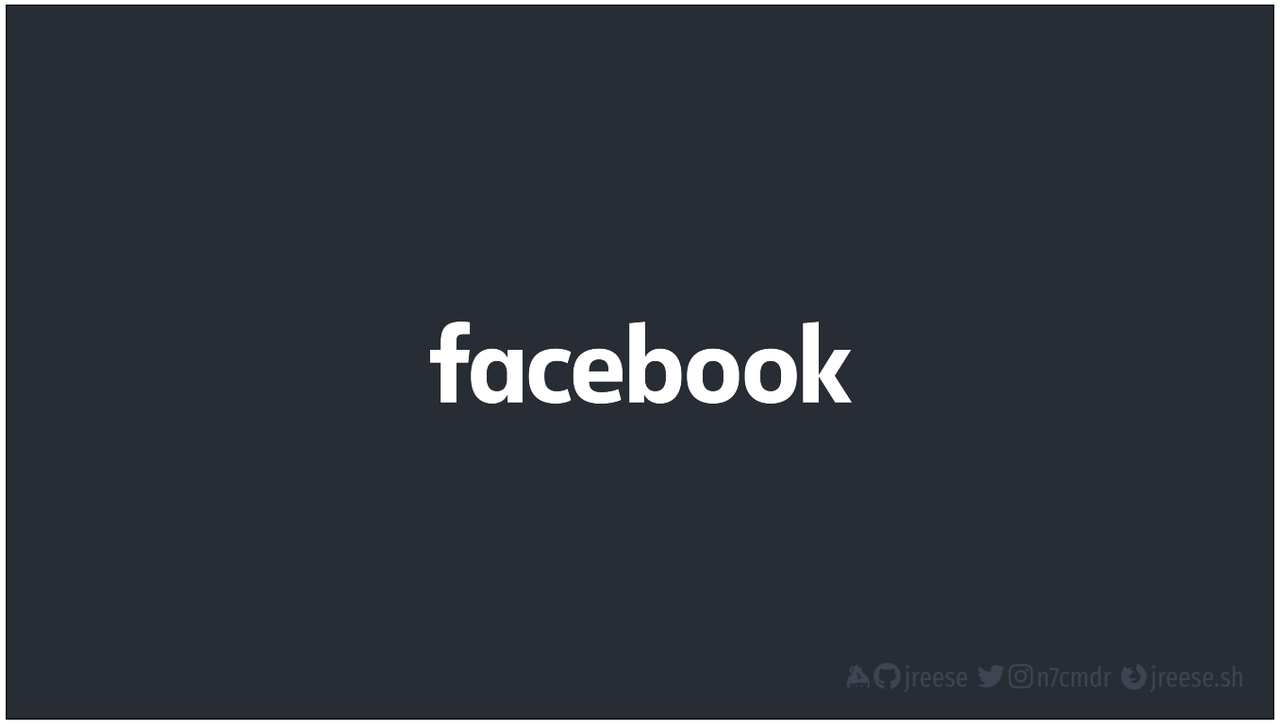
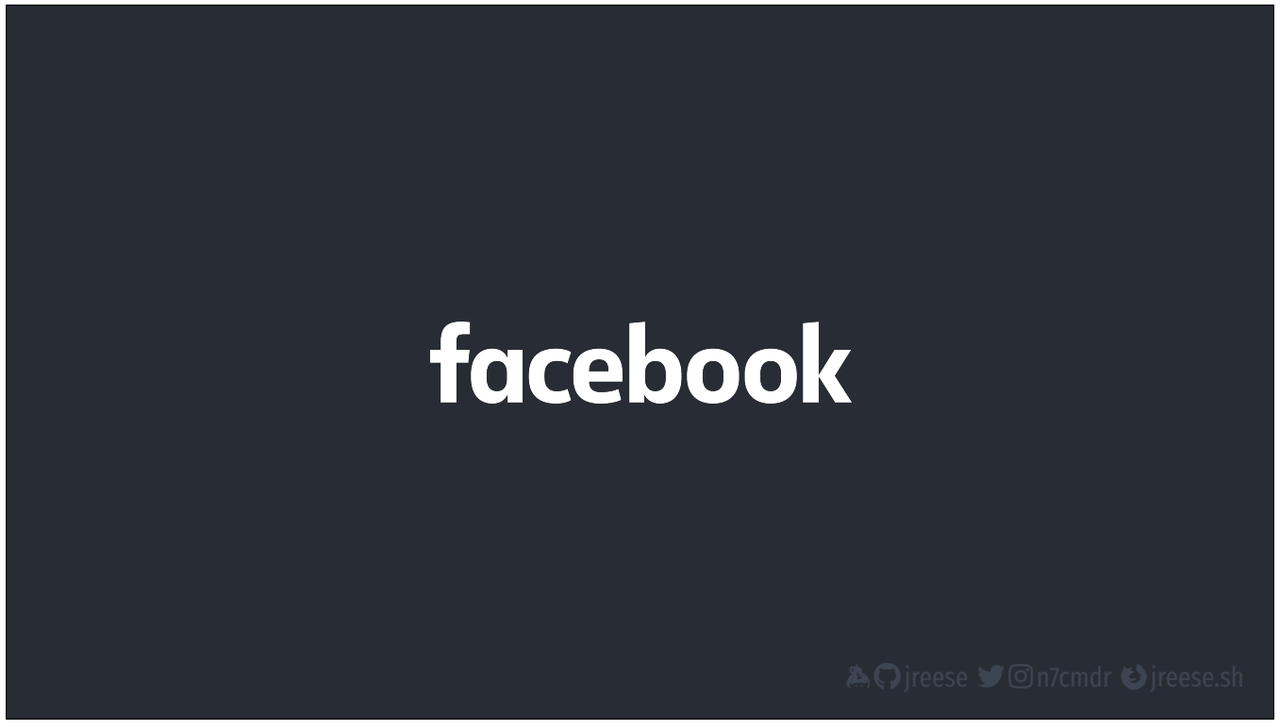
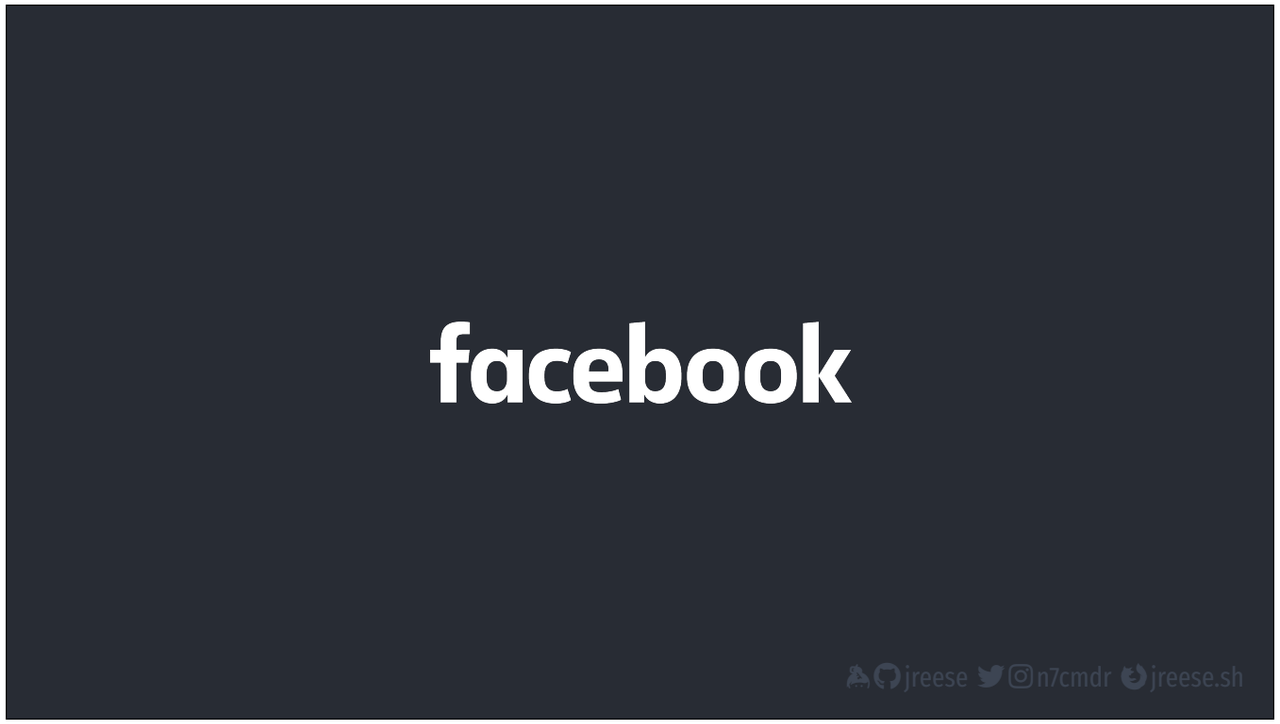


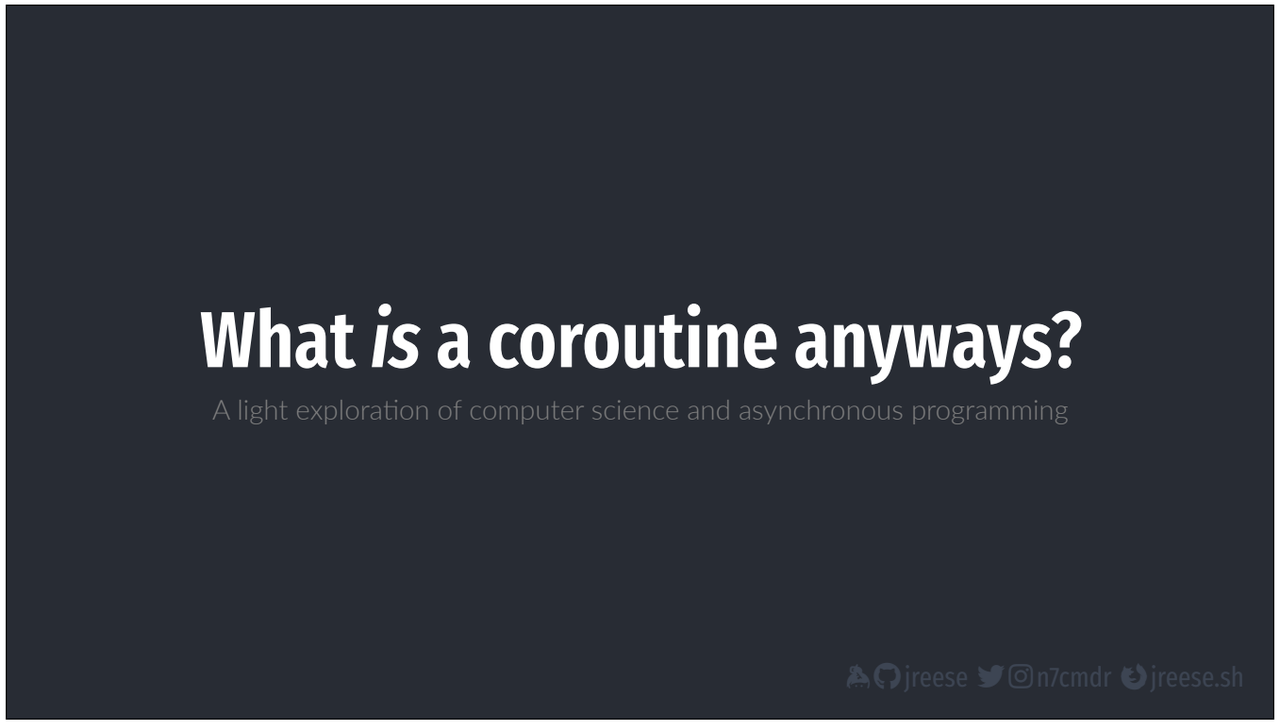
I’m John Reese
I'm a Production Engineer at Facebook
I work alongside Lisa, who you all met yesterday morning, as a member of our internal Python Foundation team



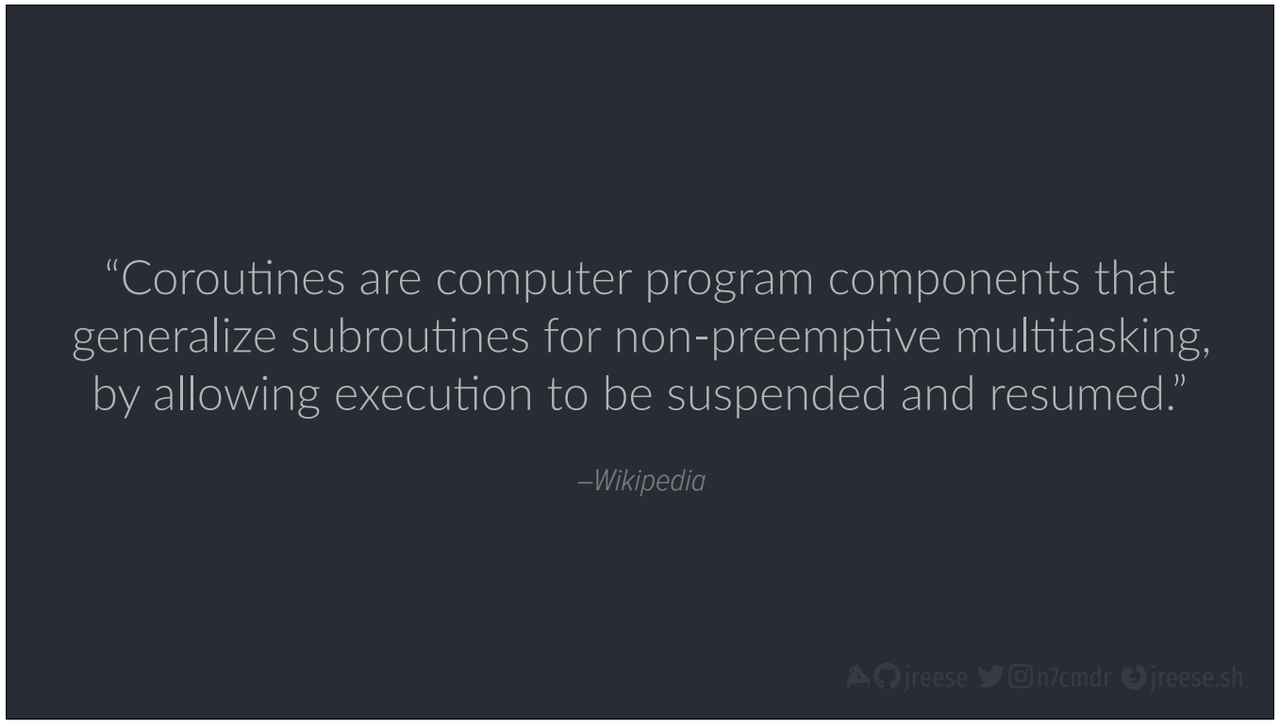
Ten years ago, I got a software engineering degree from a school that was way too expensive,
in the part of upstate New York that is way too cold and has way too much snow.
So naturally, I like to ask questions. Like, "why did I go here" and "I still owe how much?!" Anyways...



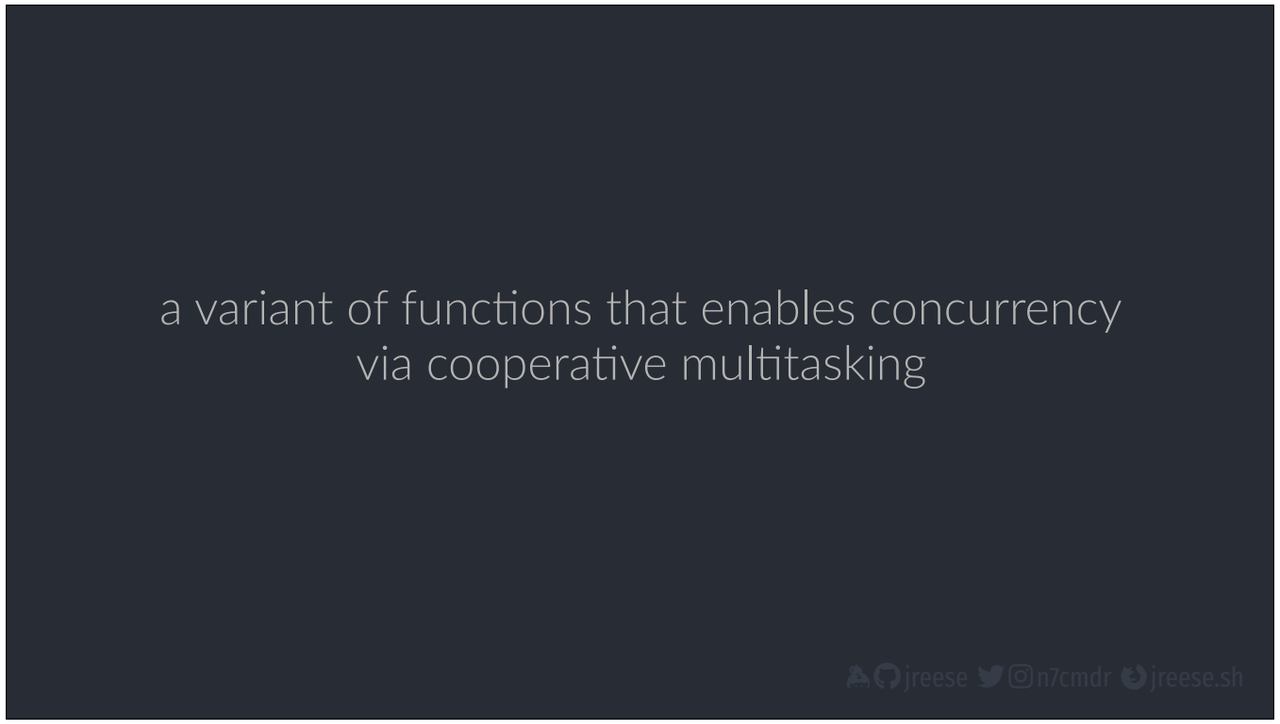
One of the reasons I love Python is how easy it is to answer questions about how things work, although the why is sometimes still beyond me.
Now, coroutines are the foundation for asynchronous programming in Python, and a fundamental building block of the AsyncIO framework.
But there can be a lot of mystery around them...



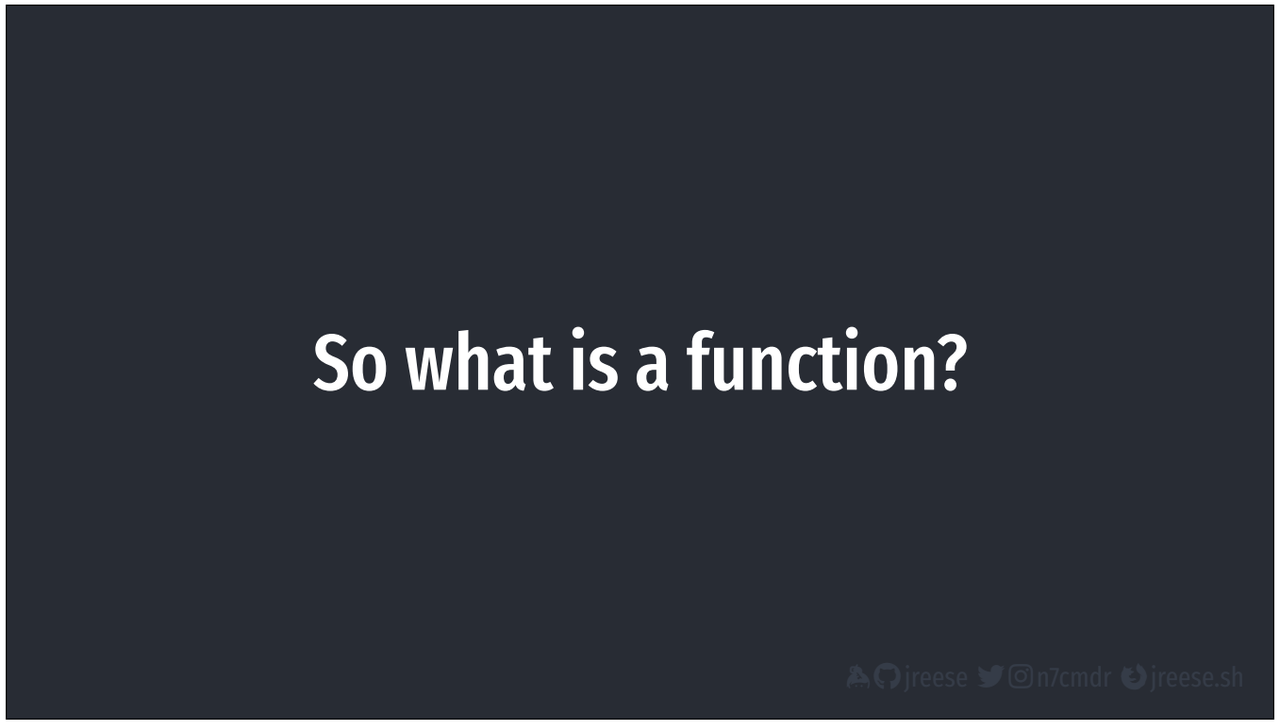
What *is* a coroutine? How do they work? Why are they so important?
I'm going to go ahead and spoil the answer here a bit,
and see what the source of all truth and knowledge has to say…



<read quote>
For the robots in the audience, this is excellent!
For the rest of us, this is an impenetrable answer.
Let’s try to break this down in something a human can understand.



A variant of functions … that enables concurrency … via cooperative multitasking
I want to explore the genesis of coroutines, and understand how they work and why we use them.
There are a lot of important concepts to cover, so I'm going to focus on the ideas,
and how they relate to Python, but I'm not going to worry about getting the details 100% correct.



So let's start from the beginning. What is a function?
I'll spare you the one true definition this time, because we're all humans here today.
Or maybe the skull-bot is still in the audience? I don't know.

a sequence of instructions that
takes inputs and returns outputs

Either way, a function is a sequence of instructions ...
that takes inputs ... and returns outputs.

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


In pretty much any useful programming language, functions are a fundamental building block
They let you organize code into reusable components that take inputs, do something, and yield outputs

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


When we execute the main function, the runtime steps through each statement …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


When it gets to a function call, the runtime “pauses” the current function’s execution …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


… sends the inputs to the function being called …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

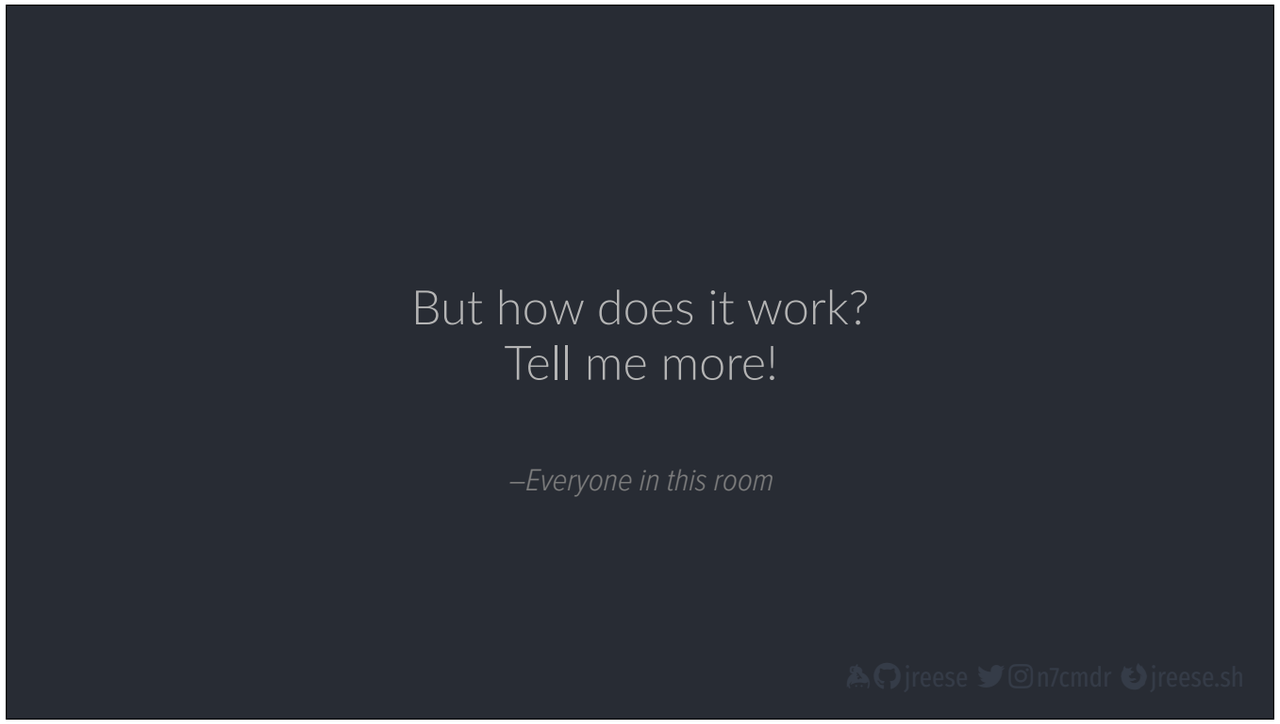

… executes the body of that function …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


… then sends the outputs back to the caller …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```


… and resumes execution of the original function where it left off.



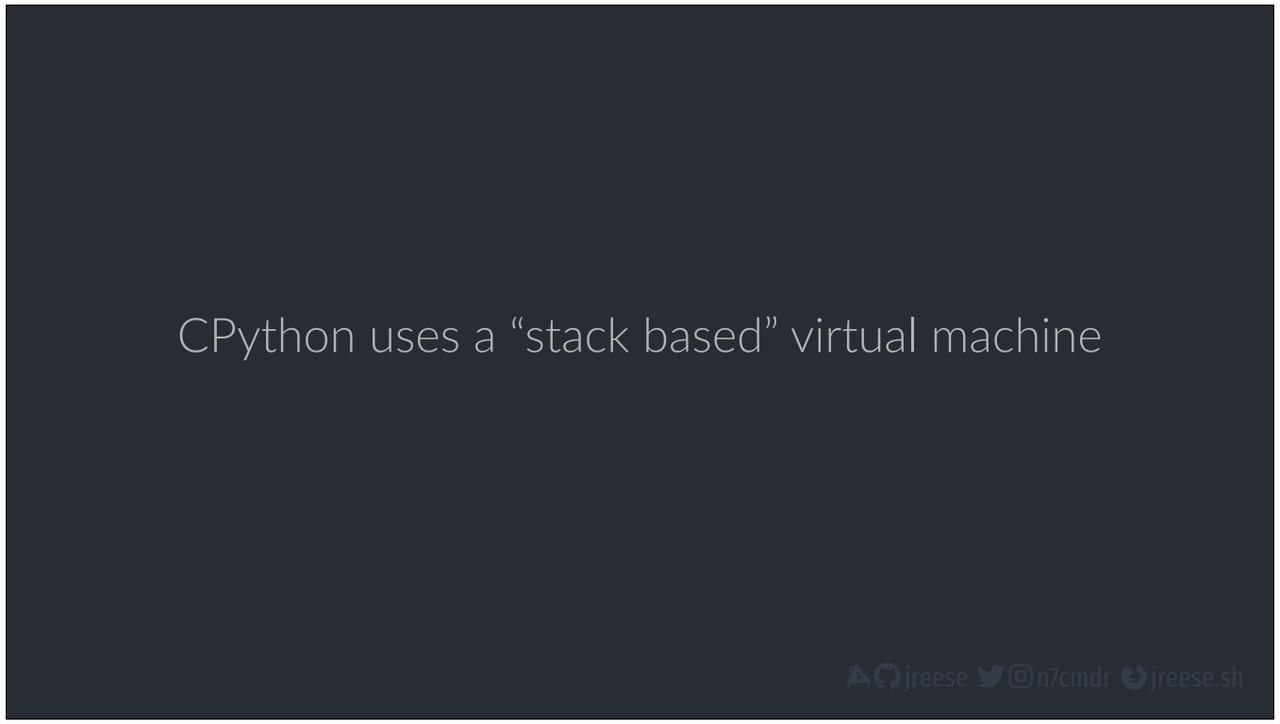
This seems reasonably straightforward, but there's a lot of details we take for granted.
We can go deeper.
Now, James stole a bit of my thunder yesterday, but hang in there.

```python
from dis import dis

print("square:")
dis(square)
print("main:")
dis(main)
```

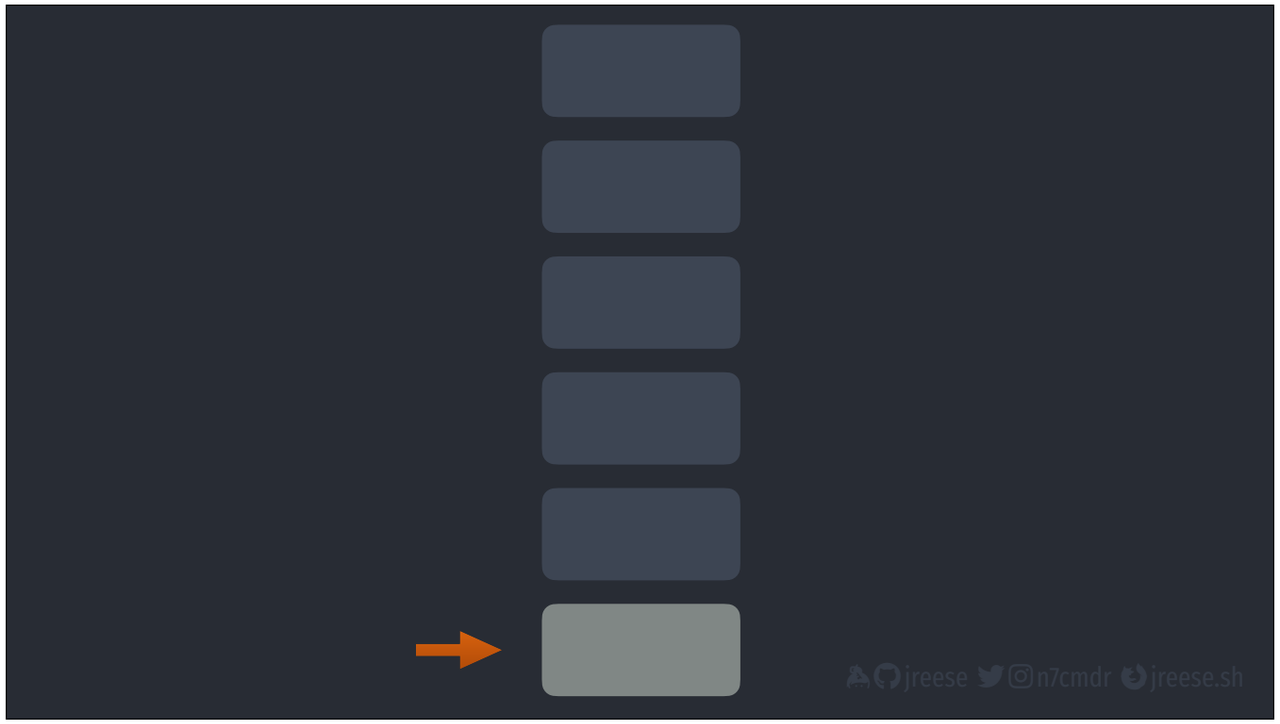

So, as you may know, CPython uses a virtual machine to execute our programs, and it provides an amazing utility to peer under the covers.
The “dis” module, short for “disassemble”, can give us a human readable listing of the compiled “byte code” for each function.
This byte code is the exact instructions used by the virtual machine, or runtime, to execute our Python code.

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

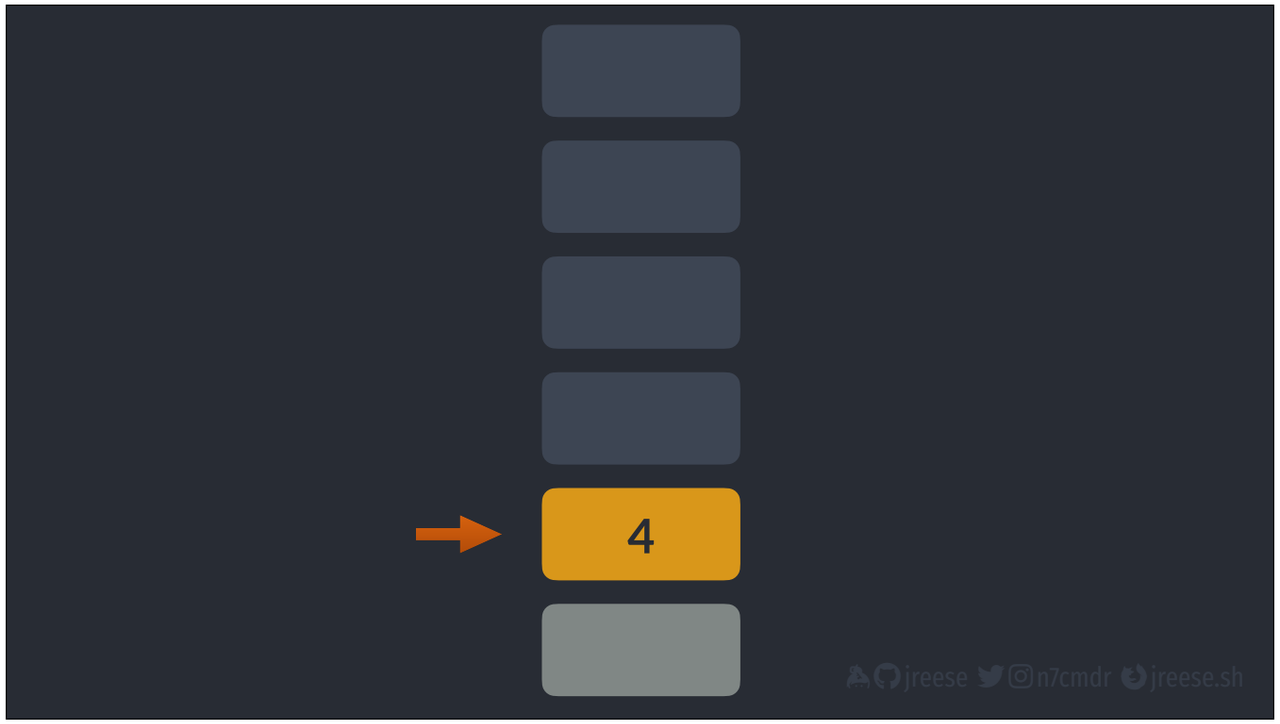

This is the output we get when disassembling our square() and main() functions
It looks more complicated like this, but if we keep asking questions and digging deeper,
it will all start to make more sense.
We just need to understand the context of how the runtime works.



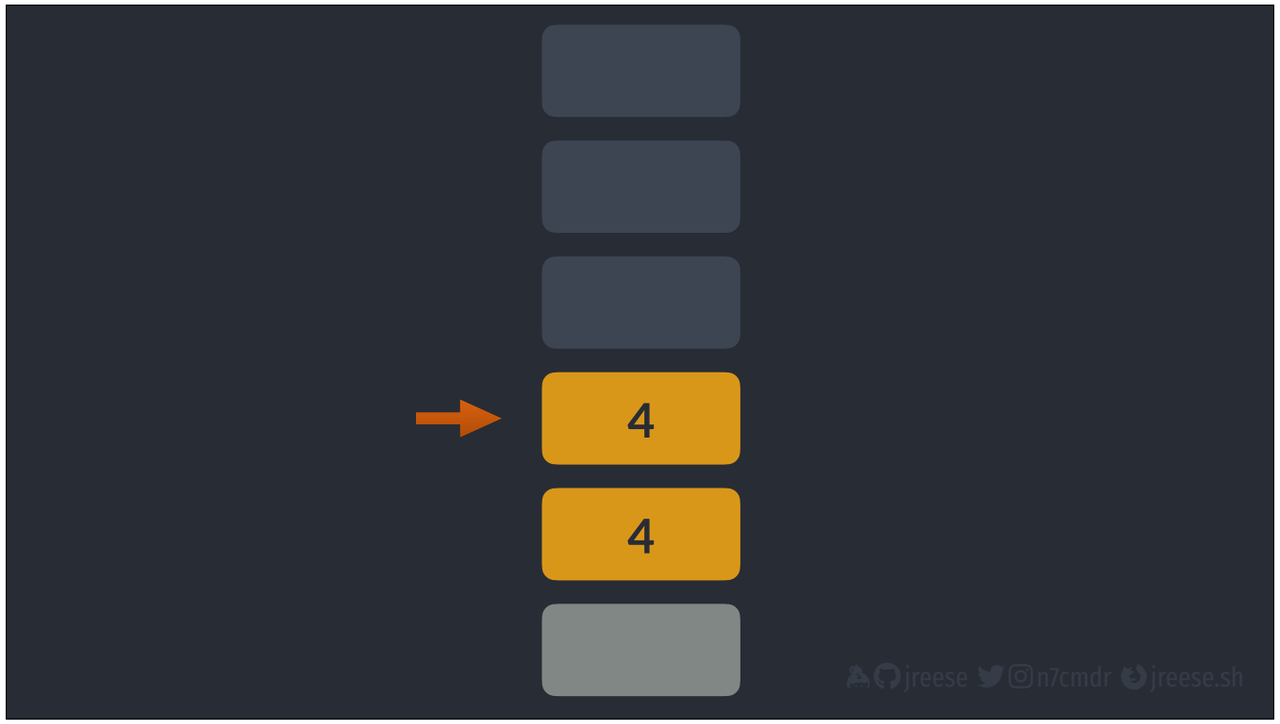
As you might also know, the CPython runtime uses a “stack based” virtual machine to execute instructions.
There are no “registers” like you might see or use in assembly programming.
If an instruction needs to operate on some piece of data, that data must be first put onto the stack.



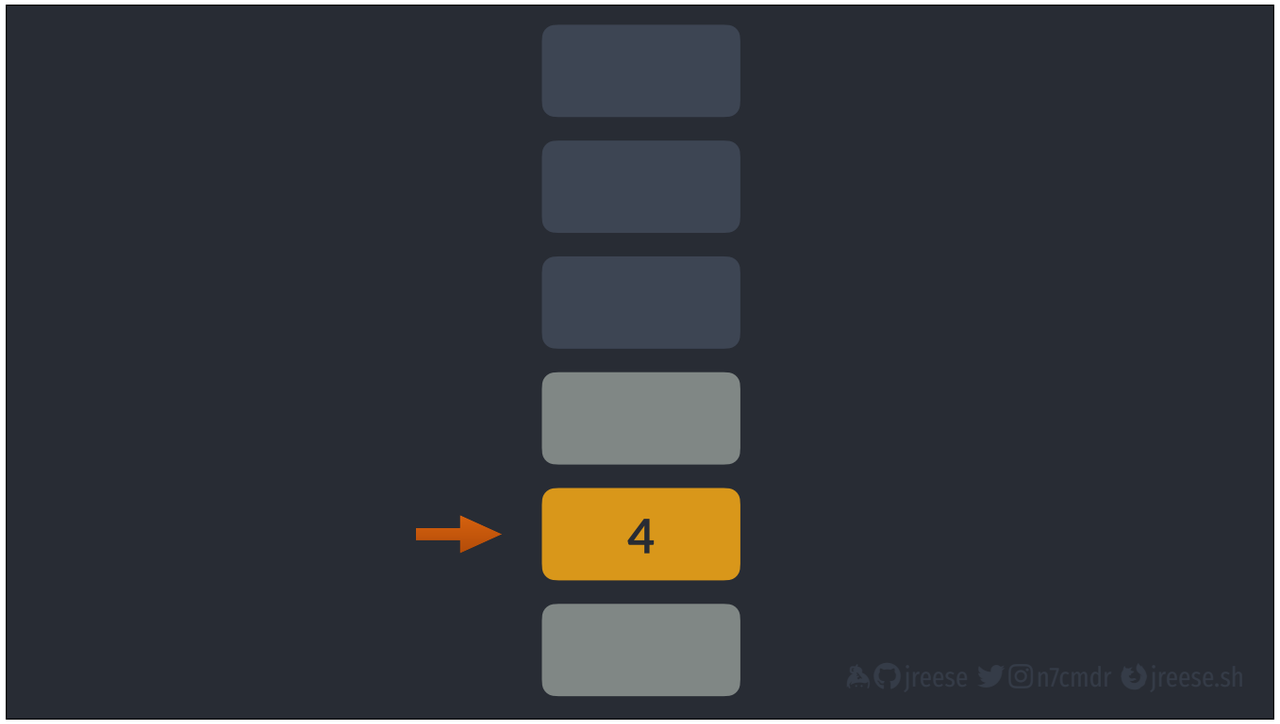
The stack itself is just a linear block of memory that contains data or references to data.
The runtime uses what’s called a “stack pointer” to keep track of where the top of the stack is.



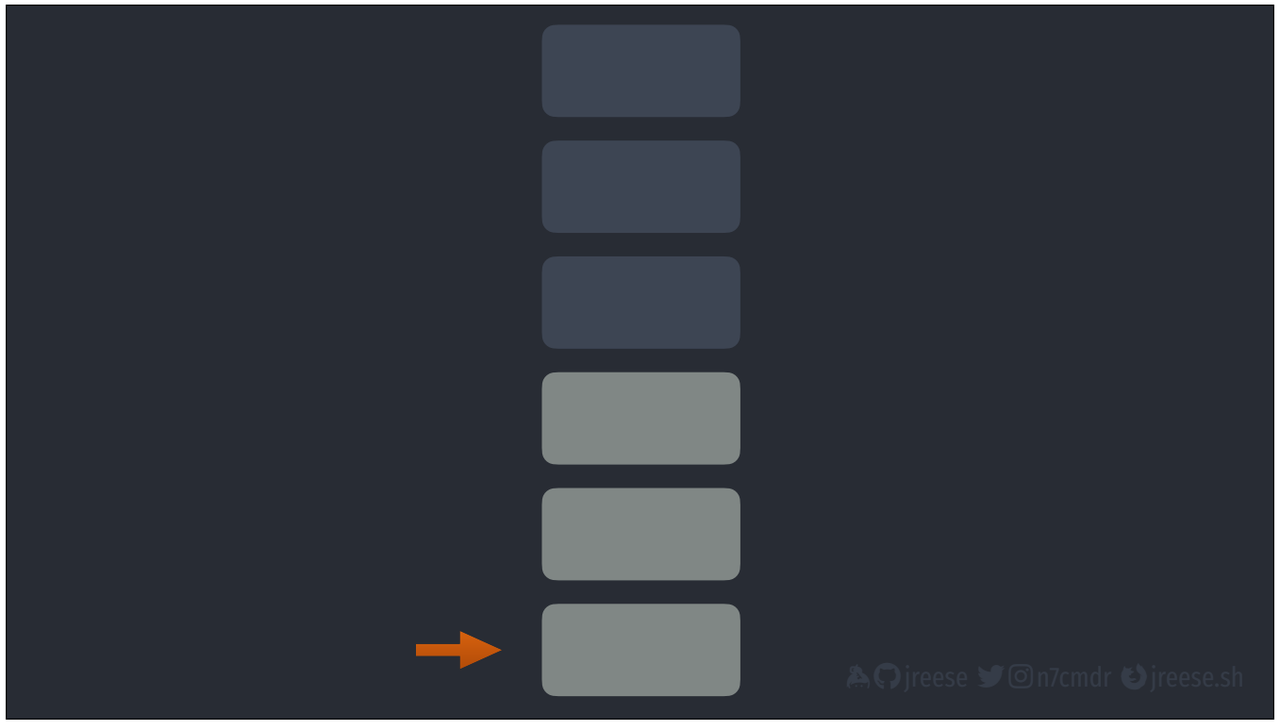
When executing instructions, it's possible to “push” data onto the top of the stack.
The LOAD_FAST instruction, for example, has one job: pushing a single value from memory.



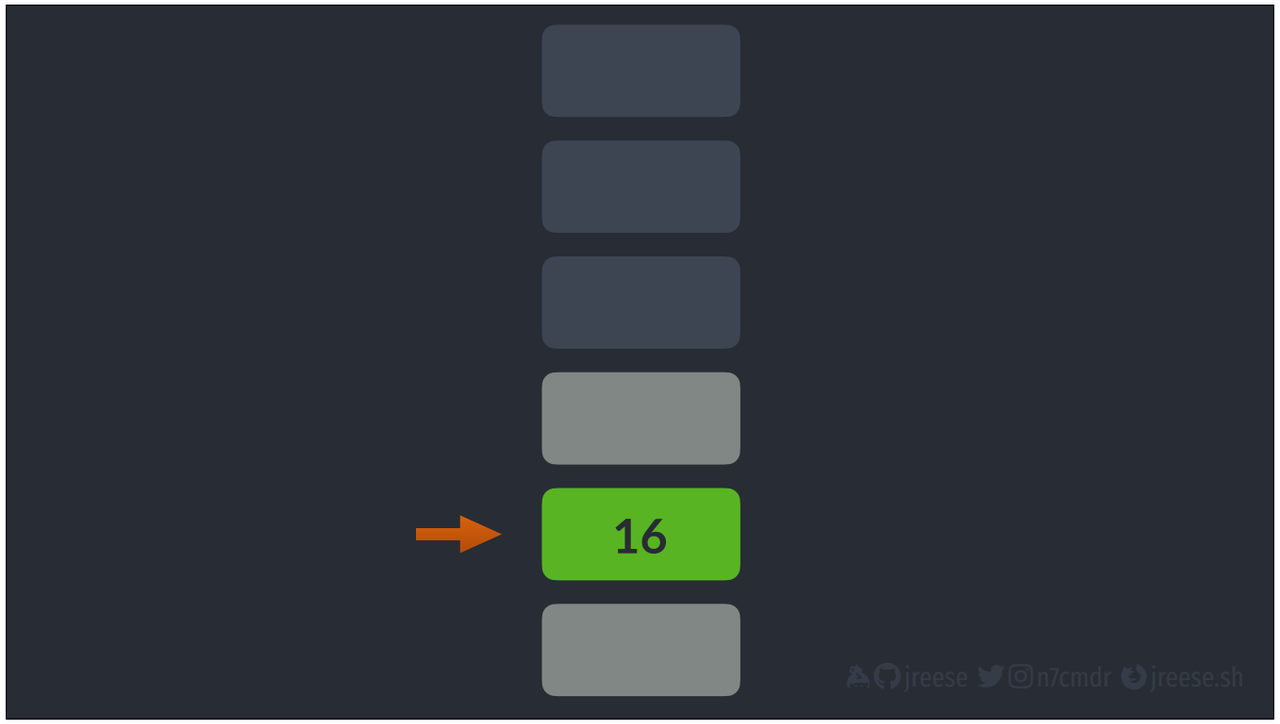
Each time this happens, the stack pointer is incremented to point at the new top of the stack.
Other instructions can then consume, or "pop" items from the stack, and even push new items at the same time.
For example, a math operation, like multiplication, would pop the top two items from the stack …



… which resets the stack pointer each time …



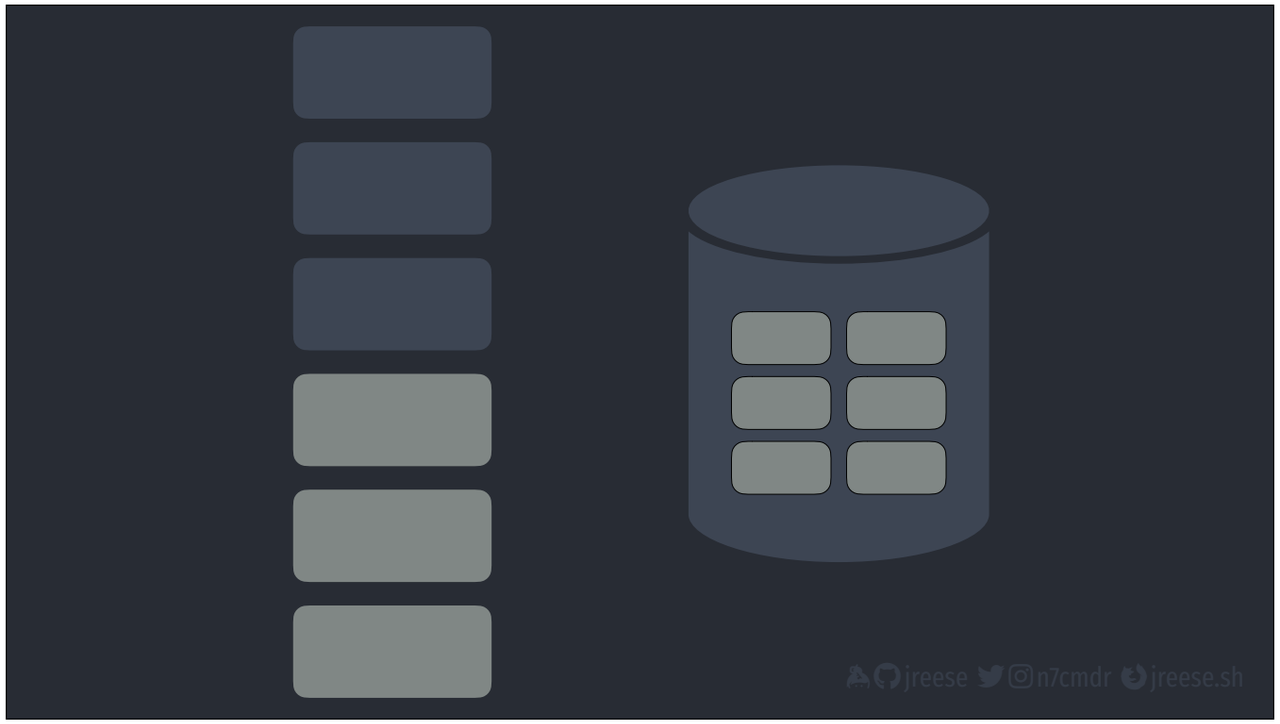
… then it performs the calculation …



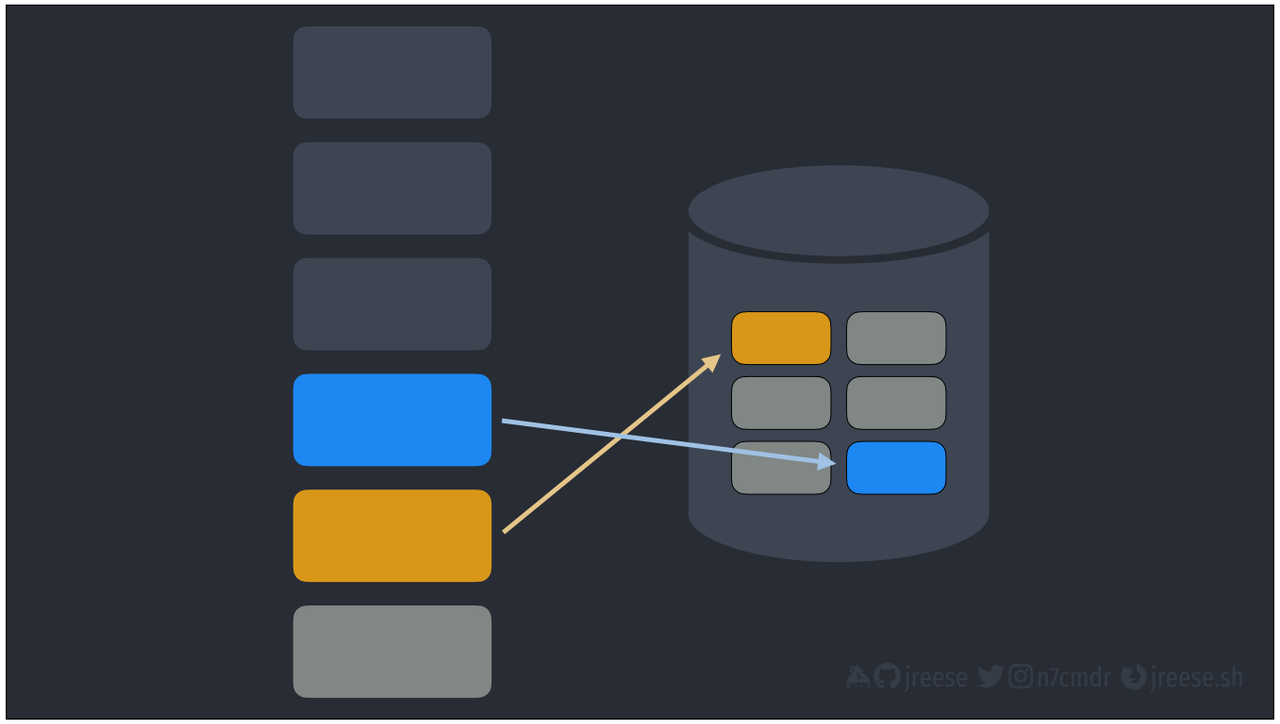
… and finally pushes the resulting value back onto the stack to be used by future instructions.

But not everything lives solely on the stack.

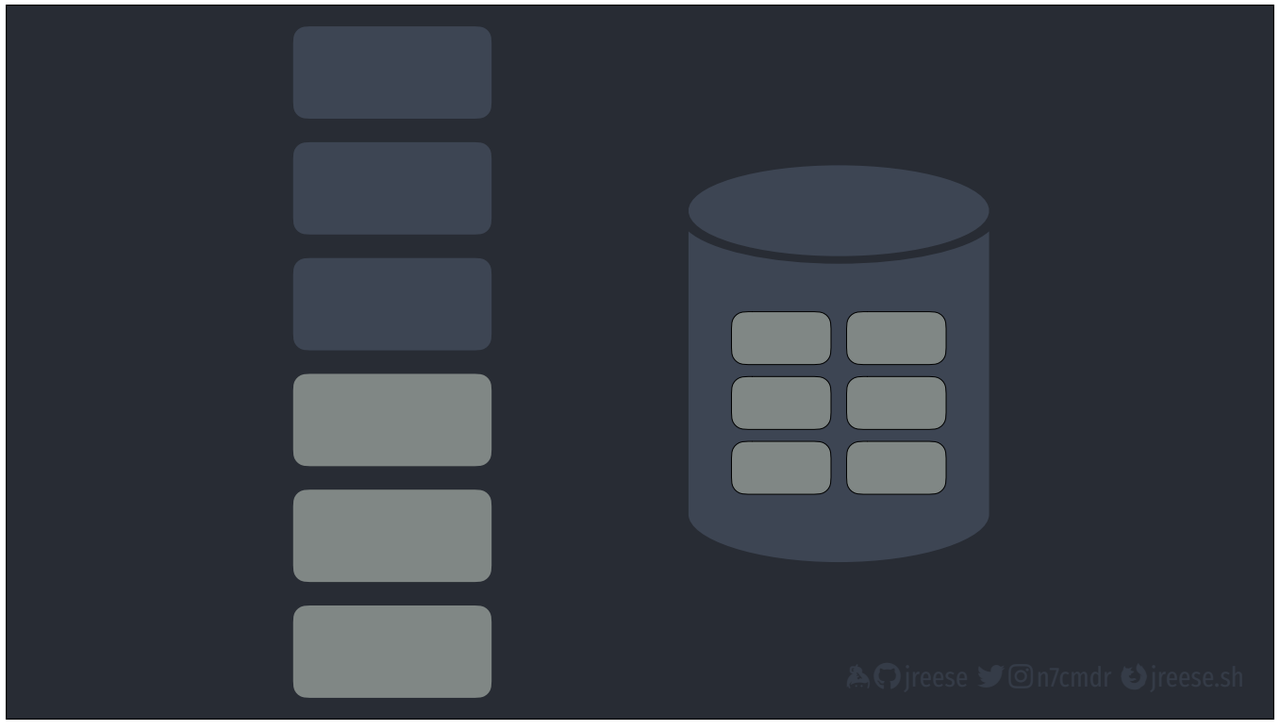
Data that outlasts the function creating it has to be stored somewhere else, so that it doesn’t go away when the function returns.



So like most languages, Python uses the heap to store objects in long term memory.
Unlike the stack, this is just an unordered space where objects can be allocated and deallocated at any time during execution.



And many times, items on the stack are just references <click> to the real objects on the heap.



But again, the exact details of this relationship is less important than knowing it exists.

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


So if we go back to our disassembled instructions from earlier,
We can start to piece together what happens when we execute the main() function.
Each line represents a single instruction for the CPython virtual machine,
It's akin to assembly code instructions, but designed for an idealized CPU architecture.

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


If we look at an individual instruction, there are four pieces of information here:

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


The line number of the original code that a group of instructions represents …

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


The instruction address (relative to the top of the function, class, or module, depending on what you disassembled) …

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


The “opcode”, which is the specific virtual machine operation that should be executed …

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


And lastly, a numerical parameter for the operation, often either a raw value
or an index into some dataset, like the list of global or local variables.
The value in parentheses is just to let us humans know what that parameter represents.

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


So if we compare this back with our original source code, we can start to see how it all relates back to the code we wrote.

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)

 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


These instructions perform the multiplication and return the result …

```
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)

 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


These instructions call the square() function and store the result in x …

```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE

main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)

 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


these instructions print the value stored in x...

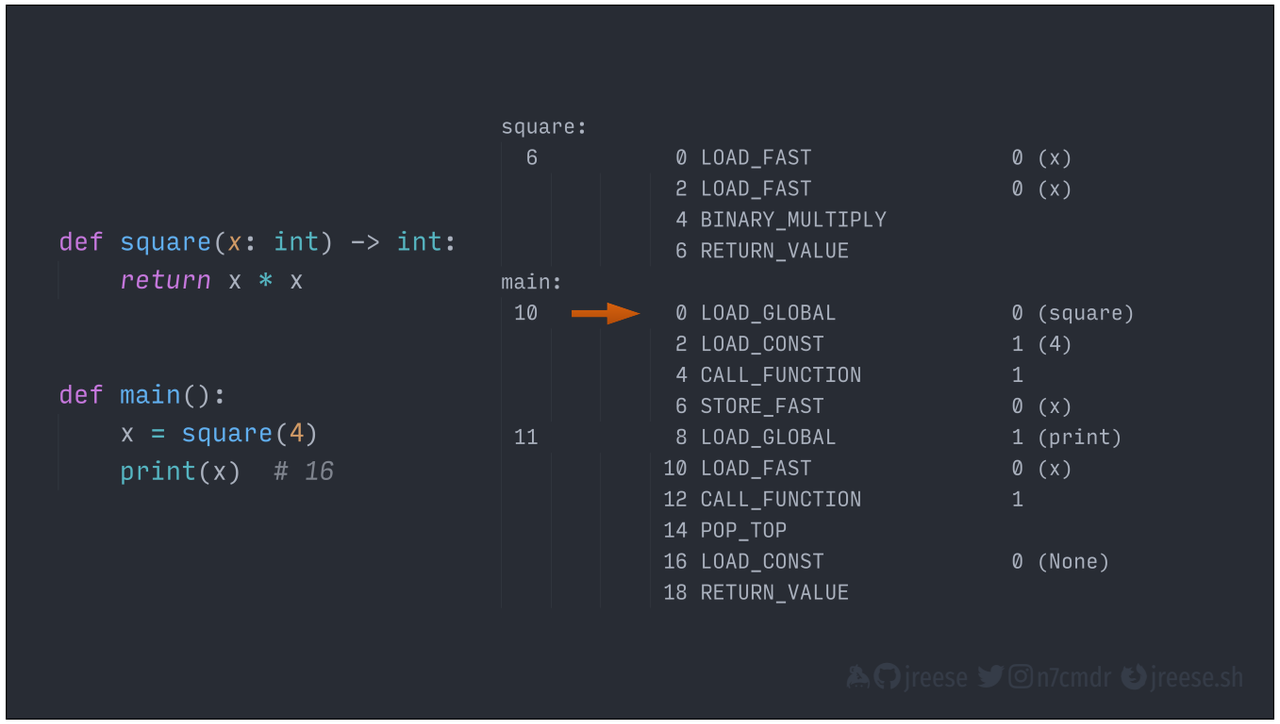
```
                                  square:
                                    6           0 LOAD_FAST                0 (x)
                                                2 LOAD_FAST                0 (x)
def square(x: int) -> int:                      4 BINARY_MULTIPLY
    return x * x                                6 RETURN_VALUE

                                  main:
                                   10           0 LOAD_GLOBAL              0 (square)
                                                2 LOAD_CONST               1 (4)
def main():                                     4 CALL_FUNCTION            1
    x = square(4)                               6 STORE_FAST               0 (x)
                                   11           8 LOAD_GLOBAL              1 (print)
    print(x)  # 16                             10 LOAD_FAST                0 (x)
                                               12 CALL_FUNCTION            1
                                               14 POP_TOP
                                               16 LOAD_CONST               0 (None)
                                               18 RETURN_VALUE

                                                    jreese n7cmdr jreese.sh
```

And finally, these last two instructions represent the implicit return statement.

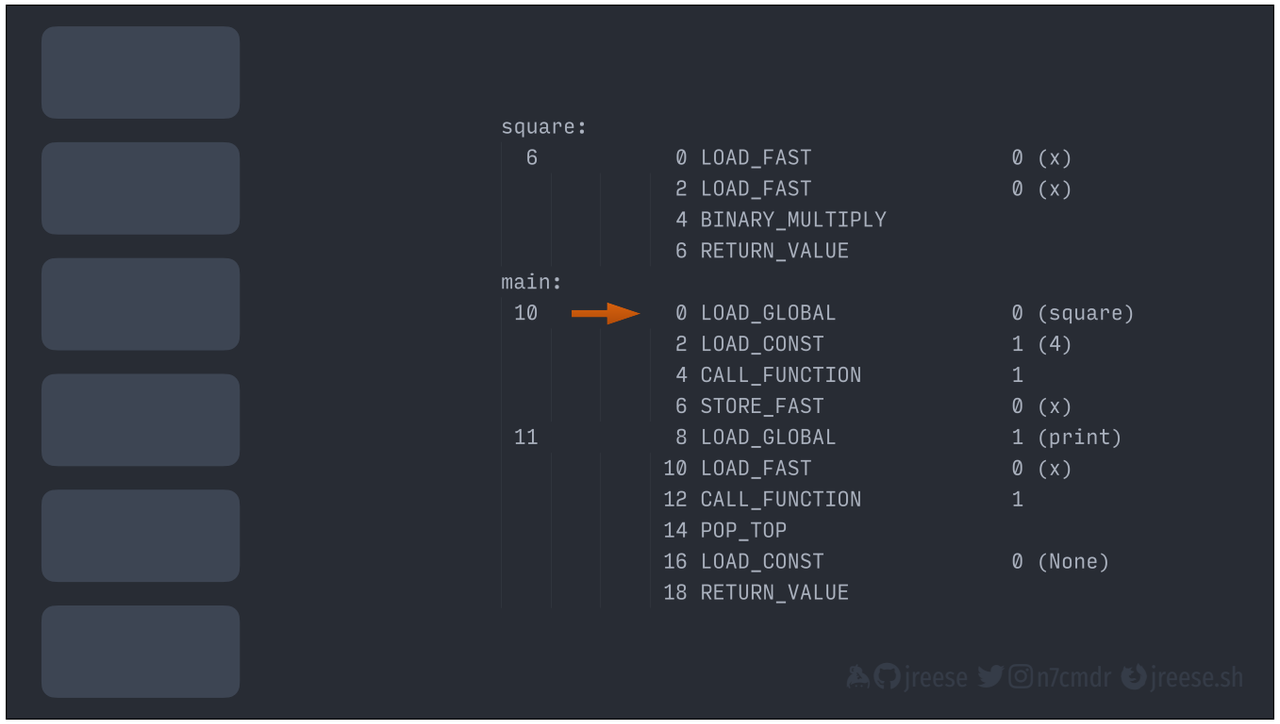
```python
def square(x: int) -> int:
    return x * x


def main():
    x = square(4)
    print(x)  # 16
```

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

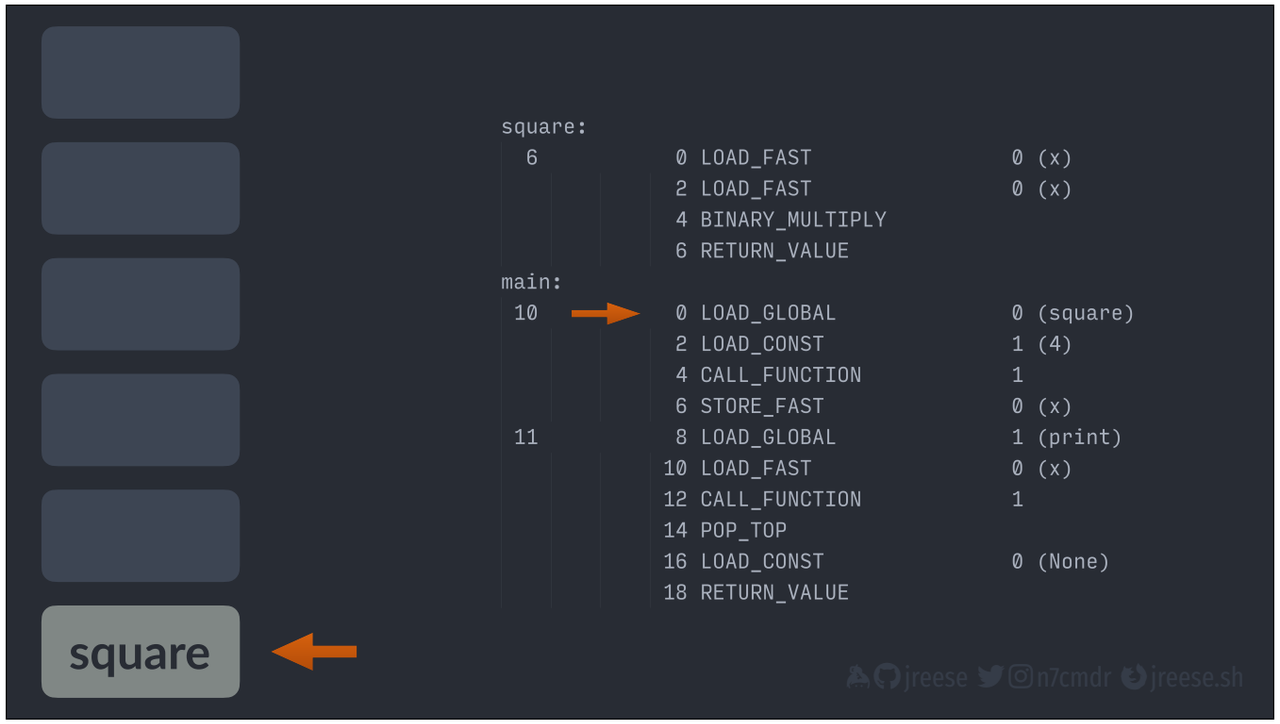

Now, when executing code, the runtime needs to keep track of its position in the set of instructions.
Each instruction is a predetermined, fixed size in memory, and therefore has its own memory address.
So the runtime will just keep track of the memory address of the next instruction.



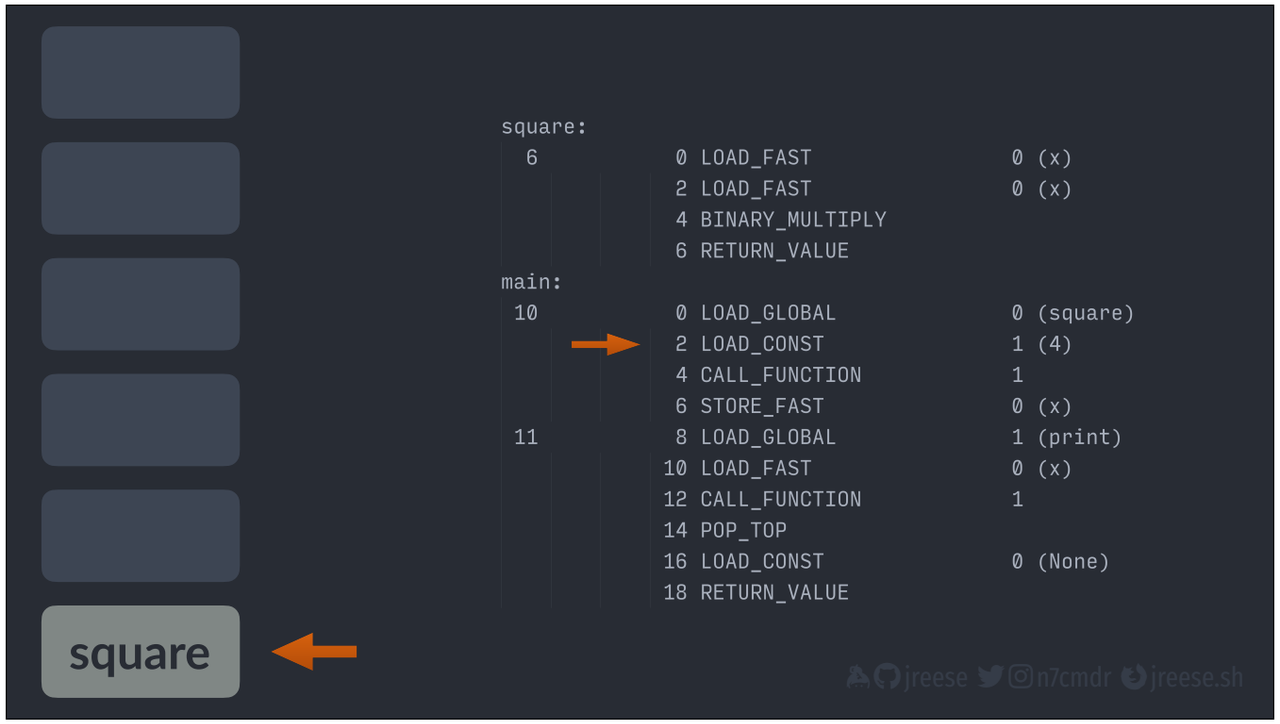
This is called the “instruction pointer”. Each time the runtime executes an instruction, it automatically increments the instruction pointer.
Special instructions can then modify the instruction pointer, allowing the runtime to jump between different sections of code.
This is the basis of all flow control, including if/else conditional blocks, for and while loops, and function calls.



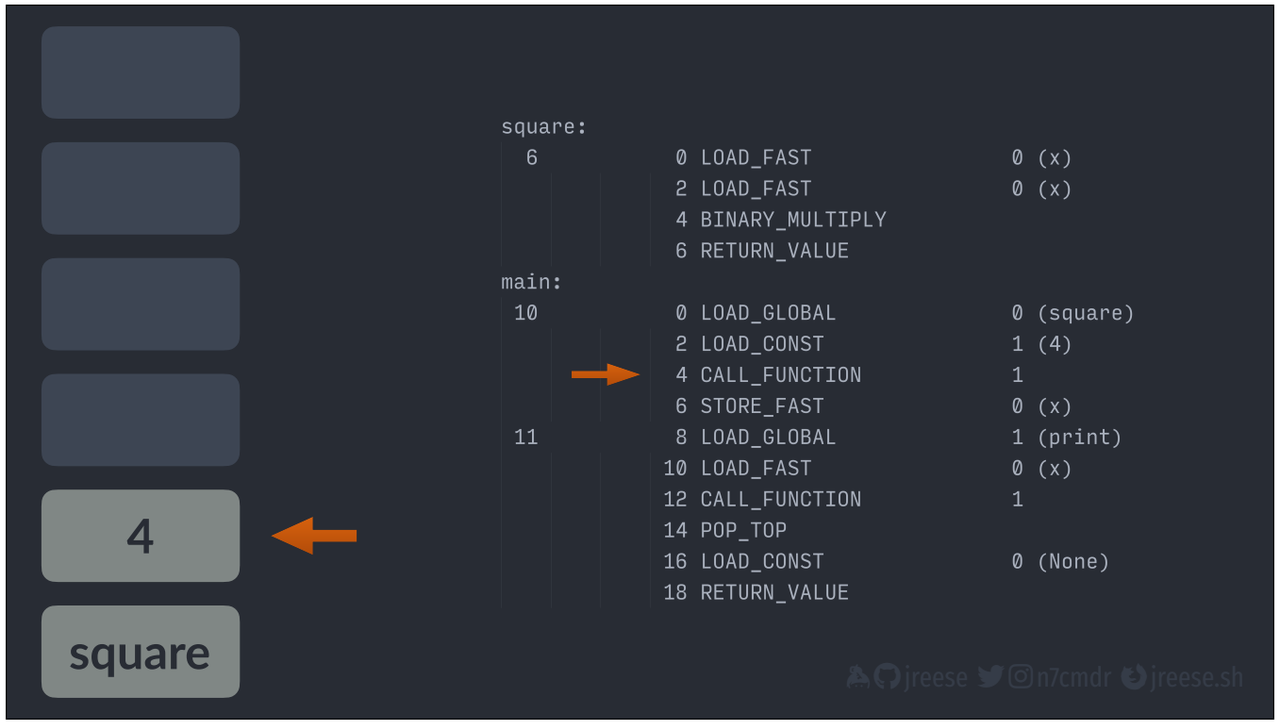
So now, let's bring back the stack from earlier, and take a high level, conceptual look at how this could be executed by the runtime.
First, we push a reference to the square() function, which is in the global scope …



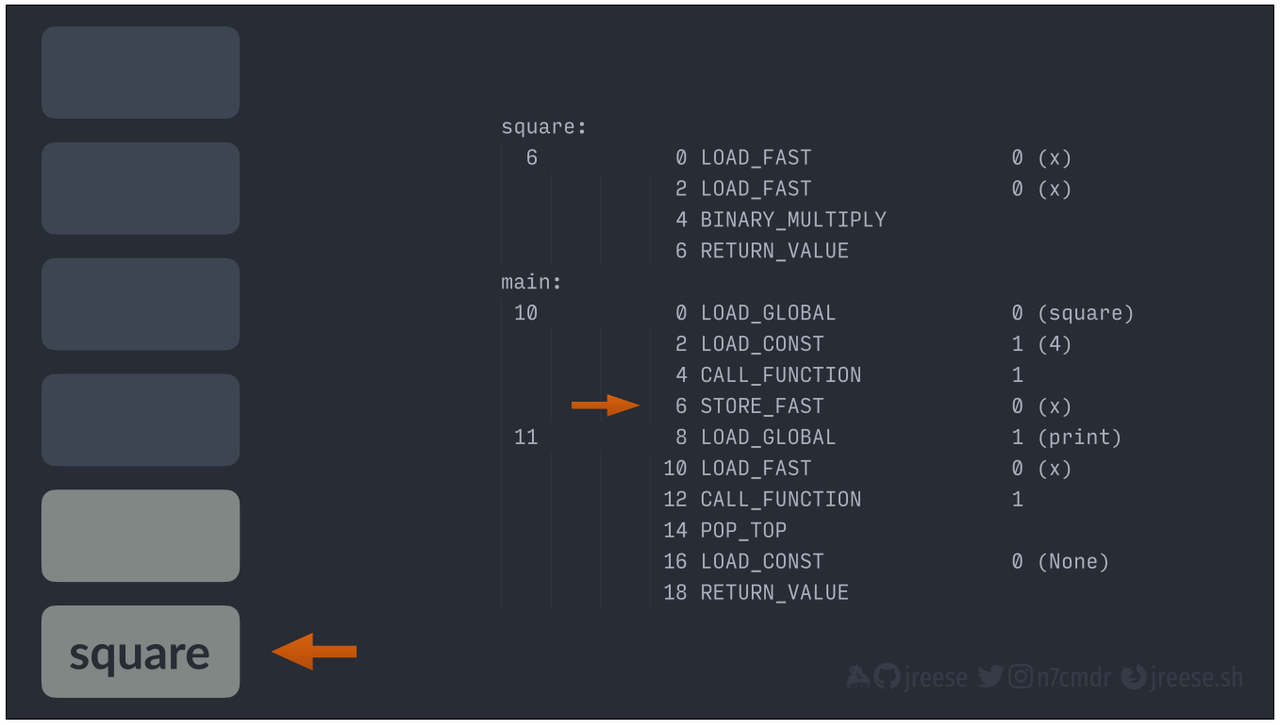
And after each operation we'll increment the instruction pointer…



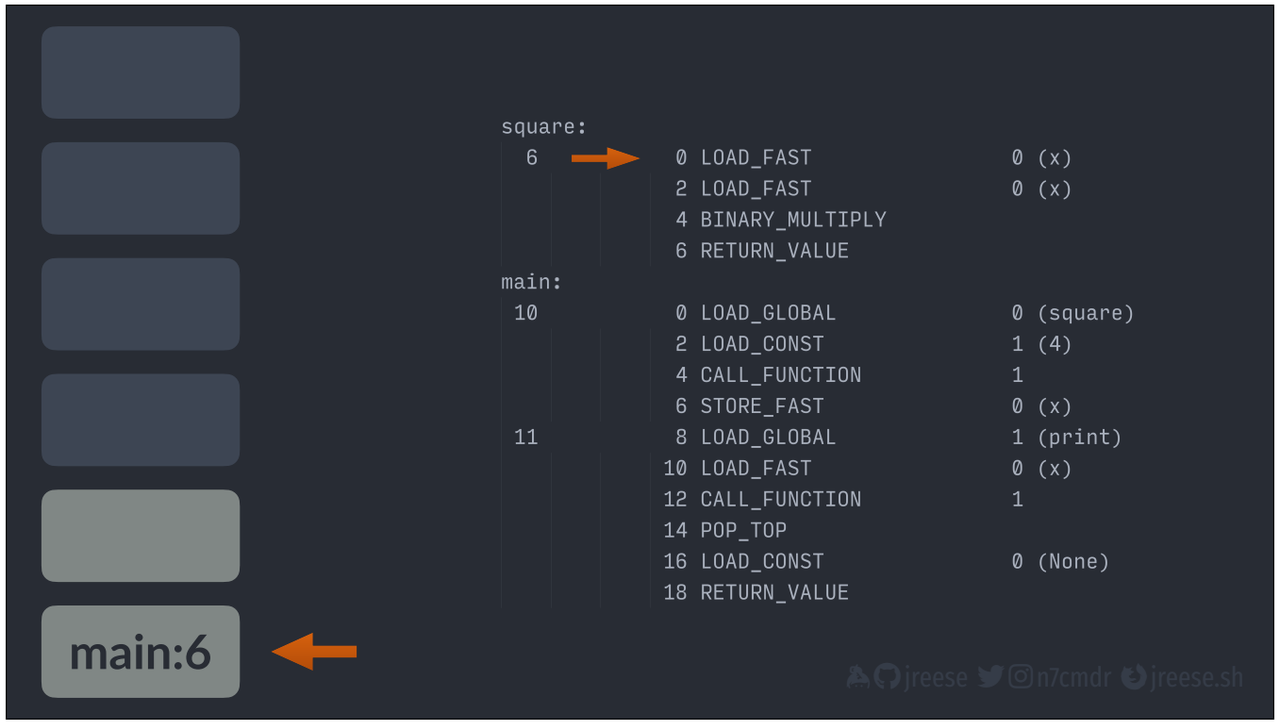
Then we load the constant value 4, the parameter that we're passing to the square() function…



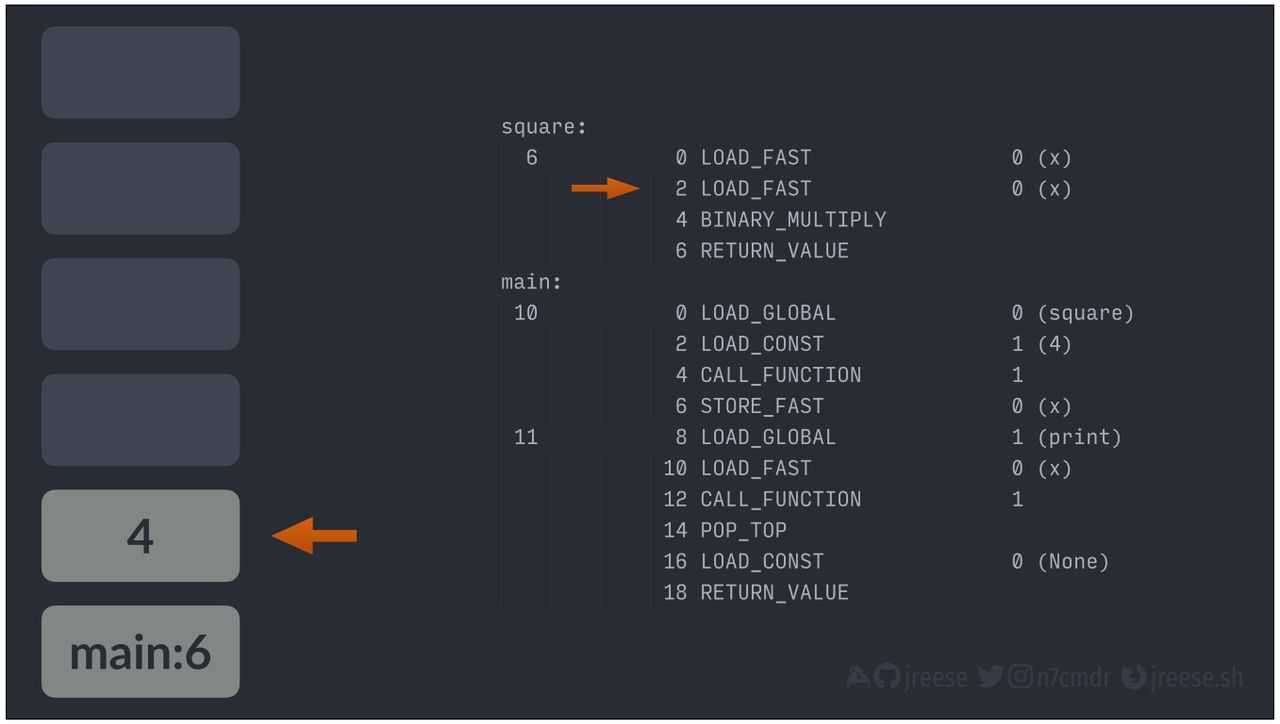
Now we run the operation to call the function. The parameter to this instruction is the number of arguments we're passing.
So the CALL_FUNCTION operation knows to consume that number of items from the stack before it finds the function to call.
The parameters it consumes will then be used to create the set of locals available inside that function.



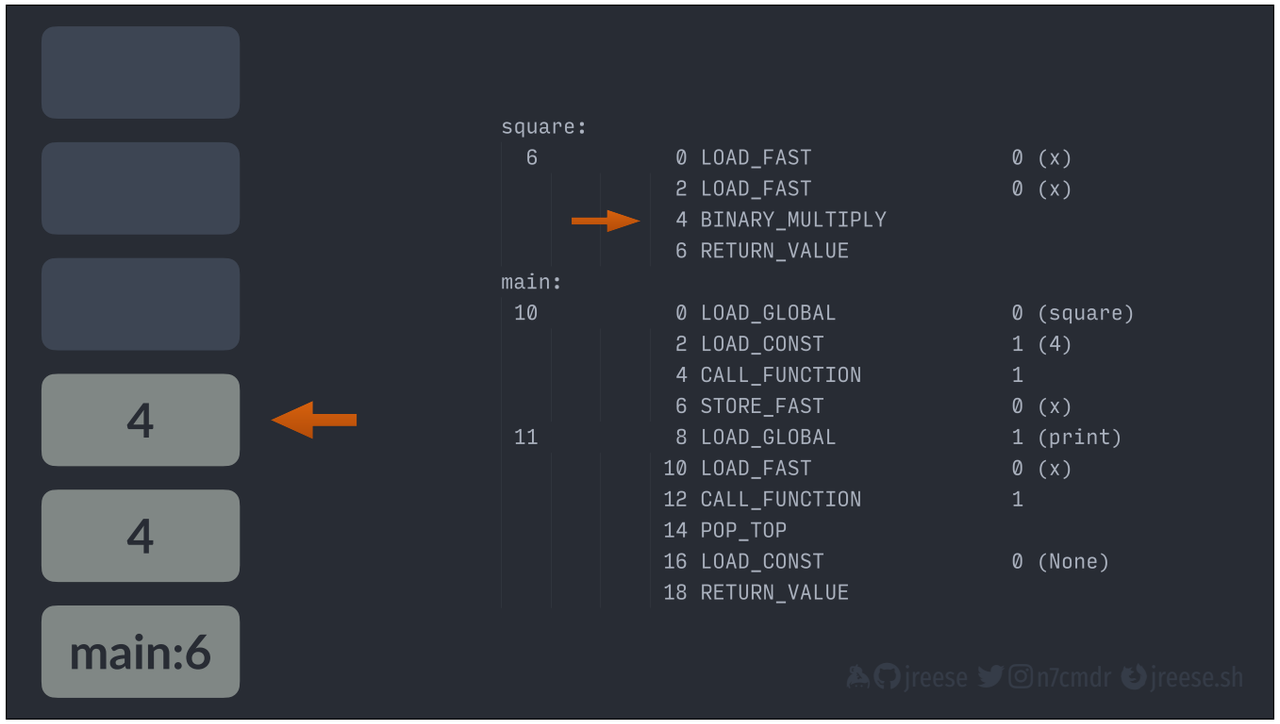
After popping the function arguments from the stack, and incrementing the instruction pointer,
The CALL_FUNCTION operation can now pop the function reference itself,
And use that value to update the instruction pointer to the first address in that function,
While storing the previous instruction pointer value to the stack, so that we know where to return to later.



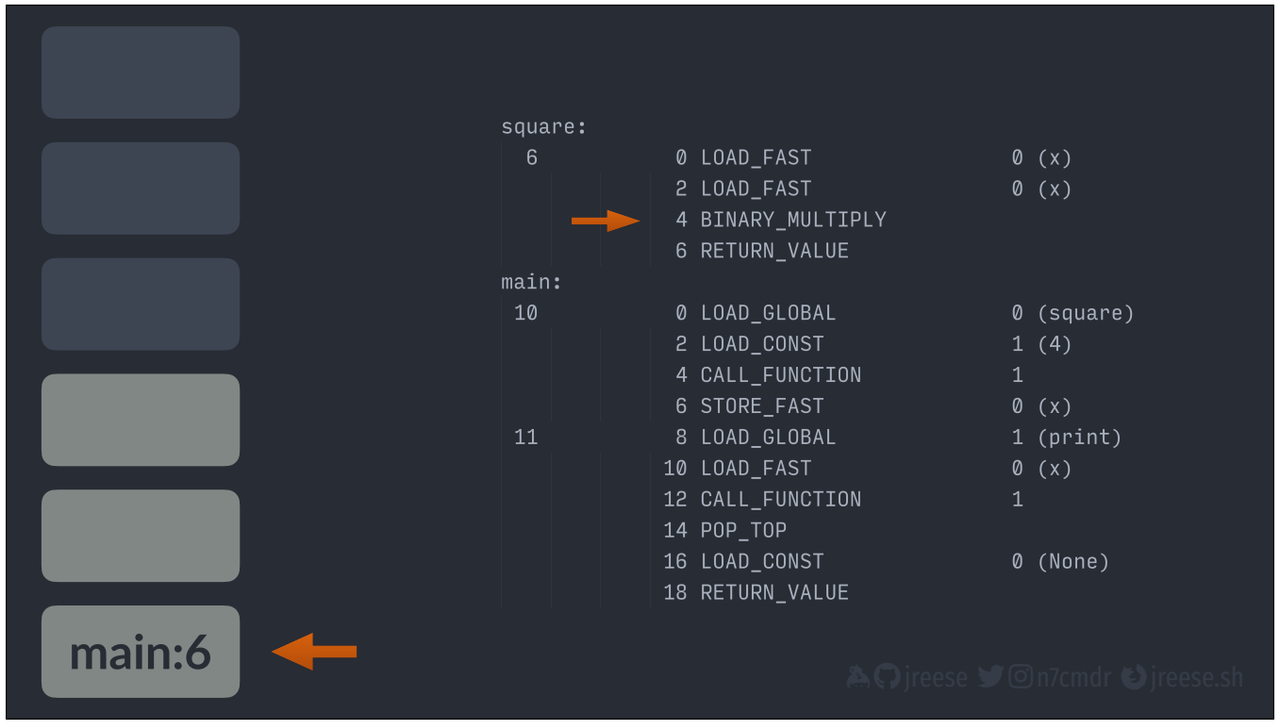
Now we’re executing the square() function. First thing is to load x onto the stack.



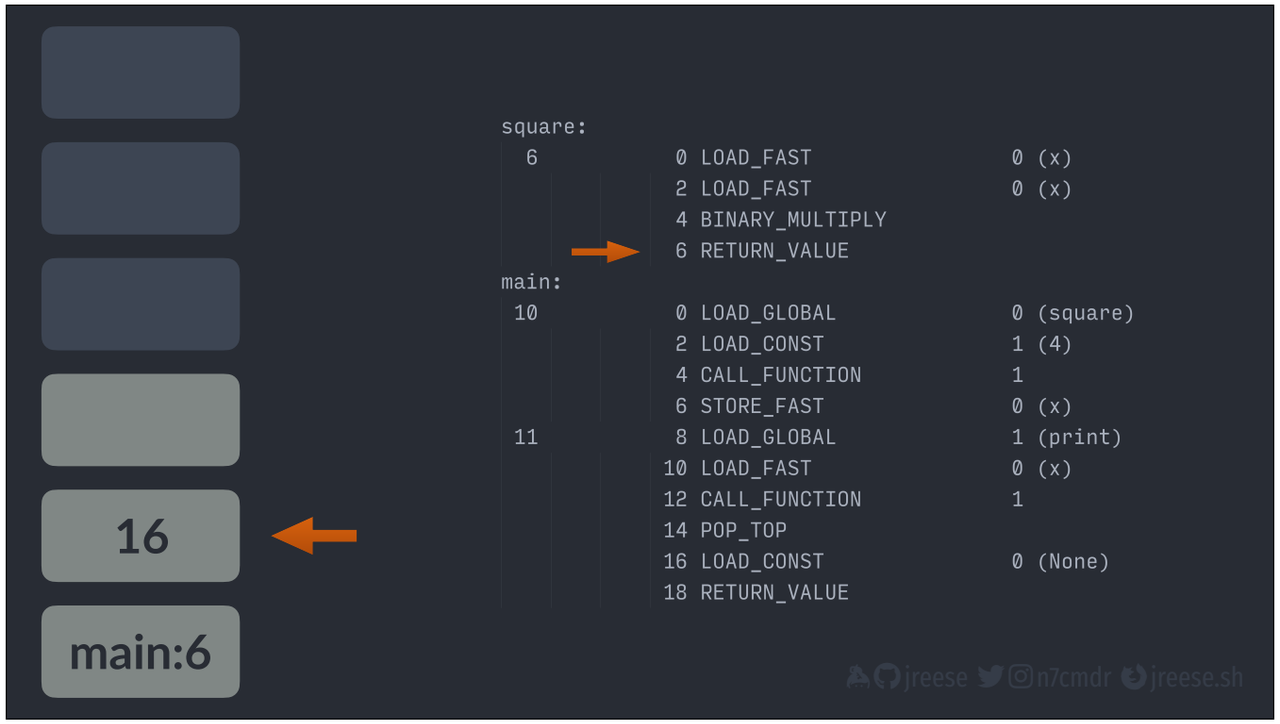
And because we’re multiplying x by itself, we’ll load x onto the stack a second time.



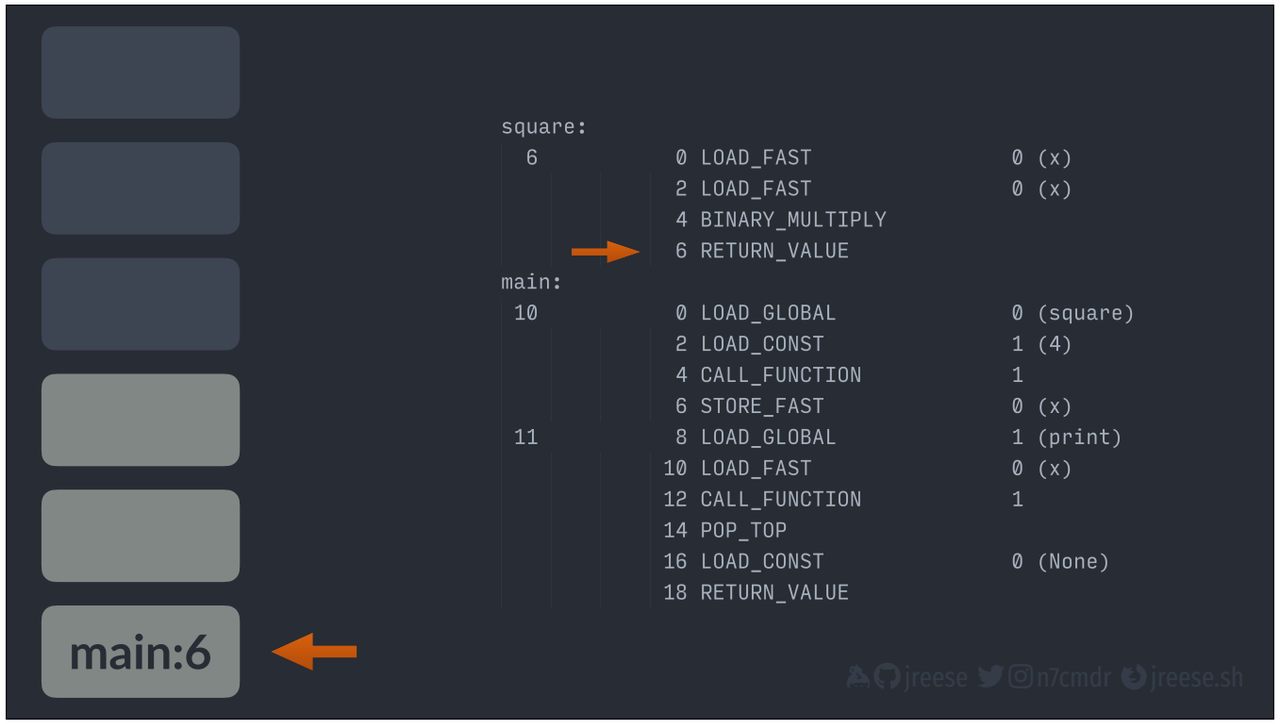
Now we can actually multiply the two values together.
This will pop both sides of the multiplication from the stack …



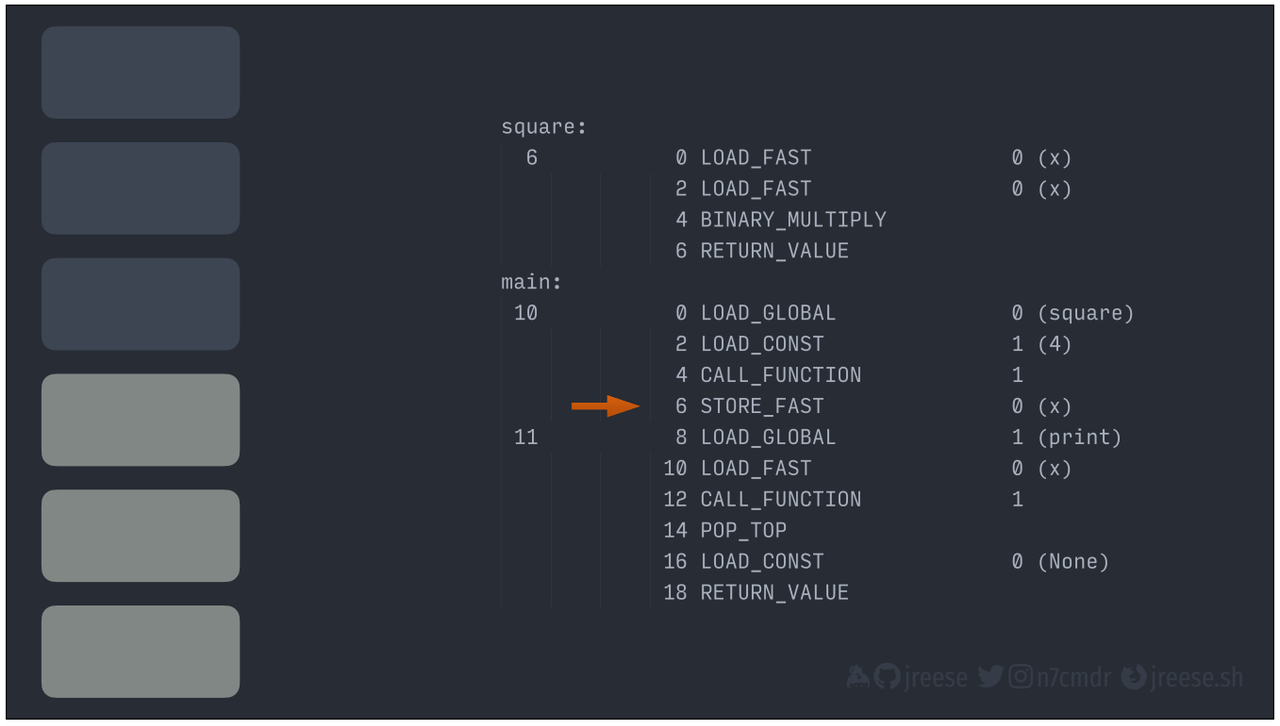
Do the actual multiplication, in this case 4 times 4, and store the resulting value back on the stack …



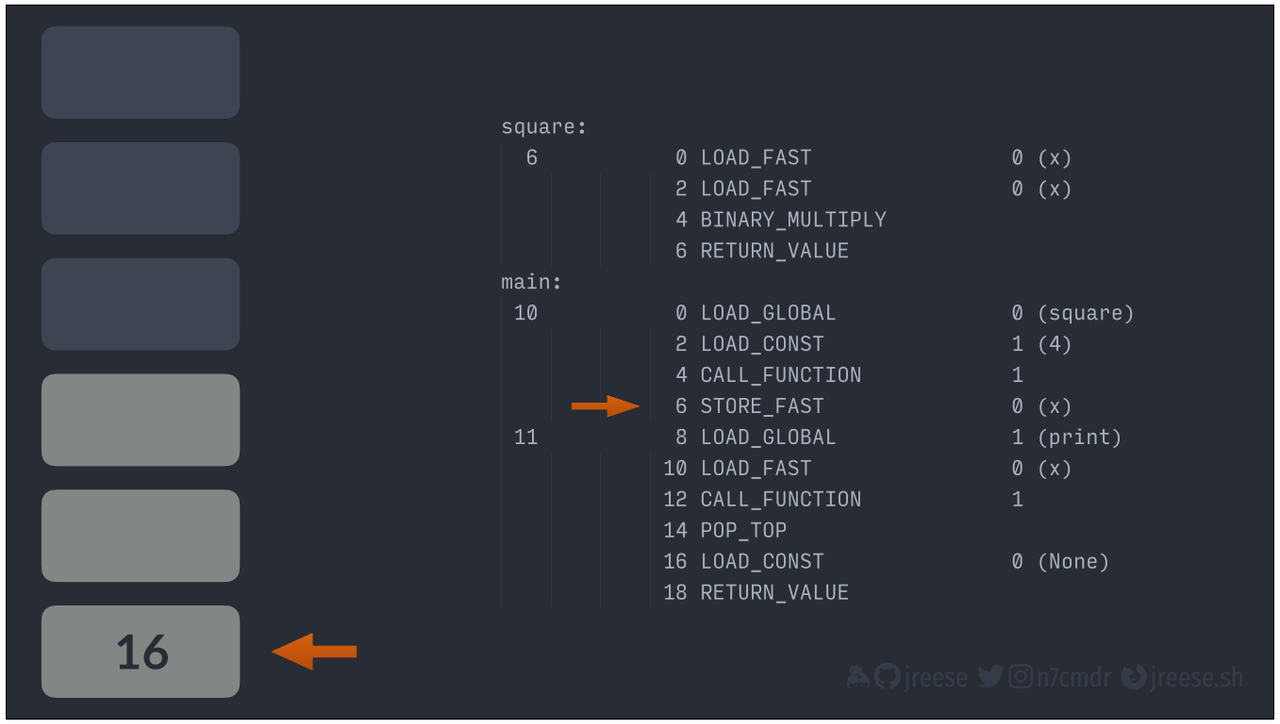
Now, we have our result, and we're ready to return it to the caller.
The RETURN_VALUE operation will pop the result from the stack…



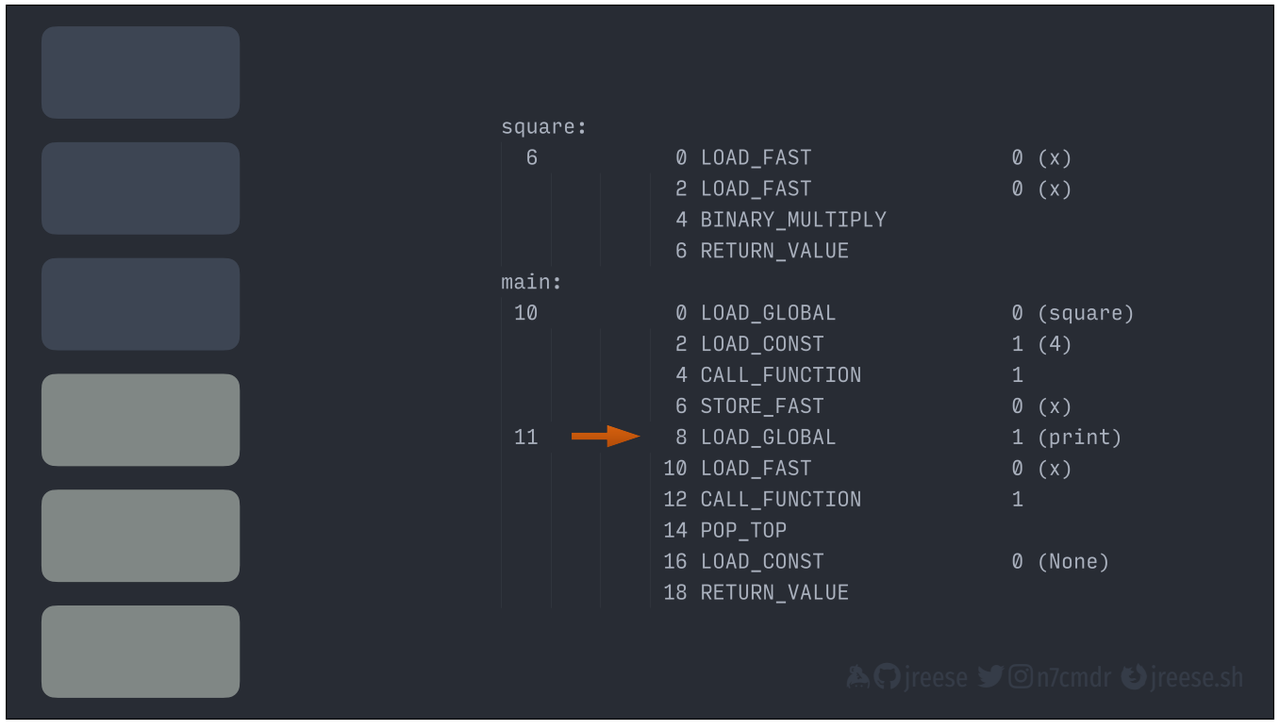
Then it will restore the instruction pointer with the next value on the stack...



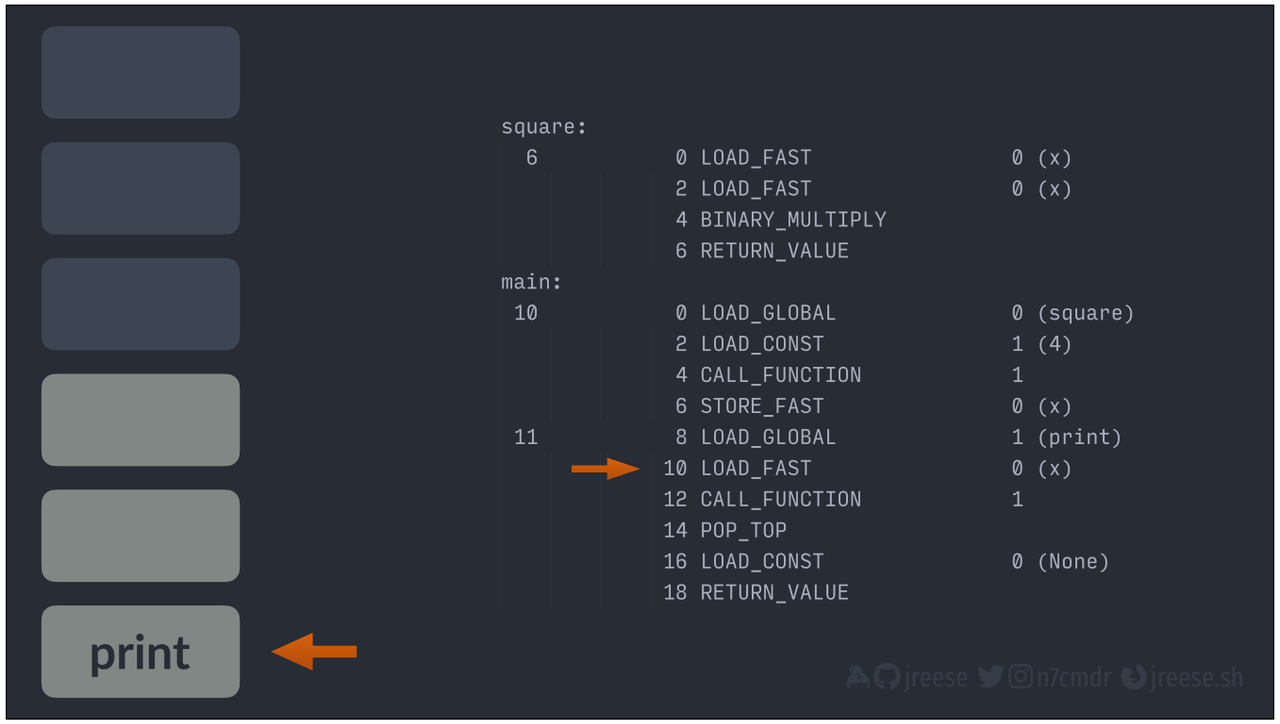
and then push the actual result back onto the stack.



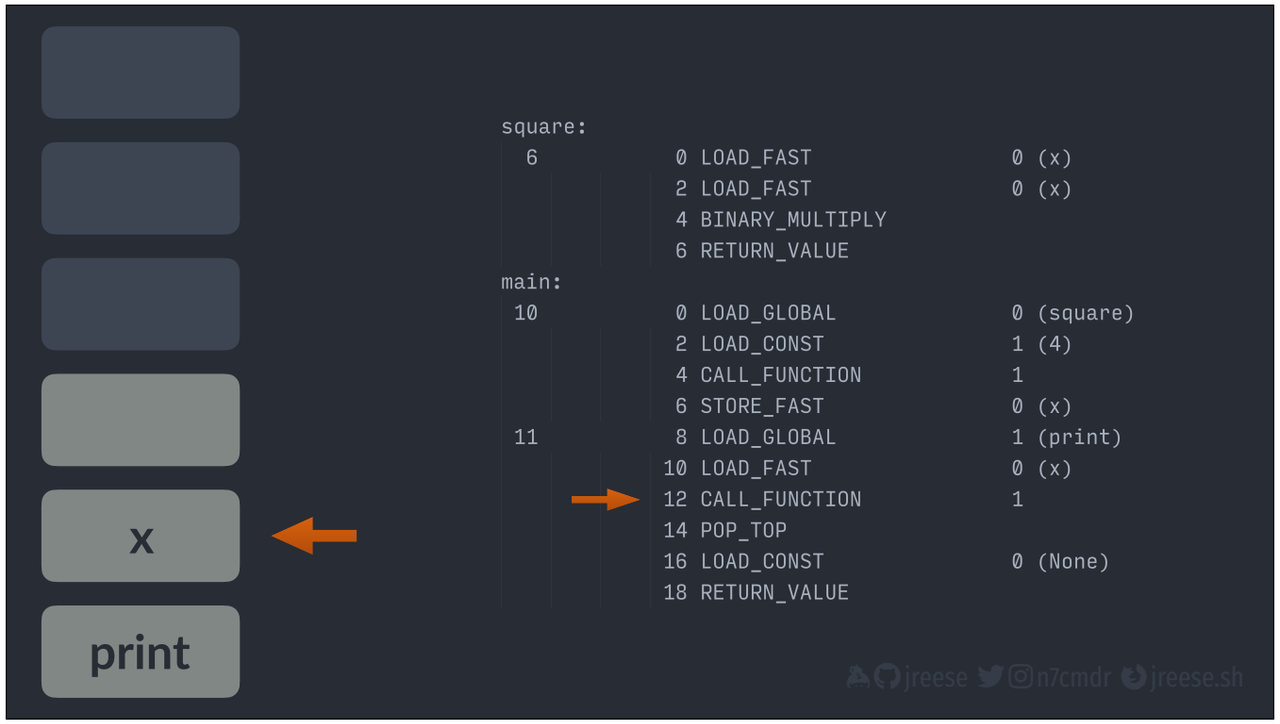
Now we’re back in our main() function where we left off
But with the result of the square() function at the top of the stack.
We can now store that value into the local variable x…



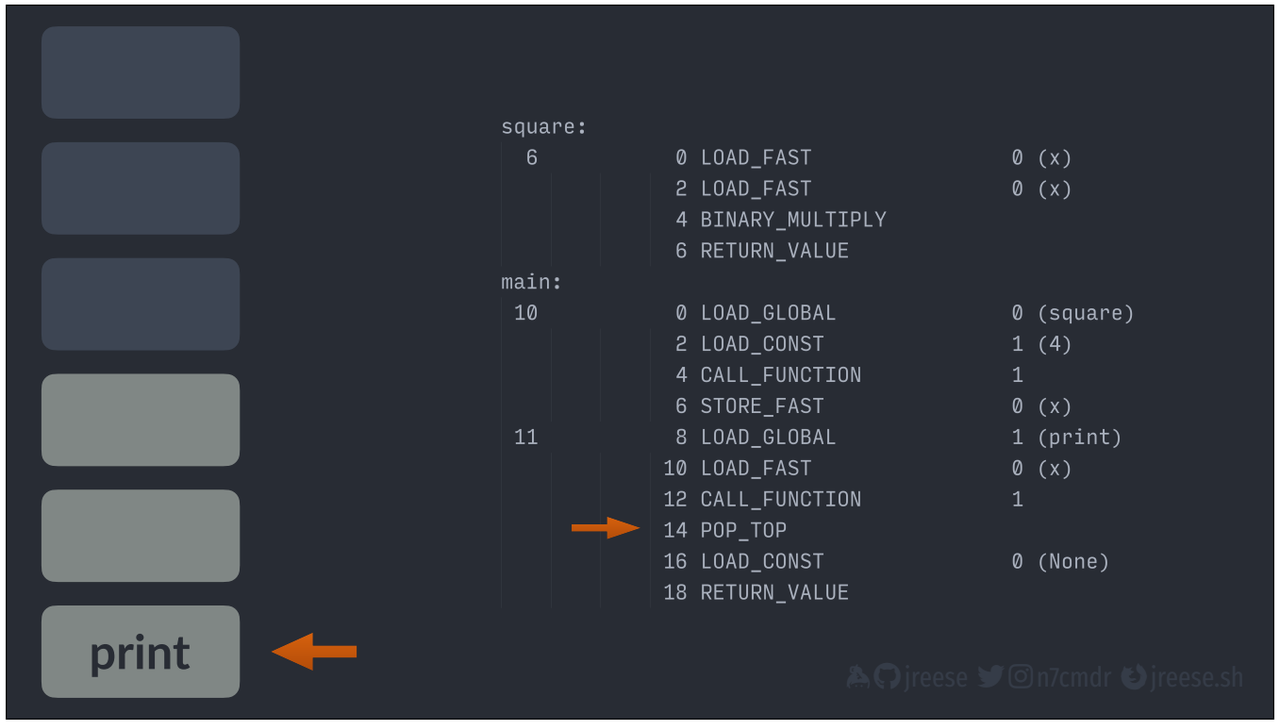
The next line of code is to print the value of x,
so we start by pushing a reference to the print() function



Next we push a reference to x onto the stack…

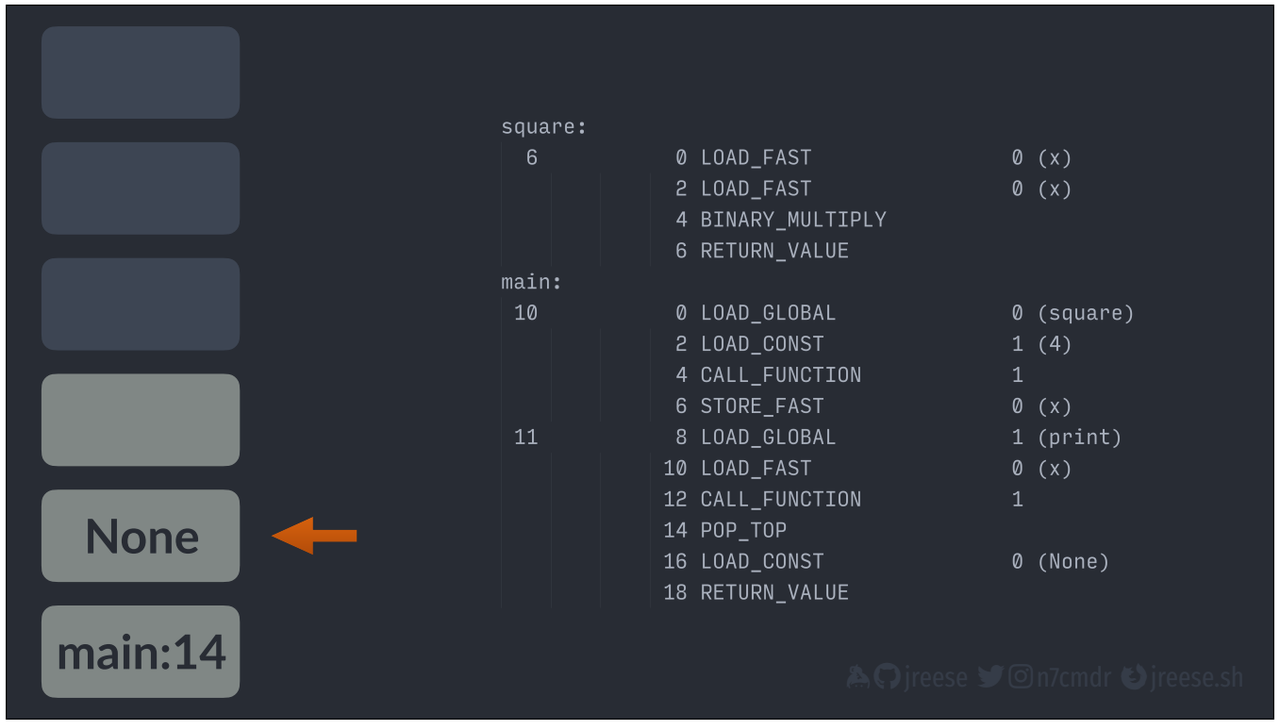


Now we execute the CALL_FUNCTION operation again with parameter 1 because there's only one argument.
This will consume the reference to x and put it in the local scope for print…

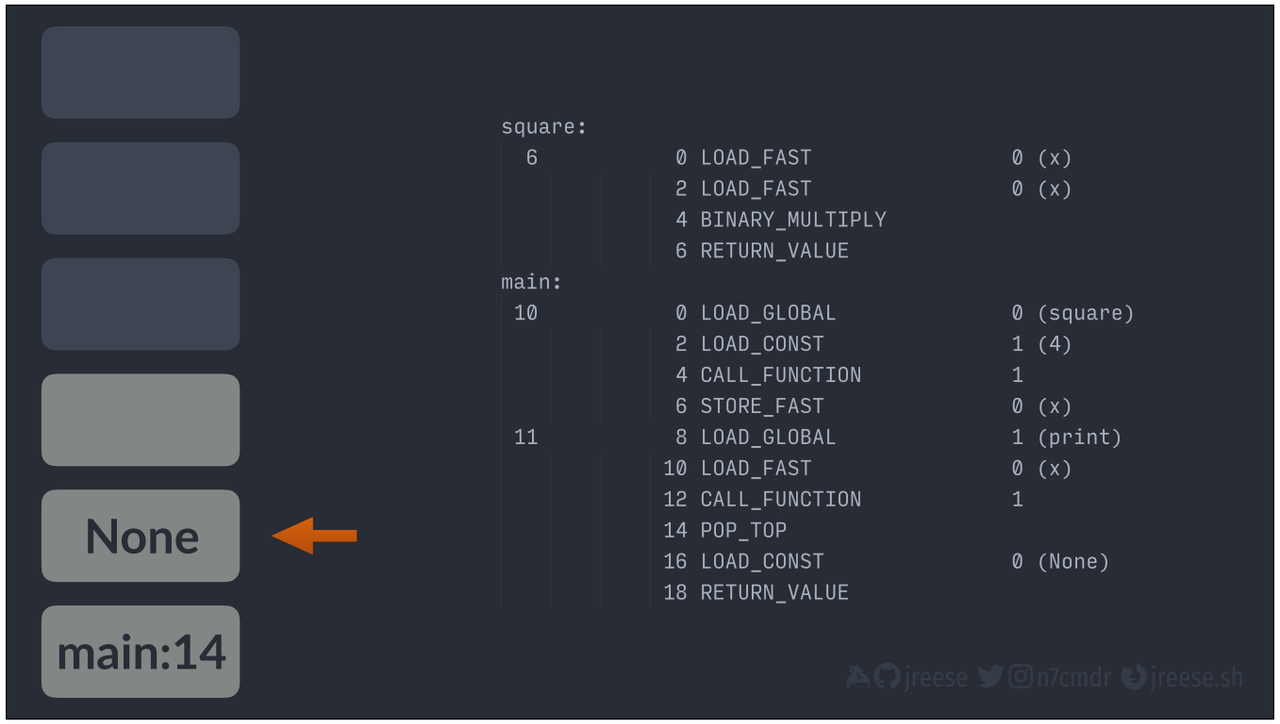


Then again, it will pop the function reference, update the instruction pointer,
And place a pointer back to the next instruction onto the stack

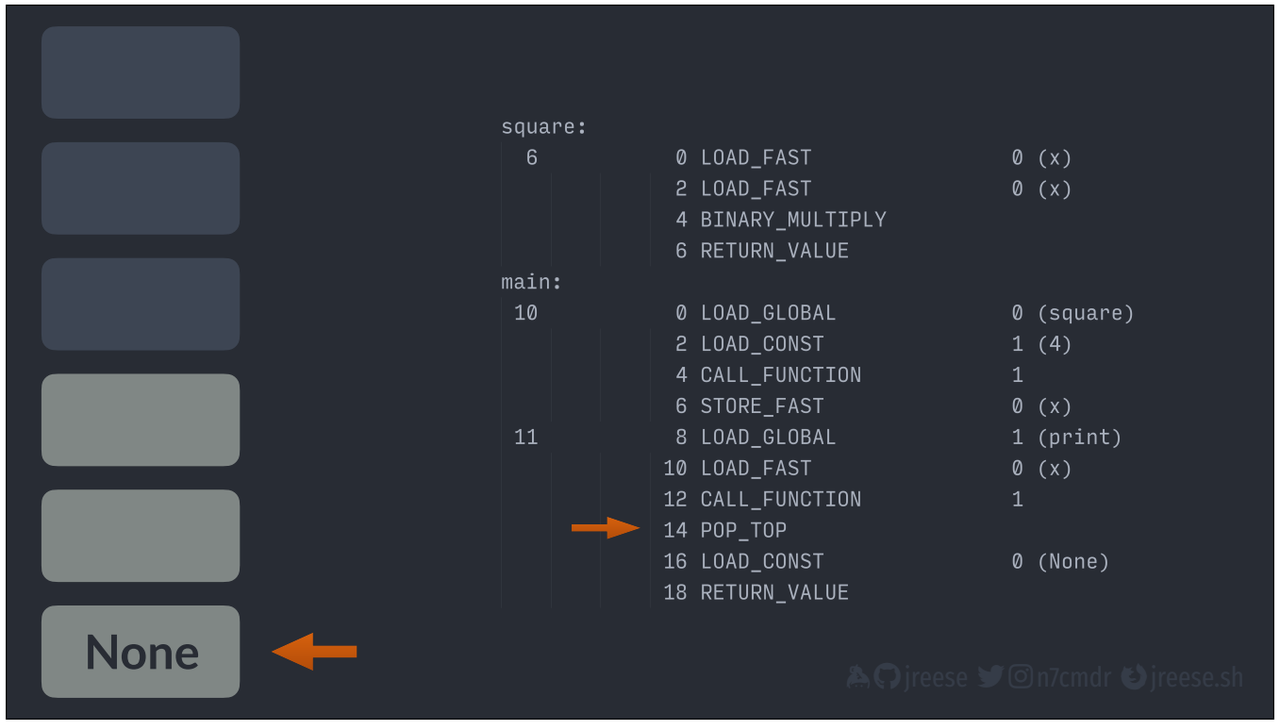
Because the print() function is implemented in C code, there is no byte code that we can follow.
It will essentially execute outside of the normal virtual machine, but results will still be placed on the stack when it returns.



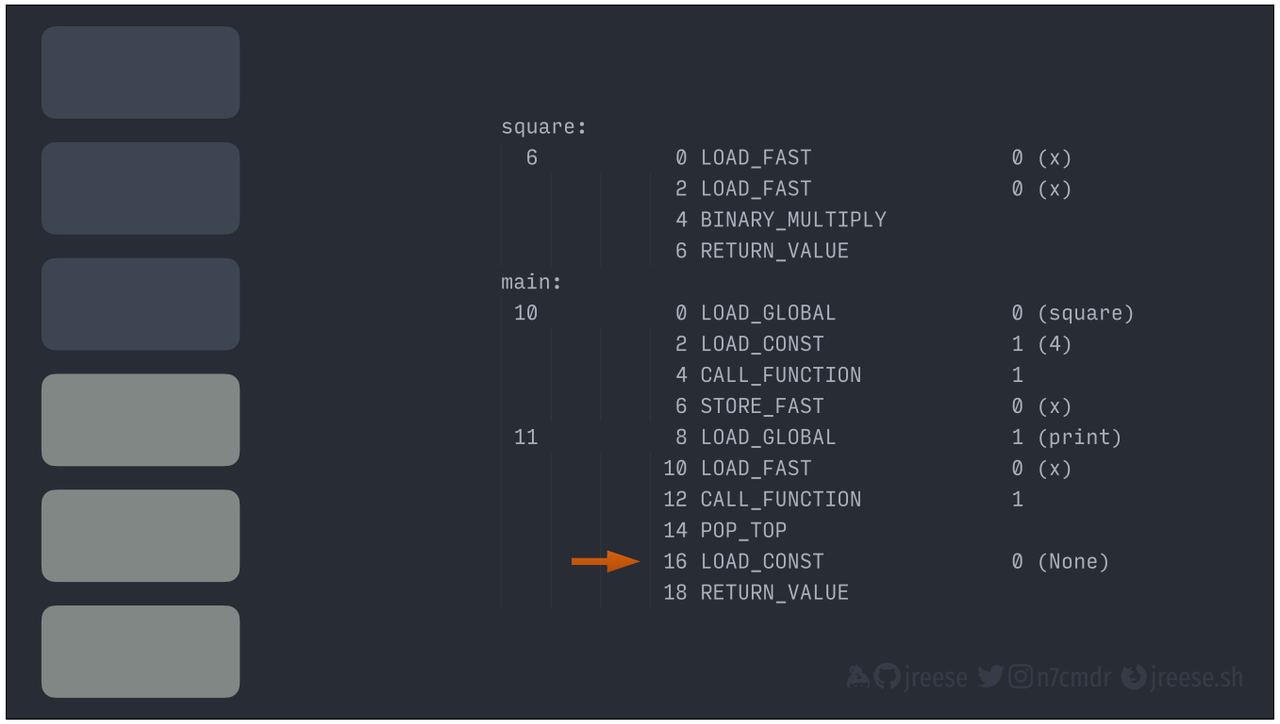
And because this is the print() function, its return value is None



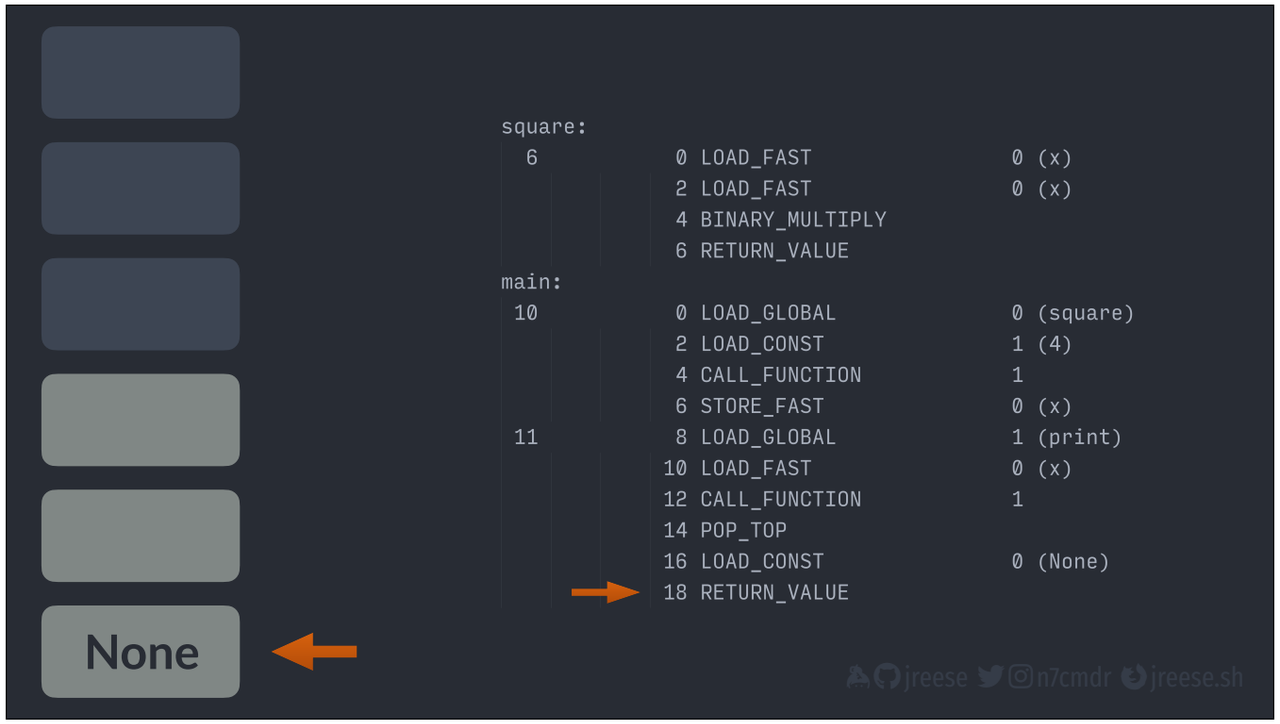
As it returns, the instruction pointer is again restored to where we left off.



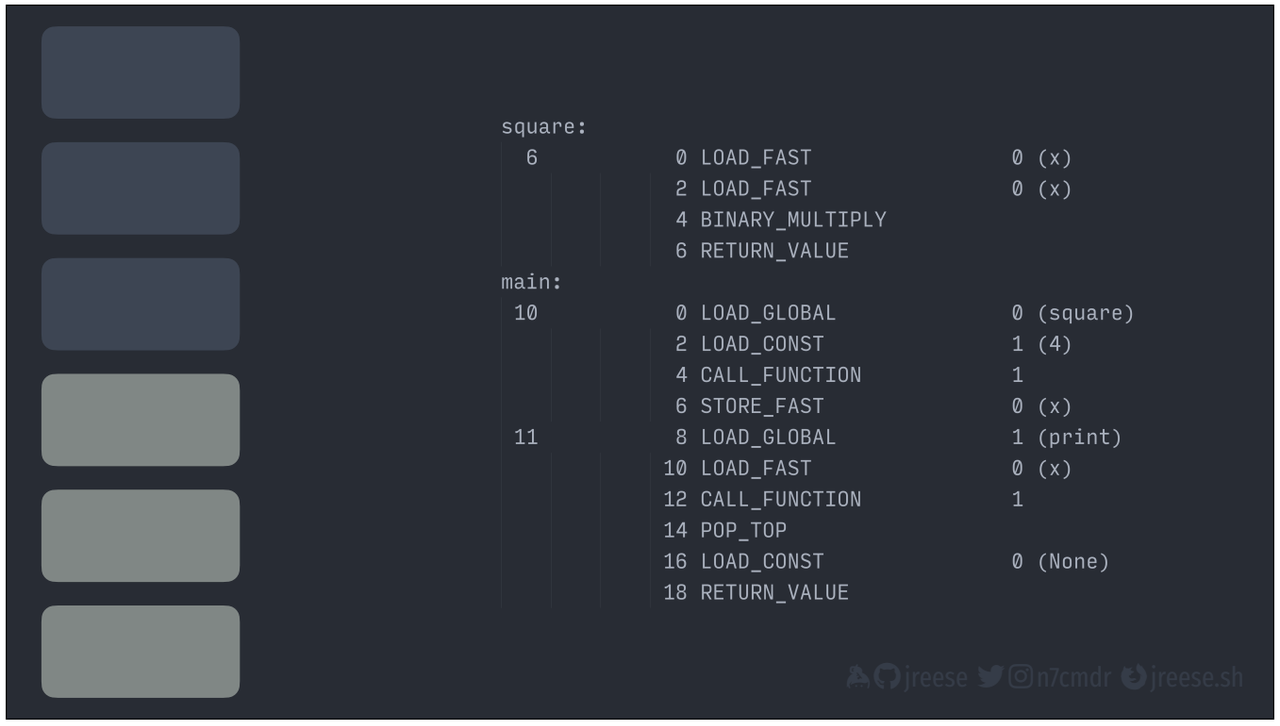
Because we aren’t assigning the result to anything, the next instruction just pops the value from the stack and discards it.



Now the main() function is completed, and we get ready to return the implicit None value.
That constant value gets pushed back onto the stack.
This is a bit ironic since we just popped None off the stack, but at bytecode compile time,
There was no way to know this or expect to be able to reuse that value.



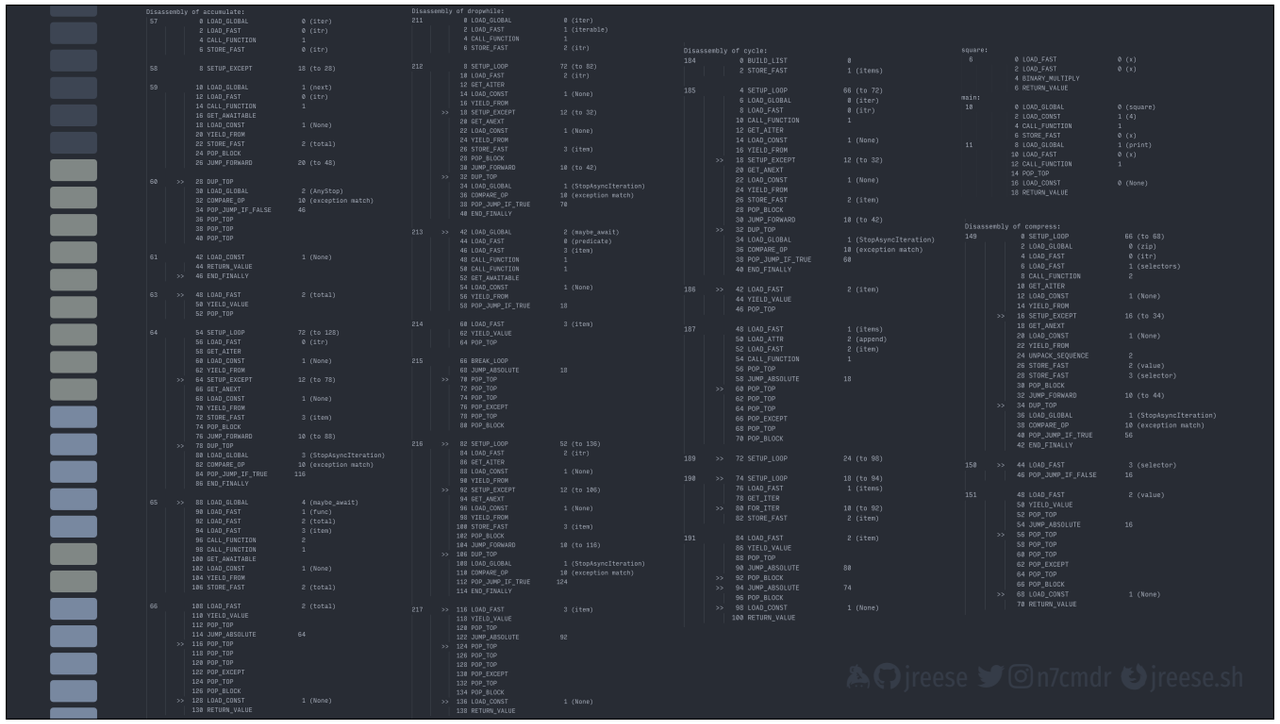
Finally, we can return control to whatever instruction is on the stack underneath our return value, and the main() function is completed.



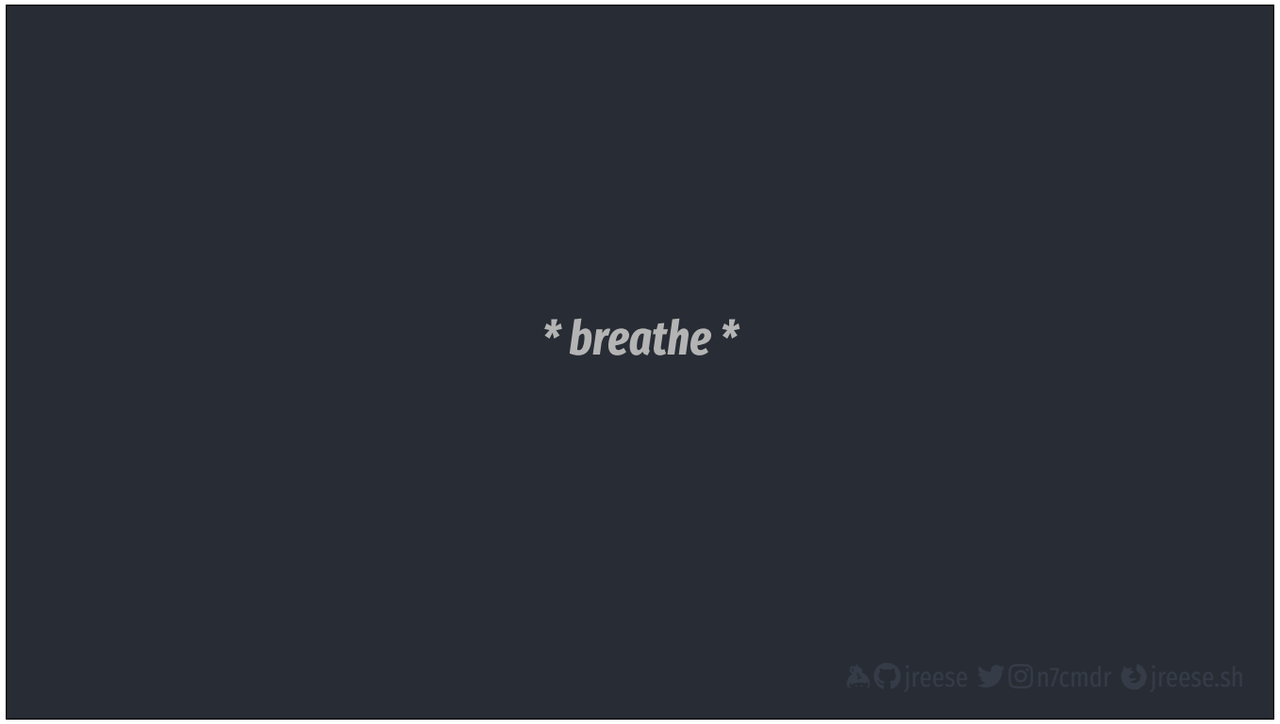
Now, in reality, our stack isn't just a handful of values…

```
square:
  6           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
main:
 10           0 LOAD_GLOBAL              0 (square)
              2 LOAD_CONST               1 (4)
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (x)
 11           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```


But potentially hundreds or thousands of items, depending on how deep the call stack goes, and how complicated the function signatures get.
And any meaningfully complex functions …



… can easily contain dozens or hundreds of instructions.
Entire applications or services are likely to cross a hundred thousand instructions and have enormous stacks.
And the CPython runtime has to keep track of everything.



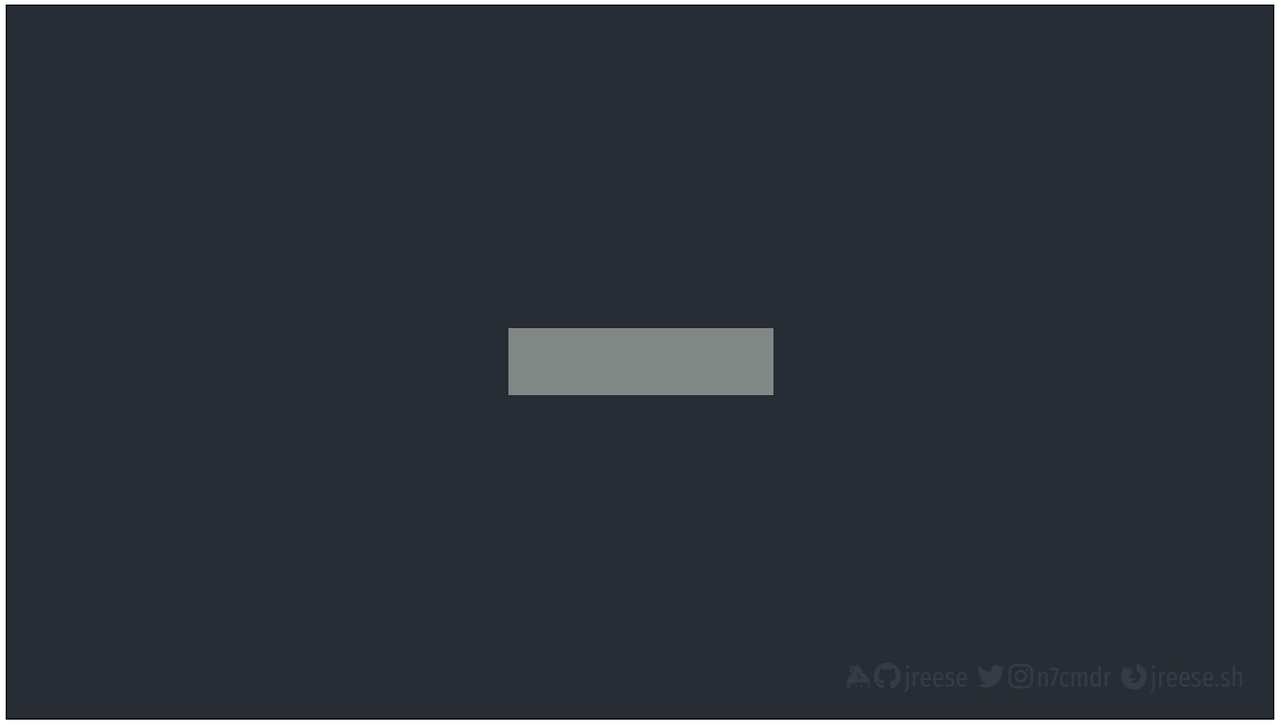
<deep breath>
That's enough bytecode for one weekend I think.

So what about concurrency?

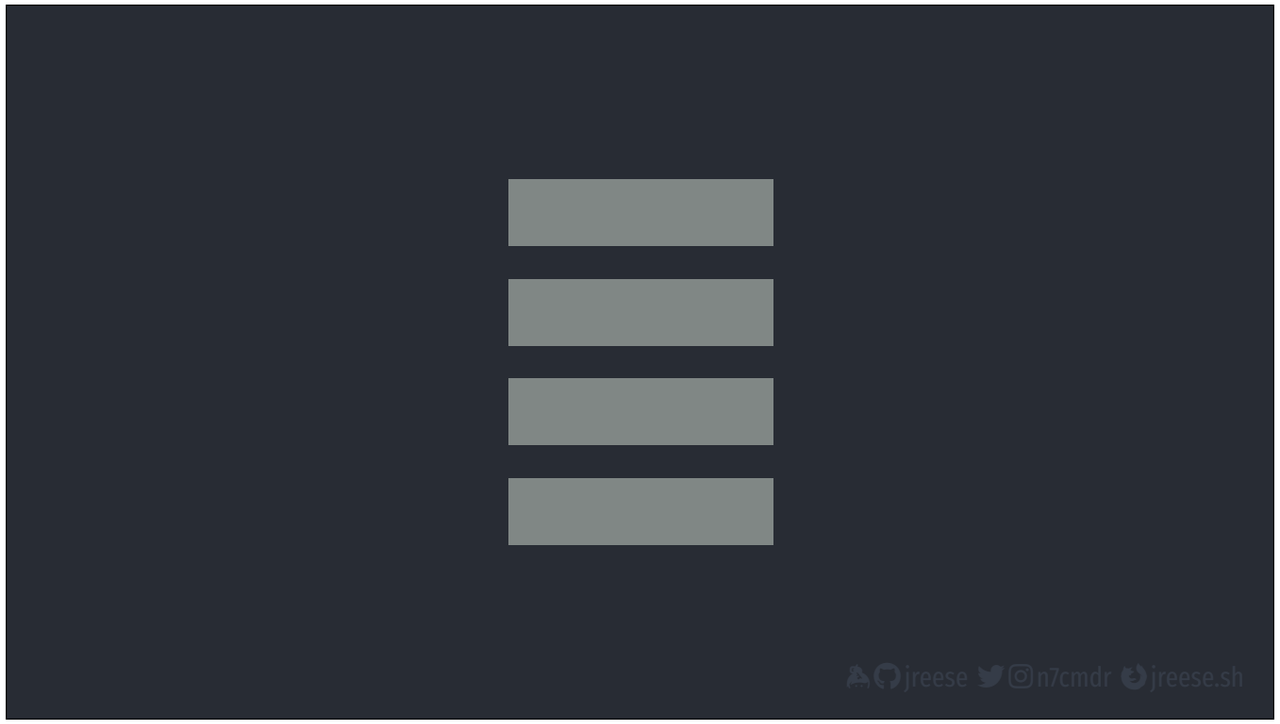
So let's talk about concurrency! It's a simple idea, but often gets conflated with other concepts.

the execution of multiple, overlapping tasks

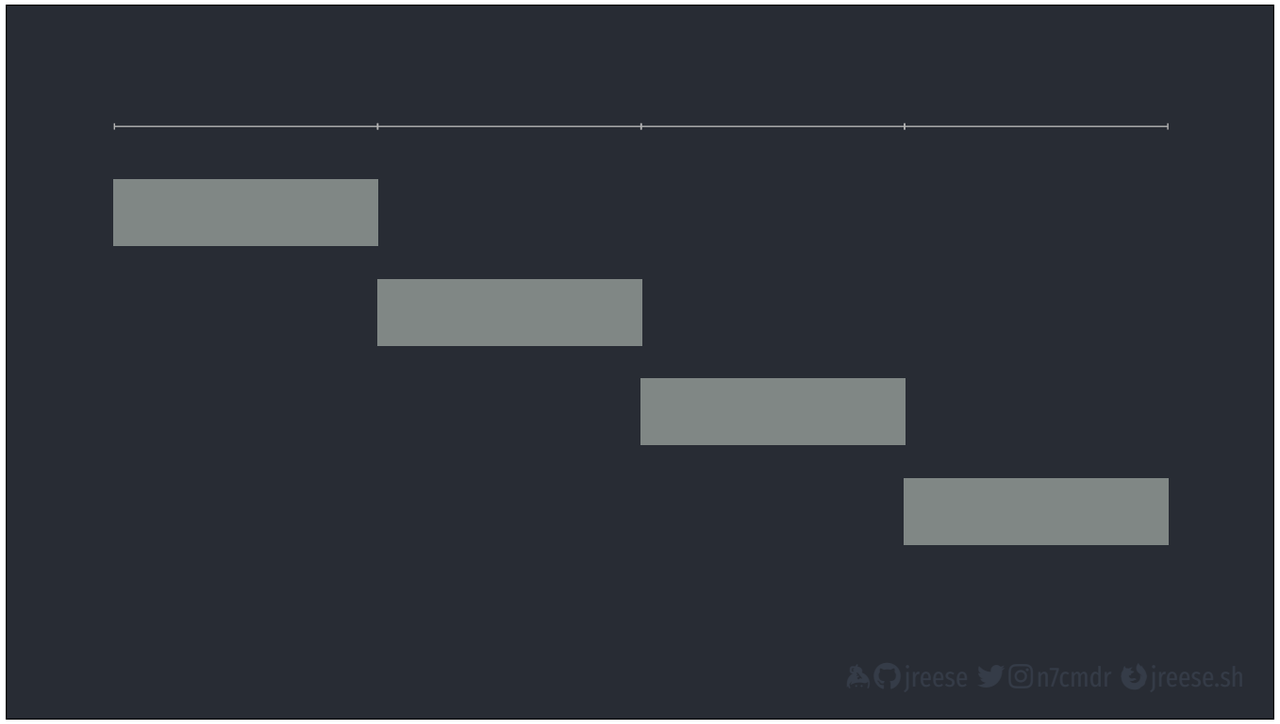
At the most basic level, concurrency is about executing multiple tasks.



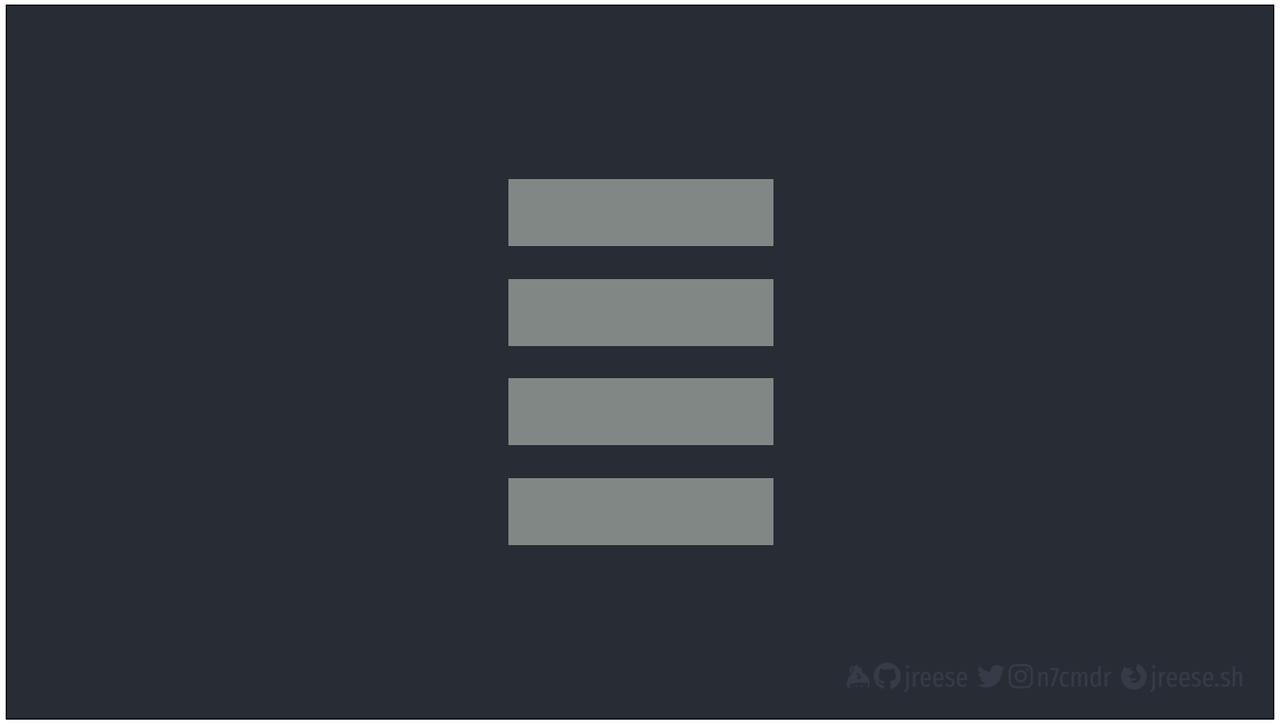
A task could be pretty much anything, but we'll represent it here with this bar, and imagine we move from left to right as we execute this task.
But no one ever has just one task...



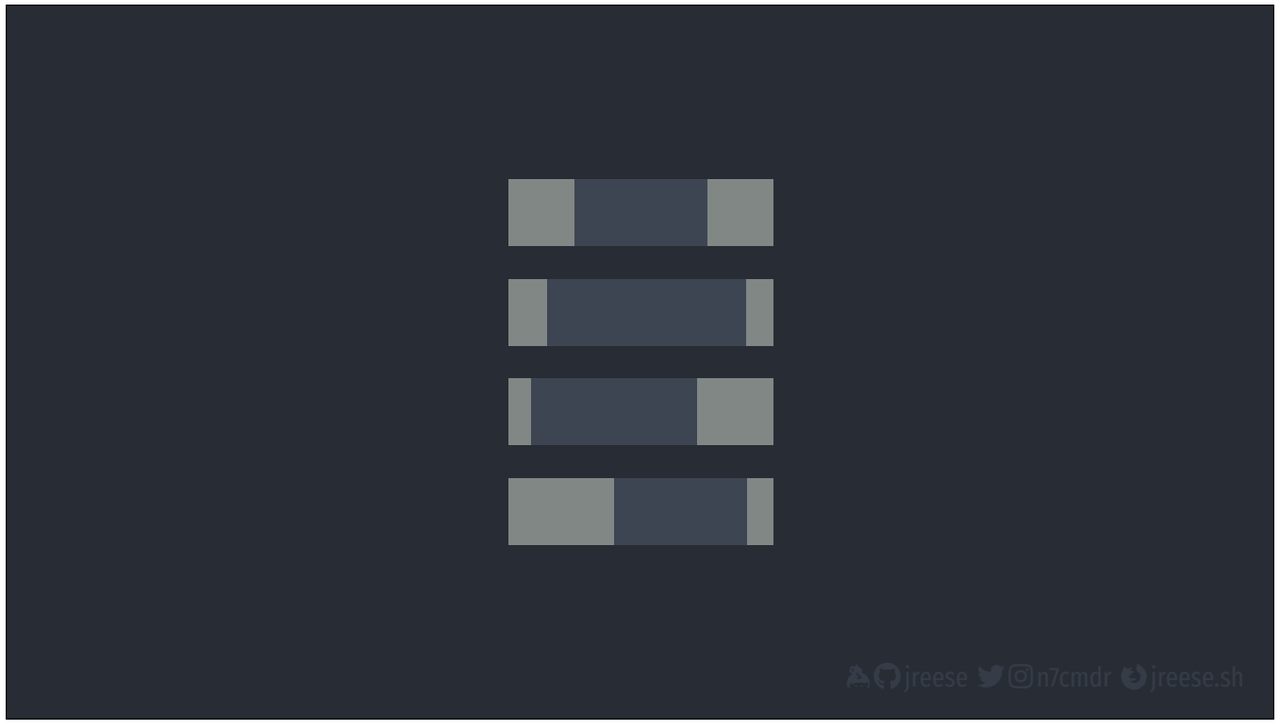
we have multiple tasks that we need to complete, often orders of magnitude more tasks.
The naive solution is to just process them in order: when one is finished, we start the next



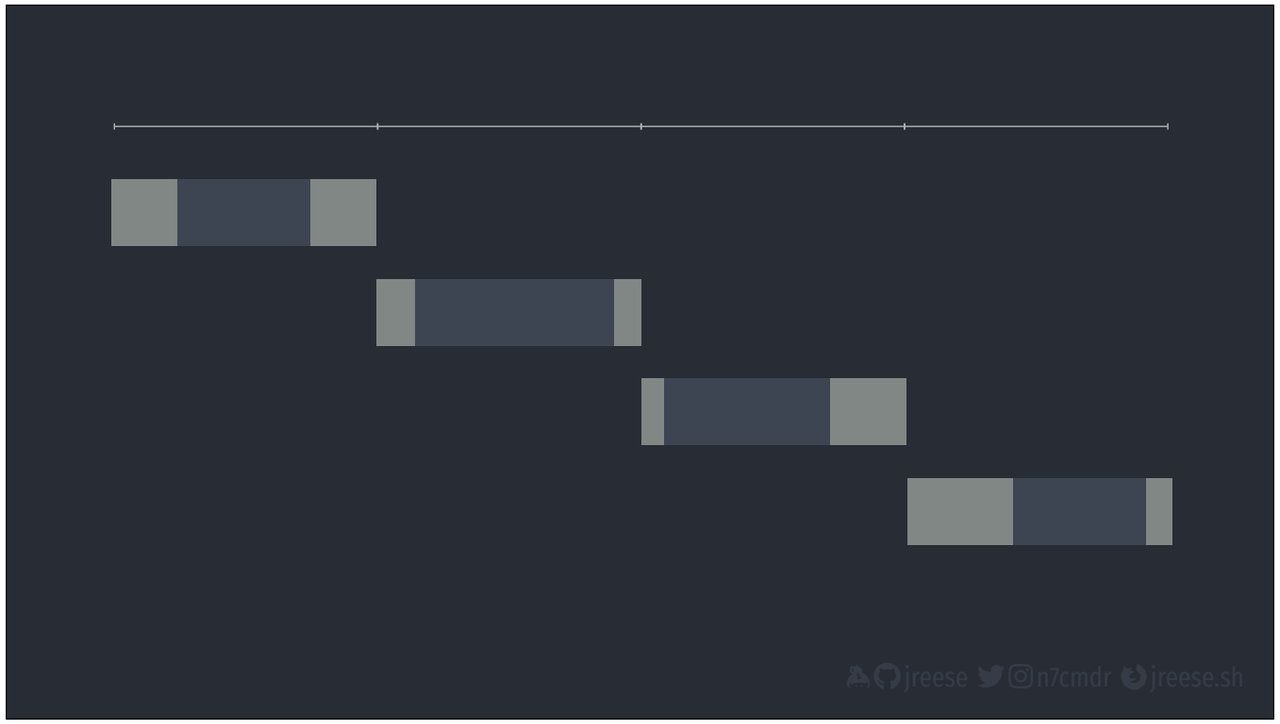
If we have four tasks, it takes four times as long as one task to finish them all.
This is easy. This makes sense.
This is not what concurrency looks like.



For many real world tasks, they also don’t look like homogenous blocks of computation.



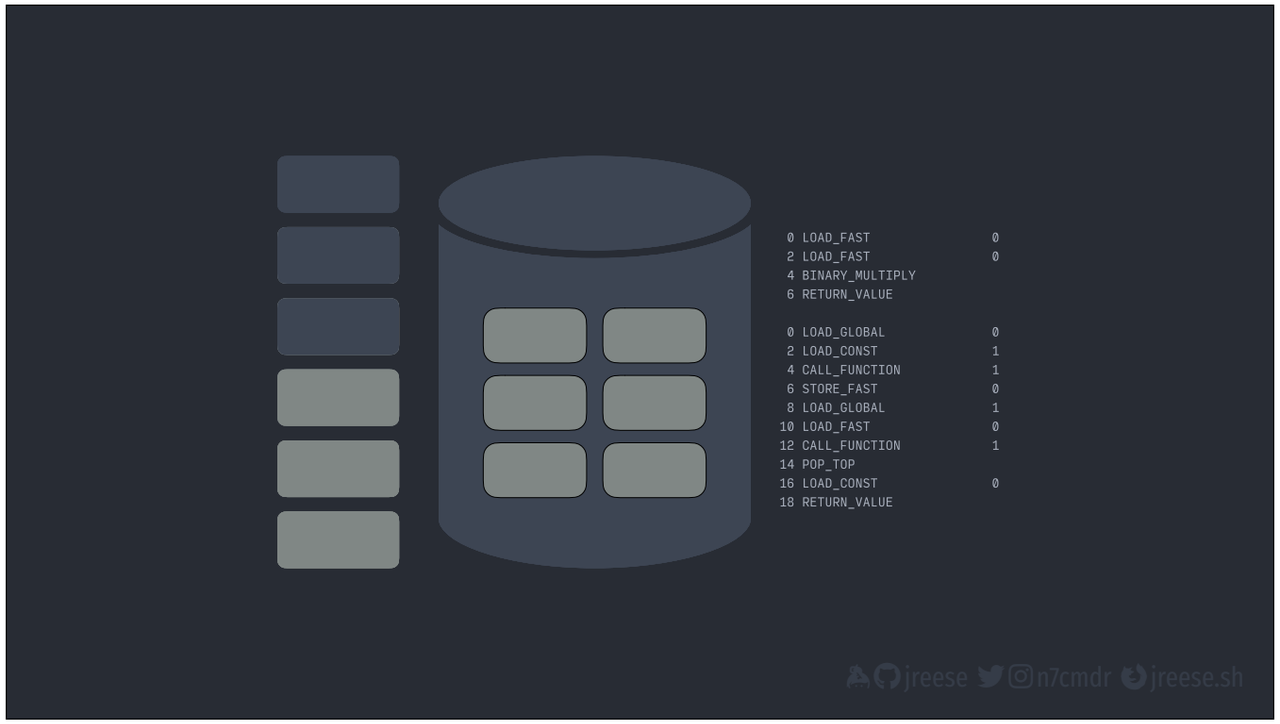
They look more like this, with big chunks of idle time spent waiting for something, like fetching data from disk, or making a network request.



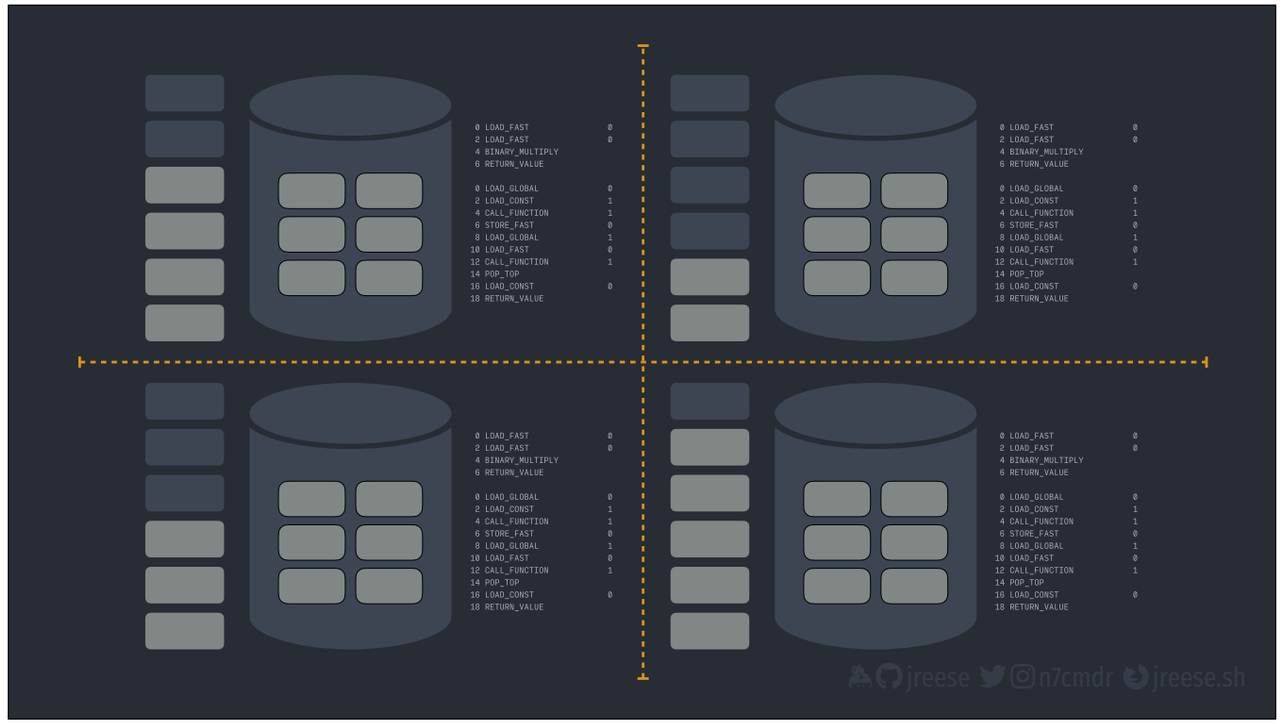
If we apply the naive method from before, and execute them sequentially,
we spend a lot of time idle, and it still takes us four times as long as a single task.

Instead, if we can fully utilize the idle moments to begin executing other tasks,
we can finish all four tasks in much less time.
The more time we spend idle on any individual task, the more tasks we can overlap to save time.
This is the core principle and goal of concurrency.

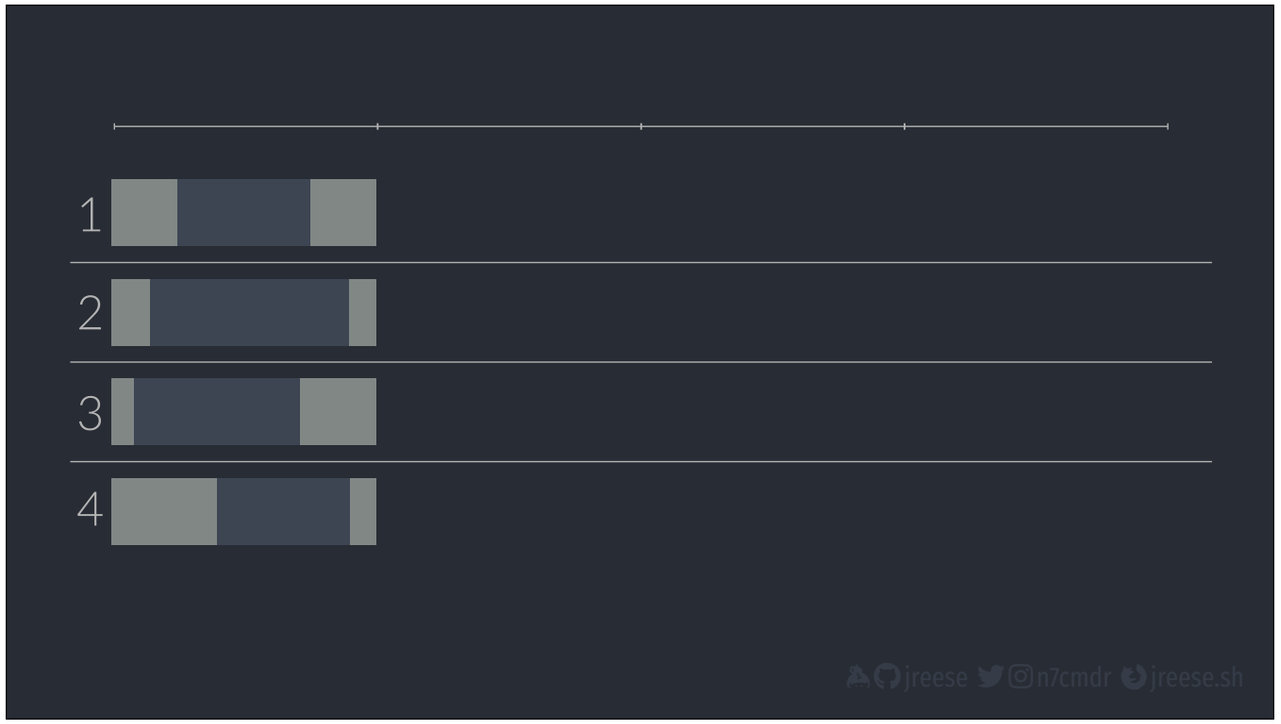
Now, there are a few different strategies for achieving concurrency.
The obvious approach is to just have multiple workers.
Each worker can then process one task at a time.



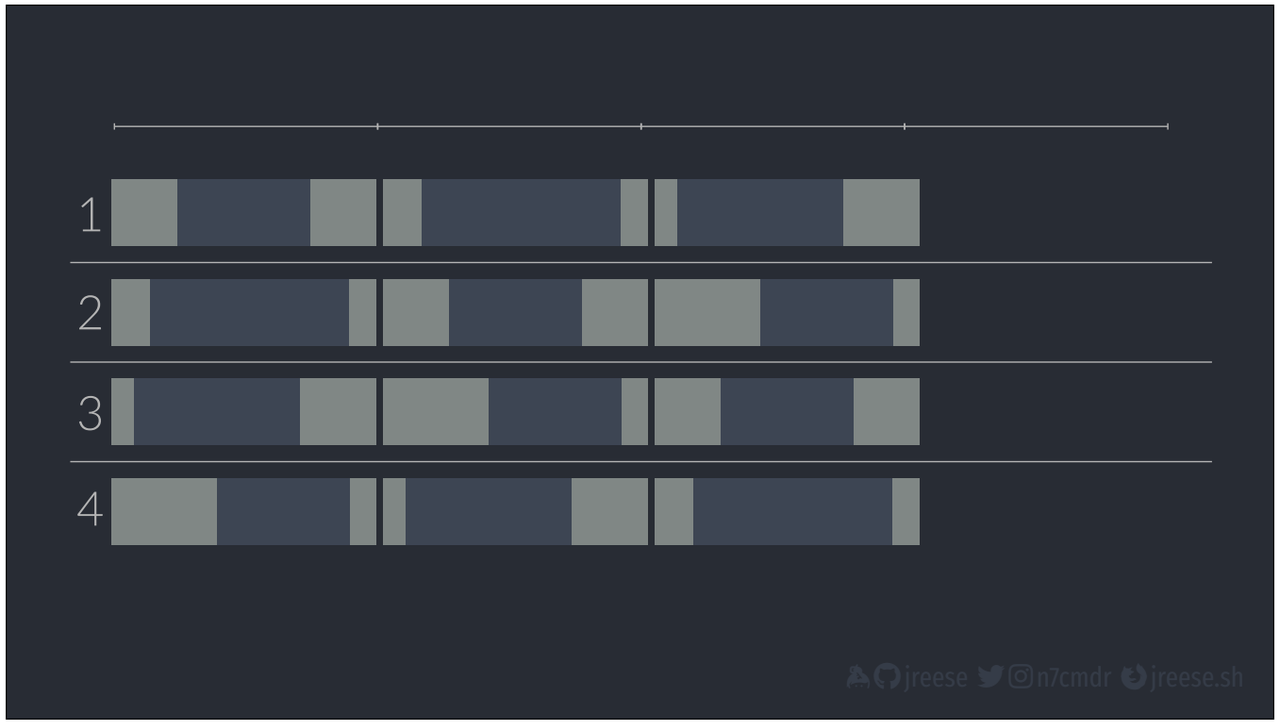
One way to implement that is with multiprocessing, where each worker is its own process.
This means each worker has its own CPython runtime, stack, heap, and compiled bytecode.
If we want to process more tasks at once, we add more workers.



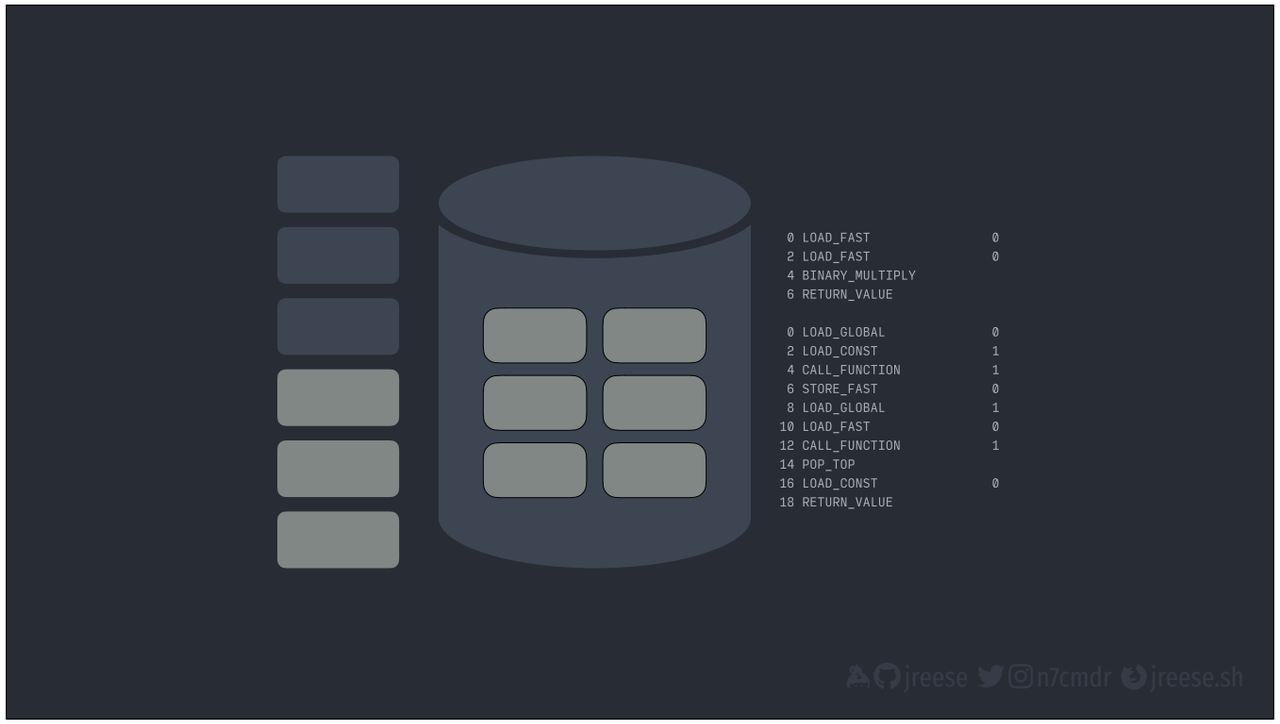
There are some costs associated with this of course. First and foremost is the duplicated memory used for each runtime.
<click>
The second is that any communication between workers has to happen by serializing and deserializing data, adding extra work for each task.



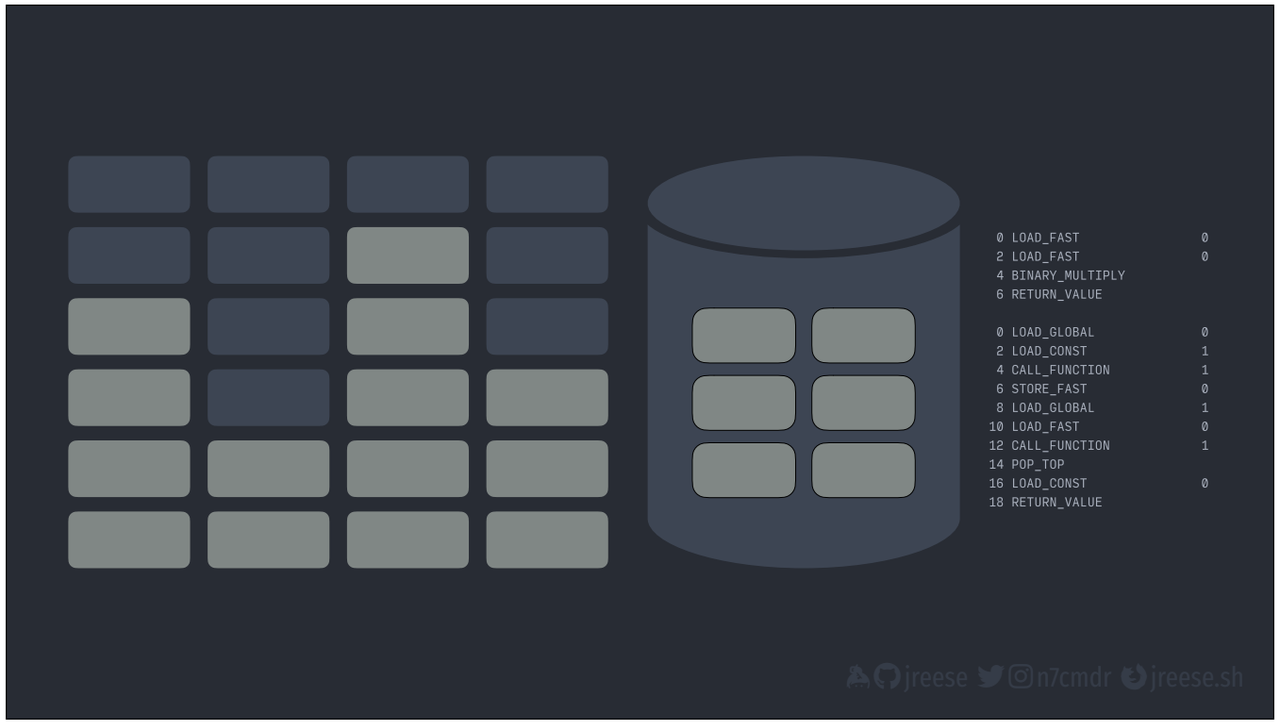
On the plus side this means each worker is able to process its task in parallel, giving us higher potential utilization of multiple cores.



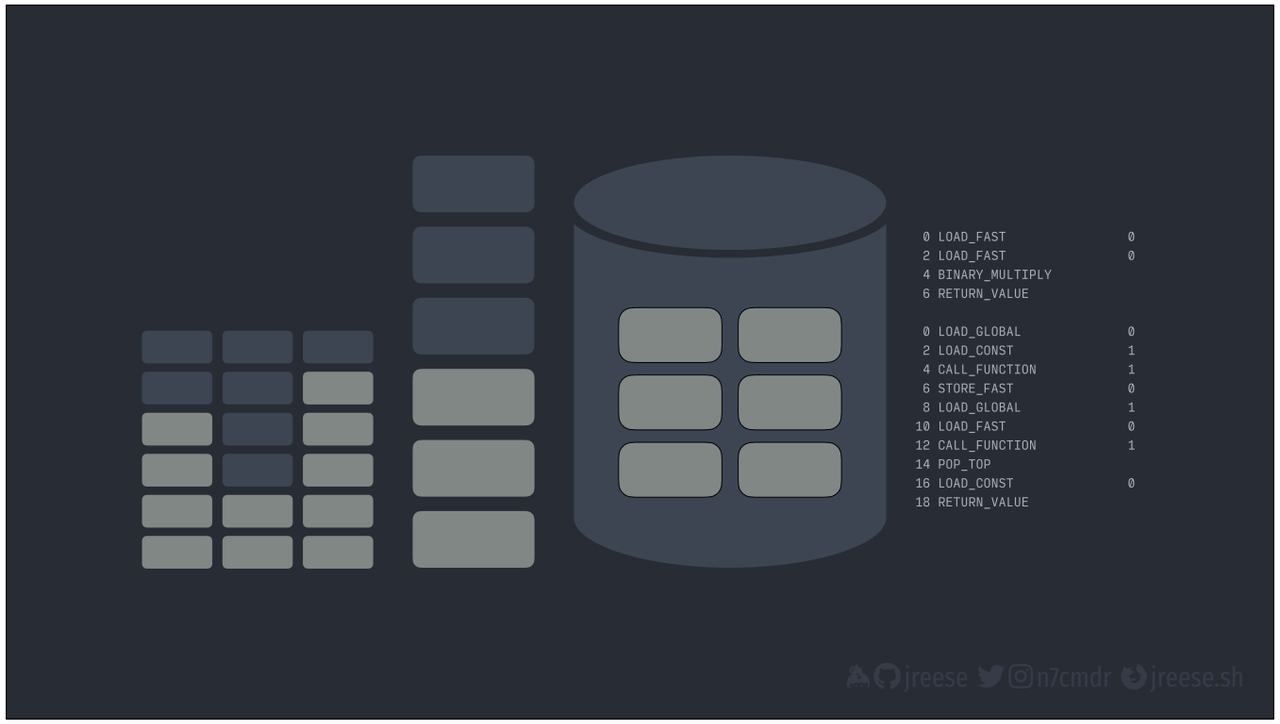
This is called parallelism, and is the younger, more popular sibling to concurrency.
But while the overall throughput generally scales with the number of workers,
each worker, and therefore each process, is still idle while a task is waiting on resources.
So another alternative is to use threads for each worker, rather than processes.



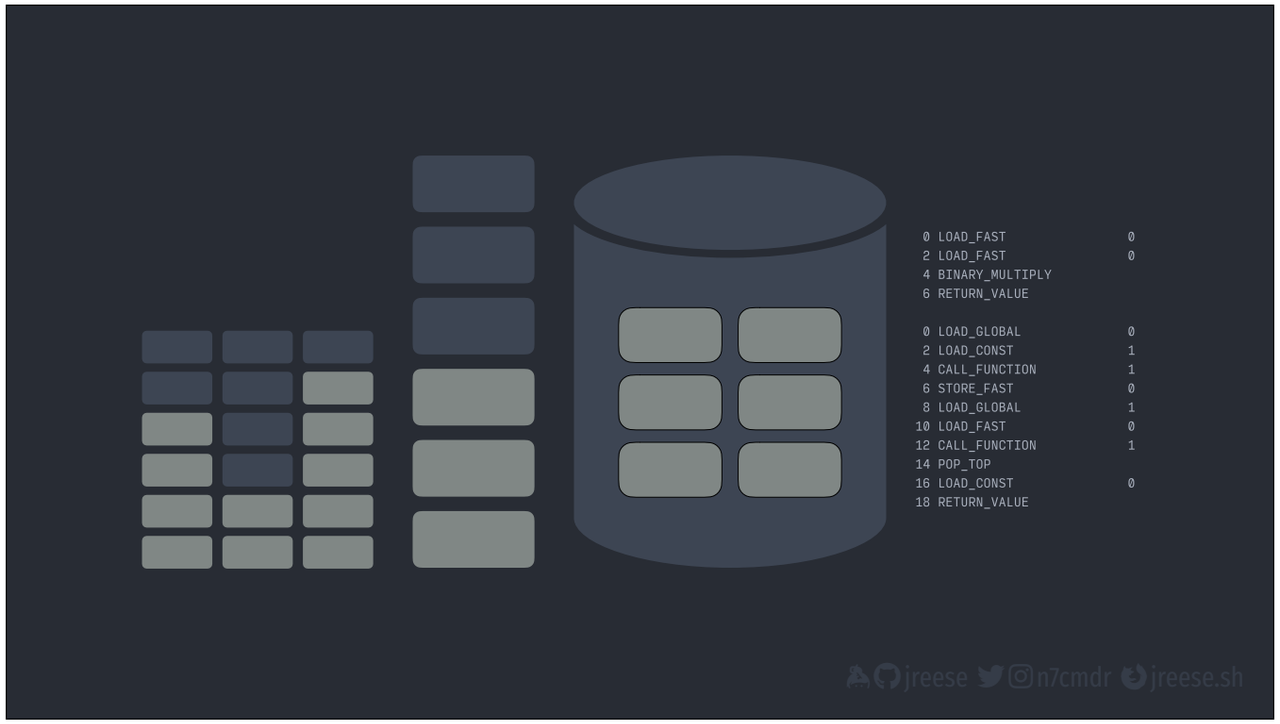
With one worker, we start from the same point as when using processes. But this time, as we add more workers …



We only add a new call stack for each thread, reducing the duplication of the heap and bytecode.
This also eliminates the need for serializing data between workers, because now they all share the same pool of memory.
But as always, this option has its own costs and tradeoffs.



Due to the *unique* way that CPython was designed, with our good friend GIL,
we can’t run multiple threads of Python code at the same time.
So all other threads have to sit idle and wait their turn while the current thread is executing.



On top of this, the runtime is in charge of scheduling threads, with no insight into what they are doing.
And every time a new thread is selected, <click> we have to wait for a costly context switch,
which involves saving the execution state of the previous thread, and restoring state for the new one

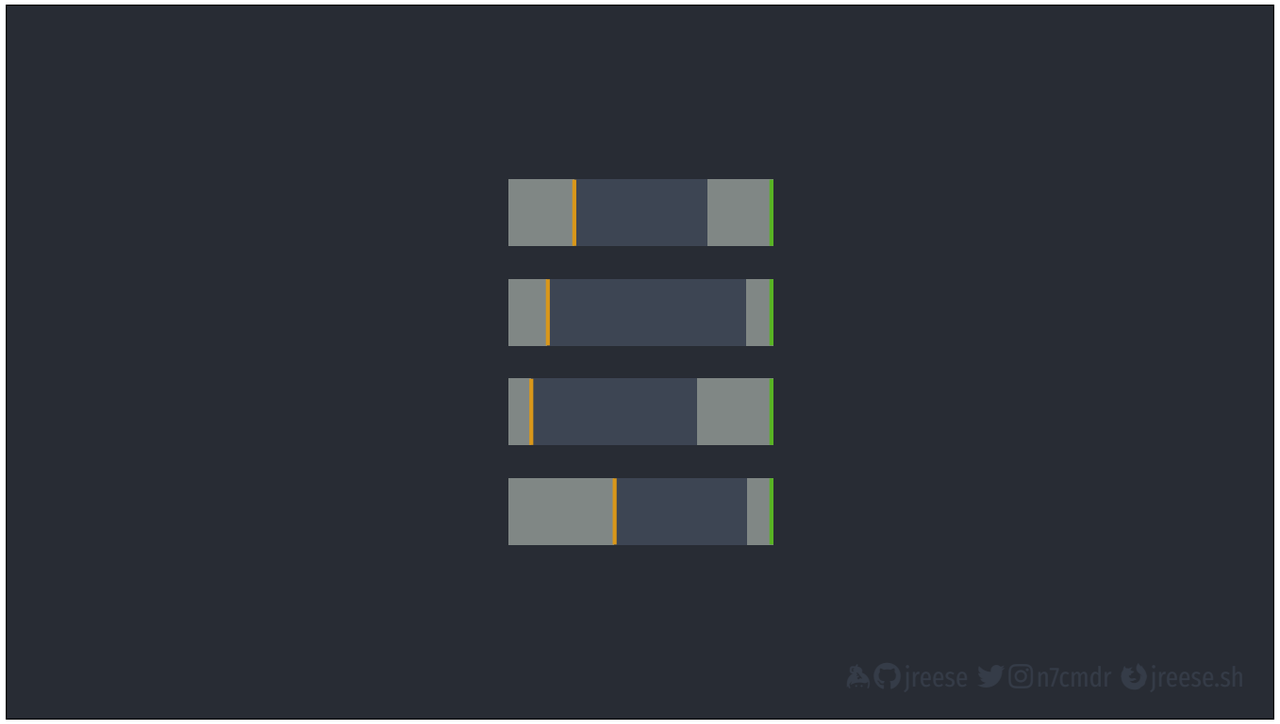
This is what’s known as “preemptive multitasking”, it guarantees “fair” access to the CPU.
This does make it easier to fully utilize a CPU core with multiple threads,
but we are more likely to switch threads at the wrong time,
resulting in suboptimal behavior, like delaying the start of network requests.

This can dramatically increase the amount of time an individual task may take,
like the top task that takes twice as long as it should to complete.
This might be a worst case example, but more threads means more context switching,
increasing the chance your tasks will be interrupted at critical times.

So both multiprocessing and threading have different advantages and disadvantages,
but neither of them begin to approach this “ideal” form of concurrency that I showed earlier.
What we really want is a system where tasks can run through their critical phases,
And only switch to the next task once they are waiting on external requests.

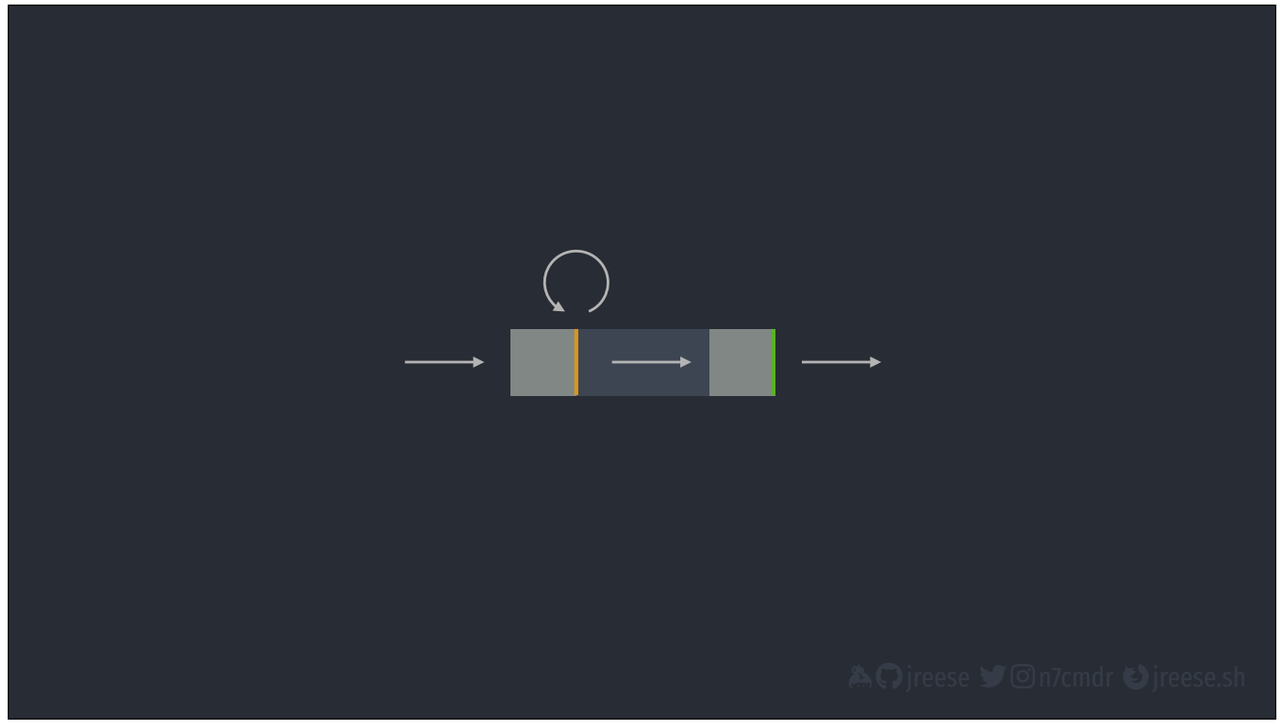
Cooperative Multitasking

This is called cooperative multitasking, and it's a simple and inexpensive form of concurrency.
The tradeoff is that it depends on each task being well-mannered,
and cooperating with other tasks for CPU time and shared resources.
But in an application where we control everything, it can result in major wins.



In practice, this means that each task needs to either complete, <click> or explicitly yield control before another task can execute.
The obvious points to yield control are when the task is ready to wait on external requests,
because there’s nothing this task can do otherwise.

Once we get this level of control from our task, and can schedule them more optimally,
we can make better use of our resources, and run all our tasks on one thread,
in one process. So we only have one shared stack of calls, and one shared heap.



So if we think about how to build cooperative tasks, we can come across a simple solution.
We can build tasks that follow a simple pattern: <click>
make progress when possible, yield when it’s not, and eventually complete and return a value.

```python
class Task:
    def __init__(self):
        self.ready = False
        self.result = NoResult

    def run(self) -> None:
        raise NotImplementedError
```


If we wanted to implement this in Python, we might start with something like this.
We can just execute the run() method repeatedly, and expect that it will return, or yield,
when it's most convenient for the individual task, and eventually set the ready attribute to True.
At that point, we can check the result for the final value.

```python
class Sleep(Task):
    def __init__(self, duration, result=None):
        super().__init__()
        self.threshold = time.time() + duration
        self.result = result

    def run(self):
        now = time.time()
        if now >= self.threshold:
            self.ready = True
```


For something even more useless on its own, we could implement a sleep task.
Given a duration and final value, every time we run this task, it will check the time.
Once enough time has passed, it marks itself as ready, indicating that the result is available.
But a task on its own isn’t useful, we want something that will execute many tasks concurrently.

```python
def wait(ts: Iterable[Task]) -> List[Any]:
    orig: List[Task] = list(ts)
    pending: Set[Task] = set(orig)
    before = time.time()

    while pending:
        for task in list(pending):
            task.run()
            if task.ready:
                pending.remove(task)

    print(f"duration = {time.time() - before:.3}")
    return [task.result for task in orig]
```

What we want is an “event loop”. We will use this as our framework for running our tasks.
It takes a list of task objects, and repeatedly calls the run() method on pending tasks.
As each task marks itself ready, the event loop will run fewer tasks in each iteration.
Once all tasks are completed, it returns a list of the final results.

```python
def main():
    tasks = [Sleep(randint(1, 3)) for _ in range(10)]
    wait(tasks)

duration = 3.0
```


Now we can create a number of tasks, in this case ones that sleep for a random amount of time
We can then run our event loop on these tasks.
all ten tasks get run to completion <click>
and results come back after the longest task is finished

```python
def main():
    tasks = [Sleep(randint(1, 3)) for _ in range(1000)]
    wait(tasks)

duration = 3.0
```

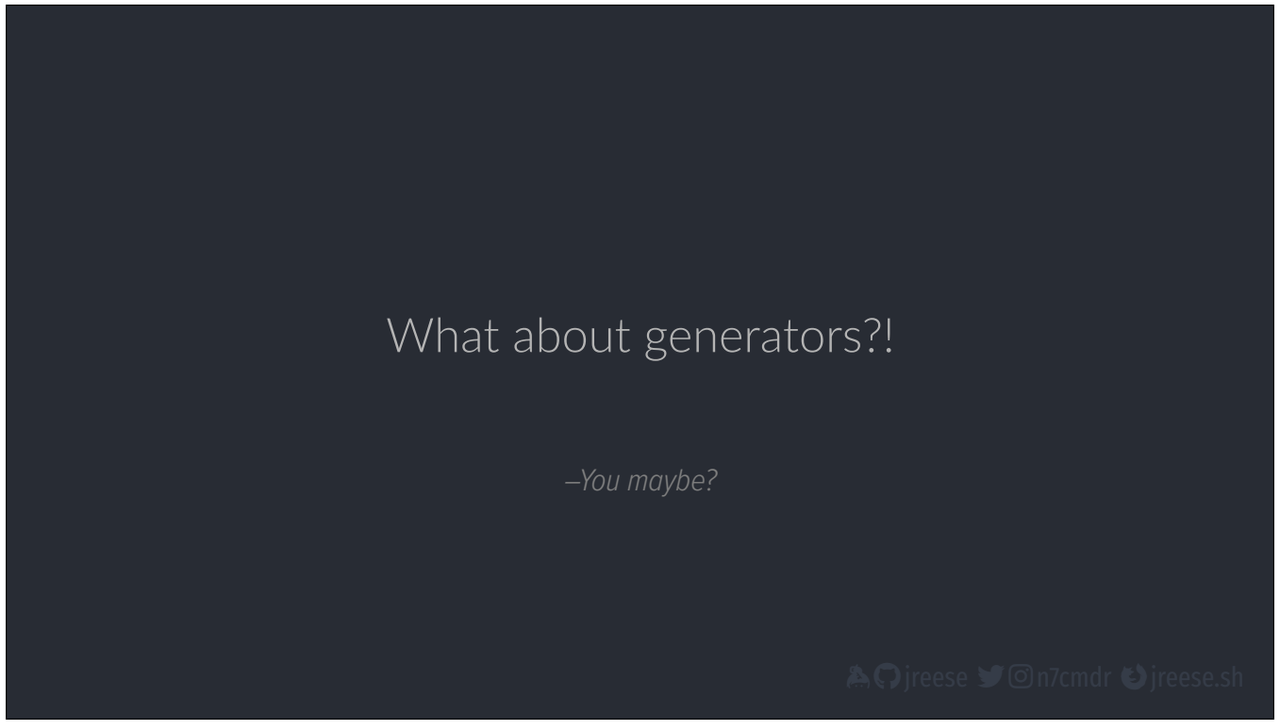

And if we give it a thousand of these tasks, <click>
even this naive event loop is still able complete them in less time than it would have taken
to spawn, run, and compile the results from a thousand threads or processes.

```python
class Task:
    def __init__(self):
        self.ready = False
        self.result = NoResult

    def run(self) -> None:
        raise NotImplementedError
```


But this task implementation is ... lacking at best.
Each task has to manually keep track of its progress,
and each task has to design its run() method to start from the beginning every time.

```python
class Task:
    def __init__(self):
        self.ready = False
        self.result = NoResult

    def run(self) -> None:
        raise NotImplementedError
```


Our framework also doesn't make it easy for tasks to call other functions that may also need to wait on their own resources.
If only there was a way write code that could yield multiple values over time, and be able to start executing from where it previously left off…



Some of you may already see where this is heading,
but generators fit this description perfectly.
Let’s look at how they work.

```python
def fib(count: int):
    a, b = 1, 0
    for _ in range(count):
        a, b = b, a + b
        yield b
```


The Fibonacci sequence is a staple of generator examples, and it's conceptually simple
ach time through the loop, we add the previous two numbers together and yield that value
resulting in the sequence [1, 1, 2, 3, 5, …]
But when we call this function, we don’t get any of these values directly

```python
def main():
    gen = fib(5)
    print(gen)
    while True:
        print(next(gen))
```

```
<generator object fib at 0x103deab30>
```


We actually get a generator object. This is a compiled version of our generator function.
The actual code in our function hasn’t started executing yet.
This generator object can then be iterated over, just like a list.
And the standard next() function can be used to iterate just once.

```
def main():
    gen = fib(5)
    print(gen)
    while True:
        print(next(gen))
```

```
<generator object fib at 0x103deab30>
1
1
2
3
5
StopIteration
```


Each time we call next(), it’s reentering the function where it left off, preserving the full state. And if the function yields another value, we get that value as the result of the next() call. When the generator function completes or returns, the generator object raises StopIteration just like any other iterator would when you reach the end.

```python
def counter(start = 0, limit = 10):
    value = start
    while value < limit:
        value += yield value
    yield value
```


Now, it’s quite common to see generators that yield values out,
but it’s also possible to communicate, or “send” values, back in to a generator from the outside.
When this happens, after returning execution to the generator function,
the yield statement itself gives the value that was sent into the generator.

```python
def main():
    gen = counter()
    gen.send(None)  # prime the generator
    while True:
        value = randint(1, 3)
        total = gen.send(value)
        print(f"sent {value}, got {total}")
```

```
sent 3, got 3
sent 2, got 5
sent 3, got 8
sent 2, got 10
```


To do this, we have to replace use of the next() function with the generator's send() method.
This will execute the function from where it left off, and just like next(), will return yielded values.
Until the function has actually been started, we can only send None.
After that, we can keep sending and receiving values until the generator completes.

Congratulations!
This is a coroutine, and Python has had them hiding in plain sight for years!
But how do we use this to actually run concurrent tasks?
As it turns out, we can adapt our event loop from earlier to make it better.

```python
def wait(tasks: Iterable[Generator]) -> List[Any]:
    pending = list(tasks)
    tasks = {task: None for task in pending}
    before = time.time()

    while pending:
        for gen in pending:
            try:
                tasks[gen] = gen.send(tasks[gen])
            except StopIteration as e:
                tasks[gen] = e.args[0]
                pending.remove(gen)

    print(f"duration = {time.time() - before:.3}")
    return list(tasks.values())
```

Instead of calling a run() method on a task, we're calling the send() method on generators
And rather than looking for a flag, we catch StopIteration, and mark those as completed.
Starting in Python 3.3, StopIteration itself contains the return value from generators, so we save those for the final combined result.
Lastly, we also capture intermediate yielded values, and send them back on the next iteration, which enables coroutines to call other coroutines.

```python
def sleep(duration: float):
    now = time.time()
    threshold = now + duration
    while now < threshold:
        yield
        now = time.time()

def bar():
    yield from sleep(0.1)
    return 123

def foo():
    value = yield from bar()
    return value
```


This means that we can “yield from” another coroutine to call into it,
and our position in the stack of coroutine calls will be preserved across yields.
Together, this makes our use of coroutines, look and feel more like standard functions,
but they are still yielding control on their terms, and get to continue when its their turn again.

```python
def sleep(duration: float):
    now = time.time()
    threshold = now + duration
    while now < threshold:
        yield
        now = time.time()

def bar():
    yield from sleep(0.1)
    return 123

def foo():
    value = yield from bar()
    return value
```

```python
def main():
    tasks = [foo(), foo()]
    print(wait(tasks))

duration = 0.1
[123, 123]
```

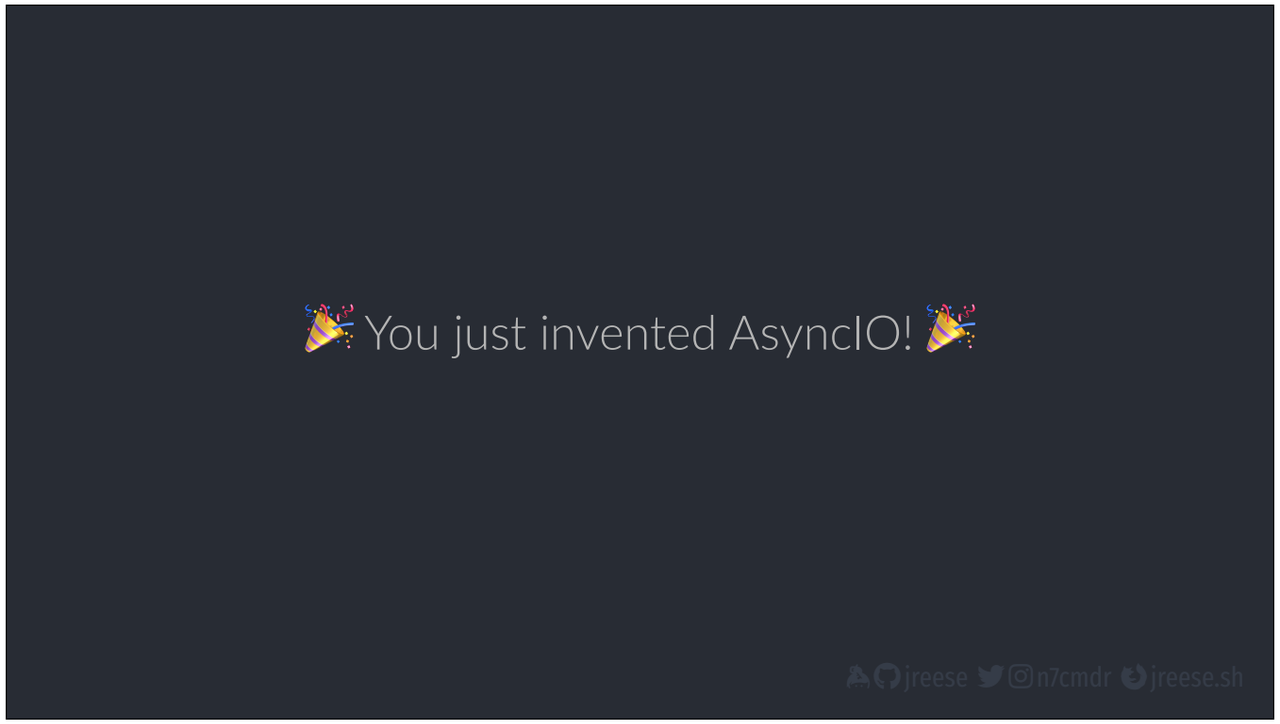

So if we create a pair of coroutines from foo(), and pass them to our event loop,
It will follow execution through foo, into bar, then into the sleep coroutine.
In there, it will continue yielding back to the event loop until the time duration is up.
Then on the next iteration through, it control will return to bar(), which returns a value back to foo, which finally completes and returns the value from bar.
And to be clear, at each yield, our event loop is cycling to the next pending task, giving us the cooperative multitasking concurrency that we've been looking for.

```python
def read(r: Response) -> bytes:
    data = b""
    for chunk in r.iter_content(SIZE):
        data += chunk
        yield
    return data

def fetch(url: str) -> str:
    with get(url, stream=True) as r:
        data = yield from read(r)
    return data.decode("utf-8")
```


Now let’s do something a bit more useful, like fetch content from a bunch of URLs.
We can write a fetch() coroutine, which initiates the connection for a single URL,
and a read() coroutine that buffers data from that connection as it becomes available.
By having the read() coroutine yield after every chunk is read, this allows other tasks the opportunity to execute while waiting on more data.

```
def main():
    coros = [fetch(url) for url in URLS]
    results = wait(coros)
    for result in results:
        print(f"{result[:20]!r}")

duration = 1.43
'\n<!DOCTYPE html>\n<ht'
'<!DOCTYPE html>\n<!--'
'<!DOCTYPE html>\n<htm'
'<html op="news"><hea'
'<!doctype html>\n<!--'
```


We can then call fetch() a bunch of times to create the generator coroutines.
Our event loop will start each one, and run them concurrently until all requests are completed.
We then have the raw response data for each request, and we can do with it as we please.



Congratulations! What we just built is a very primitive version of AsyncIO,
using the same syntax from Python 3.4, but with absolutely no bells or whistles.
Now, this example code was informative and fun to play with, but it isn't very flexible,
and it won't help us if we use it with libraries that aren't designed for it.

DIY doesn’t scale

Please, do not use this in production. There's a reason I didn't release this on PyPI.
So now that we've seen how we can build coroutines up from various features in Python,
let's take a look at how Python has built first class support for coroutines directly into the language and the standard library.

```python
async def foo():
    return 37
```


Starting with 3.5, Python now supports the "async def" syntax for declaring "native" coroutines.
Like our toy generator coroutines before, this is no longer a standard function that executes immediately when called.
Calling a coroutine function returns a coroutine object, which can be run on an event loop.

```
result = asyncio.run(coroutine)
```


This event loop is provided by the AsyncIO framework,
and it executes tasks in round robin order, just like our wait() function from before.
Here, we use the new helper from 3.7 that will create an event loop, execute our coroutine,
then close the event loop when the coroutine is completed.

```python
async def foo():
    return 37

def main():
    coro = foo()
    print(coro)
    result = asyncio.run(coro)
    print(result)
```

```
<coroutine object foo at 0x10c80d240>
37
```


So let's do that: we call the foo() coroutine function, print that coroutine object,
then run the coroutine on the event loop, and get and print the result when it's finished.
We can see the coroutine object itself we got from calling foo(),
and it only starts executing once we pass it to asyncio.run()

```python
await asyncio.sleep(delay: float)
```


Inside a coroutine, we now have a new power available to us: the ability to "await" objects.
This is similar to our use of `yield from` in our generator coroutines earlier, but even better.
We're using asyncio.sleep(), which serves the same purpose as our sleep() generator earlier.
This provides a simple way to yield control of the event loop as well when passing delay=0

```
await anything
```


But we can do more than just await other coroutines.
We can await a large variety of asynchronous objects, including "awaitables" and "futures".
This flexibility makes it easier to bridge old style, synchronous libraries, with async codebases,
and also provides us with more expressive forms of building concurrency into our applications.

```python
async def sleepy(duration: float):
    print("sleeping...")
    await asyncio.sleep(duration)
    return 37

async def foo():
    value = await sleepy(1.0)
    return value
```


So now if we take our coroutine from before, we can call and await the sleepy() coroutine.
Sleepy can itself do something, await other things, then return a value.
This value is then returned as the result of the await keyword in foo(),
and can be used or assigned as usual.

```python
results = await asyncio.gather(*futures)
```


One of the common workflows in concurrent programming is to create multiple subtasks.
If we just used the await keyword directly on each subtask, we wouldn't get any concurrency.
Instead, if we want to await multiple things at the same time, and actually run them concurrently, we need to use the asyncio.gather() helper.

```python
results = await asyncio.gather(*futures)
```


This takes a number of futures, coroutines, or awaitables, runs them concurrently,
and returns the results in the same order.
This is the direct AsyncIO equivalent of the wait() function that
we built earlier for our generator coroutines.

```python
async for value in iterable:
    pass

async with context as c:
    pass
```


We also have language support for asynchronous iterables and context managers.
Async iterables are just like standard iterables, but use a coroutine for fetching the next item.
Similarly, async context managers use coroutines instead of normal functions
when entering and exiting the contexts.

Lastly, because I love a good "yo dawg" meme, we have support for asynchronous generators.
These build on top of generators and coroutines to give us, you guessed it, more generators!
<click> These are defacto async iterables, though, which makes them fantastically useful
for building expressive async interfaces without sacrificing on readability or maintainability.

```python
async def fetch(url: str) -> str:
    async with request("GET", url) as r:
        return await r.text("utf-8")
```


So let's put this all together into one final demo, and build an AsyncIO version of URL fetching
Here, we implement fetching a single URL, using the aiohttp library.
Calling request() gives us an async context, which connects and gives the response object.
We can then await the text() method coroutine to wait for the fully body of the response.

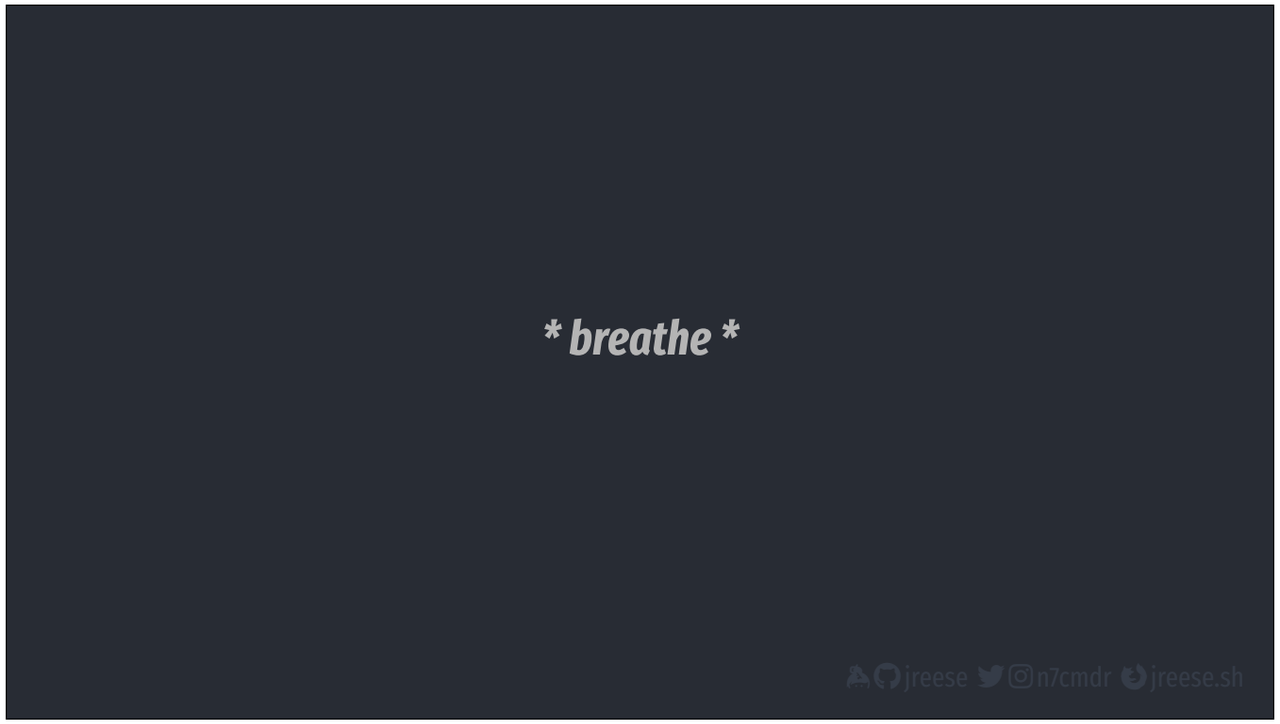
```python
async def main():
    coros = [fetch(url) for url in URLS]
    results = await asyncio.gather(*coros)
    for result in results:
        print(f"{result[:20]!r}")
```


From our main() coroutine, we can call fetch() for each URL, giving us a list of coroutine objects.
We can then pass those to asyncio.gather(), which will execute each one as a concurrent task.
Then we can take the results from awaiting the gather, and print the responses to the console.

```python
async def main():
    coros = [fetch(url) for url in URLS]
    results = await asyncio.gather(*coros)
    for result in results:
        print(f"{result[:20]!r}")
```

```
'\n<!DOCTYPE html>\n<ht'
'<!DOCTYPE html>\n<!--'
'<!DOCTYPE html>\n<htm'
'<html op="news"><hea'
'<!doctype html>\n<!--'
```


We we run this with asyncio.run(), we can then see that the results match the results we got with our generator coroutines and custom event loop.



<deep breath>

AsyncIO is not magic

jreese n7cmdr jreese.sh

Cooperative multitasking has been in use for many decades, and was even a core feature of early Mac OS and Windows.
Coroutines have been an evolving feature of the Python language for well over a decade.
AsyncIO as a framework has been around in one form or another for over six years.

AsyncIO is not magic

None of it is magic. Everything we see is the result of iterative design and development, building on top of previous features or abstractions, always learning from previous lessons. It just takes a bit of curiosity to peel back the onion, decipher the motivations, and understand how and why each decision was made, and how that influences the future. We stand on the shoulders of giants.

PEP 342 – Coroutines via Enhanced Generators

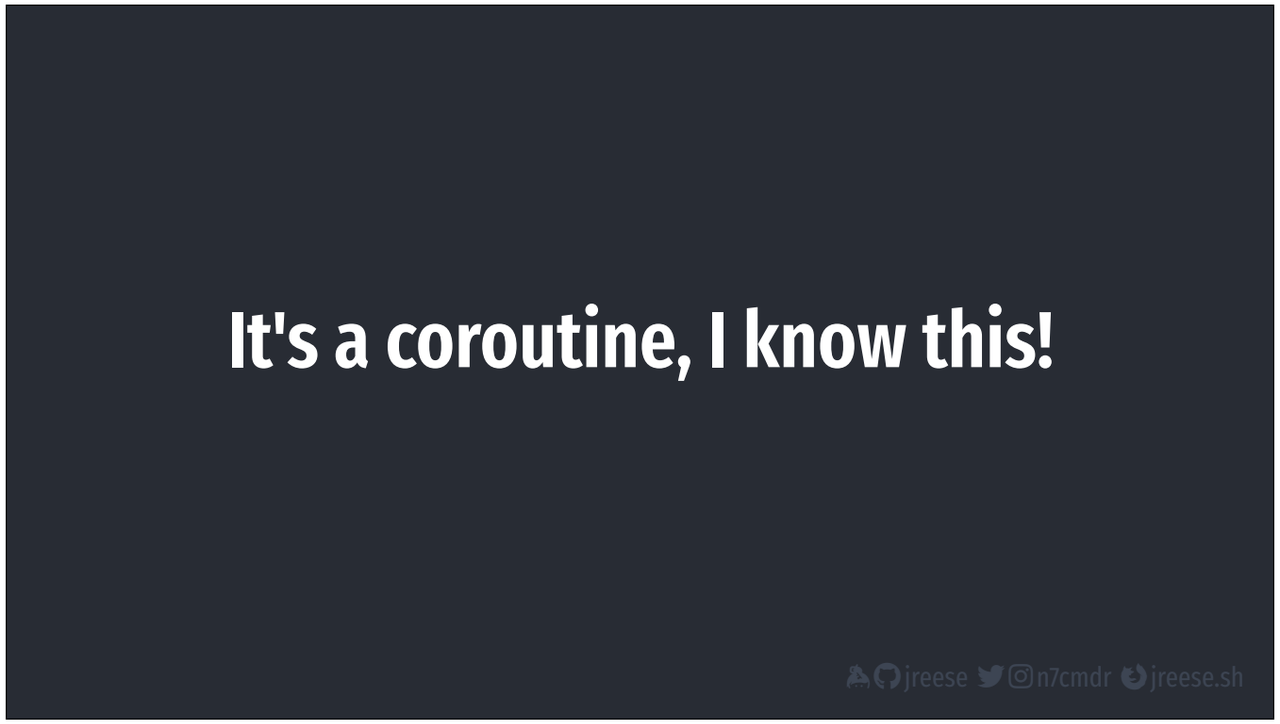
PEP 380 – Syntax for Delegating to a Subgenerator

PEP 3156 – Asynchronous IO Support Rebooted

PEP 492 – Coroutines with async and await syntax

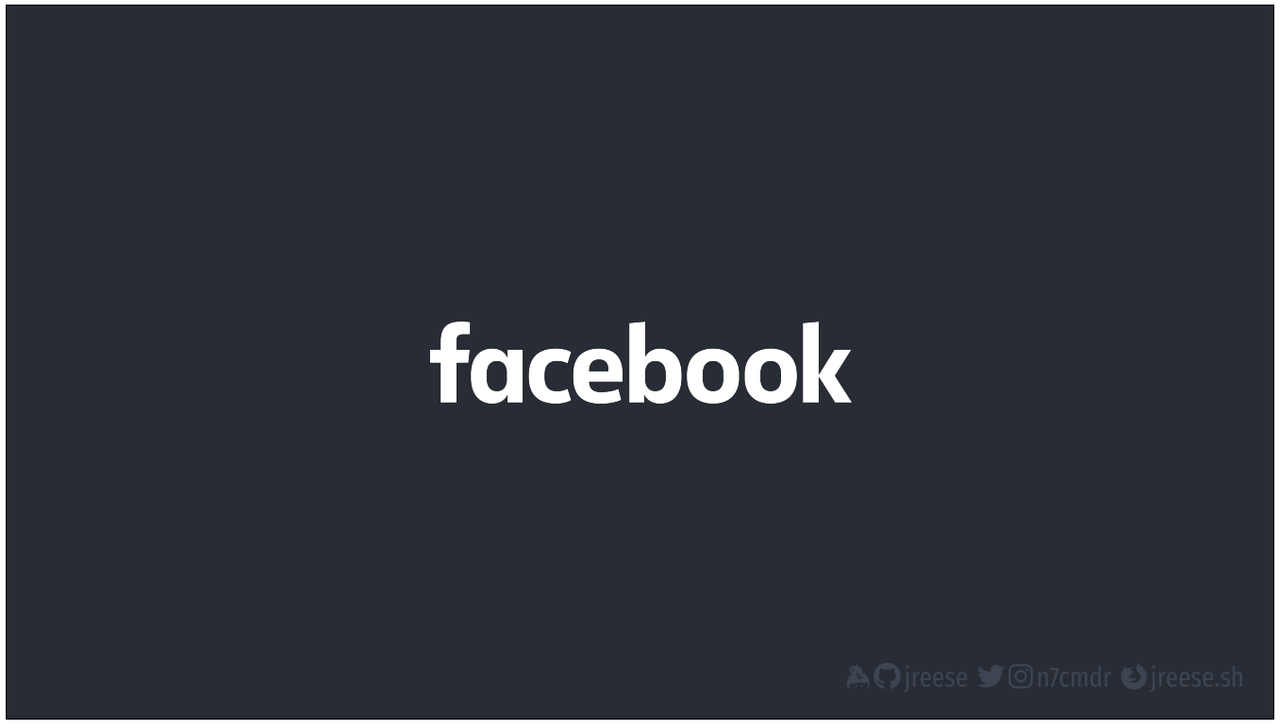
PEP 525 – Asynchronous Generators

If you enjoyed any part of this talk, you will like these PEPs. They are surprisingly readable.
And even if you're already familiar with coroutines or AsyncIO, I bet you'll learn something
that will help you, or influence the decisions you make, in the future.
Maybe next time you see someone working with AsyncIO, you can finally say ...



It's a coroutine, I know this!

If you would like to play with any of the example code, or get a copy of these slides,
they're all available on my Github repo.
Feel free to catch me outside after this talk, or during the afternoon break,
and I'll be happy to answer any questions or dig even deeper on these topics.



Cheers