

## Geography Lesson

*Our task for this exercise is to normalize latitude/longitude coordinates, translating a large number of input formats into standard Cartesian points in [x,y] form. We aim to do this with as little duplication as possible in our implementation code, taking advantage of any similarities we can find between the distinct input formats.*

### The Scene

We have a ton of geospatial data that we want to put to good use. But because it was obtained from a number of different sources, it is far from normalized.

Fortunately, Ruby is a great text processing language. With a few well placed regular expressions and some clever string manipulation techniques, we can make quick work of this task.

Rather than attacking each input format we need to handle as its own distinct case, we will look for ways to transform them into something much easier to work with. But before we go any farther, let's take a look at what exactly we need to build.

### The Requirements

We must support each format that is on the list of valid input formats provided alongside the problem description. The formats always express numerical values as some combination of degrees, minutes, and seconds. Whether the value ends up getting expressed as a positive or negative number depends on either the symbolic sign that preceeds the number, or the compass direction which may come before or after the numerical value. Finally, latitude always appears before longitude in our supported formats.

With three different ways of representing the numerical values and three ways of expressing direction, one would expect that we would need to handle nine distinct cases. But we can certainly do better than that, as you'll soon see.

Regardless of the input format, we want to end up with a normalized array of floating point numbers in [x,y] form, as you can see in the examples below:

```
>> LatLong.to_xy("45.1234 36.2345")
=> [36.2345, 45.1234]
>> LatLong.to_xy("45.1234 S 36.2345 E")
=> [36.2345, -45.1234]
>> LatLong.to_xy("N 40 30.6 W 65 36.36")
=> [-65.606, 40.51]
>> LatLong.to_xy("S 45 45 4.5 W 38 30 10.8")
=> [-38.503, -45.75125]
```

Once we have a parser that can handle all of the input formats and spit out arrays like the ones above, we're done. If we're clever, we can find a way to tie these seemingly different cases together.

## Start Simple

The most basic case is working with decimal degrees and symbolic signs. The tests below map inputs to their expected outputs:

```
require "contest"

class LatLongTest < Test::Unit::TestCase

  def assert_coords(expected, input)
    assert_equal expected, LatLong.to_xy(input)
  end

  test "parse +/--(D+).(D+) +/--(D+).(D+) as signed decimal degrees" do
    assert_coords [ 36.2345, 45.1234], "45.1234 36.2345"
    assert_coords [ 36.2345, 45.1234], "+45.1234 +36.2345"
    assert_coords [-36.2345, -45.1234], "-45.1234 -36.2345"
    assert_coords [-36.2345, 45.1234], "45.1234 -36.2345"
    assert_coords [-36.2345, 45.1234], "+45.1234 -36.2345"
  end
end
```

Here we're using Ruby's built in test/unit with a little support from the contest helper library. We've built a custom assertion to save us some typing throughout our tests, called `assert_coords()`. This simply passes our input string through `LatLong.to_xy()` and compares the return value to our expected array.

When we look at the actual assertions, its easy to see what's going on. The numbers in the strings are getting converted into Ruby floating point numbers, and their order is swapped from (y,x) to (x,y) form. In order to get our tests to pass, we don't need to do a whole lot:

```

module LatLong
  extend self

  def to_xy(input)
    lat, long = input.split(/ /)
    [Float(long), Float(lat)]
  end

end

```

By simply splitting on the empty space, we get our two component numbers as strings. We need to flip the order so that our output is in [x,y] form, and we also need to convert them into Float objects. We use the Float() method here instead of to\_f() because it is a bit more strict, but we avoid any complicated validations working under the assumption that our input data will be well formed. This code is good enough as it is to get our initial set of tests to pass.

If all the cases were this simple, there wouldn't be much left to talk about. But that doesn't necessarily mean that we'll need our trivial solution in order to account for the more complex formats. Let's see what we can do to salvage the work we've done so far as we approach some of the other cases.

## Stay Simple

Consider the second easiest case we need to handle: Decimal degrees with preceding compass directions. The following tests describe the correct mappings between input and output data:

```

test "parse N/S (D+).(D+) E/W (D+).(D+) as decimal degrees" do
  assert_coords [36.2345, 45.1234], "N 45.1234 E 36.2345"
  assert_coords [-36.2345, -45.1234], "S 45.1234 W 36.2345"
  assert_coords [-36.2345, 45.1234], "N 45.1234 W 36.2345"
end

```

If you are feeling a bit of Déjà vu here, that's a good thing. Let's compare one of the new assertions with one of the ones that is already passing:

```

# we got this working
assert_coords [-36.2345, 45.1234], "+45.1234 -36.2345"

# we want to get this to work
assert_coords [-36.2345, 45.1234], "N 45.1234 W 36.2345"

```

We want to be as lazy as possible here. If we can transform the second bit of input so that it looks like the first, we can re-use the code we have already written. Using String#gsub(), this is an easy thing to do:

```

def to_xy(input)
  text = input.gsub(/(N|E) /, "+").gsub(/(S|W) /, "-")

  lat, long = text.split(/ /)
  [Float(long), Float(lat)]
end

```

We simply substitute any instances of "N " or "E " with a "+" symbol, and any instances of "S " or "W " with a "-" symbol. This transformation normalizes the input text to look the same as the first case we handled, allowing us to reuse our original code to do the rest of the work. Since `gsub()` returns the original string when the pattern does not match, this code does not change how we deal with signed decimal inputs.

The conversion of compass directions into +/- symbols gets us far, but on its own it still won't be able to handle the following sorts of input, where the directions come after the numbers:

```

test "parse (D+).(D+) N/S (D+).(D+) E/W as decimal degrees" do
  assert_coords [ 36.2345, 45.1234], "45.1234 N 36.2345 E"
  assert_coords [-36.2345, -45.1234], "45.1234 S 36.2345 W"
  assert_coords [-36.2345, 45.1234], "45.1234 N 36.2345 W"
end

```

We follow with a similar approach as before by thinking "wouldn't it be nice if those directions were at the front of our numbers?". With a bit of pre-processing, they can be. One extra `gsub()` call gets us where we need to be:

```

module LatLong
  extend self

  def to_xy(input)
    text = normalize_text(input)

    lat, long = text.split(/ /)
    [Float(long), Float(lat)]
  end

  private

  def normalize_text(text)
    text.gsub(/^(.*)\s(N|S)\s(.*)\s(E|W)$/, '\2 \1 \4 \3')
      .gsub(/(N|E) /, "+").gsub(/(S|W) /, "-")
  end
end

```

We've pushed things down into the private method `normalize_text()` to keep things organized, but the significant change is here:

```

text.gsub(/^(.*)\s(N|S)\s(.*)\s(E|W)$/, '\2 \1 \4 \3')

```

In a string in which the directions come ahead of the numbers, this call won't produce a modified string. However, if the directions follow the numbers, this

code reorders matches within the () so that we end up with the directions in front. This collapses this third distinct case into the second, which in turn gets collapsed into the first.

At this point, we have covered the three unique ways of expressing directions in our input, and can now tackle numerical representations. Let's take a look at the sort of assertions we'll need to pass:

```
test "parse +/--(D+) (D+).(D+) +/--(D+) (D+).(D+) " +
  "as signed degrees(int) and minutes(decimal)" do
  assert_coords [ 65.606, 40.51], "40 30.6 65 36.36"
  assert_coords [ 65.606, 40.51], "+40 30.6 +65 36.36"
  assert_coords [-65.606, -40.51], "-40 30.6 -65 36.36"
  assert_coords [-65.606, 40.51], "40 30.6 -65 36.36"
  assert_coords [-65.606, 40.51], "+40 30.6 -65 36.36"
end

test "parse +/--(D+) (D+) (D+).(D+) +/--(D+) (D+) (D+).(D+)" +
  "as signed degrees(int) minutes(int) and seconds(decimal)" do
  assert_coords [ 38.503, 45.75125], "45 45 4.5 38 30 10.8"
  assert_coords [ 38.503, 45.75125], "+45 45 4.5 +38 30 10.8"
  assert_coords [-38.503, -45.75125], "-45 45 4.5 -38 30 10.8"
  assert_coords [-38.503, 45.75125], "45 45 4.5 -38 30 10.8"
  assert_coords [-38.503, 45.75125], "+45 45 4.5 -38 30 10.8"
end
```

This seems complicated at a glance, but the underlying formula is simple. Once we have a way to break the strings above into degrees, minutes, and seconds, the following code will do the trick:

```
def to_decimal(degrees, minutes=0, seconds=0)
  sign = degrees / degrees.abs

  sign*(degrees.abs + minutes/60 + seconds/3600)
end
```

Since the degrees carry the sign for all three components, we need to be a bit careful about the way we do our calculations, but it is nothing too exciting.

We use this helper in `to_xy()`, generalizing our numeric input support to cover all three cases.

```
def to_xy(input)
  text = normalize_text(input)

  numbers = text.split(/ /).map { |e| Float(e) }

  case numbers.length
  when 2
    [numbers[1], numbers[0]]
  when 4
    [to_decimal(*numbers[2..3]), to_decimal(*numbers[0..1])]
  when 6
    [to_decimal(*numbers[3..5]), to_decimal(*numbers[0..2])]
  end
end
```

## The Finished Product

At this point, we have everything we need to process all of the formats we are required to support. At the end of this document, you'll find the complete test and implementation code for reference. But a quick trip to irb verifies that things work as expected for each of the required formats:

```
>> LatLong.to_xy("45.1234 -36.2345")
=> [-36.2345, 45.1234]
>> LatLong.to_xy("45.1234 S 36.2345 E")
=> [36.2345, -45.1234]
>> LatLong.to_xy("N 45.1234 W 36.2345")
=> [-36.2345, 45.1234]
>> LatLong.to_xy("40 30.6 N 65 36.36 E")
=> [65.606, 40.51]
>> LatLong.to_xy("N 40 30.6 E 65 36.36")
=> [65.606, 40.51]
>> LatLong.to_xy("-40 30.6 -65 36.36")
=> [-65.606, -40.51]
>> LatLong.to_xy("45 45 4.5 N 38 30 10.8 W")
=> [-38.503, 45.75125]
>> LatLong.to_xy("S 45 45 4.5 W 38 30 10.8")
=> [-38.503, -45.75125]
>> LatLong.to_xy("45 45 4.5 -38 30 10.8")
=> [-38.503, 45.75125]
```

What on the surface appears to be nine distinct input formats boils down to only three key differences:

- Some formats use +/- to indicate directions, others use compass directions
- Compass directions can be expressed either before or after each number
- An individual x or y component consists of degrees, minutes, and seconds, where minutes and seconds can be omitted

These are the differences we focused on, and it turns out there's simply nothing else left to separate the formats, even as you add tests for the ones we didn't explicitly discuss here.

If all we did was blind implement each format according to its test case, we would have coded our implementation to handle 9 different cases when there are only really three issues to consider. This is a strong reminder of why refactoring is such an important part of the TDD cycle.

## Extra Credit

We assumed valid input to our parser, which allowed us to cut a lot of corners. If we couldn't assume that our input was valid, what sort of checks would we need to add? Can you come up with invalid input data that would be hard to detect given our current implementation?

## test\_latlong.rb

```
class LatLongTest < Test::Unit::TestCase

  def assert_coords(expected, input)
    assert_equal expected, LatLong.to_xy(input)
  end

  # In the following tests (D+) means "one or more digits"

  test "parse +/- (D+).(D+) +/- (D+).(D+) as signed decimal degrees" do
    assert_coords [ 36.2345, 45.1234], "45.1234 36.2345"
    assert_coords [ 36.2345, 45.1234], "+45.1234 +36.2345"
    assert_coords [-36.2345, -45.1234], "-45.1234 -36.2345"
    assert_coords [-36.2345, 45.1234], "45.1234 -36.2345"
    assert_coords [-36.2345, 45.1234], "+45.1234 -36.2345"
  end

  test "parse N/S (D+).(D+) E/W (D+).(D+) as decimal degrees" do
    assert_coords [36.2345, 45.1234], "N 45.1234 E 36.2345"
    assert_coords [-36.2345, -45.1234], "S 45.1234 W 36.2345"
    assert_coords [-36.2345, 45.1234], "N 45.1234 W 36.2345"
  end

  test "parse (D+).(D+) N/S (D+).(D+) E/W as decimal degrees" do
    assert_coords [ 36.2345, 45.1234], "45.1234 N 36.2345 E"
    assert_coords [-36.2345, -45.1234], "45.1234 S 36.2345 W"
    assert_coords [-36.2345, 45.1234], "45.1234 N 36.2345 W"
  end

  # Degrees + Minutes / 60

  test "parse (D+) (D+).(D+) N/S (D+) (D+).(D+) E/W " +
    "as degrees(int) and minutes(decimal)" do
    assert_coords [ 65.606, 40.51], "40 30.6 N 65 36.36 E"
    assert_coords [-65.606, -40.51], "40 30.6 S 65 36.36 W"
    assert_coords [-65.606, 40.51], "40 30.6 N 65 36.36 W"
  end

  test "parse N/S (D+) (D+).(D+) E/W (D+) (D+).(D+) " +
    "as degrees(int) and minutes(decimal)" do
    assert_coords [ 65.606, 40.51], "N 40 30.6 E 65 36.36"
    assert_coords [-65.606, -40.51], "S 40 30.6 W 65 36.36"
    assert_coords [-65.606, 40.51], "N 40 30.6 W 65 36.36"
  end

  test "parse +/- (D+) (D+).(D+) +/- (D+) (D+).(D+) " +
    "as signed degrees(int) and minutes(decimal)" do
    assert_coords [ 65.606, 40.51], "40 30.6 65 36.36"
    assert_coords [ 65.606, 40.51], "+40 30.6 +65 36.36"
    assert_coords [-65.606, -40.51], "-40 30.6 -65 36.36"
    assert_coords [-65.606, 40.51], "40 30.6 -65 36.36"
    assert_coords [-65.606, 40.51], "+40 30.6 -65 36.36"
  end

end
```

```

# Degrees + Minutes / 60 + Seconds / 3600

test "parse (D+) (D+) (D+).(D+) N/S (D+) (D+) (D+).(D+) E/W " +
  "as degrees(int) minutes(int) and seconds(decimal)" do
  assert_coords [ 38.503, 45.75125], "45 45 4.5 N 38 30 10.8 E"
  assert_coords [-38.503, -45.75125], "45 45 4.5 S 38 30 10.8 W"
  assert_coords [-38.503, 45.75125], "45 45 4.5 N 38 30 10.8 W"
end

test "parse N/S (D+) (D+) (D+).(D+) E/W (D+) (D+) (D+).(D+) " +
  "as degrees(int) minutes(int) and seconds(decimal)" do
  assert_coords [ 38.503, 45.75125], "N 45 45 4.5 E 38 30 10.8"
  assert_coords [-38.503, -45.75125], "S 45 45 4.5 W 38 30 10.8"
  assert_coords [-38.503, 45.75125], "N 45 45 4.5 W 38 30 10.8"
end

test "parse +/- (D+) (D+) (D+).(D+) +/- (D+) (D+) (D+).(D+)" +
  "as signed degrees(int) minutes(int) and seconds(decimal)" do
  assert_coords [ 38.503, 45.75125], "45 45 4.5 38 30 10.8"
  assert_coords [ 38.503, 45.75125], "+45 45 4.5 +38 30 10.8"
  assert_coords [-38.503, -45.75125], "-45 45 4.5 -38 30 10.8"
  assert_coords [-38.503, 45.75125], "45 45 4.5 -38 30 10.8"
  assert_coords [-38.503, 45.75125], "+45 45 4.5 -38 30 10.8"
end

end

```

## latlong.rb

```

module LatLong
  extend self

  def to_xy(input)
    text = normalize_text(input)

    numbers = text.split(/ /).map { |e| Float(e) }

    case numbers.length
    when 2
      [numbers[1], numbers[0]]
    when 4
      [to_decimal(*numbers[2..3]), to_decimal(*numbers[0..1])]
    when 6
      [to_decimal(*numbers[3..5]), to_decimal(*numbers[0..2])]
    end
  end

  private

  def normalize_text(text)
    text.gsub(/^(.*)\s(N|S)\s(.*)\s(E|W)$/, '\2 \1 \4 \3').
      gsub(/(N|E) /, "+").gsub(/(S|W) /, "-")
  end

  def to_decimal(degrees, minutes=0, seconds=0)
    sign = degrees / degrees.abs

    sign*(degrees.abs + minutes/60 + seconds/3600)
  end
end

```