## Proposal: add Geo::Coord class to standard library

Add Geo::Coord class to Ruby standard library, representing [latitude, longitude] pair + convenience methods. Add Geo standard library with additional calculations and convenience methods.

### Rationale

In modern applications, working with geographical coordinates is frequent. We propose to think of such coordinates (namely, latitude, longitude pair) as of "basic" type that should be supported by standard library - the same way as we support Time/Date/DateTime instead of having it defined by user/gems.

This type is too "small" to be defined by separate gem, so, all of existing geo gems (GeoKit, RGeo, GeoRuby, Graticule etc.) define their own Lat/Lng, or Location, or Point, whatever.

On other hand, API design for this "small" type is vague enough for all those similar types to be incompatible and break small habits and conventions when you change from one geo library to another, or try to use several simultaneously.

Additionally, many gems somehow working with geo coordinates (for weather, or timezone, or another tasks) generally prefer not to introduce type, and just work with [lat, lng] array, which is not very convenient, faithfully.

So, having "geo coordinates" functionality in standard library seems reasonable and valuable.

### Existing/reference solutions

#### Ruby:

- **GeoKit::LatLng**;
- **RGeo::Feature::Point** (with several "private" implementation classes); RGeo implements full OGC Simple Features specification, so, its points have z and m coordinates, projection and much more;
- **Graticule::Location** (not strictly a [lat,lng] wrapper);
- **Rosamary::Node** (follows naming convention of underlying OpenStreetMap API);

Other sources:

- Python: **geopy.Point**;
- **ElasticSearch** uses hash with "lat" and "lon" keys;
- Google Maps **Geocoding API** uses hash with "lat" and "lng" keys;
- **PostGIS**: pretty complicated has geometrical (projected) and geographical (lat, lng) points and stuff.

### Design decisions

While designing Geo library, our reference point was standard Time class (and, to lesser extent, Date/DateTime). It has this responsibilities:

- stores data in simple internal form;
- helps to parse and format data to and from strings;
- provides easy access to logical components of data;
- allows most simple and unambiguous calculations.

**Main type name**: as far as we can see, there's no good singular name for (lat, lng) pair concept. As mentioned above, there can be
seen names like LatLng, or Location, or Point; and in natural language just "coordinates" used frequently. We propose the name Coord, which is pretty short, easy to remember, demonstrates intentions (and looks like singular, so you can have "one coord object" and "array of coords", which is not 100% linguistically correct, yet convenient). Alternative Point name seems to be too ambiguous, being used in many contexts.

Geo::Coord object is **immutable**, there's no semantical sense in location.latitude = ... or something like this.

**Units:** Geo calculations (just like Time calculations) provide no units options, just returning numbers measured in "default" units: metres for distances (as they are SI unit) and degrees for azimuth. Latitude and longitude are stored in degrees, but radians values accessors are provided (being widely used in geodesy math).

All coordinates and calculations are thought to be in WGS 84 coordinates reference system, being current standard for maps and GPS.

There's introduced **concept of globe** used internally for calculations. Only generic (sphere) and Earth globes are implemented, but for 2016 we feel like current design of basic types should take in consideration possibility of writing Ruby scripts for Mars maps analysis. Only one geodesy formula is implemented (Vincenty, generally considered one of the most precise), as for standard library class it considered unnecessary to provide user with geodesy formulae options.

No **map projection** math was added into current proposal, but it may be a good direction for further work. No **elevation** data considered either.

**Proposal details**

**Geo::Coord class**

Represents [latitude, longitude] pair. Latitude is -90 to +90 (degrees). Longitude is -180 to +180.

Class methods:

- new(lat, lng) creates instance from two Numerics (in degrees);
- new(lat:, lng:) keyword arguments form of above;
- new(latd:, latm:, lats:, lath:, lngd:, lngm:, lngs:, lngh:) creates instance from coordinates in (deg, min, sec, hemisphere) form; hemispheres are "N"/"S" for latitude and "W"/"E" for longitude; any component except for degrees can be omitted; if hemisphere is omitted, it is decided by degrees sign (lat: positive is "N", lng: positive is "E");
- from_h(hash) creates instance from hash with "lat" or "latitude" key and "lon" or "lng" or "longitude" key (case-independent);
- from_radians(phi, la) creates instance from radian values;
- strpcoord parses string into coordinates by provided pattern (see below for pattern description);
- parse_ll parses coordinates string in "float, float" form;
- parse_dms parses coordinates string in d m s h, d m s h format (considering several widely used symbols for degrees, minutes and seconds);
- parse tries to parse string into coordinates from various formats.

Instance methods:

- lat and lng, returning Floats, signed;
- latitude and longitude as an aliases; lon as an additional aliases for longitude;
- latd, latm, lats, lath: degree, minute, second, hemisphere; latd and latm are Fixnum, lats is Float, lath is "N"/"S"; all numbers are unsigned;
- lngd, lngm, lngs, lngh: the same for longitude (hemisphere is "W"/"E");
- latdms(nohemisphere = false) returns [latd, latm, lats, lath] with nohemisphere param equal to false, and [±latd, latm, lats] with true; same with lngdms for longitude;
- phi and ϕ is latitude in radians (helpful for math), λ or λ is longitude in radians (not lambda to not confuse with Kernel method);
- to_s returning string like "50.004444,36.231389" (good for map URLs construction, for example);
- to_h(lat: :lat, lng: :lng) converts coord to hash (with desired key names);
- to_a converts coord to simple [lat, lng] pair;
- strcoord(formatstr) for complex coordinate formatting (see below for format string description);
- distance(other) calculates distance to another point (in metres);
- azimuth(other) calculates direction to target (in degrees);
- endpoint(direction, azimuth) calculates final point of the line of distance metres going in azimuth direction from current point.

**strpcoord**/strcoord

Example:
Kharkiv

strfcoord('%latd%latm%lats %lath, %lngd%lngm%lngs %lngH')
# => "50°0′16″ N, 36°13′53″ E"

Directives:
- %lat - full latitude, float; can be formatted with more control like %.4lat (four digits after point) or %+lat (explicit plus sign for positive latitudes);
- %latd - latitude degrees, unsigned, integer
- %latm - latitude minutes, unsigned, integer
- %lats - latitude seconds, unsigned, integer, but can be formatted as float: %.2lats
- %lath - latitude hemisphere, one letter ("N"/"S")
- %lng, %lngd, %lngm, %lgs, %lngs, %lngH - same for longitude
- %% literal % sign

Current implementation

Proposed implementation can be found at https://github.com/zverok/geo_coord.

It was created with thoughts of standard library, so, all docs are in RDoc format, and tests/specs are in mspec-compatible rspec flavour.

History

#1 - 05/10/2016 04:19 AM - duerst (Martin Dürst)
On rubygems, I see a lot of gems with ‘geo’ in their name. Which one is yours? How popular is it? These days, it's easy to use something as a gem, so it's not that necessary to include things into the standard library.

You make a lot of comparisons with Time. Time has things such as timezones. Geo coordinates also have all kinds of complicated details. I happen to lurk on the IETF geojson list, and there such things get discussed. For example, there seems to be Lat/Long formats and Long/Lat formats. Why did you choose Lat/Long? Also, people there often talk about something called CRS (coordinate reference system, I guess). Which one do you use.

#2 - 05/10/2016 07:40 AM - shevegen (Robert A. Heiler)
Would it actually not be better to instead add one or two methods to time, date or datatime?

I guess your proposal would add a default new toplevel namespace Geo, which I am not sure is a good idea per se since other projects may want to use it. Time/Date/DateTime on the other hand already exist more or less (at the least Time).

#3 - 05/10/2016 08:02 AM - zverok (Victor Shepelev)
Martin Dürst wrote:

On rubygems, I see a lot of gems with 'geo' in their name. Which one is yours? How popular is it? These days, it's easy to use something as a gem, so it's not that necessary to include things into the standard library.

I referenced "gem question" in original proposal: "This type is too "small" to be defined by separate gem, so, all of existing geo gems (GeoKit, RGeo, GeoRuby, Graticule etc.) define their own LatLng, or Location, or Point, whatever.".

I proposed it this way, because standard library is a "common denominator" for all gems and libraries. If some Ruby library works with time values, it returns either Time or DateTime, and if it is not -- that would be a common first suggestion for library design.

But about geographical coordinates, current state is: there is no gem which is "just geographical coordinates", well-designed and concise. All popular geo-* gems are some combines of coordinate reference systems, database access, file formats parsing, geocoding through some APIs and so on and so on. All of them (and many other geography-aware gems) frequently accept and return "pair of coordinates" type, and all of them use incompatible "basic type", or just a tuple of two values.

Whether Coord type would be in standard library, it can became common data exchange format; if it would be separate gem, it would definitely not.

So, its just a repo with proposal, not a gem.

You make a lot of comparisons with Time. Time has things such as timezones. Geo coordinates also have all kinds of complicated details. I happen to lurk on the IETF geojson list, and there such things get discussed.

Currently, I'm just following "principle of least surprise" and waiting for input from the community.

For example, there seems to be Lat/Long formats and Long/Lat formats. Why did you choose Lat/Long?
Because for "common user" the question of right order seems obsolete now, with spreading of casual GPS software and online maps, all of them seemingly using lat/long order.

Also, people there often talk about something called CRS (coordinate reference system, I guess). Which one do you use.

Same as with coordinates order, just a "common" WGS-84. Added mention of it to README and code.

#4 - 05/10/2016 08:11 AM - zverok (Victor Shepelev)

Robert A. Heiler wrote:

Would it actually not be better to instead add one or two methods to time, date or datetime?

I guess your proposal would add a default new toplevel namespace Geo, which I am not sure is a good idea per se since other projects may want to use it. Time/Date/DateTime on the other hand already exist more or less (at the least Time).

I'm not sure how coordinates processing can be added to Time/Date classes. Could you explain please?..

On other hand, I think adding Geo namespace can be justified exactly by fact of creating new "basic" types useful for everyone. And, as far as I can see, no existing and relatively popular geographic gem (RGeo, GeoRuby, Graticule, GeoKit) uses it currently

#5 - 05/10/2016 08:15 AM - zverok (Victor Shepelev)
- Description updated

#6 - 05/10/2016 09:16 AM - duerst (Martin Dürst)

Victor Shepelev wrote:

I referenced "gem question" in original proposal: "This type is too "small" to be defined by separate gem, so, all of existing geo gems (GeoKit, RGeo, GeoRuby, Graticule etc.) define their ownLatLng, or Location, or Point, whatever."

Nothing is too small to be defined as a gem. I propose you create this as a gem, then have the existing geo gems use it, and then come back to your proposal here.

#7 - 05/10/2016 09:43 AM - zverok (Victor Shepelev)

You see, my point was not "I created a cool code, lets add it to stdlib!"

My point was "We need standard Geo Coordinates type in stdlib" (because of <reasons>), I've just thought it would be easier to discuss with some sample code (but I can throw it away, if it affects the discussion).

Let's say it this way:

Proposal: We need "geo coordinates" type in standard library.

Reason:

- in modern applications (web, mobile, scientific and so on) it is pretty common type;
- there are many incompatible implementations in several popular libraries;
- for such a basic type, no "external library" would be never widely accepted (think "ok, create your own C++ string class and make it widely accepted! but in the meantime, just use char" -- std::string had to fight for years to be used in some of the libraries);
- the earlier new type became "standard", the better.

Whether my design and implementation is good or not, is not exactly relevant to proposal -- it meant to be starting point for discussion.

And that's it.

#8 - 06/13/2016 02:50 PM - matz (Yukihiro Matsumoto)
- Status changed from Open to Rejected

Although I feel it's good to have common coordination class, I don't think it's wise to pick one implementation over many and add it to the standard library. It should be done through sound competition.

If you need any help from the core team (other than adding it to the standard lib), let us know.

Matz.

#9 - 06/13/2016 02:52 PM - zverok (Victor Shepelev)

Thanks. Already did it into gem: https://github.com/zverok/geo_coord -- and will see how it is playing with ecosystem.
Although I feel it's good to have common coordination class, I don't think it's wise to pick one implementation over many and add it to the standard library. It should be done through sound competition.

To be honest, at first (it was ~February), I've just wanted to post here a mere proposal like "let's create a common class in stdlib" and call it a day.

Then, I've thought that some kind of "reference implementation" (which can be checked and discussed practically) will help to move into pragmatic matters as soon as possible.

Then, I've spent some time to create an implementation to go with the proposal.

And then, proposal was failed and rejected because it is implemented :) That's kind of funny.

Great to see the gem. With respect to

And then, proposal was failed and rejected because it is implemented :) That's kind of funny.

I think you misunderstood Matz. We could have picked just your basic data representation, without some or all of the methods. But what we would like to see is more buy-in from the geocommuting community. The best way to show such buy-in is to see your gem adopted.

We could have picked just your basic data representation, without some or all of the methods.

Thing is, I thought about my proposal as a three parts (with each next point being less important):

1. Standard library should have a class for geocoordinates
2. (Less important) It could have an interface like this, or something... Let's talk
3. (Even less important) Here is some experimental implementation, allowing to play with concept and discuss it.

I'm absolutely totally 100% OK with rejection of (3), a bit concerned about rejection of (2) with no discussion, but my only real concern about rejection of (1).

Now I'm thinking maybe if this ticket was originally created with a short text "Add geocoordinates class to Ruby standard library, representing [latitude, longitude] pair + convenience methods." and no links to proposed interface/implementation, it would be more useful for community :)

And, faithfully, do you believe such a class could be added to standard library by joined effort of several different geography gem authors?..

My only point is: let's add it to stdlib at some point in future. Interface could be any, implementation could be any, but let's have the class with this role in stdlib.