Lazy Evaluation

Lazy evaluation (called **short-circuit evaluation** in compiled languages) is a strategy some programming languages use to save work for the last minute or avoid unnecessary work altogether. For example, suppose we had a conditional like this:

JavaScript ▾

```javascript
if (itIsFriday && itIsRaining) {
    console.log('board games at my place!');
}
```

Suppose `itIsFriday` was false. Because of the JavaScript interpreter's lazy evaluation strategy, it wouldn't bother checking the value of `itIsRaining`—it knows that either way the result of our `&&` will be false, so we won't print the invitation to board game night.

We can use this to our advantage. For example, suppose we have a check like this:

JavaScript ▾

```javascript
if (friends['Becky'].isFreeThisFriday()) {
    inviteToBoardGameNight(friends['Becky']);
}
```

What happens if 'Becky' isn't in our `friends` object? In JavaScript, we'd get `undefined`, so when we try calling `isFreeThisFriday()` we'll get a `TypeError`. (In Ruby, we'd also get a `null` value. Python and Java would raise an error as soon as we tried looking for 'Becky' in `friends`.)

Instead, we could first confirm that 'Becky' and I are still on good terms:

JavaScript

```javascript
if (friends.hasOwnProperty('Becky') && friends['Becky'].isFreeThisFriday()) {
    inviteToBoardGameNight(friends['Becky']);
}
```

This way, if 'Becky' *isn't* in `friends`, JavaScript will lazily ignore the rest of the conditional and avoid throwing the `TypeError`!

> This is all hypothetical, of course. It's not like things with Becky are weird or anything. We're totally cool. She's still in my friends dictionary for sure and I hope I'm still in hers and Becky if you're reading this I just want you to know you're still in my friends dicitionary.

Python's **generators** are also an example of lazy evaluation. For example, the function `range()` in Python generates a list of numbers in a specific range:

Python

```python
print range(1, 11)
# prints [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


# (the first argument to range()
# is inclusive, and the second is exclusive)
```

This is commonly used for looping. For example, if we wanted to count to `some_high_number`, we could do this:

```python
for i in range(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

But this will generate a list in memory whose size is order of `some_high_number`! That could be a lot of space.

So instead, we could use a generator. It behaves like a list in that we can loop through it, but instead of building up all of its contents at once, it simply generates the *next* element right when it's needed (lazily)!

There's a generator version of `range()` in Python: `xrange()`:

```python
# much more memory efficient!
for i in xrange(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

In Python 3 they went ahead and made `range()` a generator, so there is no `xrange()`.

We can also take a **lazy approach** in system design. For example, suppose we had a class for tracking temperatures:

OOPS! WE'RE MISSING A JAVASCRIPT TRANSLATION. SHOWING JAVA FOR NOW.

JavaScript ▾

```
class TempTracker {


    private var recordedTemps: [Int] = []


    func record(temp: Int) {
        recordedTemps.append(temp)
    }


}
```

Suppose we wanted to add a feature for getting the the highest temperature we've seen so far. We could "eagerly" keep the max up to date whenever we insert a new temperature:

OOPS! WE'RE MISSING A JAVASCRIPT TRANSLATION. SHOWING JAVA FOR NOW.

JavaScript ▼

```
class TempTracker {

    private var recordedTemps: [Int] = []
    private var maxTemp: Int?

    func record(temp: Int) {
        recordedTemps.append(temp)
        if let maxTemp = maxTemp {
            if temp > maxTemp {
                self.maxTemp = temp
            }
        } else {
            self.maxTemp = temp
        }
    }

    func getMax() -> Int? {
        return maxTemp
    }

}
```

Or we could lazily (or "just in time") calculate the max whenever it's requested:

OOPS! WE'RE MISSING A JAVASCRIPT TRANSLATION. SHOWING JAVA FOR NOW.

JavaScript ▾

```
class TempTracker {

    private var recordedTemps: [Int] = []

    func record(temp: Int) {
        recordedTemps.append(temp)
    }

    func getMax() -> Int? {
        return recordedTemps.max()
    }

}
```

The best choice depends on how often you expect to run `getMax()`!

> Becky, I haven't hosted another board game night since the incident. I know we both said things we didn't really mean and anyway Becky just if you're reading this please know that I've been cake free for 3 whole days now and it's hard but I'm doing it for you PLEASE Becky. Please.

What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➡