Ten Essays on Fizz Buzz

Meditations on Python, mathematics, science, engineering, and design

Joel Grus

Ten Essays on Fizz Buzz

Meditations on Python, mathematics, science, engineering, and design

Joel Grus

This book is for sale at http://leanpub.com/fizzbuzz

This version was published on 2020-07-06

ISBN 978-0-9824818-2-0

Contents

Introduction

Fizz Buzz is the following (simple) problem:

> Print the numbers from 1 to 100, except that if the number is divisible by 3, instead print "fizz"; if the number is divisible by 5, instead print "buzz"; and if the number is divisible by 15, instead print "fizzbuzz".

It originated as a children's game, but has since taken on a new life as a lowest-common-denominator litmus test for assessing computer programmers.

If you are an experienced programmer, it is an extremely easy problem to solve. Because of this, it has taken on a third life as the prototypical *bad* interview problem. Everyone knows that it's the question you ask people to make sure that they're not completely incompetent as programmers. Accordingly, if your interviewer asks you to solve it, he's suggesting he thinks it possible that you're completely incompetent as a programmer. You would not be wrong to feel insulted!

My association with this problem began in 2016, when I wrote a blog post called Fizz Buzz in Tensorflow, the (possibly fictional) story of one such insulted programmer who decided to show up his interviewer by approaching Fizz Buzz as a deep learning problem. This post went modestly viral, and ever since then I have been seen as a thought leader in the Fizz Buzz space.

Accordingly, over the years I have come up with and/or collected various other stupid and/or clever ways of solving Fizz Buzz. I have not blogged about them, as I am not the sort of person who beats a joke to death, but occasionally I will tweet about them, and recently in response someone suggested that I write a book on "100 Ways of Writing Fizz Buzz in Python."

Now, I could probably come up with 100 ways of solving Fizz Buzz, but most of them would not be very interesting. Luckily for you, I was able to come up with 10 that *are* interesting in various ways, each of which turned out to be a good launching-off point for (sometimes meandering) discussions of various aspects of coding, Python, Fizz Buzz, mathematics, software design, technical interviewing, and various other topics. Hence “Ten Essays on Fizz Buzz”. [1]

In many ways this is a strange book. Its goal is not to teach you a specific field or a specific technology. I hope you will learn a lot from reading it, but it’s not really a book that you’d read in order to learn anything in particular. Most technical books are about specific technical topics; this one sort of isn’t.

Nonetheless, it is a technical book. Each essay contains code that implements a different solution of Fizz Buzz. Each essay uses code to illustrate its ideas. Each essay represents my current best thinking about how to solve problems using code. If you have a coding job, you should feel no reluctance to expense this book to your employer.

As I primarily code in Python, all of these solutions will be in Python. Some of them will use features that were only introduced in Python 3.6, and some of them will demonstrate concepts and techniques that are mostly specific to Python. On some level this is deeply a *Python* book. However, my goal was to write a book that would be interesting and enlightening even if you are not a Python programmer. I’ll leave it to you to judge how successful I was.

If I have explored more ways of solving Fizz Buzz than others, it is because I stood on the shoulders of giants. The “if / elif / elif / else” solution is the canonical one; the “100 print statements” solution is obvious; the “random guessing” solution I learned about from a Stack Overflow question; the remaining solutions I’m pretty sure I came up with myself, although for many of them it is a near certainty that other people have independently come up with similar (if not basically identical) solutions. That’s how programming works.

The solutions from this book are all available at github.com/joelgrus/fizzbuzz, in case you don’t feel like typing them into the computer yourself.

[1] After I started writing this book I discovered there is a blog post “Twenty Ways to Fizz Buzz”, very few of which overlap with these.

1. 100 Print Statements

Fizz Buzz originated as a game for children. The idea was that the children would sit in a circle and go around in sequence calling out the next number; substituting “fizz” or “buzz” or “fizzbuzz” as appropriate; punishing mistakes according to house rules with varying levels of cruelty.

Well, if children can come up with the correct outputs without using a computer, then so can we. This suggests what is possibly the least imaginative solution – figure out the correct outputs by hand and explicitly print each one:

```python
print('1')
print('2')
print('fizz')
print('4')
print('buzz')
print('fizz')
print('7')
print('8')
print('fizz')
print('buzz')
print('11')
print('fizz')
print('13')
print('14')
print('fizzbuzz')
print('16')
print('17')
print('fizz')
print('19')
print('buzz')
print('fizz')
print('22')
print('23')
print('fizz')
print('buzz')
print('26')
print('fizz')
print('28')
print('29')
print('fizzbuzz')
print('31')
print('32')
print('fizz')
print('34')
print('buzz')
print('fizz')
print('37')
print('38')
print('fizz')
print('buzz')
print('41')
print('fizz')
print('43')
print('44')
print('fizzbuzz')
print('46')
print('47')
print('fizz')
print('49')
print('buzz')
print('fizz')
print('52')
print('53')
print('fizz')
print('buzz')
print('56')
print('fizz')
print('58')
print('59')
print('fizzbuzz')
print('61')
print('62')
print('fizz')
print('64')
print('buzz')
print('fizz')
print('67')
print('68')
print('fizz')
print('buzz')
print('71')
print('fizz')
print('73')
print('74')
print('fizzbuzz')
print('76')
print('77')
print('fizz')
print('79')
print('buzz')
print('fizz')
print('82')
print('83')
print('fizz')
print('buzz')
print('86')
print('fizz')
print('88')
print('89')
print('fizzbuzz')
print('91')
print('92')
print('fizz')
print('94')
print('buzz')
print('fizz')
print('97')
print('98')
print('fizz')
print('buzz')
```

Technically this is a solution. It solves the stated problem of printing the numbers 1 to 100 except in specific circumstances substituting “fizz” or “buzz” or “fizzbuzz”.

Yet this is a solution that’s not particularly satisfying. For example, we would not expect an interviewer to be impressed by it.[1]

In this first chapter we’ll discuss why that’s the case.

Algorithm and Abstraction

One thing the interviewer is (presumably) looking for is your ability to think algorithmically; that is, to come up with an efficient process for solving the problem by turning it into code.

But this isn’t really a process for solving the problem. It’s only a process for printing out a precomputed solution. It relies on already having a solution to the problem.

Nor is it particularly efficient. There is a sense in which this is about as inefficient as possible, since each of the 100 outputs is generated by its own code – nothing is shared. (There are many other senses in which there are far more inefficient solutions, and we will see some such solutions throughout the book.)

Here is a slight improvement:

[1]Although I would be pretty impressed if you used this solution in an interview.

```
FIZZ_BUZZ = [
    '1', '2', 'fizz', '4', 'buzz', 'fizz', '7', '8', 'fizz',
    'buzz', '11', 'fizz', '13', '14', 'fizzbuzz', '16', '17',
    'fizz', '19', 'buzz', 'fizz', '22', '23', 'fizz', 'buzz',
    '26', 'fizz', '28', '29', 'fizzbuzz', '31', '32', 'fizz',
    '34', 'buzz', 'fizz', '37', '38', 'fizz', 'buzz', '41',
    'fizz', '43', '44', 'fizzbuzz', '46', '47', 'fizz', '49',
    'buzz', 'fizz', '52', '53', 'fizz', 'buzz', '56', 'fizz',
    '58', '59', 'fizzbuzz', '61', '62', 'fizz', '64', 'buzz',
    'fizz', '67', '68', 'fizz', 'buzz', '71', 'fizz', '73',
    '74', 'fizzbuzz', '76', '77', 'fizz', '79', 'buzz',
    'fizz', '82', '83', 'fizz', 'buzz', '86', 'fizz', '88',
    '89', 'fizzbuzz', '91', '92', 'fizz', '94', 'buzz',
    'fizz', '97', '98', 'fizz', 'buzz'
]

for fizz_buzz in FIZZ_BUZZ:
    print(fizz_buzz)
```

The values are still manually specified, but at least we're using a `for` loop and a single call to `print` instead of 100 calls to `print`. And while we still didn't use algorithmic thinking to produce the output values, we do have the values in a list to use and re-use however we see fit.

However, this is still not a good solution, for a couple of reasons. For one thing, all of the Fizz Buzz logic here was performed by a human (me). We'd like the computer to handle that logic. What's the point of writing software if we don't let the computer do the parts it's good at?

The second reason this is not a good solution is that it's not at all *extensible*. If we suddenly needed the corresponding outputs for the numbers 101 to 200 we couldn't re-use any part of this (other than the `print` statement), and we'd have to do that work starting from nothing.

Nonetheless, this is an important improvement, for a couple of reasons.

Reusability and Changeability

One nice thing about solving problems with software is that (compared with many other kinds of solutions) it's relatively easy to change your solution when requirements change. For example, imagine that your interviewer decides she'd rather have the words printed in ALL CAPS.

With the "100 Print Statements" solution, you would have to go through and make that change line-by-line. Whereas with this second solution you could just make a small change:

```
for fizz_buzz in FIZZ_BUZZ:
    print(fizz_buzz.upper())
```

Or if you wanted the results to start at 100 and count down to 1:

```
for fizz_buzz in reversed(FIZZ_BUZZ):
    print(fizz_buzz)
```

Or if you needed to print the words without vowels ("Fzz Bzz"):

```
import re

for fizz_buzz in FIZZ_BUZZ:
    print(re.sub("[aeiouAEIOU]", "", fizz_buzz))
```

Or if you wanted the output in Spanish:

```
for fizz_buzz in FIZZ_BUZZ:
    print(fizz_buzz
          .replace("fizz", "efervescencia")
          .replace("buzz", "zumbido"))
```

Or if you wanted to write the results out to a file:

```
with open('fizzbuzz.txt', 'w') as f:
    for fizz_buzz in FIZZ_BUZZ:
        f.write(f"{fizz_buzz}\n")
```

However, there's a limit to the sorts of modifications that this solution admits. Imagine that the interviewer decided that "buzz" should now replace multiples of 7, and "fizzbuzz" multiples of 21. There's no obvious way to modify our solution to accomplish this; probably we'd have to create an entirely new list of values:

```
FIZZ_BUZZ_7 = [1, 2, 'fizz', 4, 5, 'fizz', 'buzz', 8, ...]
```

And, as mentioned previously, if we wanted the outputs for the numbers 101 to 200, there's no way for us to reuse this work (other than the `print` statement).

Throughout the book we'll see various other solutions that make some of these changes much simpler. (Some of our solutions won't make these changes any simpler but will be interesting for other reasons.)

Testability and Fizz Buzz

Another important reason why the "100 print statements" solution is not great is that it's very hard to test. Is it correct? Did I make a mistake and accidentally write "fizz" when I meant to write "buzz"? The only way to know is to go through line by line and check each answer.

As we explore various ways of solving this problem, we'll want to check that our solutions are correct. After you finish this book and go on to solving other problems, you'll also want to check that your solutions are correct.

When developing software we prefer to use *automated tests*; that is, we write *test cases* that we expect to “pass” if our code is correct and that we hope will “fail” if our code is not correct.

One simple way of doing this in Python is using `assert` statements, which will raise an exception (`AssertionError`) if the condition they’re asserting is false:

```
# these assertions all pass
assert True
assert 2 > 1
assert "iz" in "FizzBuzz"

# these assertions all fail
assert False
assert 1 > 2
assert "I" in "TEAM"
```

Typically it’s difficult to create test cases that are 100% comprehensive, but here we only have 100 input-output pairs to check, so we can write a test that covers every possible input / output.

We’ll use the `FIZZ_BUZZ` list from this chapter as the source of truth. This means you should convince yourself that it’s entirely correct.

The solutions in this book will take two general forms. Some solutions will generate a list of the 100 Fizz Buzz outputs. We’ll write a function to check that such a list is correct. It first checks that the list actually has 100 elements. After that it checks that each element of the provided list is the same as the corresponding element of `FIZZ_BUZZ`.

We do that by generating a list containing all the incorrect outputs and then `assert`-ing that the list is empty. The reason we do it this way is so that when the test fails, it fails with an explicit list of all the outputs that were wrong:

```python
# We need this to type-annotate lists.
from typing import List

def check_output(output: List[str]) -> None:
    assert len(output) == 100, "output should have length 100"

    # Collect all the errors in a list
    # The i+1 reflects that output[0] is the output for 1,
    # output[1] is the output for 2, and so on
    errors = [
        f"({i+1}) predicted: {output[i]}, actual: {FIZZ_BUZZ[i]}"
        for i in range(100)
        if output[i] != FIZZ_BUZZ[i]
    ]

    # And assert that the list of errors is empty
    assert not errors, f"{errors}"
```

Other solutions will result in *functions* that take in a number `n` and return the correct Fizz Buzz output for that `n`. We'll test such a function by generating the list of 100 outputs and then using our previous test. (This is another example of us re-using previous work.)

```python
# We need this to type-annotate functions.
from typing import Callable

def check_function(fizz_buzz: Callable[[int], str]) -> None:
    """
    The type annotation says that `fizz_buzz` needs to be
    a function that takes a single argument (which is an `int`)
    and returns a `str`.
    """
    output = [fizz_buzz(i) for i in range(1, 101)]
    check_output(output)
```

As we explore more interesting solutions, we'll use these functions to check that they're correct, so make sure you understand them.

Testability Beyond Fizz Buzz

Throughout this book we'll apply this idea of testability to more than our various Fizz Buzz solutions. We'll also apply it to our intermediate steps and even to our digressions.

For example, imagine that one of our solutions involves checking that two words are anagrams. (None of our solutions involves this, but bear with me.)

We dutifully craft a solution of this subproblem:

```python
def anagrams(s1: str, s2: str) -> bool:
    return sorted(s1) == sorted(s2)
```

How do we know it works? By writing test cases for it:

```python
assert anagrams("dale", "lead")
assert anagrams("time", "mite")
assert not anagrams("made", "deem")
assert not anagrams("time", "miter")
```

This is not a particularly comprehensive set of tests, but it involves a couple of positive examples and a couple of negative examples. At the very least, these tests would not all pass if we'd made a stupid mistake. If they all pass, our mistake would have to be somewhat subtle.

Generating the `print` Statements

Imagine that you're writing a technical book, and that one of the chapters involves solving Fizz Buzz with 100 `print` statements, and that you don't feel like writing them out by hand.

Why not use Python to generate the `print` statements?

```python
def make_print_statement(fizz_buzz: str) -> str:
    return f"print('{fizz_buzz}')"

assert make_print_statement("10") == "print('10')"
assert make_print_statement("fizz") == "print('fizz')"
assert make_print_statement("buzz") == "print('buzz')"
assert make_print_statement("fizzbuzz") == "print('fizzbuzz')"
```

After which it's easy to print them out and then copy and paste them into your manuscript:

```python
for fizz_buzz in FIZZ_BUZZ:
    print(make_print_statement(fizz_buzz))
```

2. Euclid’s Solution

Euclid was an ancient Greek mathematician sometimes called the father of geometry. He predated Fizz Buzz by thousands of years, but this is how I like to think he would have solved it:

```python
def fizz_buzz(n: int) -> str:
    hi, lo = max(n, 15), min(n, 15)

    while hi % lo > 0:
        hi, lo = lo, hi % lo

    return {1: str(n), 3: "fizz", 5: "buzz", 15: "fizzbuzz"}[lo]
```

In this chapter we’ll explore why this works and why Euclid might have solved it this way.

Prime Numbers

A positive number is *prime* if it cannot be written as the product of two smaller numbers.

3 and 5 are both prime numbers, since there’s no way to write them as such a product. 15 is not a prime number, as we can write it as 3 * 5. By convention, 1 is also not a prime number, since we can only write 1 * 1.

One way to check whether a number is prime by trying to divide it by every number smaller than itself, starting at 2:

```python
def is_prime(n: int) -> bool:
    """
    n is prime if it's at least 2 and if it's not
    divisible by any smaller number (other than 1)
    """
    return (
        n >= 2 and
        all(n % d > 0 for d in range(2, n))
    )

assert all(is_prime(n) for n in [2, 3, 7, 11, 83, 89, 97])
assert not any(is_prime(n) for n in [4, 50, 91])
```

In order to check whether some number `n` is prime, this function has to take (in the worst case) approximately `n` actions: first a check that `n` is at least two, and then a divisibility check for each of 2, 3, … `n - 1`. (In the best case it will take a lot fewer actions; for example, `is_prime(1000)` will stop as soon as it computes `1000 % 2 == 0`.)

This means that if we wanted a list of the *all* the primes up to `n`, a worst case estimate is that it would take approximately `n * n` actions.

We can do a little better, though. We really only need to check for divisors up to `math.sqrt(n)`, since if `a` and `b` are both larger than `math.sqrt(n)` then their product is larger than `n`, and in particular is not `n`:

```python
import math

def int_sqrt(n: int) -> int:
    return int(math.sqrt(n))

def is_prime2(n: int) -> bool:
    return (
        n >= 2 and
        all(n % d > 0 for d in range(2, int_sqrt(n) + 1))
    )
```

Let's see how slow this is:

```python
from typing import List

def primes_up_to(n: int) -> List[int]:
    return [i for i in range(2, n + 1) if is_prime2(i)]
```

As always, we write a couple of test cases:

```python
assert primes_up_to(20) == [2, 3, 5, 7, 11, 13, 17, 19]
assert primes_up_to(100)[-3:] == [83, 89, 97]
```

Then, if you're using IPython you can use the `%timeit` magic to see how this scales:

```
%timeit primes = primes_up_to(100)
```

On my laptop this results in

```
100           .077 ms
1000         1.051 ms (14x the previous)
10000       13.4   ms (13x)
100000     209     ms (16x)
1000000   4860     ms (23x)
```

That is, it takes about 5 seconds to find all the primes up to 1 million. Given the way it's scaling I don't particularly want to check 10 million and beyond.

A more efficient way is to use a trick called the "sieve", which is based on the observation that if a number is not prime than it's necessarily divisible by a smaller *prime* number.

Start with the numbers 2, …, n as "candidate" primes. The smallest element (that is, 2) must be prime. Remove it (and remember it) and then eliminate all further multiples of 2 as not prime. The new smallest element 3 is now our next prime. Remove it and then eliminate all the multiples of 3 as not prime. Keep repeating this process. At each step the new smallest element was not divisible by any smaller prime, hence must be prime itself.

```
def primes_up_to(n: int) -> List[int]:
    candidates = range(2, n + 1)
    primes = []

    while candidates:
        # The smallest remaining number must be prime,
        # because it wasn't divisible by any smaller prime.
        p = candidates[0]
        primes.append(p)

        # Remove further multiples of p as not-prime
        candidates = [n for n in candidates if n % p > 0]

    return primes
```

And we’ll repeat the test cases:

```
assert primes_up_to(20) == [2, 3, 5, 7, 11, 13, 17, 19]
assert primes_up_to(100)[-3:] == [83, 89, 97]
```

But it turns out that this version doesn’t scale well either! Using `%timeit` again I get the following:

```
100          0.0337 ms
1000         0.764  ms  (22x the previous)
10000       45.9    ms  (60x)
100000    2590      ms  (56x)
```

That is, it takes over 2.5 seconds to find all the primes up to 100,000. This is even worse than the “slow” version! The sieve was supposed to be fast. What did we do wrong?

Performance Optimization

We can use a tool called `cProfile` to investigate where our code is spending its time:

```
import cProfile
cProfile.run('primes_up_to(100_000)')
```

The output (severely edited to fit in the book) looks sort of like

```
ncalls   tottime   function
1        0.078     <ipython-input-18-7da95fe8b6cd>:1(primes_up_to)
9592     2.138     <ipython-input-18-7da95fe8b6cd>:12(<listcomp>)
1        0.000     <string>:1(<module>)
1        0.000     {built-in method builtins.exec}
9592     0.000     {method 'append' of 'list' objects}
1        0.000     {method 'disable' of '_lsprof.Profiler' objects}
```

Almost all of the time is spent in `<listcomp>`; that is, doing 9600 list comprehensions.

This makes sense. Recreating the list of candidates each time by inspecting every element is overkill. When we are doing the sieve for p = 5, there’s no need to check 7, since it’s not a multiple of 5. But in order to recreate the list of candidates we have to iterate over every element.

How can we eliminate this? An alternative approach makes the following changes:

1. We maintain a list `candidates` of booleans of length n + 1. `candidates[i]` is `True` if we haven’t yet ruled out `i` as a prime number. We will systematically go through the list and set every composite (not prime) number to `False`.
2. We check every candidate from 2 to `math.sqrt(n)`. This is sufficient because if some `m <= n` is composite and can be written as `a * b` then at least one of `a` and `b` must be smaller than `sqrt(n)`, and we’ll set `m` to `False` when considering that number (or one of its prime factors).
3. If a candidate `p` has already been eliminated as a candidate prime, we skip it
4. Otherwise it’s a prime number. Then we eliminate `p ** 2`, `p ** 2 + p`, `p ** 2 + 2 * p` and so on. (Smaller multiples of `p` will have already been eliminated by the time we get to `p`.)

```
def primes_up_to(n: int) -> List[int]:
    #             0      1          2, ..., n
    candidates = [False, False] + [True] * (n - 1)

    for p in range(2, int(math.sqrt(n))):
        # if we haven't already eliminated p as a prime
        if candidates[p]:
            # eliminate all multiples of p, starting at p ** 2
            for m in range(p * p, n + 1, p):
                candidates[m] = False

    # return the indices that weren't eliminated
    return [n for n, prime in enumerate(candidates) if prime]
```

Once again we can check the timings:

```
100          0.0067 ms
1000         0.0884 ms  (13x the previous)
10000        0.911  ms  (10x)
100000       9.16   ms  (10x)
1000000    139      ms  (15x)
10000000  2240      ms  (16x)
```

This version is faster for small input sizes, but more importantly it grows much more slowly as the input size increases. You can see that it finds all prime numbers up to 10,000,000 faster than the previous sieve implementation found all the prime numbers up to 100,000.

Mathematically both versions are the same algorithm, but the implementation differences turn out to be pretty crucial to performance.

Factorization

It turns out that every positive integer can be written in a unique way as a product of prime numbers. (Here "unique" means "order doesn't matter", as plainly 3 * 5 == 5 * 3 and so on.) We call this product the "prime factorization".

Using the sieve it's easy enough to find the prime factorization of a number:

```python
from typing import List

def factorize(n: int) -> List[int]:
    primes = primes_up_to(n)

    factors = []

    for p in primes:
        # p might divide n more than once
        while n % p == 0:
            factors.append(p)
            n = n // p
        # once we reach 1 there are no more prime factors
        if n == 1:
            break

    return factors
```

And as always we write a few test cases:

```python
assert factorize(15) == [3, 5]
assert factorize(150) == [2, 3, 5, 5]
assert factorize(7) == [7]
```

What Prime Factorization Has to Do with Fizz Buzz

If some number `n` is divisible by 3, then 3 is necessarily a prime factor of `n`, and vice versa. Same thing goes for 5.

So here's another "simple" solution:

```
def fizz_buzz_factorization(n: int) -> str:
    prime_factors = factorize(n)

    if 3 in prime_factors and 5 in prime_factors:
        return "fizzbuzz"
    elif 3 in prime_factors:
        return "fizz"
    elif 5 in prime_factors:
        return "buzz"
    else:
        return str(n)
```

Why the scare quotes? Because this "simple" solution required us to first implement a prime number sieve and factorization algorithm. The solution at the beginning of the chapter certainly didn't do all that!

Greatest Common Divisors and Least Common Multiples

Frequently in mathematics we would like to know the greatest common divisor of two numbers; that is, the largest number that divides both of them. For example, we do this when we reduce fractions.

Imagine we're given the fraction 24 / 44. We calculate (don't worry about how right now) that the greatest common divisor of 24 and 44 is 4, and then we cancel out a factor of 4 from the numerator and denominator:

```
24 / 44 = (6 * 4) / (11 * 4) = 6 / 11
```

One way to find the greatest common divisor is to factorize the two numbers and take the product of all the common factors. In this case, 24 factorizes as `[2, 2, 2, 3]` and 44 as `[2, 2, 11]`, so the common factors are `[2, 2]`, and their product is 4. More generally:

```
def gcd_factorize(n: int, m: int) -> int:
    n_factors = factorize(n)
    m_factors = factorize(m)
    gcd = 1

    # Stop when either list is empty
    while n_factors and m_factors:
        # Greatest remaining factors of both are equal
        # so multiply the gcd by that factor
        if n_factors[-1] == m_factors[-1]:
            gcd *= n_factors.pop()
            m_factors.pop()
        # Largest factor of m is not a factor of n
        elif n_factors[-1] < m_factors[-1]:
            m_factors.pop()
        # Largest factor of n is not a factor of m
        else:
            n_factors.pop()

    return gcd
```

Let’s check a few cases:

```
assert gcd(24, 44) == 4
assert gcd(44, 24) == 4
assert gcd(3, 5) == 1
assert gcd(100, 10) == 10
```

It would probably feel more natural to go through the factors in the order they’re given (that is, front to back). We start from the end because `pop()`-ing the last element off a list is a very cheap operation; removing the first element requires (in essence) making a copy of the list.

With the sizes of lists we’re dealing with it doesn’t really matter, but it’s good practice to get in the habit of using efficient idioms and avoiding inefficient ones. (This will be a common theme throughout this book.)

Sometimes we also care about the *least common multiple* of two numbers; that is, the smallest number that is itself divisible by both. For example, we use this to find a common denominator when adding fractions:

```
3 / 10 + 3 / 4 = 6 / 20 + 15 / 20
               = 21 / 20
```

Here we used the fact that the least common multiple of 10 and 4 is 20.

We were able to compute the gcd of two numbers by taking the intersection of their prime factors. We can similarly find the lcm by taking the union of their prime factors:

```python
def lcm_factorize(n: int, m: int) -> int:
    n_factors = factorize(n)
    m_factors = factorize(m)
    lcm = 1

    # Stop when both lists are empty
    while n_factors or m_factors:
        # no more prime factors of n
        if not n_factors:
            lcm *= m_factors.pop()
        # no more prime factors of m
        elif not m_factors:
            lcm *= n_factors.pop()
        # same largest prime factor, only use it once
        elif n_factors[-1] == m_factors[-1]:
            lcm *= n_factors.pop()
            m_factors.pop()
        # largest factor of m is not a factor of n
        elif n_factors[-1] < m_factors[-1]:
            lcm *= m_factors.pop()
        # largest factor of n is not a factor of m
        else:
            lcm *= n_factors.pop()

    return lcm
```

And as usual we write some tests:

```
assert lcm_factorize(10, 4) == 20
assert lcm_factorize(4, 10) == 20
assert lcm_factorize(3, 5) == 15
assert lcm_factorize(10, 100) == 100
```

One final observation: gcd(n, m) is the product of only the common factors. lcm(n, m) is the product of all the factors of both, but with common factors only counted once (even though they appear twice). And n * m is the product of all the factors of both as many times as they appear.

This means that `lcm(n, m) * gcd(n, m) == n * m`. So if you know how to compute one, you know how to compute the other. (Indeed, currently Python has a `math.gcd` function but no `math.lcm` function, although this is tentatively scheduled to change in Python 3.9.)

gcd and Fizz Buzz

What does any of this have to do with Fizz Buzz?

Well, `gcd(n, m)` gives us the product of the common factors of `n` and `m`. And the factors of 15 are `[3, 5]`.

That means that `gcd(n, 15)` is either 1 (if `n` has no factors of 3 or 5), 3 (if `n` has a factor of 3 but not a factor of 5), 5 (if `n` has a factor of 5 but not a factor of 3), or 15 (if `n` has a factor of 3 and a factor of 5).

Of course, saying "n has a factor of 3" is the same as saying "n is divisible by 3". Now this is starting to look like Fizz Buzz.

Using Python's `math.gcd` function, you could just do:

```
import math

def fizz_buzz(n: int) -> str:
    choices = {1: str(n), 3: 'fizz', 5: 'buzz', 15: 'fizzbuzz'}
    return choices[math.gcd(n, 15)]
```

So that’s the idea behind our solution. But that wasn’t our solution.

Euclid’s Algorithm

How would we go about computing the greatest common divisor of two numbers without computing their prime factorizations first? (Obviously we can use `math.gcd`, but what if we didn’t have that?)

The brute force way is to take all numbers between 1 and the smaller number, keep only the ones that divide both of the numbers, and take the largest:

```
def gcd_slow(n: int, m: int) -> int:
    return max(i
               for i in range(1, min(n, m) + 1)
               if n % i == m % i == 0)
```

If `n` and `m` are both large, that’s a lot of numbers to check. For example, to compute

```
gcd_slow(97039801, 97313179)
```

on my laptop takes about 8 seconds. (Those are both prime numbers, so their gcd is 1.) You could also check from highest to lowest and stop when you find one, but that would still take a long time when the gcd is small.

A more efficient way to compute the gcd is using *Euclid’s algorithm*. You remember Euclid from the start of the chapter.

Say we have two numbers `m <= n` with greatest common divisor `d`. Then we can do "long division" and write `n = c * m + r` where `0 <= r < m`. That is, `c == n // m` is the result of integer division, and `r == n % m` is the remainder.

I claim that `d == gcd(m, n) == gcd(r, m)`. That is, the greatest common divisor of `m` and `n` is the same as the greatest common denominator of `m` and `n % m`. Why is this?

Well, if `x` is any number that divides both `m` and `n`, then `c * m` is also divisible by `x` (since `m` is), which means that `r = n - c * m` must be divisible by `x`.

On the other hand, if `y` is any number that divides both `m` and `r`, then `c * m` is also divisible by `y` (since `m` is), which means that `n = c * m + r` must be divisible by `y`.

That is, {common divisors of m and n} and {common divisors of m and r} are the exact same sets of numbers. Any number in the set on the left is also in the set on the right, and any number in the set on the right is also in the set on the left.

In particular, they have the same greatest element. The greatest element of the set on the left is (by definition) gcd(m, n), and the greatest element of the set on the right is (by definition) gcd(m, r), so these gcds must be the same.

So now imagine we are trying to compute gcd(n, m) with `n >= m`. We just showed that this is is the same as gcd(m, n % m). If `n % m == 0`, then this equals `m`, and we're done. In particular, if `m == n`, this is the case. Otherwise, necessarily `m < n` and we repeat.

At each step, either the larger of the pair gets smaller, or we're done.

This leads to a much faster implementation:

```python
def gcd(n: int, m: int) -> int:
    # Want n >= m
    n, m = max(m, n), min(m, n)

    # gcd(n, m) = gcd(m, n % m)
    while n % m > 0:
        n, m = m, n % m

    # When n % m == 0, n is a multiple of m, so m is the gcd
    return m
```

For the same two large prime numbers this takes about 1/1000 of a second.

“Euclid’s solution” was just an explicit invocation of this algorithm with m = 15. That is, it was an extremely opaque (but fast) computation of `gcd(n, 15)`, which allows us to choose the correct Fizz Buzz result.

3. Infinite Iterables

One day I was thinking about `itertools`, as I am prone to do, and this elegant solution struck me out of the blue:

```python
import itertools

fizzes = itertools.cycle(['', '', 'fizz'])
buzzes = itertools.cycle(['', '', '', '', 'buzz'])
numbers = itertools.count(1)

fizz_buzzes = ((fizz + buzz) or str(n)
                for fizz, buzz, n in zip(fizzes, buzzes, numbers))

output = [next(fizz_buzzes) for _ in range(100)]
```

In this chapter we'll explore how this solution works.

Iterables, Iterators, and Lazy Infinite Sequences

Typically, one of the first things you learn in Python is lists:

```python
xs = ['a', 'b', 'c', 'd', 'e']
```

There are a couple of common ways to access the elements of a list. The first is to retrieve an element at a specific position:

```
# lists are indexed starting at 0
assert xs[2] == 'c'
```

The second is to *iterate* over the list, for example using a `for` loop:

```
for x in xs:
    assert 'a' <= x <= 'e'
```

But lists are not the only things you can iterate over. For example, we've used `enumerate` to get the elements of a list along with their indices. But it's certainly not a list:

```
es = enumerate(xs)

try:
    es[0]
except TypeError:
    print("'enumerate' object is not subscriptable")
```

In fact, there is a much broader class of objects ("iterables") that can be iterated over. It turns out that anything can be iterated over if it knows how to generate an "iterator".

The distinction in Python between iterators and iterables is subtle and confusing (in particular, because every iterator is an iterable but not vice versa) and so we'll go through it in somewhat excruciating detail.

Iterators

Something is an *iterator* if you can call `next()` on it. Calling `next()` will either return some "next" element, or it will raise a `StopIteration` exception. You can think of an iterator as a stream of elements that can be traversed only once, in order, and that may end at some point.

Why only once in order? Because the only way to access the elements of an iterator is by calling `next()` to get the next element. There is no way to "go back" nor to access any element that's not the "next" one.

One iterator that you may have dealt with in Python is a generator expression:

```
# generator containing 1, 2, 3
it = (x for x in [1, 2, 3])

assert next(it) == 1  # next element is 1
assert next(it) == 2  # next element is 2
assert next(it) == 3  # next element is 3
try:
    next(it)
    assert False
except StopIteration:
    print("no more elements")
```

Another is a function that `yield`s values:

```
from typing import Iterator

def upto(n: int) -> Iterator[int]:
    for i in range(n):
        yield i

it = upto(3)

assert next(it) == 0
assert next(it) == 1
assert next(it) == 2

try:
    next(it)
    assert False
```

```
except StopIteration:
    pass
```

In practice you rarely call `next()` yourself (although we will sometimes be doing so in this chapter), and so in practice you rarely work with iterators as iterators.

Iterables

Something is an *iterable* if it knows how to give you an iterator. In particular, every iterator is an iterable, since it can just give you itself.

You call the `iter()` function on an iterable to get its iterator. When you call it on an iterator it just returns itself:

```
# it is an iterator and therefore also an iterable
it = (x for x in [1, 2, 3])

# iterator is its own iterator
assert iter(it) == it
```

Of course you can also get iterators for other iterables:

```
# xs is not an iterator, calling next(xs) would give an error
xs = [1, 2, 3]

# but it is an iterable
it = iter(xs)

assert next(it) == 1
assert next(it) == 2
assert next(it) == 3
```

Notice that while the iterator is consumed, the list is not. At any time you can get a fresh iterator, or even multiple iterators at once:

```python
xs = [1, 2, 3]
it1 = iter(xs)
it2 = iter(xs)
assert next(it1) == next(it2) == 1
assert next(it1) == next(it2) == 2
```

Typically we'll work using tools like `for` loops. Behind the scenes you could imagine them doing something like:

```python
from typing import Iterable, Callable

def for_each(xs: Iterable[int], do: Callable) -> None:
    it = iter(xs)

    try:
        while True:
            do(next(it))
    except StopIteration:
        pass

for_each([1, 2, 3], print)  # works with lists
for_each(upto(3), print)    # works with iterators
```

Working with the iterable abstraction allows us not to care whether we have lists or generators or whatever.

Infinite Iterables

So far all of the iterables we've used eventually stop. But it's easy to make an iterator that goes on forever:

```
def every_number() -> Iterator[int]:
    n = 0
    while True:
        yield n
        n += 1
```

You shouldn't use a `for` loop with this iterator, since `for x in every_number()` would never stop (unless your loop had a `break` in it, or something similar).

Instead you have to use other techniques. For example, you could define a function to take the first `n` results from an iterable and return them in a list:

```
from typing import List

def head(xs: Iterable[int], n: int) -> List[int]:
    results = []
    it = iter(xs)

    for _ in range(n):
        try:
            results.append(next(it))
        except StopIteration:
            break

    return results
```

This works on infinite iterables:

```
assert head(every_number(), 5) == [0, 1, 2, 3, 4]
```

and finite ones:

```
assert head([2, 3, 4, 5], 2) == [2, 3]
```

Working with iterables in this way is somewhat clumsy. Luckily, there's a better way.

`itertools`

Python comes with the `itertools` library, a collection of functions for working with iterators and iterables. Generally speaking, these functions will take iterables (or nothing) as inputs and return iterators as outputs.

For example, itertools contains an `islice` function which is roughly the equivalent of the `head` function we wrote in the previous section (except that it returns an iterator not a list):

```
import itertools

def head2(xs: Iterable[int], n: int) -> List[int]:
    return list(itertools.islice(xs, n))

assert head(every_number(), 5) == head2(every_number(), 5)
assert head([2, 3, 4], 2) == head2([2, 3, 4], 2)
```

The functions in `itertools` do many interesting things, and you are encouraged to read the documentation and/or check out my 2015 presentation on Stupid Itertools Tricks for Data Science.

Here we only need two of its functions.

The first is `itertools.count` which returns an infinite iterator that "counts" based on the supplied `start` (default: 0) and `step` (default: 1). That is, `itertools.count()` is the iterator that generates the values 0, 1, 2, … and keeps going.

```
assert head(itertools.count(), 5) == [0, 1, 2, 3, 4]
assert head(itertools.count(start=10, step=2), 3) == [10, 12, 14]
```

The other one we'll need is `itertools.cycle`, which takes an iterable and returns its elements repeatedly in a cycle forever:

```
guesses = itertools.cycle(['yes', 'no', 'maybe'])
assert head(guesses, 5) == ['yes', 'no', 'maybe', 'yes', 'no']
```

`zip` VS `map`

`zip` is a commonly used function in Python. It lazily "zips" together multiple iterables into an iterator of tuples, stopping whenever the first is exhausted. (There is an `itertools.zip_longest` in case you want it to keep going as long as one of the iterables is not exhausted.)

So, for example,

```
pairs = zip([1, 2], ['a', 'b', 'c'])
assert list(pairs) == [(1, 'a'), (2, 'b')]

triplets = zip([1, 2], ['a', 'b', 'c'], [True, False])
assert list(triplets) == [(1, 'a', True), (2, 'b', False)]
```

One way to combine multiple iterables is using `zip` and a list comprehension. For example,

```
names = ['Alice', 'Bob', 'Carol']
titles = ['Professor', 'Doctor', 'President']

greetings = [f"{title} {name}"
             for title, name in zip(names, titles)]

assert greetings = [
    "Professor Alice",
    "Doctor Bob",
    "President Carol"
]
```

Another way is to use `map`, which takes a mapping function and some iterables to use as its arguments one value at a time:

```
greetings2 = map(lambda title, name: f"{title} {name}",
                 names,
                 titles)

assert list(greetings2) == greetings
```

I generally avoid `map`, for a couple of reasons. One is that it returns a lazy iterable, so that you have to remember to wrap it in `list()` if you actually want a list. (`zip` has the same issue, but you typically use it inside a `for` loop, so it doesn't end up mattering.)

The second reason is that I find Python code that uses `map` to be "unpythonic"; that is, non-idiomatic. This is a judgment call on my part, but I'm the one writing the book.

Putting It All Together

How do we use all these concepts to solve the problem? First, we define

```python
# An infinite iterator whose every third element is 'fizz'
# and the rest of whose elements are the empty string
fizzes = itertools.cycle(['', '', 'fizz'])
```

and

```python
# An infinite iterator whose every fifth element is 'buzz'
# and the rest of whose elements are the empty string
buzzes = itertools.cycle(['', '', '', '', 'buzz'])
```

and finally

```python
# The infinite iterator 1, 2, 3, 4, ...
numbers = itertools.count(1)
```

Now consider the iterator `zip(fizzes, buzzes, numbers)`. Its first several elements are

```
('', '', 1)
('', '', 2)
('fizz', '', 3)
('', '', 4)
('', 'buzz', 5)
('fizz', '', 6)
```

and its 15th element is `('fizz', 'buzz', 15)`.

That is, whenever the number is divisible by 3, the first element will be ‘fizz’, and whenever the number is divisible by 5 the second element will be ‘buzz’, which means that whenever the number is divisible by 15 the first two elements will be ‘fizz’ and ‘buzz’.

If the first two elements are not both empty, the correct answer is their concatenation (since `'fizz' + '' == 'fizz'`). If they are both empty, the right answer is `str()` of the number. We can get this with

```
fizz_buzzes = ((fizz + buzz) or str(n)
               for fizz, buzz, n in zip(fizzes, buzzes, numbers))
```

This works because `or` returns its left-hand value if that value is truthy, and its right-hand value otherwise. Here, if either `fizz` or `buzz` is non-empty, `fizz + buzz` is a non-empty string (which is truthy) and so that's what gets returned. If both `fizz` and `buzz` are empty strings, `fizz + buzz` is an empty string (which is not truthy), and so `str(n)` gets returned.

Notice that since the inputs are infinite iterables we have to use a generator comprehension. If we used a list comprehension it would try to materialize this infinite sequence as a list and would run forever.

This actually solves more than the original problem, since it generates an infinite stream of Fizz Buzz outputs. We only need the first 100:

```
output = [next(fizz_buzzes) for _ in range(100)]
```

Bonus: PowerFizz

Here's a variant I call "PowerFizz":

> Print the numbers 1 to N, except that if the number is a perfect square, print "fizz"; if the number is a perfect cube, print "buzz"; and if the number is a sixth power, print "fizzbuzz".

The obvious solution doesn't work because of the usual floating point reasons:

```
def is_sixth_power(n: int) -> bool:
    sixth_root = n ** (1 / 6)
    return sixth_root == int(sixth_root)

assert is_sixth_power(1 ** 6)
assert is_sixth_power(2 ** 6)
assert is_sixth_power(3 ** 6)
assert not is_sixth_power(4 ** 6)  # !!
```

Of course, you could use the try-every-number method:

```
def is_kth_power(n: int, k: int) -> bool:
    for i in itertools.count(1):
        if i ** k == n:
            return True
        # Once we pass n, it's not a k-th power.
        elif i ** k > n:
            return False

assert is_kth_power(10 ** 6, 6)
assert is_kth_power(3 ** 5, 5)
assert not is_kth_power(3 ** 4, 5)
```

But we can also easily solve this variant using iterators. We create a `squares` iterator consisting of all squares and a `cubes` iterator consisting of all cubes. Any time `n` equals the first element of `squares` we know `n` must be a square, and then we advance that iterator and are ready to find the next square. Any time `n` equals the first element of `cubes` we know `n` must be a cube, and then we advance that iterator and are ready to find the next cube.

There's no easy way to "peek" at the first element of an iterator, so we just use `next()` to get the first element and store it in a "buffer" variable:

```python
def power_fizz() -> Iterator[str]:
    squares = (n ** 2 for n in itertools.count(1))
    cubes = (n ** 3 for n in itertools.count(1))

    # Buffer for next element of each iterator
    next_square, next_cube = next(squares), next(cubes)

    for n in itertools.count(1):
        fizz = buzz = ''
        if n == next_square:
            fizz = 'fizz'
            next_square = next(squares)
        if n == next_cube:
            buzz = 'buzz'
            next_cube = next(cubes)
        yield (fizz + buzz) or str(n)
```

When `n` is a square, `fizz` is `"fizz"`; otherwise it's an empty string. When `n` is a cube, `buzz` is `"buzz"`; otherwise it's an empty string. And when `n` is a sixth power (that is, both a square and a cube) then `fizz` is `"fizz"` and `buzz` is `"buzz"`, as desired:

```python
# Put a None at the beginning so output[1] is the output for 1
output = [None] + head(power_fizz(), 1000)

assert output[7 ** 2] == "fizz"
assert output[6 ** 3] == "buzz"
assert output[2 ** 6] == "fizzbuzz"
assert output[2 ** 7] == str(2 ** 7)
```

About the Author

Joel Grus is the author of the beloved book Data Science from Scratch, the beloved blog post Fizz Buzz in Tensorflow, and the polarizing JuypterCon presentation I Don't Like Notebooks.

He's also the co-host of the Adversarial Learning podcast.

He very occasionally blogs at joelgrus.com and spends most of the rest of his time on Twitter @joelgrus. If you'd like to stay up to date on his various thoughts and projects, sign up for his mailing list at joelgrus.substack.com.

And if you have feedback on the book, he would love to hear from you:

joelgrus@gmail.com

Acknowledgements

Thanks for reading! I hope you enjoyed it. Please recommend it to your friends and let me know if you have any feedback, positive or negative.

Although I have thought about Fizz Buzz quite a lot over the past four years, I never considered writing a book about it until @RoccoMeli suggested it on Twitter. Blame him for putting the idea in my head.

Thanks to Jeremy Kun and (especially) Tim Hopper for early reading and feedback. The book is much better thanks to their help.

Thanks as always to Ganga and Madeline for putting up with me writing another book.

The cover photo is from https://www.pexels.com/photo/bee-on-a-plant-2047420/ . Thank you to the photographer for allowing its free use.