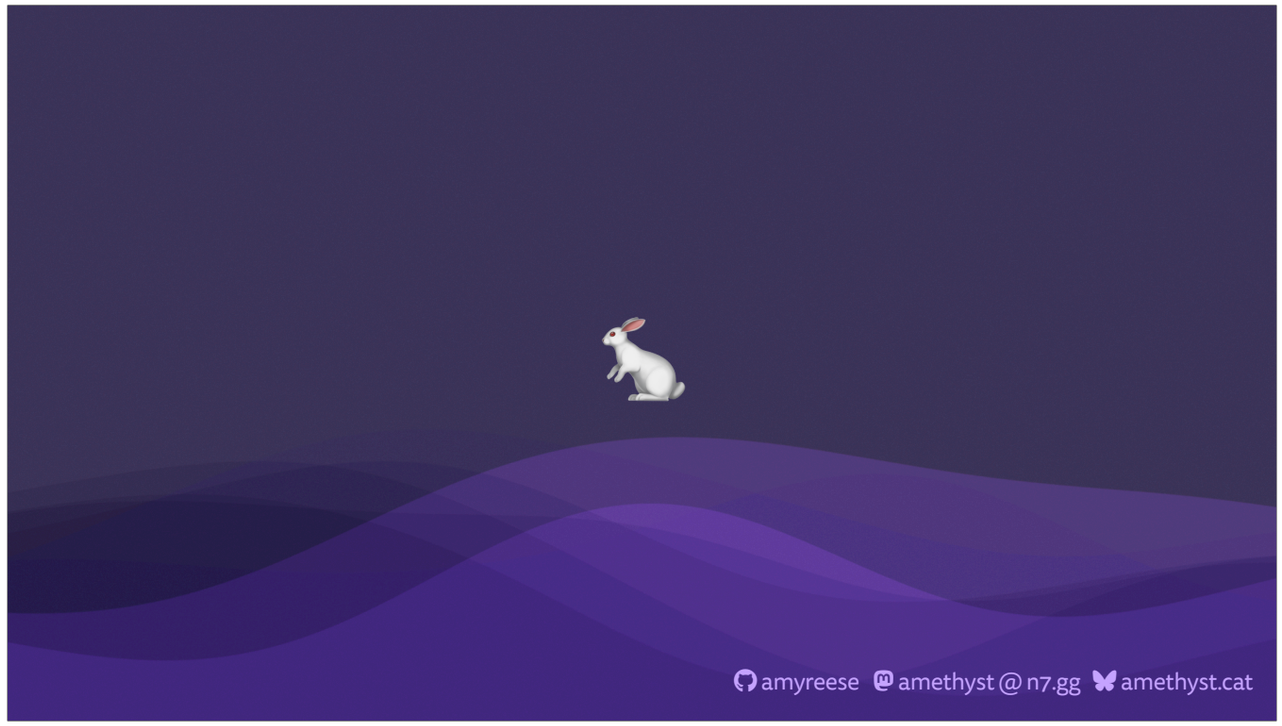


The else block runs when the condition of the loop evaluates as false.



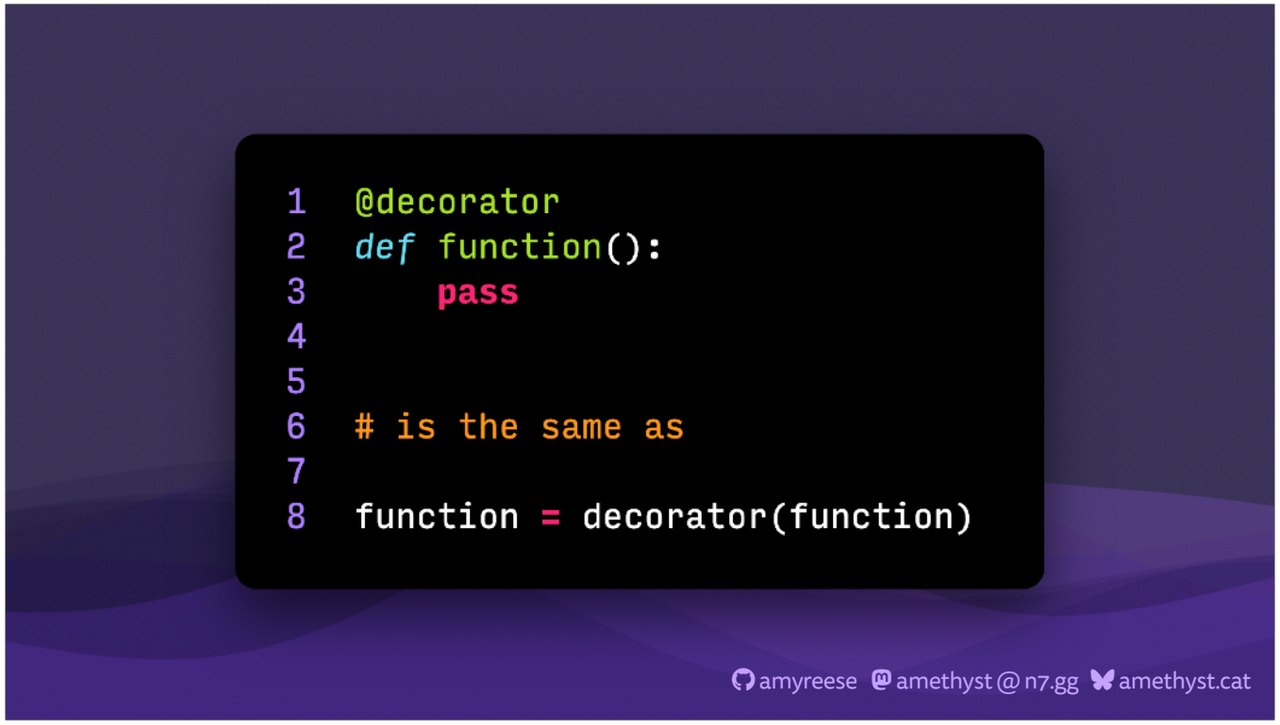
Functionally, it’s equivalent to an infinite while loop, where you nest an if-else block inside, and break out of the loop once that condition is False.

But again, once a break statement is reached, the loop condition won’t be reevaluated, and the else block will not be executed.

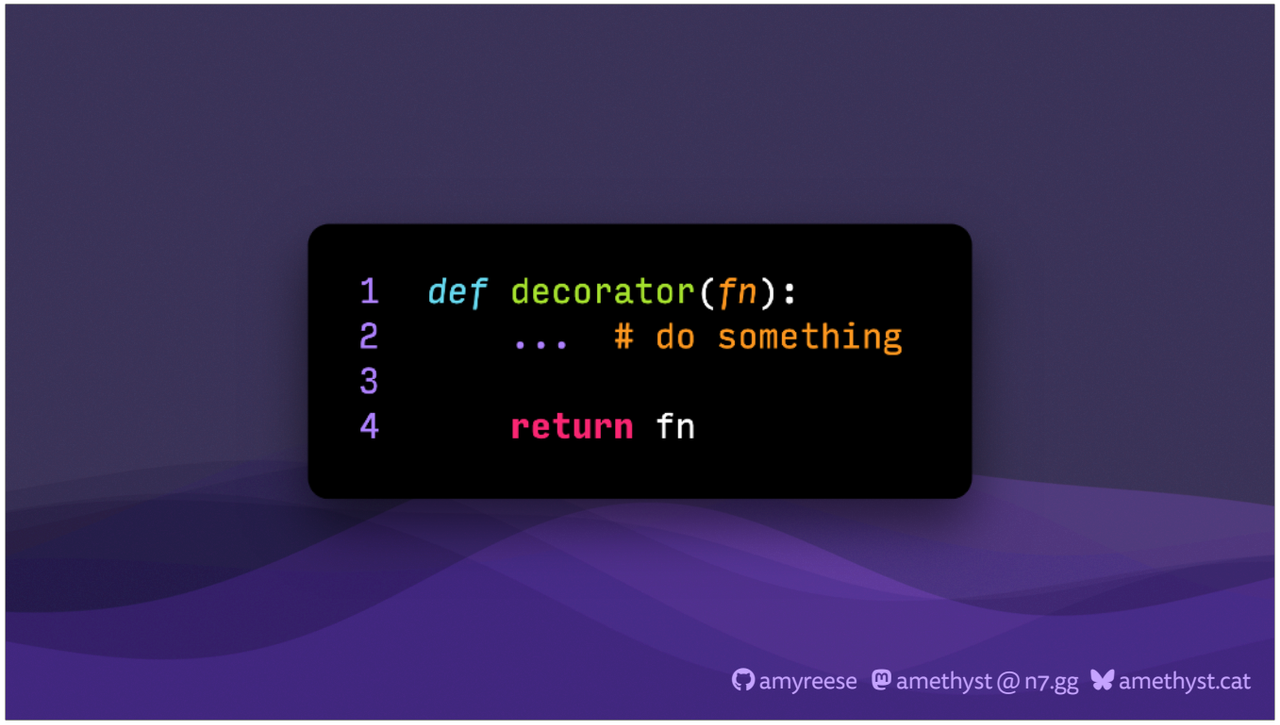


<drink> Now, we're going to jump down the rabbit hole. Let's look at the ways we can use — or abuse — some of the tools that Python gives us. Things that we might not think of as control flow.

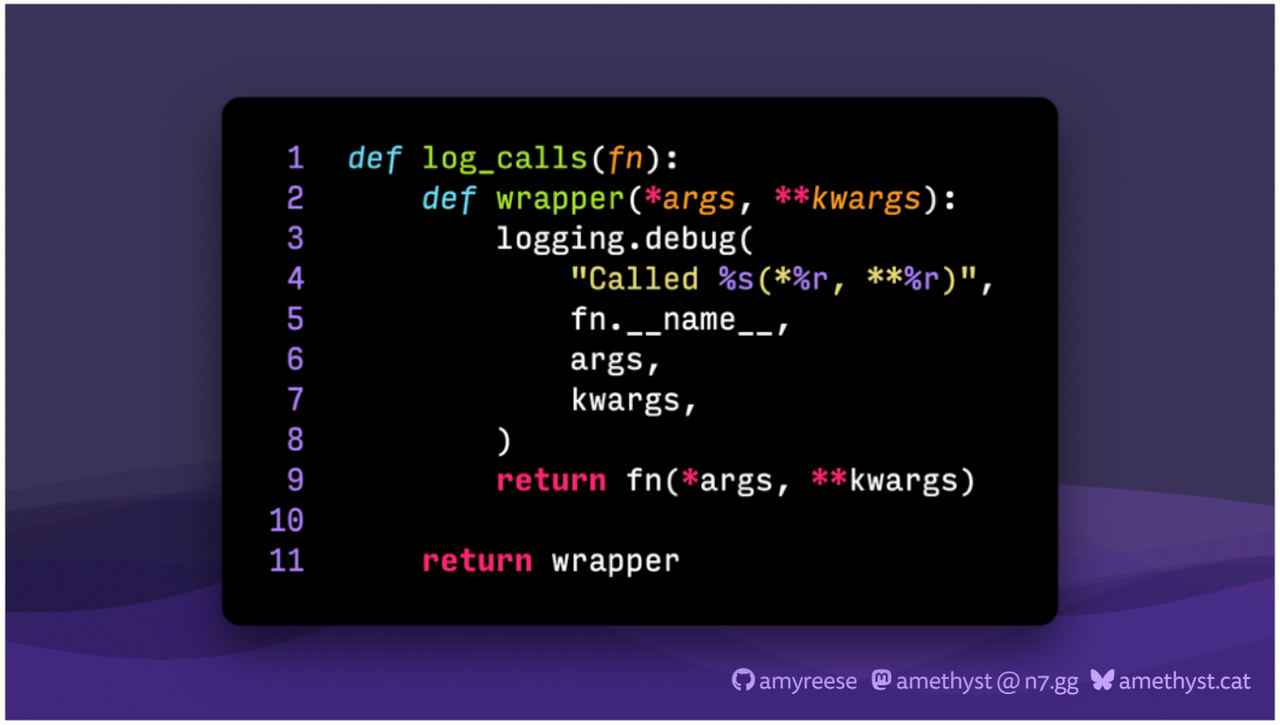
Like decorators, the fun little things we can tack on to our functions and methods and classes. Some of the best inventions in Python have come out of decorators, like dataclasses.



But really, decorators are just callable objects, using syntactic sugar from the language. Adding the “at decorator” above our function is shorthand for calling the decorator, passing it our function, and then overwriting our function with whatever the decorator returns.



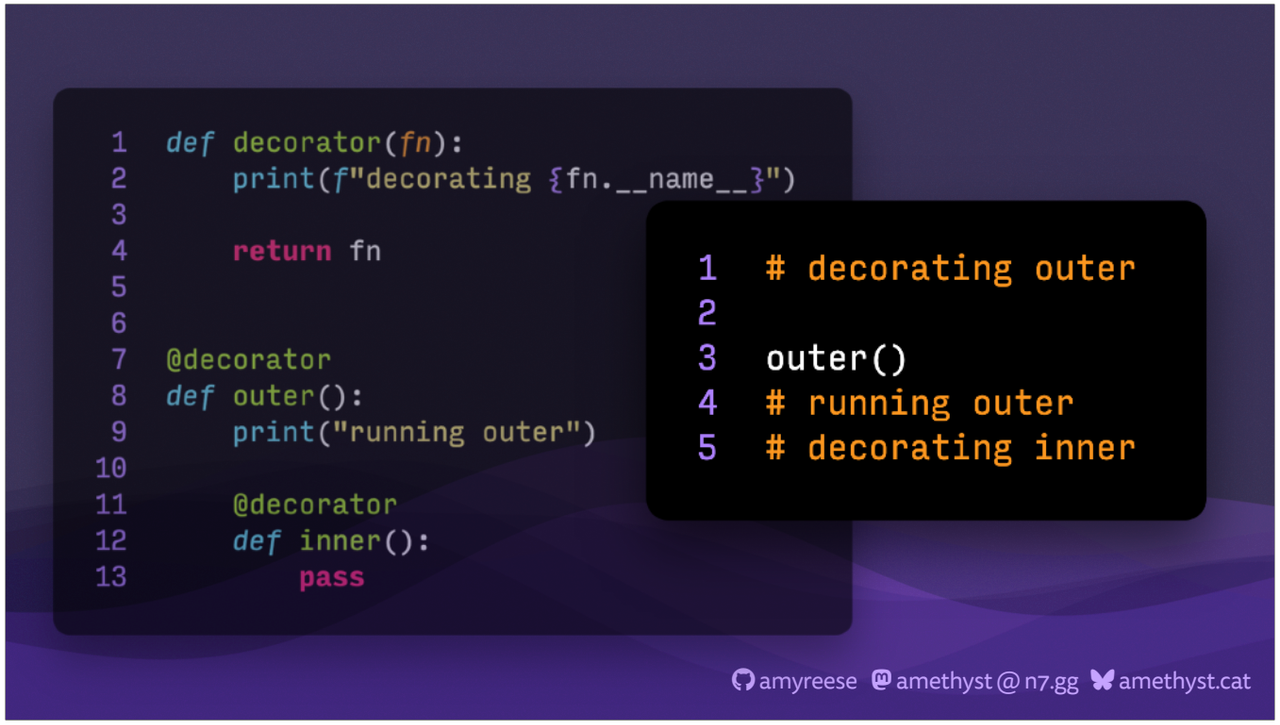
Making a decorator is as simple as making a function, only this function takes a function or class as an argument, does *something* with it, and returns the new object when finished.



The most common decorators are function wrappers — they define an inner function wrapping the original one, do something interesting, and then call the original and return the results up the stack. This might be something as simple as logging every time a function is called.

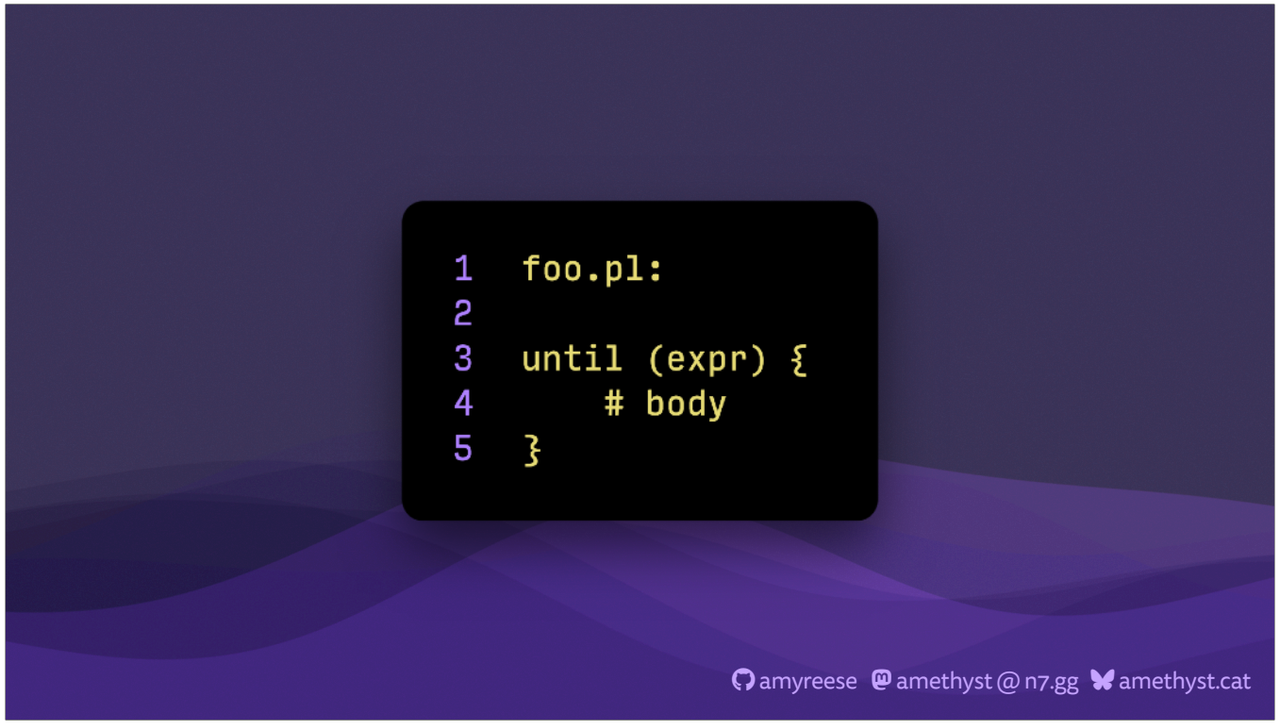
Our greet function can then add the “log calls” decorator, and now every time we call greet, we automatically get logging output including the name of the function and the arguments it was called with.

But what makes a decorator *control flow* is key: using the “at decorator” syntax is literally calling and executing the decorator’s code at the moment the function is defined in scope. That means our decorator will execute before the function it’s wrapping is ever used, and if we import this code...



We'll see that before we ever even call the outer function, the decorator has printed its message — at import time no less! And when we do call the outer function, the decorator runs again to process the inner function, even though it is never called or used.

But decorators don’t have to return functions, or methods, or classes, or a callable — they don’t even need to return anything at all! All that matters is that using the “at” syntax implicitly calls your decorator function.



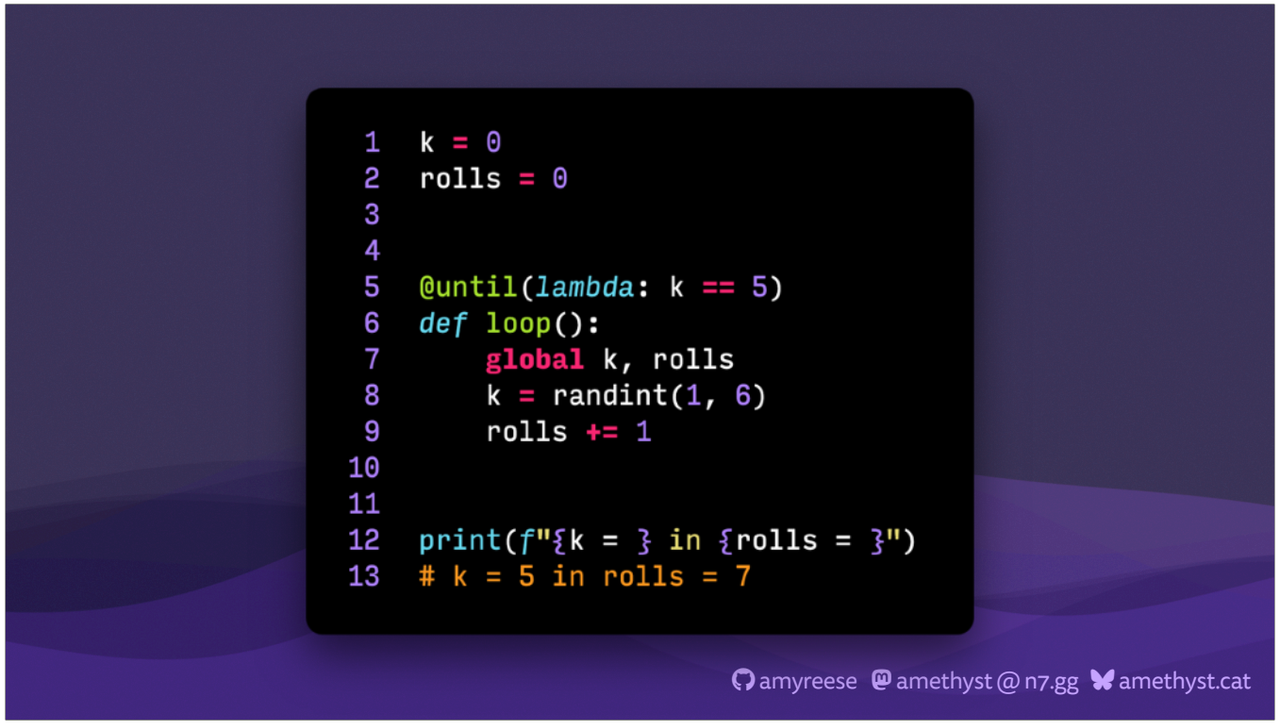
Now, a quick side note: one of the first programming language I learned as a kid was Perl. It has a few variants of loops that invert the expected conditions of the loop expression. The “until” loop is the inverse of the while loop, and will repeat the body of the loop *until* the expression evaluates *true*, rather than looping until it’s false. I think that’s kinda neat.

So I replicated the until loop in Python using a decorator. In its most simple form, it takes a callable as the loop predicate, and calls the wrapped function repeatedly until the predicate evaluates true.

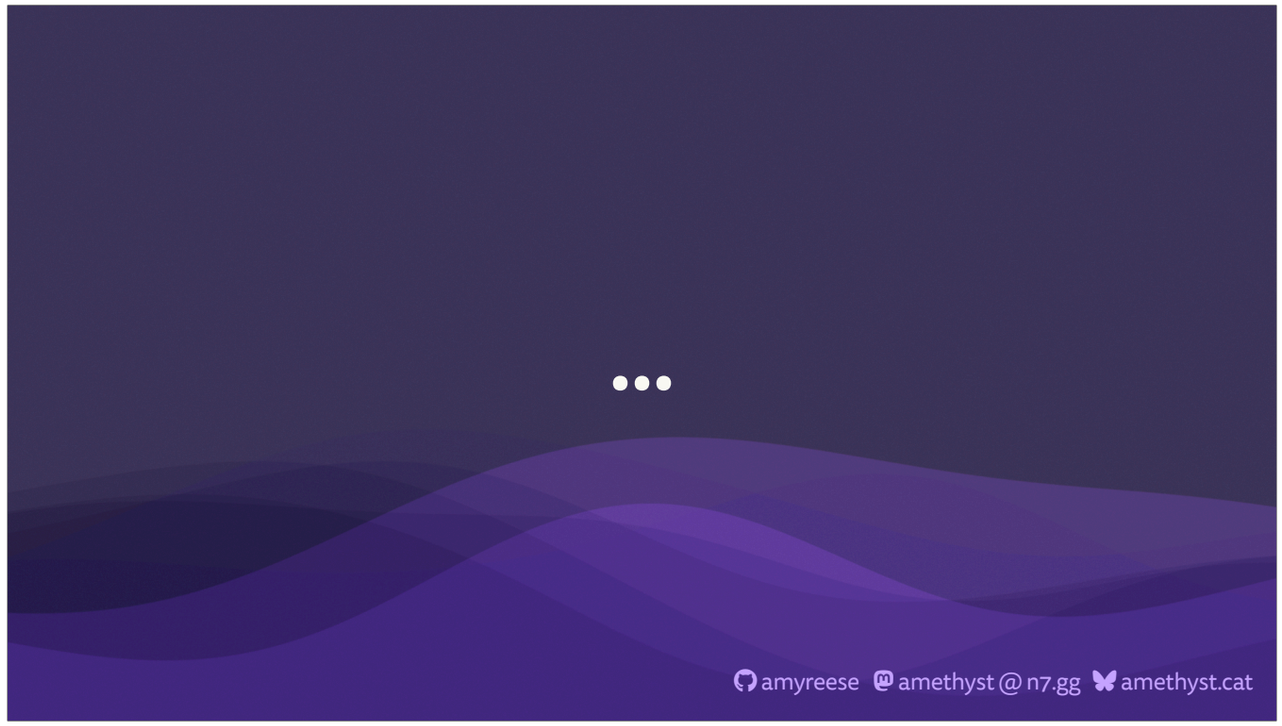
```python
def until(predicate):
    def wrapper(fn):
        def wrapped():
            while not predicate():
                fn()

        wrapped()

    return wrapper
```

amyreese amethyst@n7.gg amethyst.cat

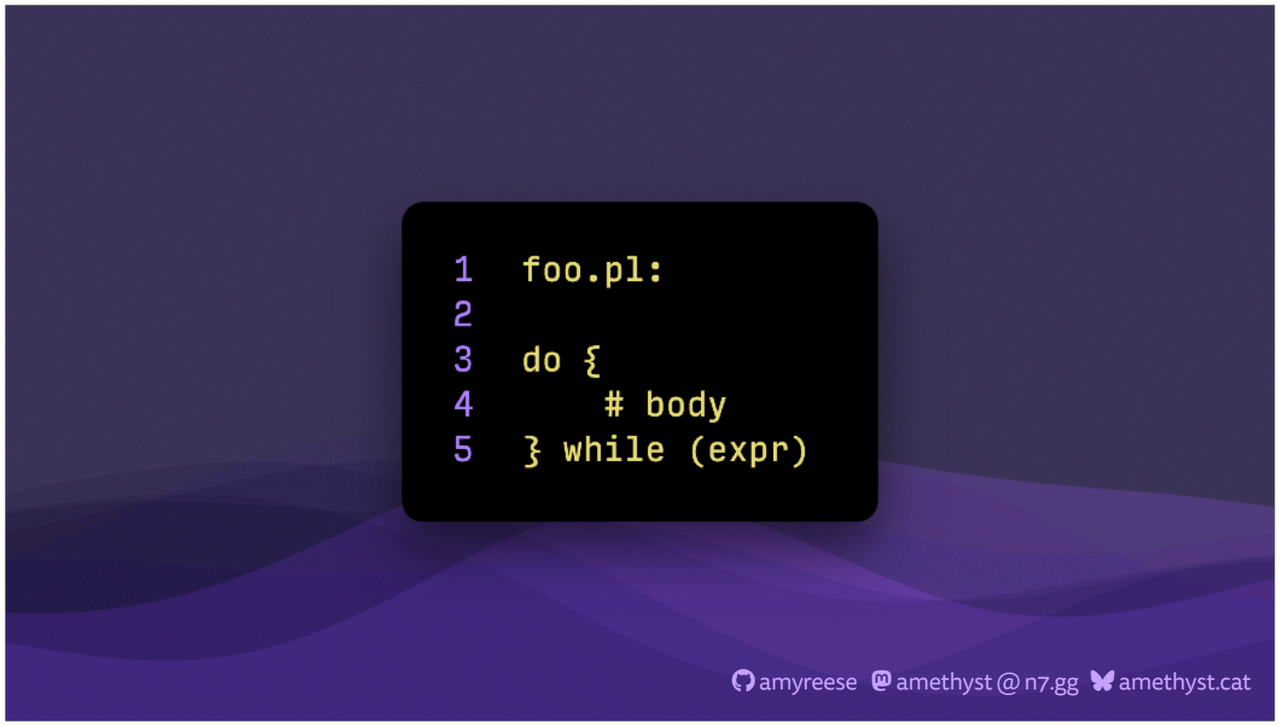
But THIS decorator doesn’t return a new function waiting to be called — it starts running immediately on definition!



Now we can write “until” loops, but in Python! The loop body will be called until the predicate — the lambda — is true. Now, yes, it’s a bit ugly, and we have to take care to use non-local state to persist data between iterations, and this is all just to satisfy nostalgia and avoid a negation in the loop condition. But this is a glimpse into what decorators can really do.

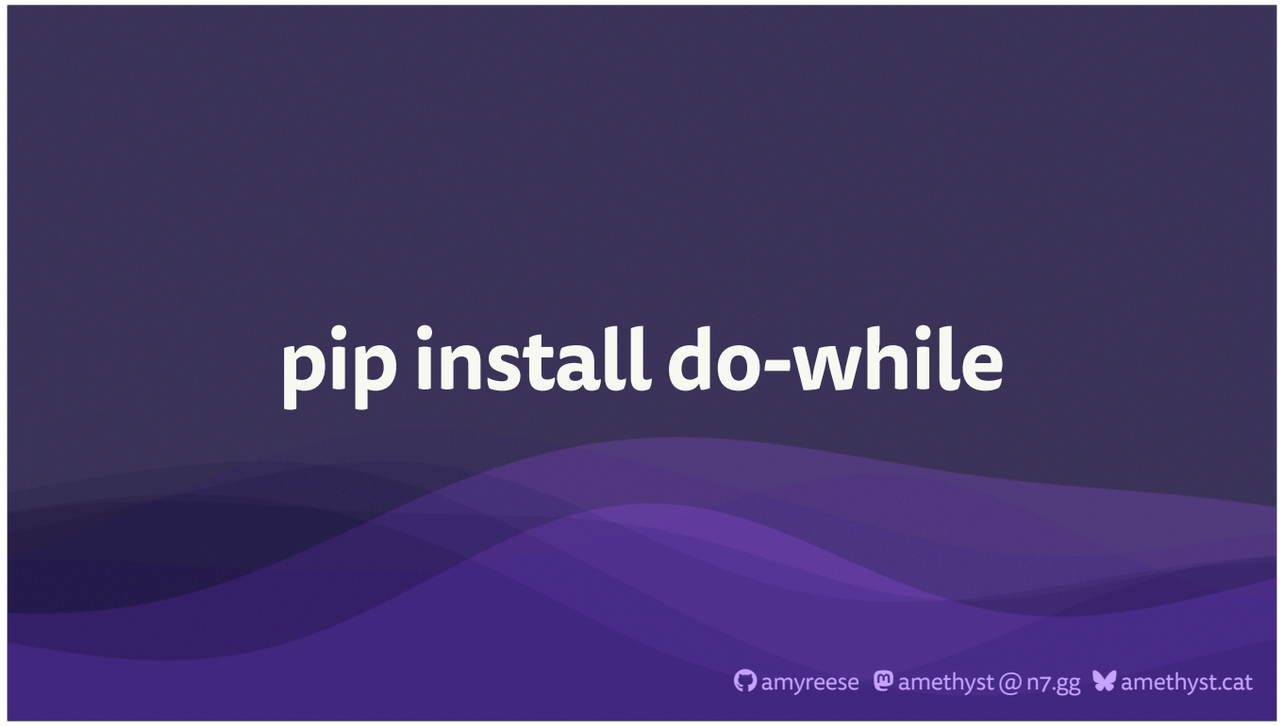


just wait ... <drink> ...

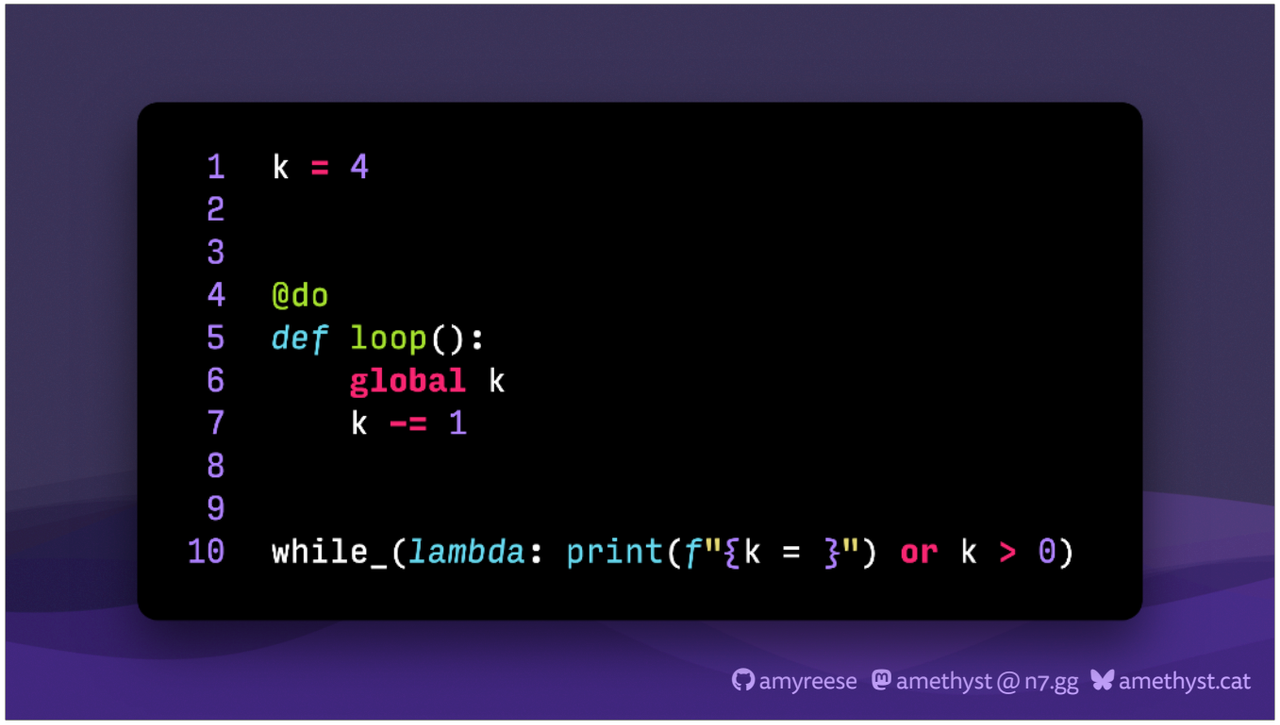


So Perl also has this gem, familiar to many C programmers. This is a do-while loop. It always executes the body at least once, *and then* it evaluates the loop condition before repeating. In certain cases, this is helpful, because it reduces duplication of setup code. Python has no native syntax for doing this.

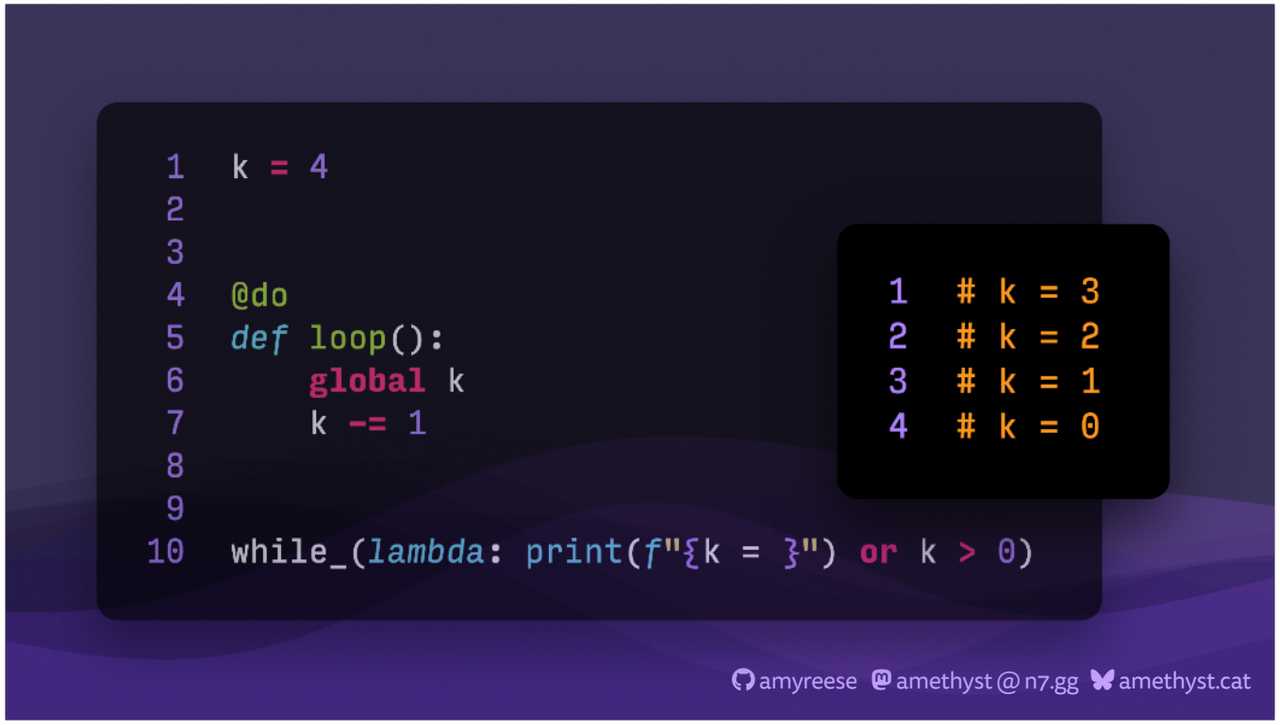
22 years ago, PEP 315 was proposed with new syntax to add a do-while loop to Python. For Python *2.5*. The pep was deferred for years, and finally rejected in 2013. I rediscovered it while writing this talk, and I think it’s finally time we do justice for PEP 315.



So I’m announcing the public release of the “do while” package on PyPI, available now. You better believe I did this for the bit. Let’s see it in action!

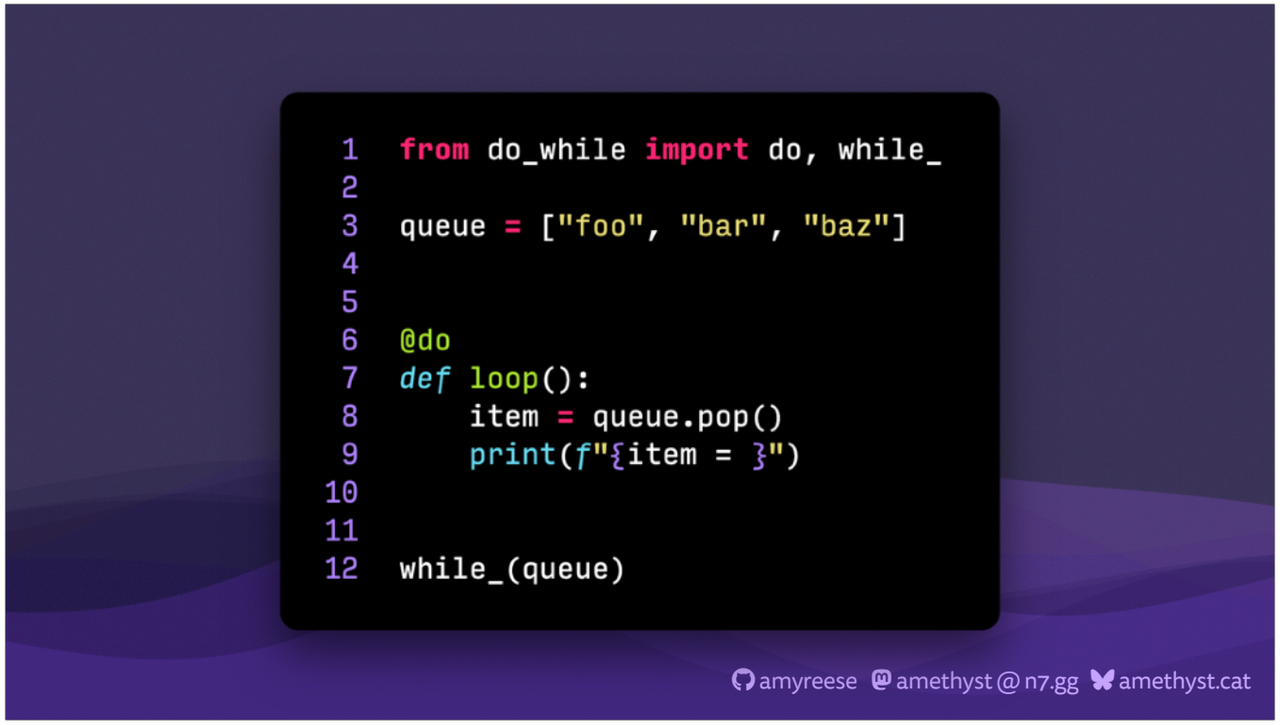


This new package lets us finally write a do-while loop in Python, with the “@do” decorator wrapping the body of the loop, followed by a call to the while function with the appropriate predicate. This predicate also prints the value so we can confirm when it’s being evaluated.

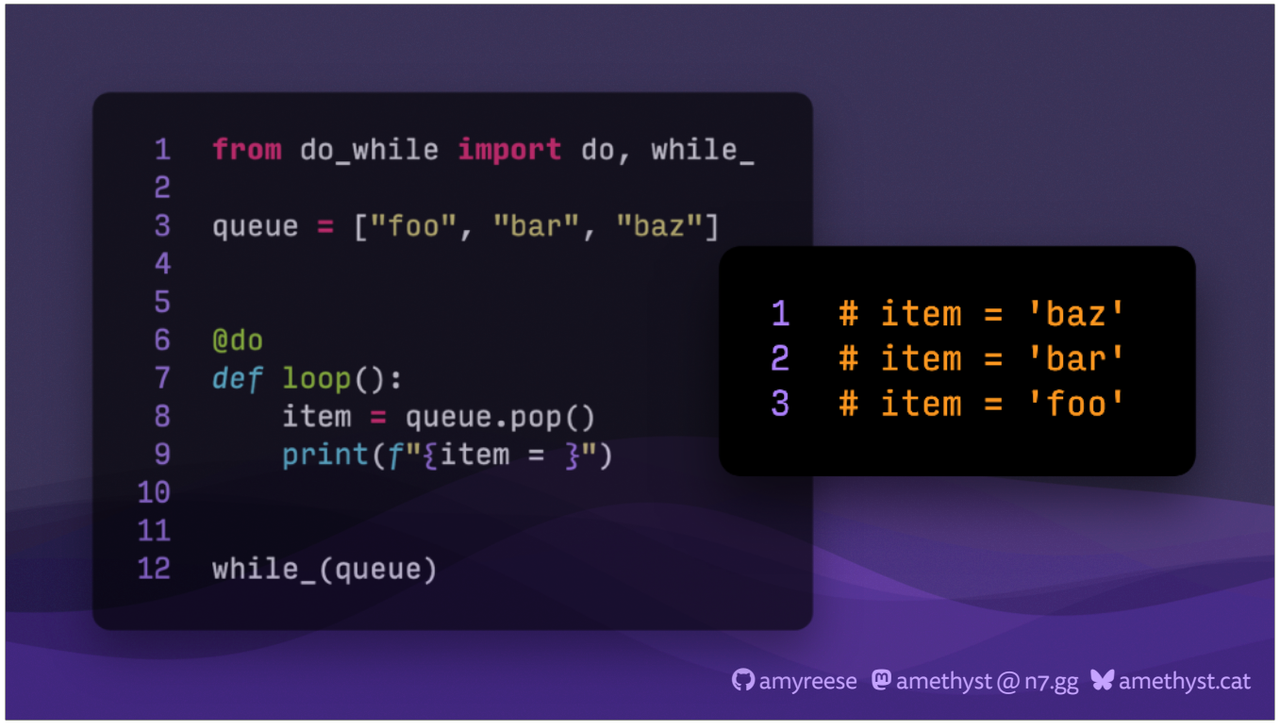


And when we run the code, we can confirm that the predicate is evaluated *after* the first iteration, thus ensuring the body is always run at least once, even if the predicate was always false. How does it work? It’s actually quite simple.

The real package has a bit more to it, but it can be boiled down to this fragment. The “do” decorator function doesn’t return anything, and it doesn’t run the loop. It merely creates and saves a wrapper, and then waits for the “while” function to be called. Only then does the wrapper execute, using the provided predicate to control the loop.



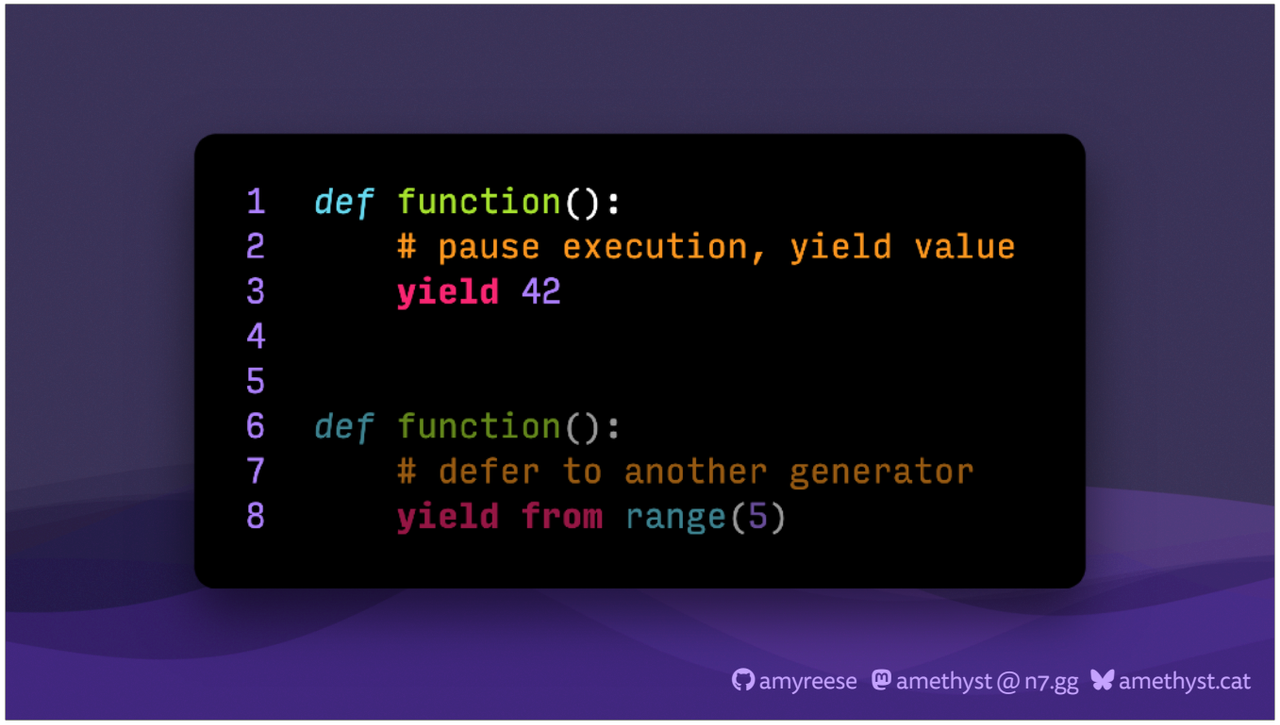
The release version also supports collections or other truthy objects as predicate, running the loop until those objects become falsey. This lets us clean up some uses of global or nonlocal, and get by without a lambda for the predicate. I think this is about as nice of a do-while loop as we can get, without modifying the Python runtime itself.



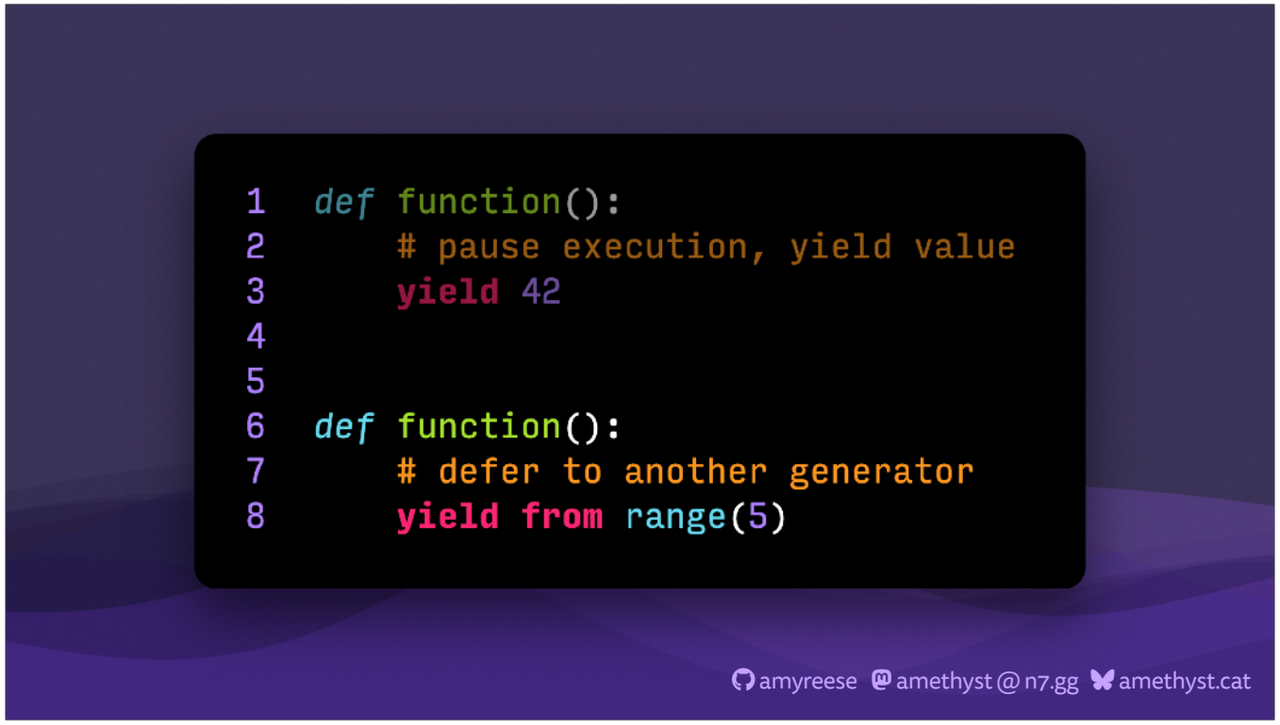
And again, we can see the loop working as we expect, with all of its post-evaluation glory. Great success!

It’s 2025, what did y’all expect? <drink> Ok, I get it. Let’s climb back out of the rabbit hole — decorators are passé. What we really want is control flow we can use for good, not evil. Something like ...

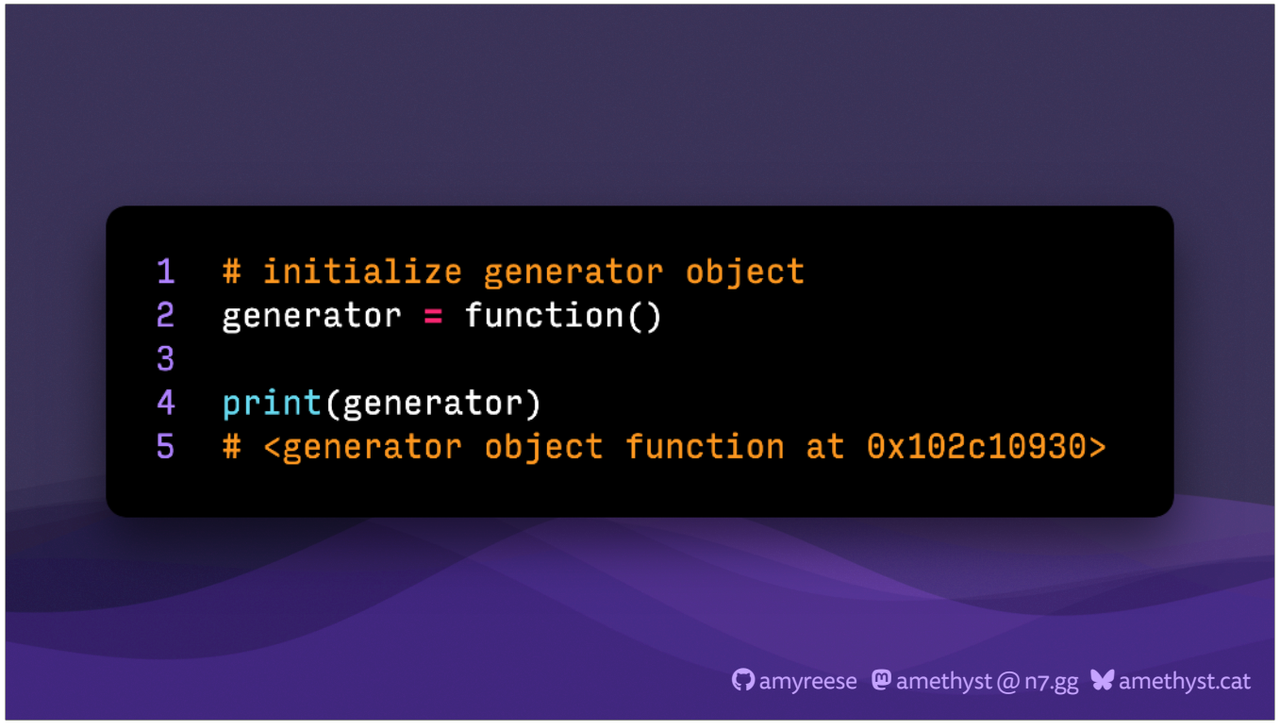
Generators! Generators are a special type of function that can “pause” execution, yield arbitrary values, and then resume from where they left off again.



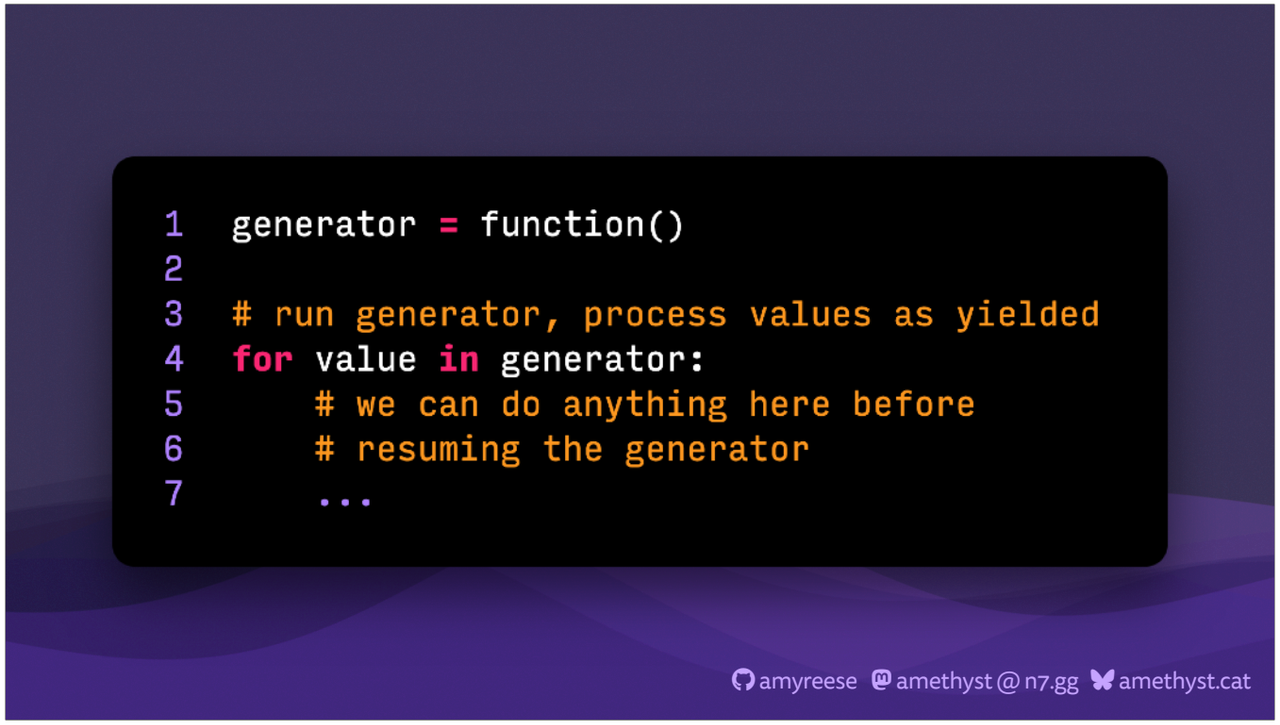
Writing any function with the “yield” keyword automatically turns it into a generator. When reaching a yield statement, the runtime will pause execution of the function, and pass that value back to whatever is consuming the generator.



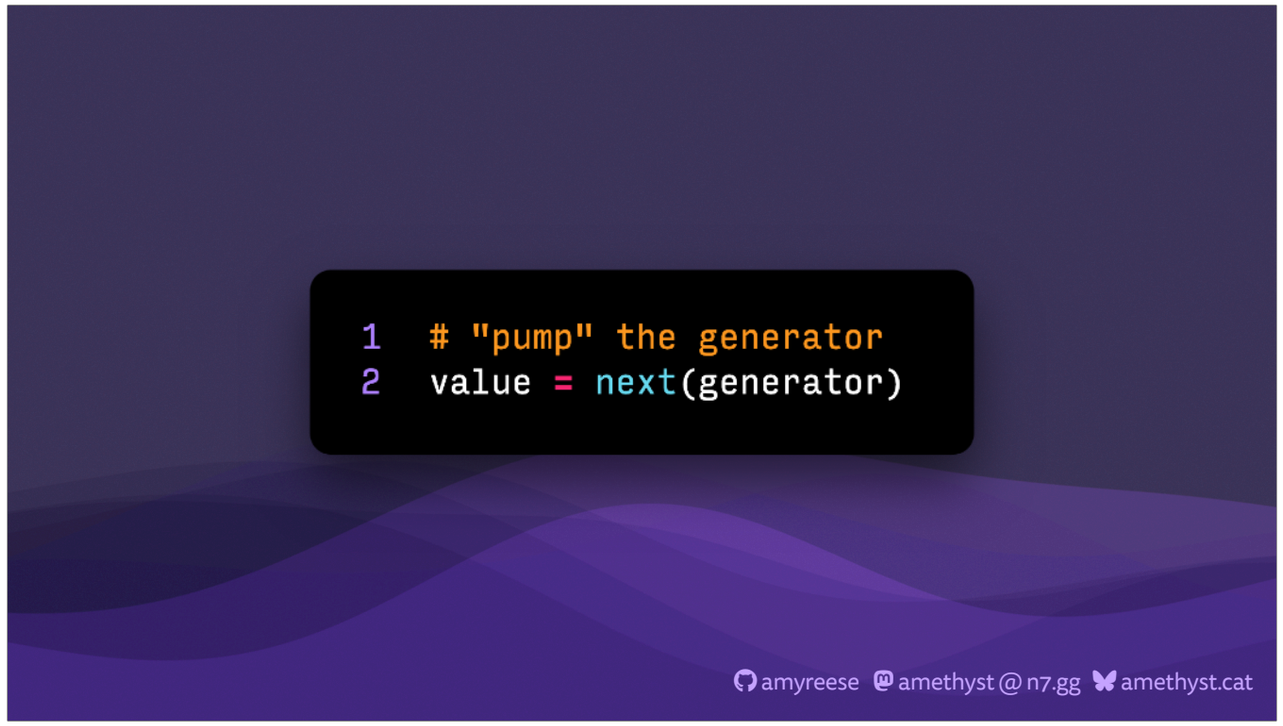
We can also use “yield from”, which will again pause our function, but now it will defer to an arbitrary iterable, yielding each of those values until it is exhausted, and only then resume our function afterwards.



But unlike normal functions, when we call a generator function, it doesn’t immediately start running. Instead, we get a “generator object”, and we need to do something with that object to make it start running and yield those values.

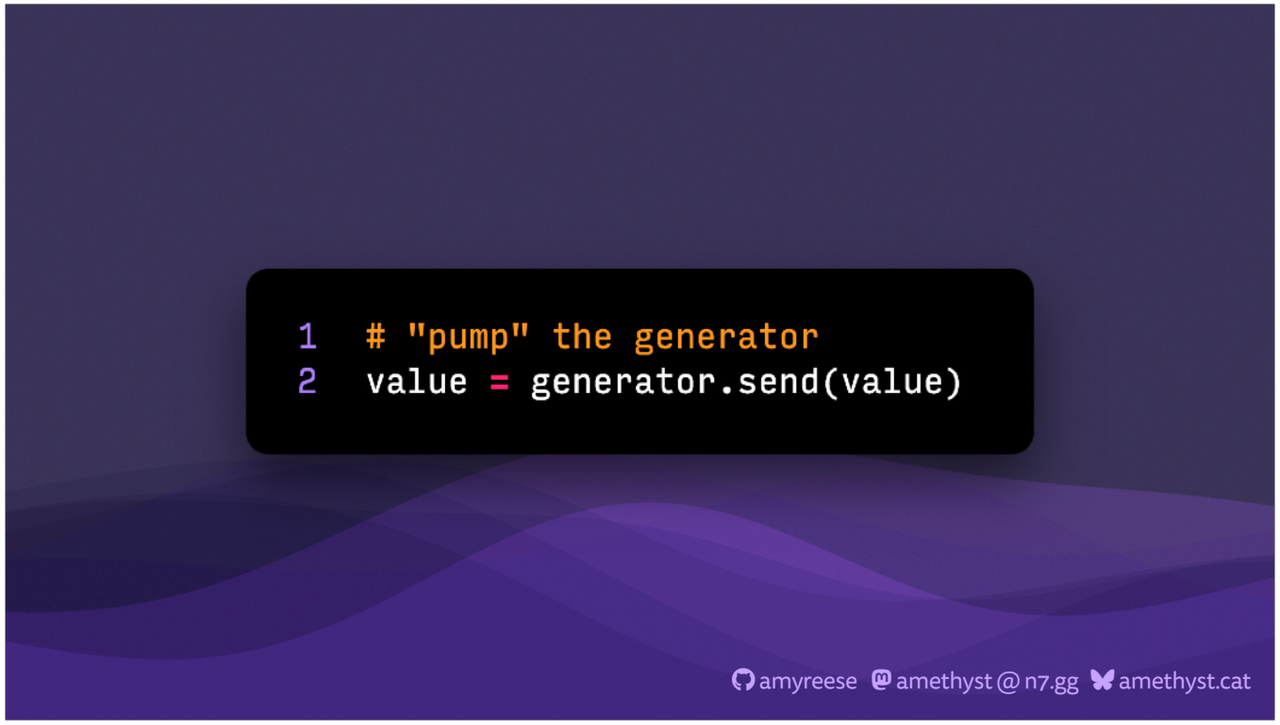


Generator objects are themselves iterable, so the easiest thing to do is to loop over them. For each iteration of the loop, it runs the generator until it either yields a value or finishes. Once a generator yields a value, we can do anything we want in the body of the loop with that value.

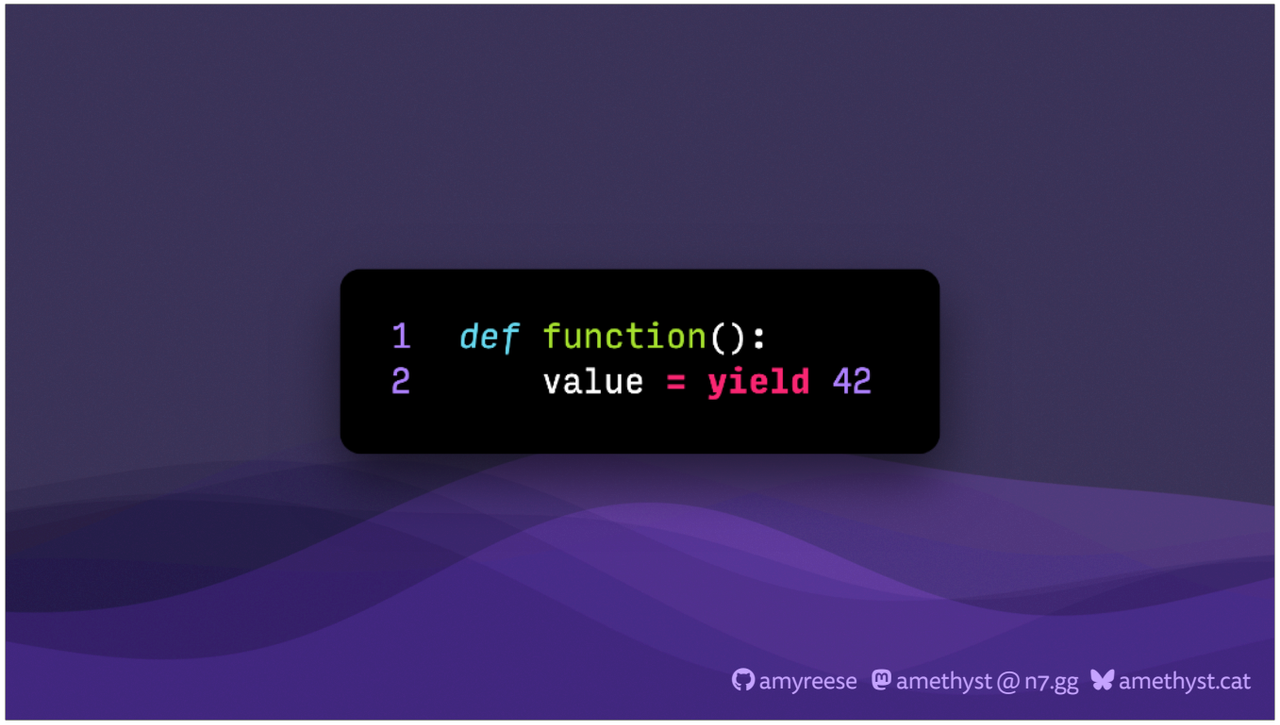


Alternately, we can call next on the generator object to run it until it yields a value, and again we are free to run arbitrary code until we’re ready to resume the generator.

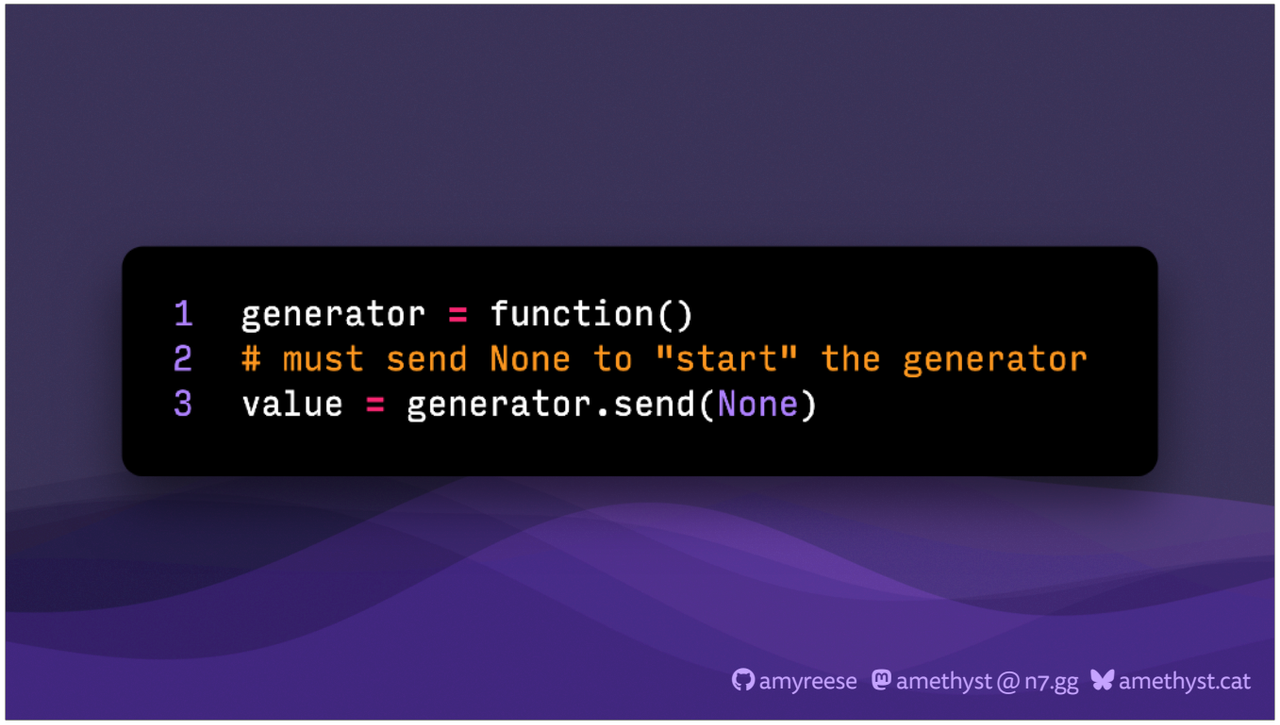
But perhaps unexpectedly, we can also send data *back in* to a running generator, in between the yielded values we receive from it. However, we need to stop treating generators like normal iterators in loops.



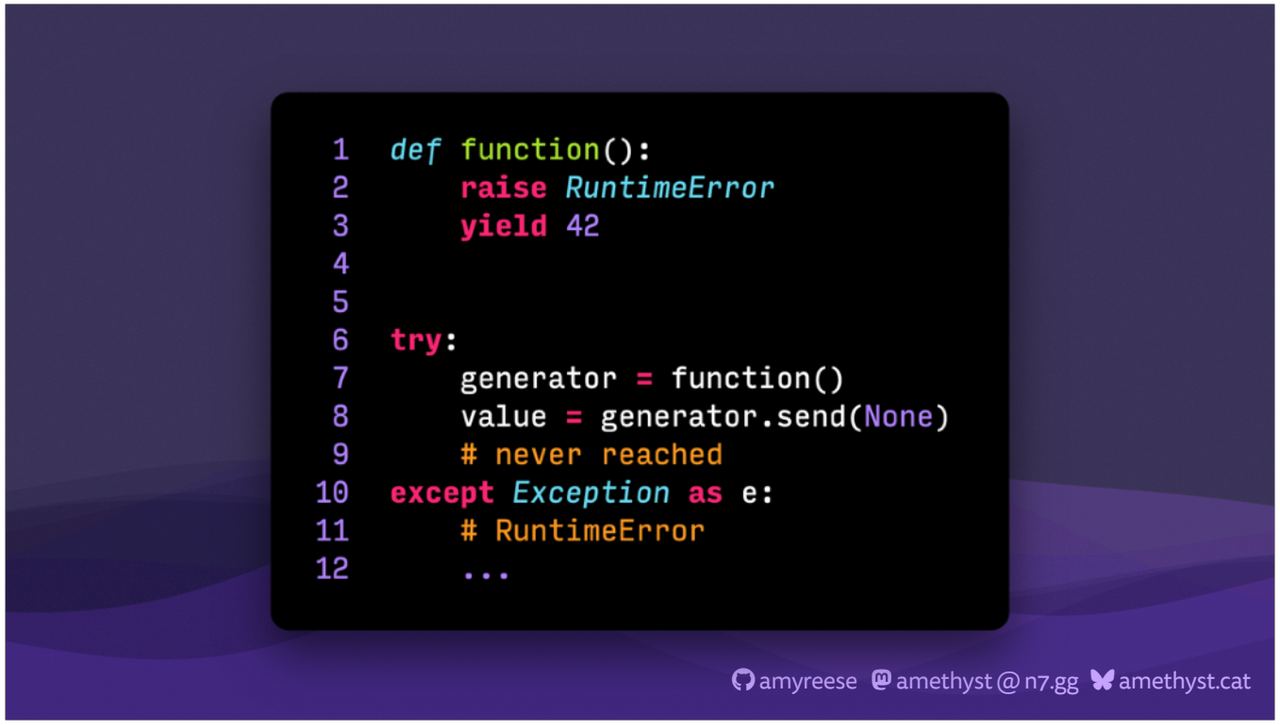
We can do this by calling the “send” method on a generator object. The value we pass will be sent into the generator, the generator function will run until it yields again, and that yielded value will be returned to us from the send method.



On the function side, values sent into the generator are presented as the “return value” of the yield statement itself.

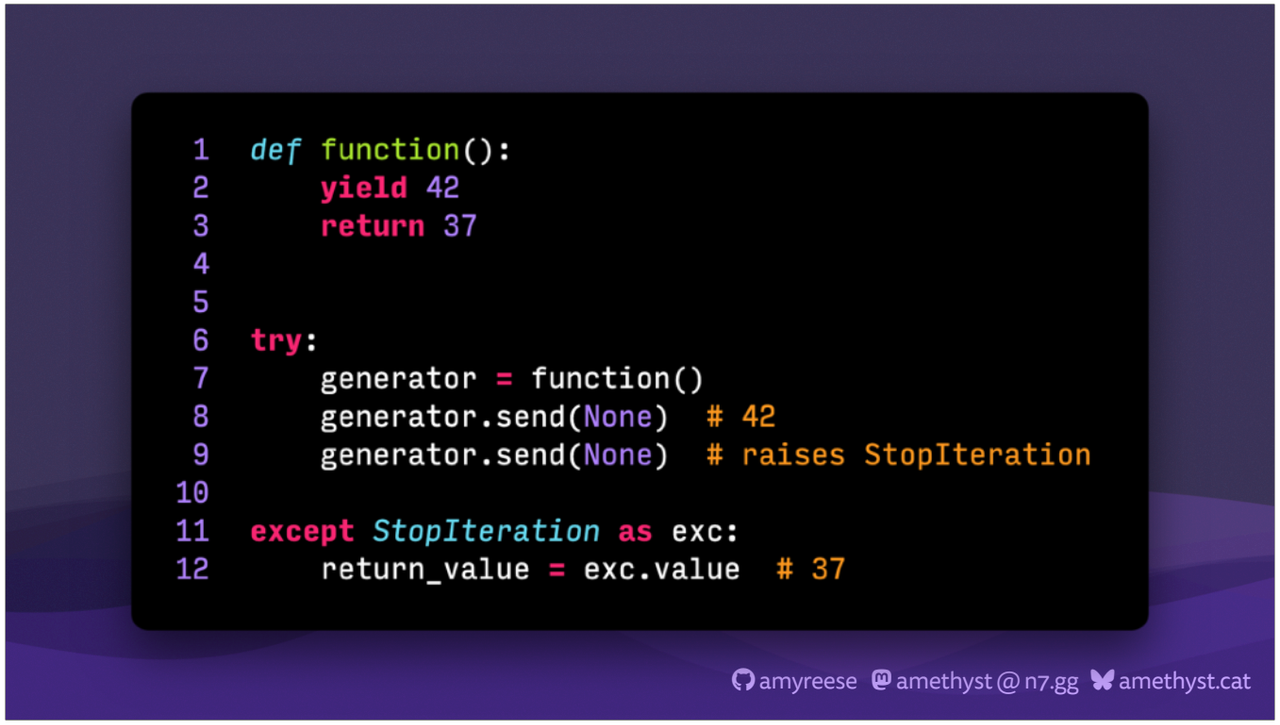


Remember that when we first call a generator object, the function hasn’t started executing, so there’s no current yield statement to receive a value. That means the first time we call the send method, we must send None. This is what actually starts execution of the generator function so that it can yield its first value.



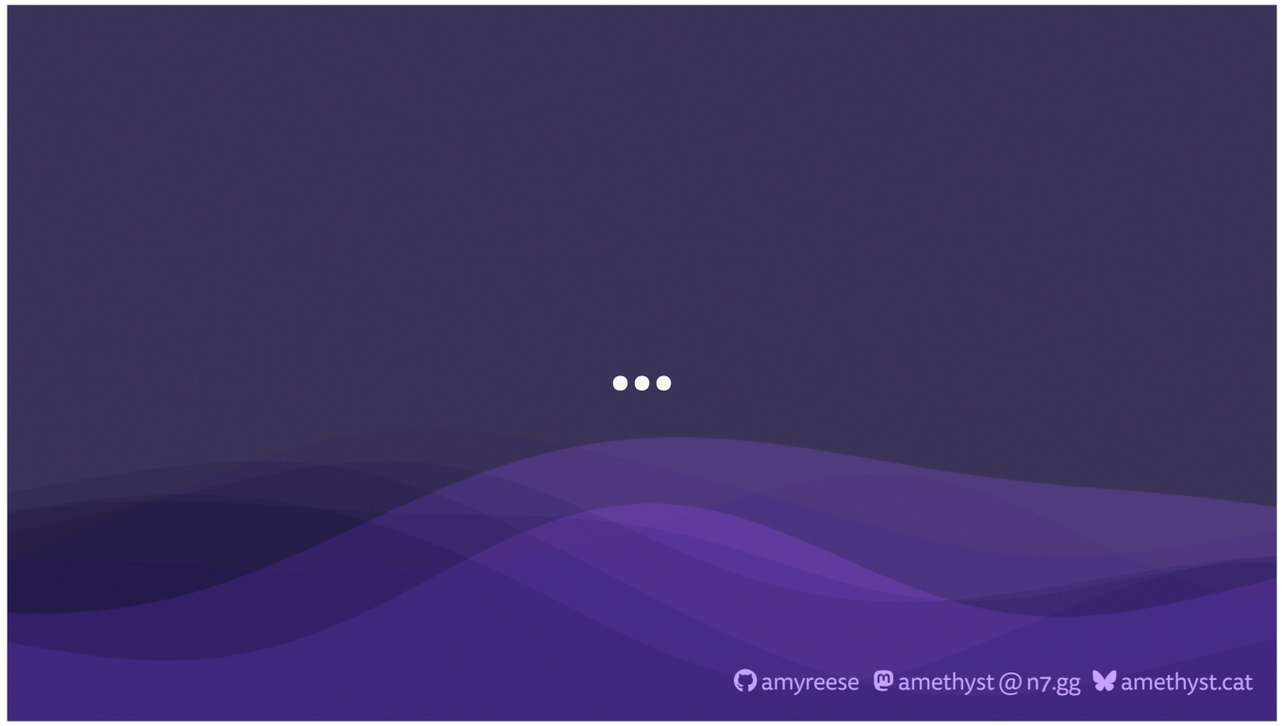
Now, if the generator raises an exception, that exception is also raised from the send method, and we can catch and handle that exception. But like sending values back into the generator, we can also inject exceptions into the generator.

The throw method will raise the given exception inside the generator function, triggering the normal exception handling process. But beware — if the generator doesn’t catch the exception, it will be raised right back out from our throw method. Otherwise, we get the next yielded value in return.

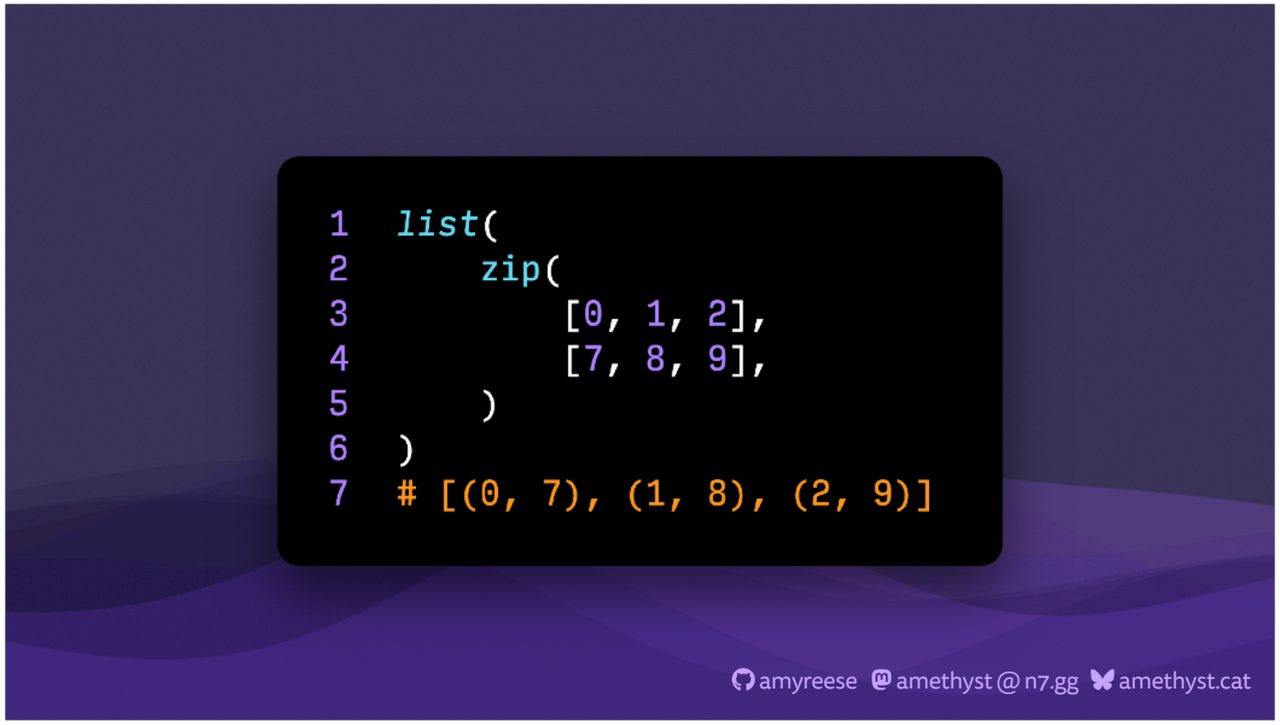


Lastly, when a generator function returns or finishes executing, it will raise a StopIteration exception, just like any other iterable. We see that exception raised when we call send enough times to reach the return statement, and the value from our generators return statement is available as a property on the exception object.

Altogether, this gives us bidirectional communication between generators and the code wrapping them. This forms the foundation for concurrent, interleaved execution of two or more functions, commonly known as coroutines. asyncio itself was originally built on top of generators, with event loops that round robin execution between multiple active coroutines...



But that’s a different rabbit hole for a different day, and I already gave that talk back in 2019. So what else can we do with this knowledge?



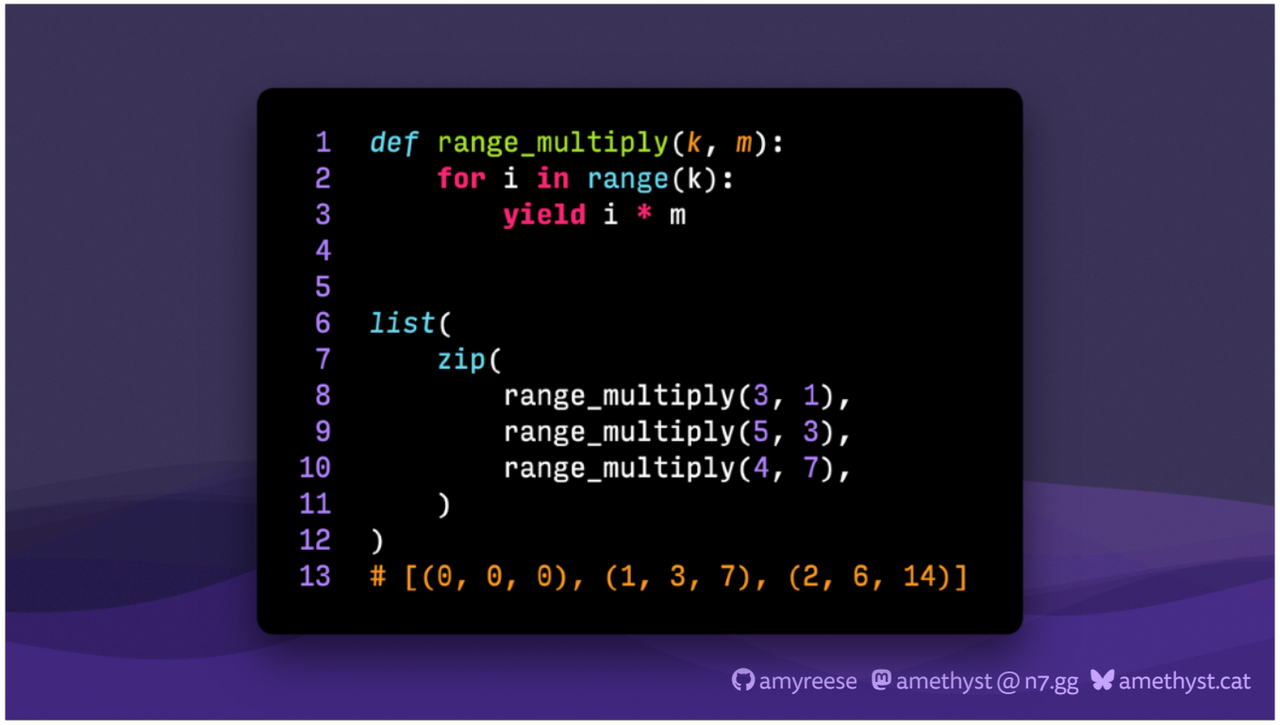
I’m sure many of you are familiar with, or have at least seen, the zip function before. It takes one or more iterables, and produces tuples combining the values from each step of those iterables. We can build a version specifically for generators.

```python
def zip(*generators):
    while True:
        try:
            # yield each group of values
            values = [gen.send(None) for gen in generators]
            yield tuple(values)

        except StopIteration:
            # handle unfinished generators
            values = [gen.close() for gen in generators]
            return tuple(values)
```

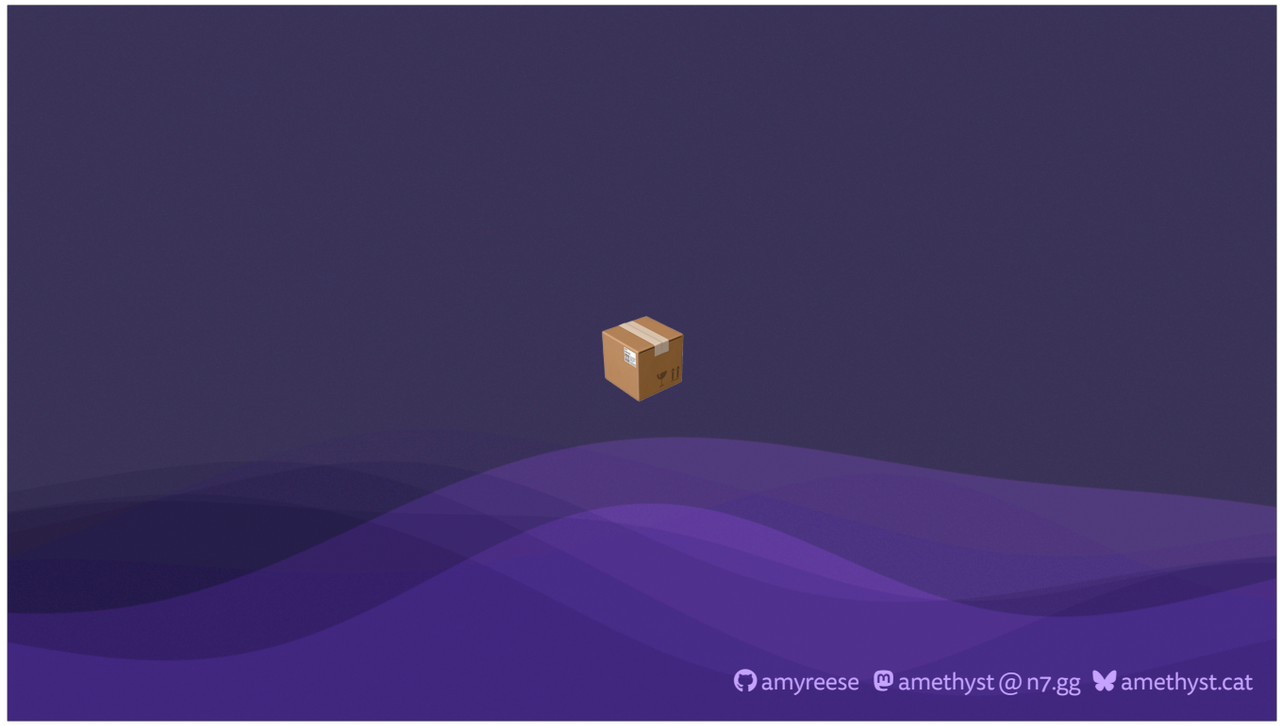

Our version of zip can be relatively simple. We take any number of generators, and call send on all of them, and yield a tuple with the values gathered from each of those generators.

```python
def zip(*generators):
    while True:
        try:
            # yield each group of values
            values = [gen.send(None) for gen in generators]
            yield tuple(values)

        except StopIteration:
            # handle unfinished generators
            values = [gen.close() for gen in generators]
            return tuple(values)
```

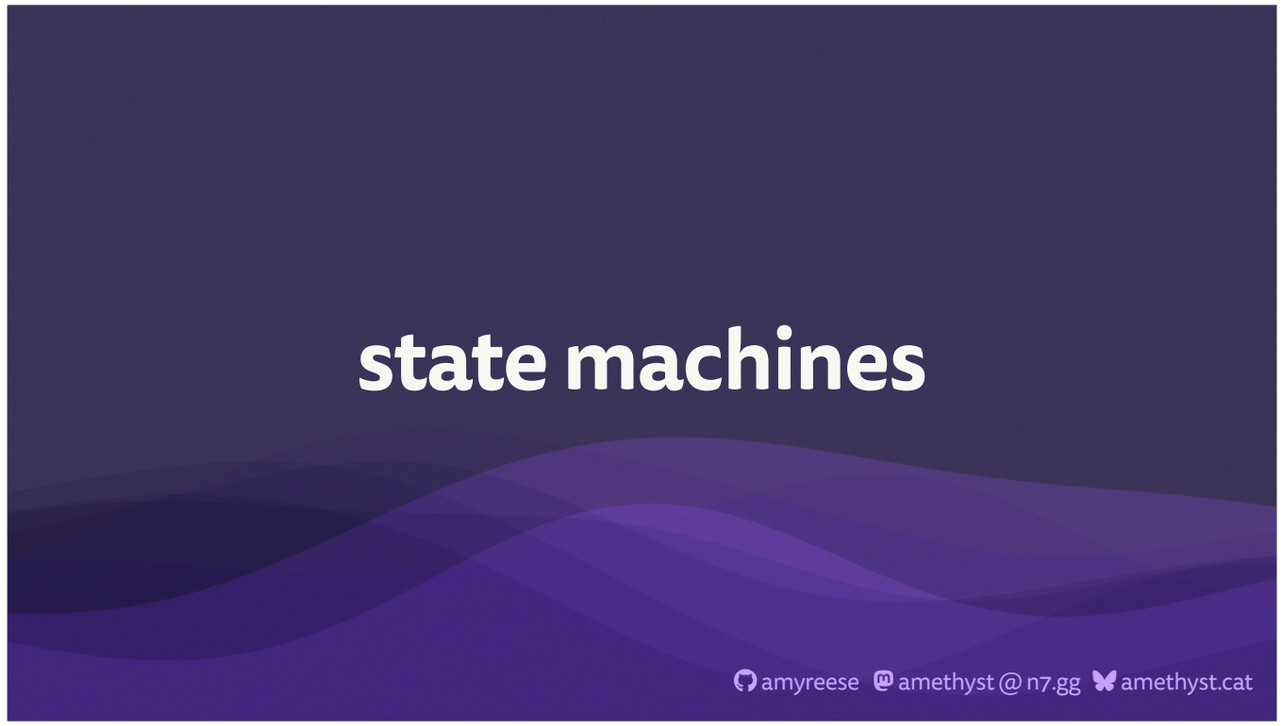

We repeat that until one of the generators finishes and raises a StopIteration error. At that point, we clean up any unfinished generators by calling “close” on all of them, ensuring that each generator has the chance to cleanly exit.



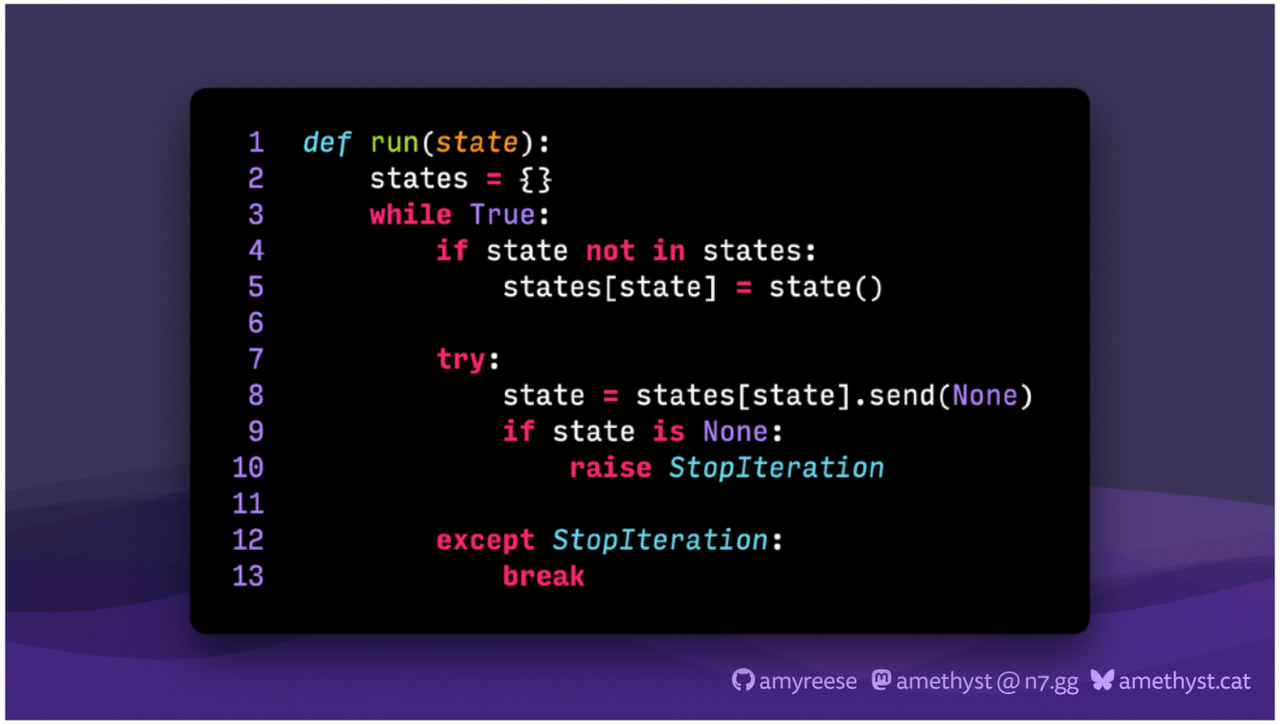
We can use that with our toy generator function here, and see that it generates tuples accordingly, and also stops correctly once the shortest generator is completed. It’s not really groundbreaking or exciting though.



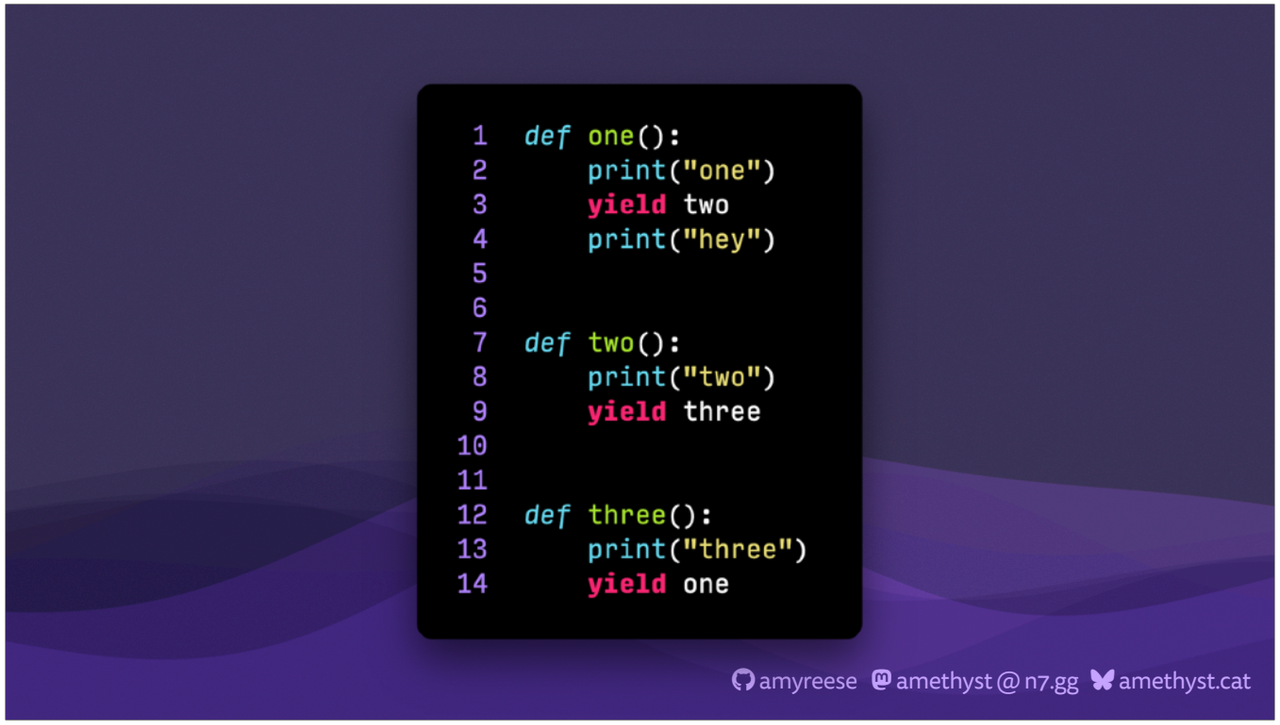
So let’s try to think outside the box. We know how to start generators, and we can switch execution between multiple concurrent generators....



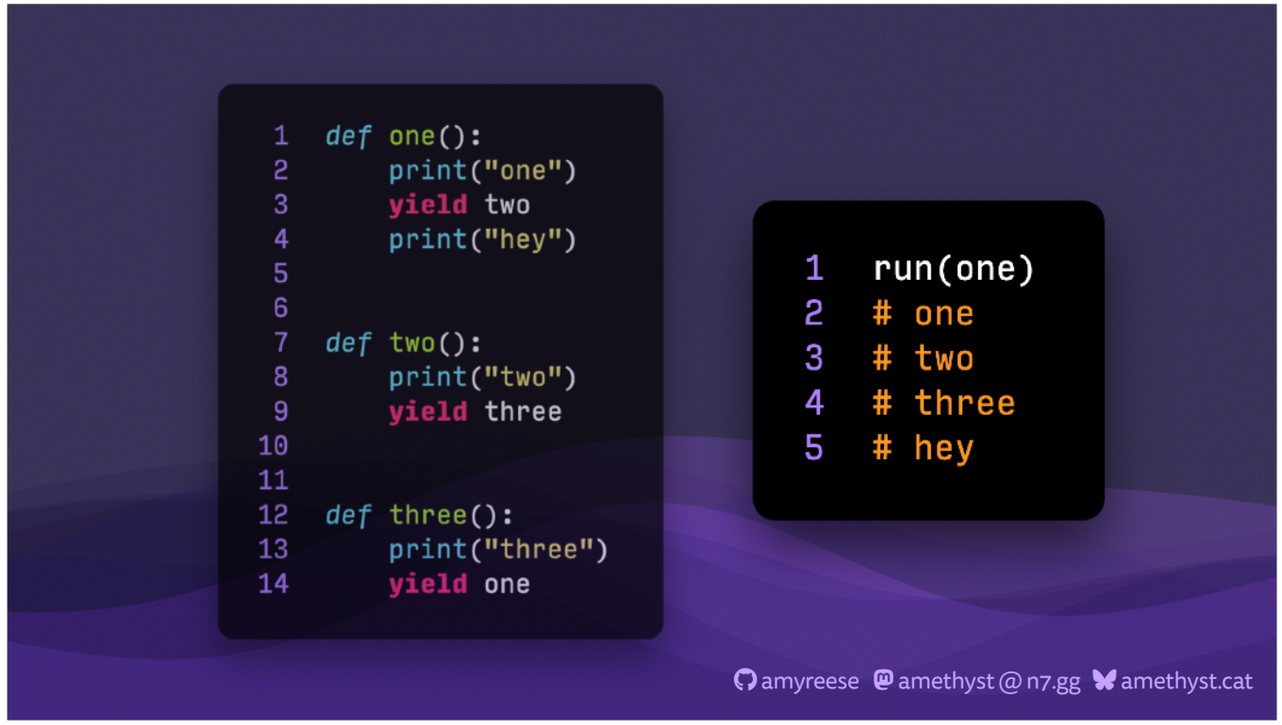
What about a state machine? We could have each “state” be a separate generator function, and use the yield statement to transition from one state to another.



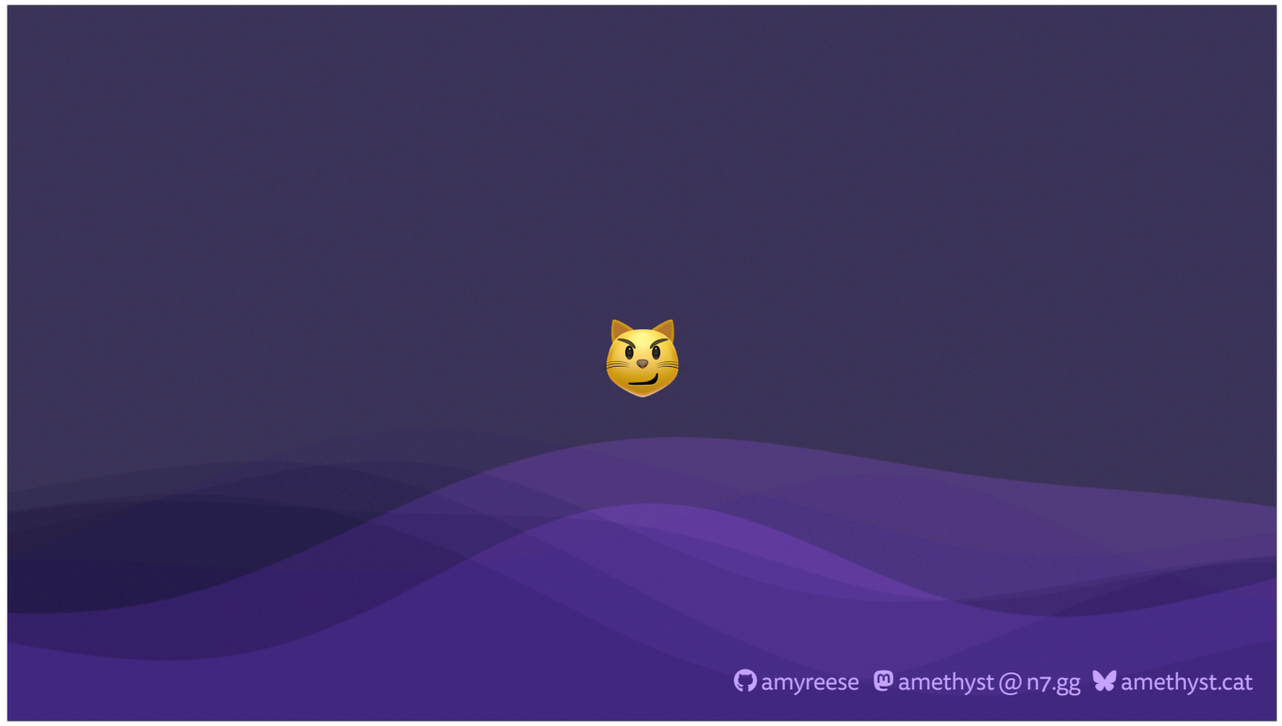
We can even build a barebones state machine in a single function. Given a starting state, we initialize generator functions as-needed, running the current state until it yields a new one, and then repeat. We could also implement message passing between states, but we don’t need that yet.



This lets us write simple implementations for each state as individual generator functions that yield the name of the function to transition to. And when we transition back to an active state, it continues running exactly where it left off.



And when we start our state machine from state one, we can see it transition to each new state, coming back around to state one, resuming its execution, and then eventually our program ends once a function returns without yielding a new transition.



<drink> Now let’s do something *fun* with this... Introducing...

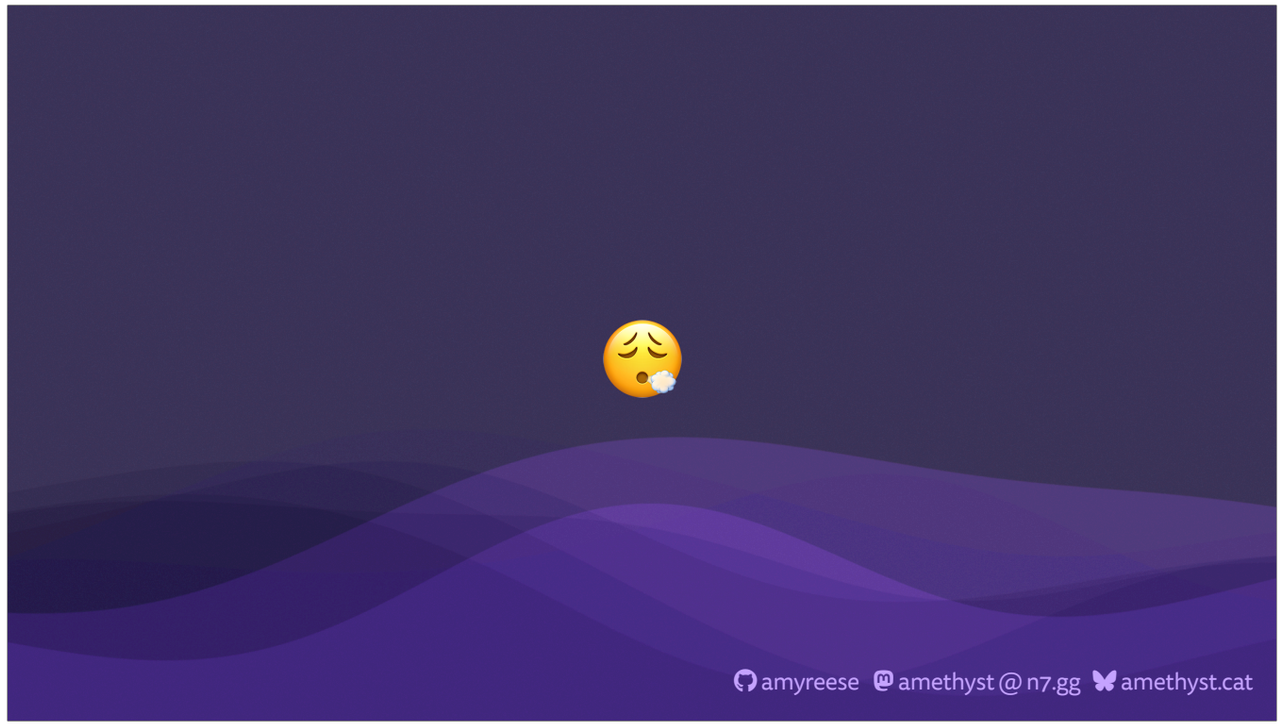
... the Great Python Control Flow Text Adventure! Text adventures were some of the earliest PC games ever made, and many are just a collection of rooms and actions combined into one giant state machine. Perfect for our toy project!

```python
@repeat
def closet(game):
    choice = prompt(
        "You find yourself in a closet. It is dark.",
        ["east", "light"],
    )

    if choice == "east":
        yield hallway

    elif choice == "light":
        print("You pull the chain. The bulb must be burned out.")
```


We can build a simple text adventure using our generator state machine, with a few small changes. Each room can be its own state; we can use a shared “game” object with data that we track across rooms; and the logic within a room’s function can interact with the user, change the game state, or yield a different function to transition the player from one room to another.

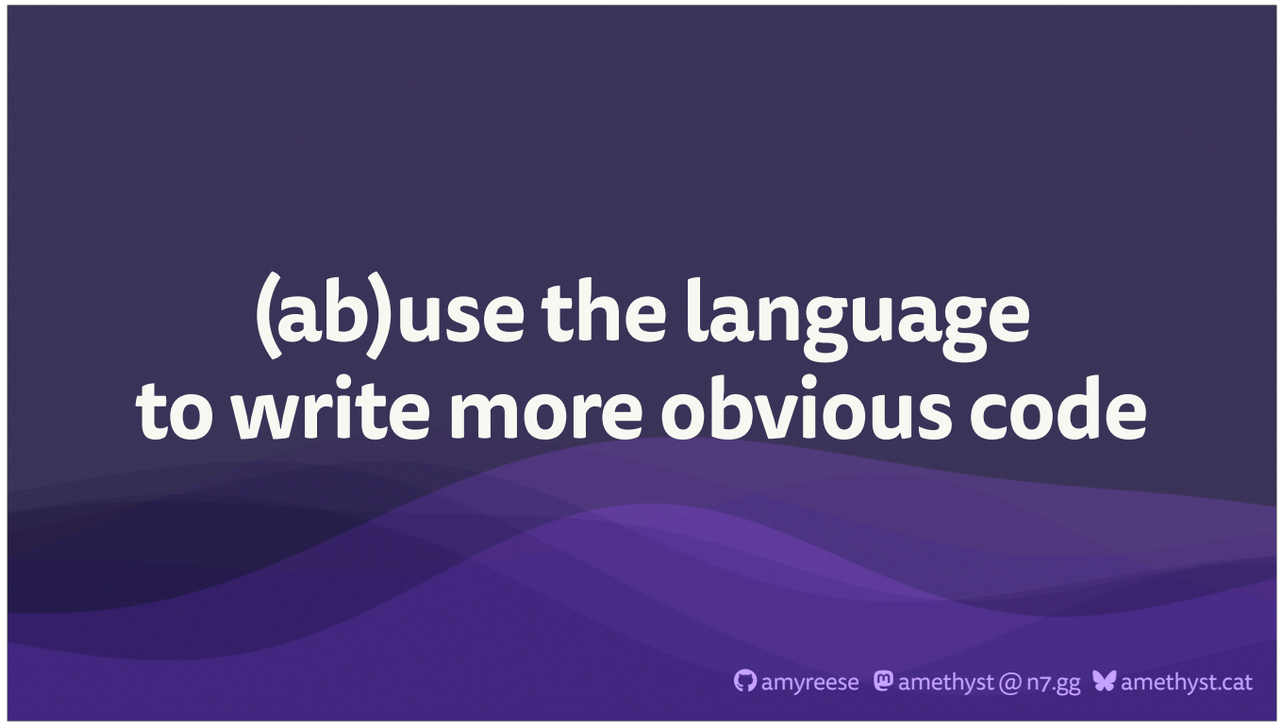
So let’s play our game! If you have UV installed, you can play along from your own machine easily with this command. The URL simply redirects to the latest version from my github repo. <play demo>

For those wondering, the full “game engine” for our text adventure weighs in at a hefty 28 lines of code, or about double that including doc strings and whitespace. And the entire file, with the engine and generator functions for all rooms is just under 200 lines in total.



Ok, we're running out of time, so where do we go from here?

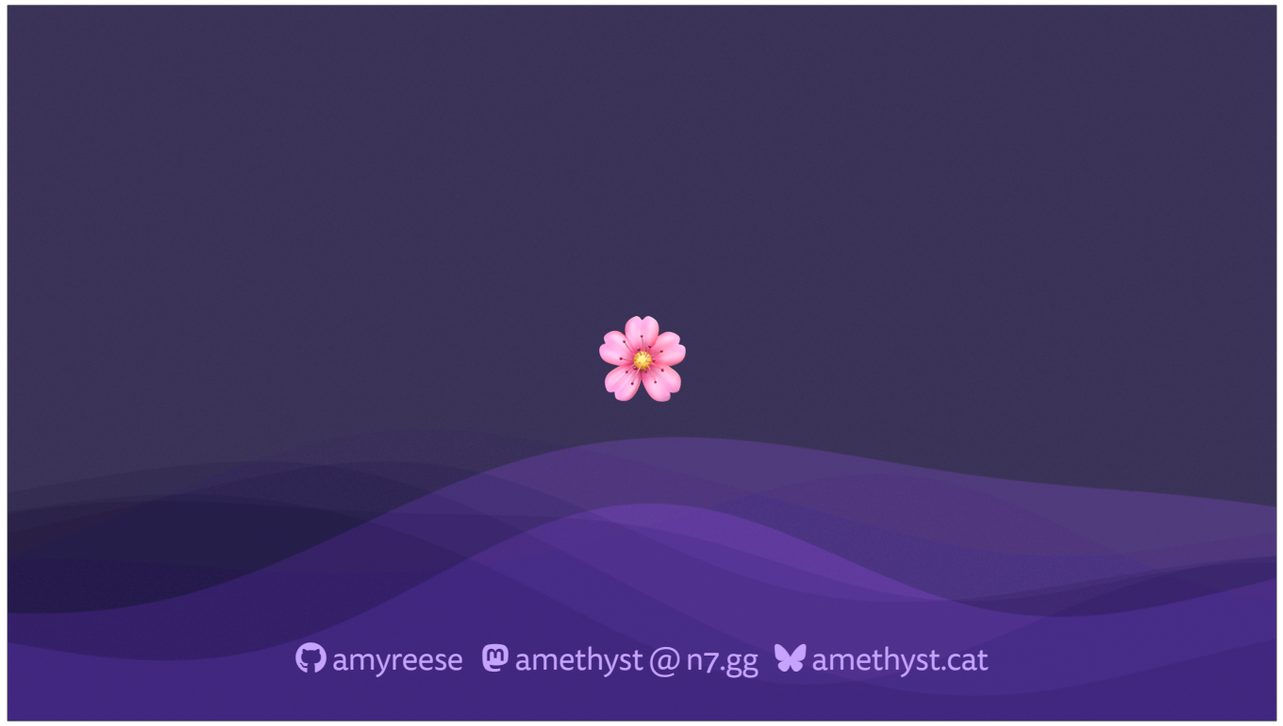
If there’s one lesson here, it’s that you should always write *obvious* code. Clear is better than clever. Everyone who has to read your code in the future, including you, will appreciate obvious code. Often, this takes more time than just writing code that works — take that time.



Think about ways that you can use — or abuse — your language to make your code more obvious. Make the important code clear and concise and bullet proof.

Just don’t jump the shark. Maybe think twice before writing your loops with decorators. Above all else, have some compassion for your coworkers and your future self.

As promised, the slides and example code, including the text adventure game, are available on my github repo. This also includes all of my past talks, so if you really do want to dive into python grammar, syntax trees, or the nerdy details of coroutines and asyncio, then there's something here for you!



Thank you all for sticking around. There’s some great lightning talks lined up in hall B after this. Hope you all have a great evening!!