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
website: crebma.com

I 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 May 20-21st, 2016!

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
- scala
- 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

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

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

Our Kata in 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

```
@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

```
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 - Project Structure

Gemfile

The Gemfile is where we declare our external dependencies for our project.

```
source 'https://rubygems.org'

gem 'rspec', '~> 3.4'
```

Ours includes a source (where to find gems), as well as a reference to rspec (a bdd testing library) at version 3.4 or higher.

Ruby - The Basics

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

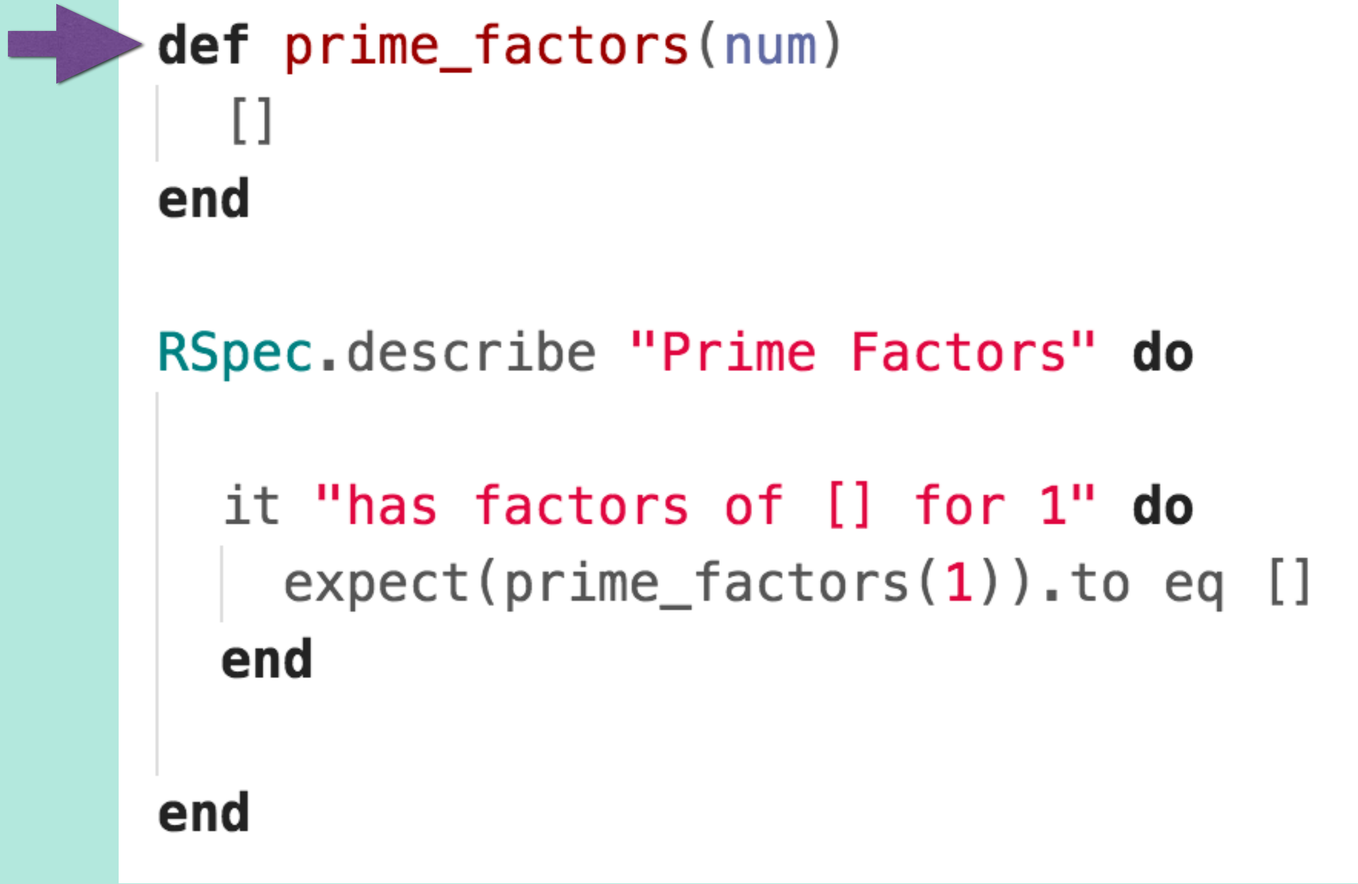
The _spec bit is convention for saying that this is a test file, as opposed to production code.

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 - 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
```

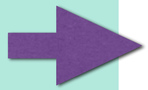
Ruby - Basic Anatomy

```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

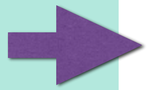


```ruby
def prime_factors(num)
  []
end

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
```

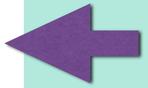
Ruby - Basic Anatomy



```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
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
```

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

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

```
it "has factors of [5] for 5" do
  expect(prime_factors(5)).to eq [5]
end
```

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

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

```
it "has factors of [7] for 7" do
  expect(prime_factors(7)).to eq [7]
end
```

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

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

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

```
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 - Project Structure

project.clj

This is where we define various things about our project, as well as our dependencies, using clojure.

```
(defproject prime_factors "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.6.0"]])
```

Ours has a lot of generated stuff that we don’t really care about right now, as well as a dependency for clojure 1.6.0.

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.

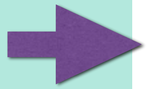
All clojure files must end in *.clj.

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) [])))
```

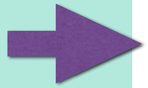
Clojure - Basic Anatomy



```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) [])))
```

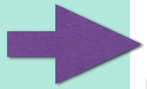
Clojure - Basic Anatomy

```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

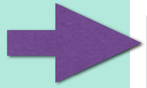
```clojure
(ns prime-factors.core)

(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

```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
(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])))
```

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

```clojure
(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])))
```

```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])))
```

```
(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]))
  (is (= (primes 7) [7])))
```

```
(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])))
```

```
(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.

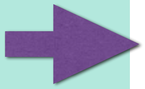
All haskell files must end in *.hs.

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]
```

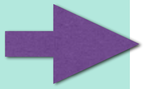
Haskell - Basic Anatomy



```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]
```

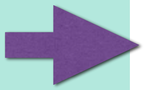
Haskell - Basic Anatomy



```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

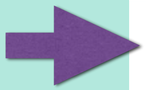


```haskell
module PrimeFactors (primeFactors) where

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

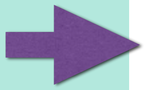
```haskell
module PrimeFactors (primeFactors) where

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]
```



```
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]
```

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

```
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]
```

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

```
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]
```

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

```
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)

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

```
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]
```

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

```
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)

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

```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]
```

```
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]
```

```
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.

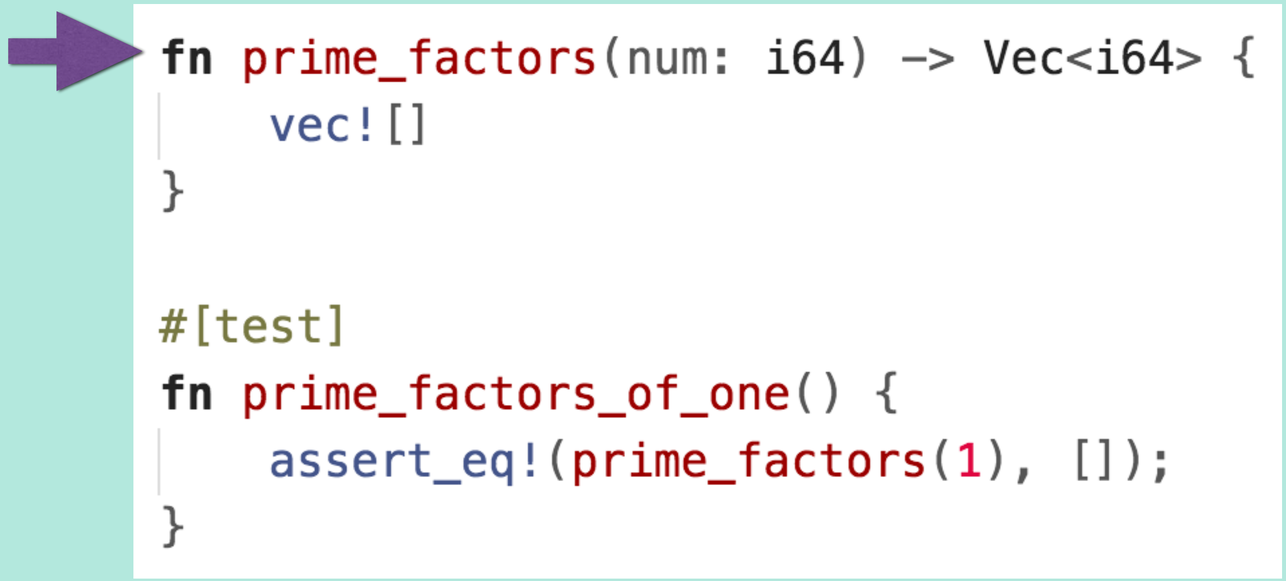
The _Test bit is convention for saying that this is a test file, as opposed to production code.

All rust files must end in *.rs.

Rust - Basic Anatomy

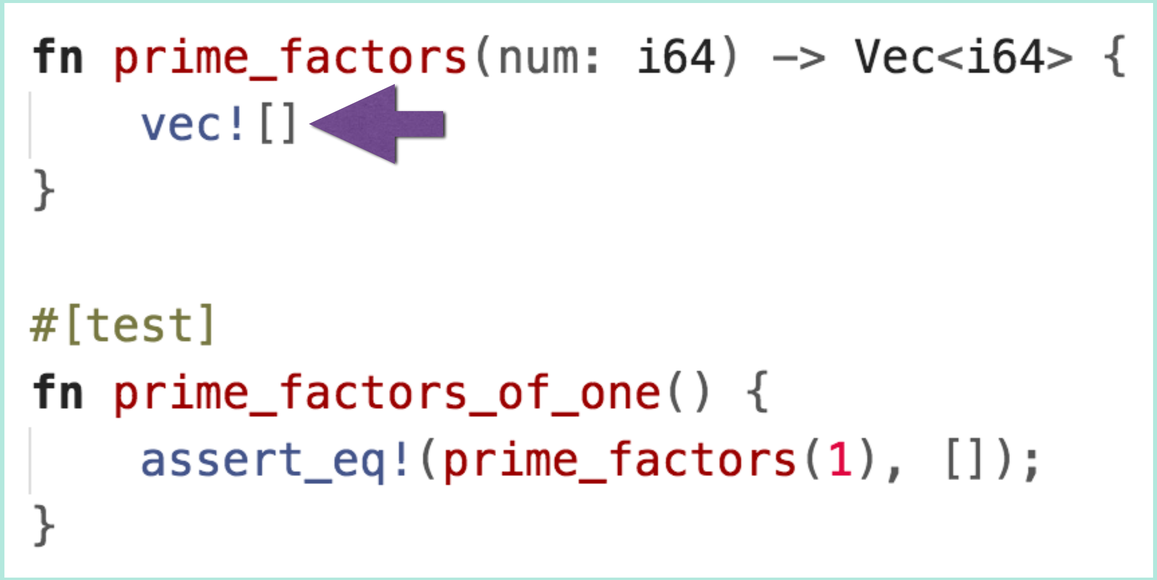
```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

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

Rust - Basic Anatomy

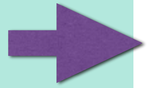


```rust
fn prime_factors(num: i64) -> Vec<i64> {
    vec![]
}

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

Rust - Basic Anatomy



Rust - 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
#[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_five() {
    assert_eq!(prime_factors(5), [5]);
}
```

```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_seven() {
    assert_eq!(prime_factors(7), [7]);
}
```

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

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]);
}

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?

Scala

Scala

Scala is a statically typed language, but has type inference, which can make it feel kind of dynamic when you’re using it. It also runs on top of the JVM, like clojure.

Scala - The Basics

Scala convention is to use camel case for method names and variables. It uses camel case for objects and classes, starting with a capital letter.

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

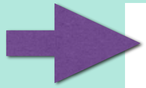
All scala files must end in *.scala.

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

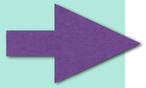
Scala - Basic Anatomy



```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy



```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

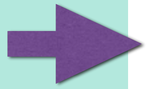
Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```



```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

Scala - Basic Anatomy

```scala
package com.crebma.primeFactors

object PrimeFactors {
  def factors(num: Int) : List[Int] = {
    List[Int]()
  }
}
```

```scala
package com.crebma.primeFactors

import org.scalatest._

class PrimeFactorsSpec extends FlatSpec with Matchers {
  it should "give factors of 1 as []" in {
    PrimeFactors.factors(1) should equal (List())
  }
}
```

```
it should "give factors of 2 as [2]" in {
  PrimeFactors.factors(2) should equal (List(2))
}
```

```
def factors(num: Int) : List[Int] = {
  if (num == 1) {
    List[Int]()
  } else {
    List[Int](2)
  }
}
```

```
it should "give factors of 3 as [3]" in {
  PrimeFactors.factors(3) should equal (List(3))
}
```

```
def factors(num: Int) : List[Int] = {
  if (num == 1) {
    List[Int]()
  } else {
    List[Int](num)
  }
}
```

```
it should "give factors of 4 as [2,2]" in {
  PrimeFactors.factors(4) should equal (List(2,2))
}
```

```
def factors(num: Int) : List[Int] = {
  if (num == 1) {
    List[Int]()
  } else {
    if (num % 2 == 0 && num > 2) {
      List[Int](2,2)
    } else {
      List[Int](num)
    }
  }
}
```

```
it should "give factors of 5 as [5]" in {
  PrimeFactors.factors(5) should equal (List(5))
}
```

```
it should "give factors of 6 as [2,3]" in {
  PrimeFactors.factors(6) should equal (List(2,3))
}
```

```
def factors(num: Int) : List[Int] = {
  if (num == 1) {
    List[Int]()
  } else {
    if (num % 2 == 0 && num > 2) {
      List[Int](2,num/2)
    } else {
      List[Int](num)
    }
  }
}
```

```
it should "give factors of 7 as [7]" in {
  PrimeFactors.factors(7) should equal (List(7))
}
```

```
it should "give factors of 8 as [2,2,2]" in {
  PrimeFactors.factors(8) should equal (List(2,2,2))
}
```

```
def factors(num: Int) : List[Int] = {
  var primes = List[Int]()
  var number = num
  if (number != 1) {
    while (number % 2 == 0 && number > 2) {
      primes = primes :+ 2
      number /= 2
    }

    if (number != 1) {
      primes = primes :+ number
    }
  }
  primes
}
```

```
it should "give factors of 9 as [3,3]" in {
  PrimeFactors.factors(9) should equal (List(3,3))
}
```

```
def factors(num: Int) : List[Int] = {
  var primes = List[Int]()
  var number = num
  for (candidate <- 2 to number) {
    while (number % candidate == 0) {
      primes = primes :+ candidate
      number /= candidate
    }
  }
  primes
}
```

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.

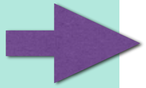
All elixir files must end in either *.ex or *.exs (for a script).

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



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

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

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

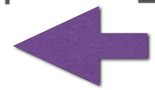
Elixir - Basic Anatomy

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

```elixir
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
```

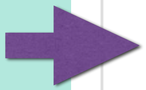
Elixir - Basic Anatomy

```
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
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
```

```
test "factors of 5 are [5]" do
  assert PF.PrimeFactors.prime_factors(5) == [5]
end
```

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

```
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 7 are [7]" do
  assert PF.PrimeFactors.prime_factors(7) == [7]
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
```

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

```
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.

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.

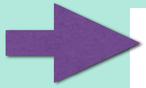
All go files must end in*.go.

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{}))
  })
})
```

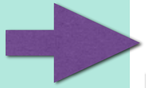
Go - Basic Anatomy



```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{}))
  })
})
```

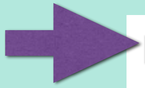
Go - Basic Anatomy

```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

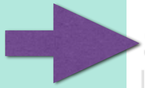
```go
package prime_factors

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

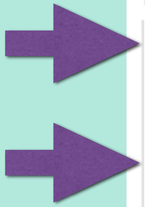
```go
package prime_factors

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{}))
  })
})
```

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

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

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

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

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

```
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{}
  }
}
```

```
It("has factors of 5 as [5]", func() {
  Expect(prime_factors.Factors(5)).To(Equal([]int{5}))
})
```

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

```
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{}
  }
}
```

```
It("has factors of 7 as [7]", func() {
  Expect(prime_factors.Factors(7)).To(Equal([]int{7}))
})
```

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

```
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
}
```

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

```
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.com
@crebma
@testdouble
@selfconference