7 Languages
in
7 Hours

$> whois crebma

I’m Amber Conville! You can find me on the internet at these places:

twitter: @crebma
email: crebma@gmail.com
github: crebma

I help others develop software at Test Double (@testdouble), which is an awesome group of consultants if you’re looking for work or your company might need help!

I also run Self.conference (@selfconference) in Detroit, MI - scheduled for June 7-8th, 2019!

What Are We Doing Today?

Today, we'll be looking at the basic syntax of 7 different languages and then working through the same kata in each of them. The kata we'll be working through is the Prime Factors kata. The languages we'll be exploring are:

- ruby
- clojure
- haskell
- rust
- elm
- elixir
- go

The Kata

Given an integer, return a list of factors for that integer that are prime numbers. For example:

Given	Result
1	[]
2	[2]
3	[3]
4	[2, 2]

And so on.

The Rules

1. TDD
   - Write the simplest test you can to make progress
   - Write only enough code to make your test pass
   - Only refactor on green!
2. Pairing is encouraged, but you don't have to. Here's an example of ping-pong style pairing if you want to try it out:
   - You write a very simple failing test (non-compiling counts as failing)
   - You pass the keyboard to your pair to make it pass writing only enough code as is necessary
   - Your pair writes another simple failing test and passes the keyboard back
3. You may not finish the exercise in the allotted time - that's fine! You still have the rest of your life outside of this session to work on it. :)
4. Don't be a jerk.

Our Kata in Java

```
@Test
public void factorsOf1() {
    List<Integer> expected = new ArrayList<Integer>();

    assertEquals(expected, factors(1));
}
```

Our Kata in Java

```
public static List<Integer> factors(int number) {
    return new ArrayList<Integer>();
}
```

Our Kata in Java

```
@Test
public void factorsOf2() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(2);

    assertEquals(expected, factors(2));
}
```

Our Kata in Java

```
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    if (number > 1) {
        primes.add(2);
    }
    return primes;
}
```

Our Kata in Java

```
@Test
public void factorsOf3() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(3);

    assertEquals(expected, factors(3));
}
```

Our Kata in Java

```java
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    if (number > 1) {
        primes.add(number);
    }
    return primes;
}
```

Our Kata in Java

```
@Test
public void factorsOf4() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(2);
    expected.add(2);

    assertEquals(expected, factors(4));
}
```

Our Kata in Java

```
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    if (number > 1) {
        if (number % 2 == 0 && number > 2) {
            primes.add(2);
            primes.add(2);
        } else {
            primes.add(number);
        }
    }
    return primes;
}
```

Our Kata in Java

```
@Test
public void factorsOf5() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(5);

    assertEquals(expected, factors(5));
}
```

Our Kata in Java

```
@Test
public void factorsOf6() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(2);
    expected.add(3);

    assertEquals(expected, factors(6));
}
```

Our Kata in Java

```
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    if (number > 1) {
        if (number % 2 == 0 && number > 2) {
            primes.add(2);
            primes.add(number / 2);
        } else {
            primes.add(number);
        }
    }
    return primes;
}
```

Our Kata in Java

```
@Test
public void factorsOf7() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(7);

    assertEquals(expected, factors(7));
}
```

Our Kata in Java

```java
@Test
public void factorsOf8() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(2);
    expected.add(2);
    expected.add(2);

    assertEquals(expected, factors(8));
}
```

Our Kata in Java

```java
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    if (number > 1) {
        while (number % 2 == 0 && number > 2) {
            primes.add(2);
            number = number / 2;
        }

        if (number > 1) {
            primes.add(number);
        }
    }
    return primes;
}
```

Our Kata in Java

```
@Test
public void factorsOf9() {
    List<Integer> expected = new ArrayList<Integer>();
    expected.add(3);
    expected.add(3);

    assertEquals(expected, factors(9));
}
```

Our Kata in Java

```
public static List<Integer> factors(int number) {
    List<Integer> primes = new ArrayList<Integer>();
    for (int candidate = 2; candidate <= number; candidate++) {
        for (; number % candidate == 0; number /= candidate) {
            primes.add(candidate);
        }
    }
    return primes;
}
```

Ruby

Ruby

Ruby is a dynamic language that is relatively easy to read and write. It supports recursion and has many functional concepts, though it is not a functional language.

Ruby - The Basics

Ruby convention is to use underscores in place of spaces. This is true for file names, variable names, method names, everything except class names, which use camel case.

The _spec bit is convention for saying that this is a test file, as opposed to production code.

All ruby files must end in *.rb.

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Basic Anatomy

```ruby
def prime_factors(num)
  []
end

RSpec.describe "Prime Factors" do

  it "has factors of [] for 1" do
    expect(prime_factors(1)).to eq []
  end

end
```

Ruby - Helpful Things to Google

map

flatten

collect

ranges

default arguments

recursion

how to run tests

how to work with arrays (.push)

irb, the repl!

```ruby
it "has factors of [2] for 2" do
  expect(prime_factors(2)).to eq [2]
end
```

```ruby
def prime_factors(num)
  return [] if num == 1
  [2]
end
```

```ruby
it "has factors of [3] for 3" do
  expect(prime_factors(3)).to eq [3]
end
```

```ruby
def prime_factors(num)
  return [] if num == 1
  [num]
end
```

```ruby
it "has factors of [2,2] for 4" do
  expect(prime_factors(4)).to eq [2,2]
end
```

```ruby
def prime_factors(num)
  return [] if num == 1
  if num % 2 == 0 && num > 2
    [2, 2]
  else
    [num]
  end
end
```

```ruby
it "has factors of [2,3] for 6" do
  expect(prime_factors(6)).to eq [2,3]
end
```

```ruby
def prime_factors(num)
  return [] if num == 1
  if num % 2 == 0 && num > 2
    [2, num/2]
  else
    [num]
  end
end
```

```ruby
it "has factors of [2,2,2] for 8" do
  expect(prime_factors(8)).to eq [2,2,2]
end
```

```ruby
def prime_factors(num)
  return [] if num == 1
  if num % 2 == 0 && num > 2
    [2] + prime_factors(num / 2)
  else
    [num]
  end
end
```

```ruby
it "has factors of [3,3] for 9" do
  expect(prime_factors(9)).to eq [3,3]
end
```

```ruby
def prime_factors(num, candidate = 2)
  return [] if num == 1
  if num % candidate == 0
    [candidate] + prime_factors(num / candidate, candidate)
  else
    prime_factors(num, candidate + 1)
  end
end
```

Thoughts?

Clojure

Clojure

Clojure is another dynamic programming language, although perhaps a little harder to read and write at first blush. It runs on top of the Java Virtual Machine (JVM), and is a Lisp.

Clojure - The Basics

Clojure convention is to use dashes in place of spaces. This is true for file names, variable names, method names, everything.

The _test bit is convention for saying that this is a test file, as opposed to production code.

All clojure files must end in *.clj.

Clojure - Basic Anatomy

```clojure
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```clojure
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```clojure
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```clojure
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```clojure
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```clojure
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Basic Anatomy

```clojure
(ns prime-factors.core)

(defn primes
  ([num] []))
```

```clojure
(ns prime-factors.core-test
  (:require [clojure.test :refer :all]
            [prime-factors.core :refer :all]))

(deftest prime-factors
  (is (= (primes 1) [])))
```

Clojure - Helpful Things to Google

lein test

zero?

rem

cons

cond/else

inc

lein repl

```clojure
(deftest prime-factors
  (is (= (primes 1) []))
  (is (= (primes 2) [2])))
```

```clojure
(defn primes
  ([num]
    (cond
      (< num 2) []
      :else [2])))
```

```clojure
(deftest prime-factors
  (is (= (primes 1) []))
  (is (= (primes 2) [2]))
  (is (= (primes 3) [3])))
```

```clojure
(defn primes
  ([num]
    (cond
      (< num 2) []
      :else [num])))
```

```
(deftest prime-factors
  (is (= (primes 1) []))
  (is (= (primes 2) [2]))
  (is (= (primes 3) [3]))
  (is (= (primes 4) [2,2])))
```

```
(defn primes
  ([num]
    (cond
      (< num 2) []
      (and (zero? (rem num 2)) (> num 2)) [2, 2]
      :else [num])))
```

```clojure
(deftest prime-factors
  (is (= (primes 1) []))
  (is (= (primes 2) [2]))
  (is (= (primes 3) [3]))
  (is (= (primes 4) [2,2]))
  (is (= (primes 5) [5]))
  (is (= (primes 6) [2,3])))
```

```clojure
(defn primes
  ([num]
    (cond
      (< num 2) []
      (and (zero? (rem num 2)) (> num 2)) [2, (/ num 2)]
      :else [num])))
```

```clojure
(is (= (primes 3) [3]))
(is (= (primes 4) [2,2]))
(is (= (primes 5) [5]))
(is (= (primes 6) [2,3]))
(is (= (primes 7) [7]))
(is (= (primes 8) [2,2,2])))
```

```clojure
(defn primes
  ([num]
    (cond
      (< num 2) []
      (and (zero? (rem num 2)) (> num 2))
        (cons 2 (primes (/ num 2)))
      :else [num])))
```

```
(is (= (primes 3) [3]))
(is (= (primes 4) [2,2]))
(is (= (primes 5) [5]))
(is (= (primes 6) [2,3]))
(is (= (primes 7) [7]))
(is (= (primes 8) [2,2,2]))
(is (= (primes 9) [3,3])))
```

```
(defn primes
  ([num] (primes num 2))
  ([num candidate]
    (cond
      (< num candidate) []
      (zero? (rem num candidate))
        (cons candidate (primes (/ num candidate) candidate))
      :else (primes num (inc candidate)))))
```

Thoughts?

Haskell

Haskell

Haskell is a statically typed, purely functional programming language. This one gets a little trickier for those starting out with it, but like the rest of them, is very fun to play with.

Haskell - The Basics

Haskell convention is to use camel case, likeThis. Its modules and types are all camel cased, but start with a capital letter, unlike functions and variables.

The _Test bit is convention for saying that this is a test file, as opposed to production code.

All haskell files must end in *.hs.

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Basic Anatomy

```haskell
module PrimeFactors (primeFactors) where

primeFactors :: Integer -> [Integer]
primeFactors x = []
```

```haskell
module PrimeFactors_Test where

import PrimeFactors (primeFactors)
import Test.HUnit

test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)

main = runTestTT $ TestList [test1]
```

Haskell - Helpful Things to Google

stack test

`where` for helper functions

`where` for variables

infix vs prefix

guards

lists

how to add two lists together

stack repl

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)

main = runTestTT $ TestList [test1, test2]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors 1 = []
primeFactors x = [2]
```

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)
test3 = TestCase $ assertEqual "factors of 3 are [3]" [3] (primeFactors 3)

main = runTestTT $ TestList [test1, test2, test3]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors 1 = []
primeFactors x = [x]
```

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)
test3 = TestCase $ assertEqual "factors of 3 are [3]" [3] (primeFactors 3)
test4 = TestCase $ assertEqual "factors of 4 are [2,2]" [2,2] (primeFactors 4)

main = runTestTT $ TestList [test1, test2, test3, test4]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors 1 = []
primeFactors x
  | (x `mod` 2 == 0) && (x > 2) = [2, 2]
  | otherwise = [x]
```

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)
test3 = TestCase $ assertEqual "factors of 3 are [3]" [3] (primeFactors 3)
test4 = TestCase $ assertEqual "factors of 4 are [2,2]" [2,2] (primeFactors 4)
test5 = TestCase $ assertEqual "factors of 5 are [5]" [5] (primeFactors 5)
test6 = TestCase $ assertEqual "factors of 6 are [2, 3]" [2,3] (primeFactors 6)

main = runTestTT $ TestList [test1, test2, test3, test4, test5, test6]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors 1 = []
primeFactors x
  | (x `mod` 2 == 0) && (x > 2) = [2, x `div` 2]
  | otherwise = [x]
```

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)
test3 = TestCase $ assertEqual "factors of 3 are [3]" [3] (primeFactors 3)
test4 = TestCase $ assertEqual "factors of 4 are [2,2]" [2,2] (primeFactors 4)
test5 = TestCase $ assertEqual "factors of 5 are [5]" [5] (primeFactors 5)
test6 = TestCase $ assertEqual "factors of 6 are [2, 3]" [2,3] (primeFactors 6)
test7 = TestCase $ assertEqual "factors of 7 are [7]" [7] (primeFactors 7)
test8 = TestCase $ assertEqual "factors of 8 are [2,2,2]" [2,2,2] (primeFactors 8)

main = runTestTT $ TestList [test1, test2, test3, test4, test5, test6, test7, test8]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors 1 = []
primeFactors x
  | (x `mod` 2 == 0) && (x > 2) = [2] ++ primeFactors (x `div` 2)
  | otherwise = [x]
```

```haskell
test1 = TestCase $ assertEqual "factors of 1 are []" [] (primeFactors 1)
test2 = TestCase $ assertEqual "factors of 2 are [2]" [2] (primeFactors 2)
test3 = TestCase $ assertEqual "factors of 3 are [3]" [3] (primeFactors 3)
test4 = TestCase $ assertEqual "factors of 4 are [2,2]" [2,2] (primeFactors 4)
test5 = TestCase $ assertEqual "factors of 5 are [5]" [5] (primeFactors 5)
test6 = TestCase $ assertEqual "factors of 6 are [2, 3]" [2,3] (primeFactors 6)
test7 = TestCase $ assertEqual "factors of 7 are [7]" [7] (primeFactors 7)
test8 = TestCase $ assertEqual "factors of 8 are [2,2,2]" [2,2,2] (primeFactors 8)
test9 = TestCase $ assertEqual "factors of 9 are [3,3]" [3,3] (primeFactors 9)

main = runTestTT $ TestList [test1, test2, test3, test4, test5, test6, test7, test8, test9]
```

```haskell
primeFactors :: Integer -> [Integer]
primeFactors x = primeFactors' x 2
  where primeFactors' x c
                | x < c = []
                | x `mod` c == 0 = [c] ++ primeFactors' (x `div` c) c
                | otherwise = primeFactors' x (c + 1)
```

Thoughts?

Rust

Rust

Rust is a statically typed language, but has type inference, which can make it feel kind of dynamic when you’re using it.

Rust - The Basics

Rust convention is to use underscores in place of spaces. This is true for file names, variable names, method names, everything.

The _Test bit is convention for saying that this is a test file, as opposed to production code.

All rust files must end in *.rs.

Rust - Basic Anatomy

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

#[test]
fn prime_factors_of_one() {
    assert_eq!(prime_factors(1), []);
}
```

Rust - Basic Anatomy

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

#[test]
fn prime_factors_of_one() {
    assert_eq!(prime_factors(1), []);
}
```

Rust - Basic Anatomy

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

#[test]
fn prime_factors_of_one() {
    assert_eq!(prime_factors(1), []);
}
```

Rust - Basic Anatomy

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

#[test]
fn prime_factors_of_one() {
    assert_eq!(prime_factors(1), []);
}
```

Rust - Basic Anatomy

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

#[test]
fn prime_factors_of_one() {
    assert_eq!(prime_factors(1), []);
}
```

Rust - Helpful Things to Google

cargo test

let vs let mut (mutability)

borrowing & ownership

no tail call recursion

how to append to vectors

&mut for the previous

has while

for

```rust
#[test]
fn prime_factors_of_two() {
    assert_eq!(prime_factors(2), [2]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    if num != 1 {
        vec![2]
    } else {
        vec![]
    }
}
```

```rust
#[test]
fn prime_factors_of_three() {
    assert_eq!(prime_factors(3), [3]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    if num != 1 {
        vec![num]
    } else {
        vec![]
    }
}
```

```rust
#[test]
fn prime_factors_of_four() {
    assert_eq!(prime_factors(4), [2,2]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    if num != 1 {
        if num % 2 == 0 && num > 2 {
            vec![2, 2]
        } else {
            vec![num]
        }
    } else {
        vec![]
    }
}
```

```rust
#[test]
fn prime_factors_of_six() {
    assert_eq!(prime_factors(6), [2,3]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    if num != 1 {
        if num % 2 == 0 && num > 2 {
            vec![2,num / 2]
        } else {
            vec![num]
        }
    } else {
        vec![]
    }
}
```

```rust
#[test]
fn prime_factors_of_eight() {
    assert_eq!(prime_factors(8), [2,2,2]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    let mut number = num;
    let mut primes = vec![];
    if number != 1 {
        while number % 2 == 0 && number > 2 {
            primes.append(&mut vec![2]);
            number = number / 2
        }
        if number != 1 {
            primes.append(&mut vec![number]);
        }
    }
    primes
}
```

```rust
#[test]
fn prime_factors_of_nine() {
    assert_eq!(prime_factors(9), [3,3]);
}
```

```rust
fn prime_factors(num: i64) -> Vec<i64> {
    let mut number = num;
    let mut primes = vec![];
    for candidate in 2..number + 1 {
        while number % candidate == 0 {
            primes.append(&mut vec![candidate]);
            number = number / candidate
        }
    }
    primes
}
```

Thoughts?

Elm

Elm

Elm is a functional language that looks a bit like Haskell and compiles to javascript. It is based around the idea of being easy to use and learn, and its error messages really speak to that.

Elm - The Basics

Elm convention is to use camel case for function names and variables. It uses camel case for modules, starting with a capital letter.

The Test bit is convention for saying that this is a test file, as opposed to production code.

All elm files must end in *.elm.

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```elm
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```elm
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```elm
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```elm
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```elm
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```elm
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
    [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Basic Anatomy

```
module PrimeFactors exposing (..)

primeFactors : Int -> (List Int)
primeFactors x = []
```

```
module PrimeFactorsTest exposing (..)

import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)
import PrimeFactors exposing (primeFactors)


suite : Test
suite =
  describe "Prime Factors"
      [ test "has factors of [] for 1" <|
        \_ ->
          Expect.equal [] (primeFactors 1) ]
```

Elm - Helpful Things to Google

elm-test

Modulus in elm 0.19 (remainderBy, modBy)

How to append to lists

How to add two lists together

elm repl

```
, test "has factors of [2] for 2" <|
  \_ ->
    Expect.equal [2] (primeFactors 2)
```

```
primeFactors : Int -> (List Int)
primeFactors x =
  if x < 2 then []
  else [2]
```

```
, test "has factors of [3] for 3" <|
  \_ ->
    Expect.equal [3] (primeFactors 3)
```

```
primeFactors : Int -> (List Int)
primeFactors x =
  if x < 2 then []
  else [x]
```

```
, test "has factors of [2,2] for 4" <|
  \_ ->
    Expect.equal [2,2] (primeFactors 4)
```

```
primeFactors : Int -> (List Int)
primeFactors x =
  if x < 2 then []
  else if x > 2 && remainderBy 2 x == 0 then
    [2, 2]
  else [x]
```

```
, test "has factors of [6] for 6" <|
  \_ ->
    Expect.equal [2,3] (primeFactors 6)
```

```
primeFactors : Int -> (List Int)
primeFactors x =
  if x < 2 then []
  else if x > 2 && remainderBy 2 x == 0 then
    [2, x // 2]
  else [x]
```

```
, test "has factors of [2,2,2] for 8" <|
  \_ ->
    Expect.equal [2,2,2] (primeFactors 8)
```

```
primeFactors : Int -> (List Int)
primeFactors x =
  if x < 2 then []
  else if x > 2 && remainderBy 2 x == 0 then
    [2] ++ primeFactors (x // 2)
  else [x]
```

```
, test "has factors of [3,3] for 9" <|
  \_ ->
    Expect.equal [3,3] (primeFactors 9)
```

```
primeFactors : Int -> (List Int)
primeFactors x = primeFactorsCandidate x 2

primeFactorsCandidate : Int -> Int -> (List Int)
primeFactorsCandidate x c =
  if x < c then []
  else if remainderBy c x == 0 then
    [c] ++ primeFactorsCandidate (x // c) c
  else primeFactorsCandidate x (c + 1)
```

Thoughts?

Elixir

Elixir

Elixir is a dynamic, functional language. It runs on top of the Erlang VM.

Elixir - The Basics

Elixir convention is to use underscores in place of spaces, for method names and variables. It uses camel case starting with a capital letter for modules, and namespaces are just capital letter acronyms.

The _test bit is convention for saying that this is a test file, as opposed to production code.

All elixir files must end in either *.ex or *.exs (for a script).

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Basic Anatomy

```
defmodule PF.PrimeFactors do
  def prime_factors(num) do
    []
  end
end
```

```
defmodule PF.PrimeFactorsTest do
  use ExUnit.Case

  test "factors of 1 are []" do
    assert PF.PrimeFactors.prime_factors(1) == []
  end
end
```

Elixir - Helpful Things to Google

mix test

int vs float math (/ vs div() and rem())

no while or for loops at all - recursion only

can use if

can also use cond

default arguments

iex (elixir’s repl!)

```elixir
test "factors of 2 are [2]" do
  assert PF.PrimeFactors.prime_factors(2) == [2]
end
```

```elixir
def prime_factors(num) do
  cond do
    num == 1 -> []
    true -> [2]
  end
end
```

```elixir
test "factors of 3 are [3]" do
  assert PF.PrimeFactors.prime_factors(3) == [3]
end
```

```elixir
def prime_factors(num) do
  cond do
    num == 1 -> []
    true -> [num]
  end
end
```

```elixir
test "factors of 4 are [2,2]" do
  assert PF.PrimeFactors.prime_factors(4) == [2,2]
end
```

```elixir
def prime_factors(num) do
  cond do
    num == 1 -> []
    rem(num, 2) == 0 and num > 2 -> [2,2]
    true -> [num]
  end
end
```

```elixir
test "factors of 6 are [2,3]" do
  assert PF.PrimeFactors.prime_factors(6) == [2,3]
end
```

```elixir
def prime_factors(num) do
  cond
    num == 1 -> []
    rem(num, 2) == 0 and num > 2 -> [2, div(num, 2)]
    true ->[num]
end
```

```elixir
test "factors of 8 are [2,2,2]" do
  assert PF.PrimeFactors.prime_factors(8) == [2,2,2]
end
```

```elixir
def prime_factors(num) do
  cond do
    num == 1 -> []
    rem(num, 2) == 0 and num > 2 -> [2] ++ prime_factors(div(num, 2))
    true -> [num]
  end
end
```

```elixir
test "factors of 9 are [3,3]" do
  assert PF.PrimeFactors.prime_factors(9) == [3,3]
end
```

```elixir
def prime_factors(num, candidate \\ 2) do
  cond do
    num < candidate -> []
    rem(num, candidate) == 0 ->
      [candidate] ++ prime_factors(div(num, candidate), candidate)
    true -> prime_factors(num, candidate + 1)
  end
end
```

Thoughts?

Go

Go

Go is a statically typed language. It does not optimize for tail call recursion, and is pretty good at doing small distributed tasks fast.

Go - The Basics

Go convention is to use underscores in place of spaces for packages and file names. It uses camel case starting with a capital letter for methods which are public.

The _test bit is convention for saying that this is a test file, as opposed to production code.

All go files must end in *.go.

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Basic Anatomy

```go
package prime_factors

func Factors(num int) []int {
  return []int{}
}
```

```go
package prime_factors_test

import (
  "prime_factors"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)

var _ = Describe("Factors", func() {
  It("has factors of 1 as []", func() {
    Expect(prime_factors.Factors(1)).To(Equal([]int{}))
  })
})
```

Go - Helpful Things to Google

how to run tests

declare and initialize locally (:=)

NO while loops

for loops only

how to add to a slice (append, which makes a new slice of the old slice with the new element added onto the end)

recursion is supported

has collection functions like map, filter, reduce

has anonymous functions

```go
It("has factors of 2 as [2]", func() {
  Expect(prime_factors.Factors(2)).To(Equal([]int{2}))
})
```

```go
func Factors(num int) []int {
  if num > 1 {
    return []int{2}
  } else {
    return []int{}
  }
}
```

```go
It("has factors of 3 as [3]", func() {
  Expect(prime_factors.Factors(3)).To(Equal([]int{3}))
})
```

```go
func Factors(num int) []int {
  if num > 1 {
    return []int{num}
  } else {
    return []int{}
  }
}
```

```go
It("has factors of 4 as [2,2]", func() {
  Expect(prime_factors.Factors(4)).To(Equal([]int{2,2}))
})
```

```go
func Factors(num int) []int {
  if num > 1 {
    if num % 2 == 0 && num > 2 {
      return []int{2,2}
    } else {
      return []int{num}
    }
  } else {
    return []int{}
  }
}
```

```go
It("has factors of 6 as [6]", func() {
  Expect(prime_factors.Factors(6)).To(Equal([]int{2,3}))
})
```

```go
func Factors(num int) []int {
  if num > 1 {
    if num % 2 == 0 && num > 2 {
      return []int{2, (num / 2)}
    } else {
      return []int{num}
    }
  } else {
    return []int{}
  }
}
```

```go
It("has factors of 8 as [2,2,2]", func() {
  Expect(prime_factors.Factors(8)).To(Equal([]int{2,2,2}))
})
```

```go
func Factors(num int) []int {
  primes := []int{}
  if num > 1 {
    for num % 2 == 0 && num > 2 {
      primes = append(primes, 2)
      num /= 2
    }
    if num > 1 {
      primes = append(primes, num)
    }
  }
  return primes
}
```

```go
It("has factors of 9 as [3,3]", func() {
  Expect(prime_factors.Factors(9)).To(Equal([]int{3,3}))
})
```

```go
func Factors(num int) []int {
  primes := []int{}
  for candidate := 2; candidate <= num; candidate++ {
    for ; num % candidate == 0; num /= candidate {
      primes = append(primes, candidate)
    }
  }
  return primes
}
```

Thoughts?

Thanks for coming!

Amber Conville
@crebma
@testdouble
@selfconference