pyturf Documentation

Release 0.5.0

pyturf

Jun 05, 2020

CONTENTS:

`pyturf` is a powerful geospatial library written in python, based on turf.js, a popular library written in javascript. It follows the same modular structure and maintains the same functionality as the original modules in that library for maximum compatibility.

It includes traditional geospatial operations, as well as helper functions for creating and manipulating GeoJSON data.

CHAPTER

ONE

INSTALLATION

```
$ pip install pyturf
```

CHAPTER

TWO

USAGE

Most `pyturf` modules expect as input GeoJSON features or a collection of these, which can be the following:

- Point / MultiPoint
- LineString / MultiLineString
- Polygon / MultiPolygon

These can either be defined as a python dictionary or as objects from `pyturf` helper classes.

```
# example as a dictionary:

point1 = {
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Point",
    # Note order: longitude, latitude.
    "coordinates": [-73.988214, 40.749128]
  }
}

...

# Example using objects from helper classes

from turf import point

# Note order: longitude, latitude.
point1 = point([-73.988214, 40.749128])
```

In order to use the modules, one can import directly from `pyturf`, such as:

```
from turf import distance, point

point1 = point([-73.988214, 40.749128])
point2 = point([-73.838432, 40.738484])

dist = distance(point1, point2, {"units": "miles"})
```

2.1 Measurement

2.1.1 along

`turf.along`(*line*, *dist*, *options=None*)

Takes a LineString and returns a Point at a specified distance along the line

Parameters

- **`line`** – input LineString
- **`dist`** – distance along the line
- **`options`** – optional parameters [options[“units”]=”kilometers”] can be degrees, radians, miles, or kilometers

Returns Point *dist units* along the line

2.1.2 area

`turf.area`(*features*)

Takes one or more features and returns their area in square meters.

Parameters **`features`** – geojson input GeoJSON feature(s)

Returns area in square meters

2.1.3 bbox

`turf.bbox`(*features*)

Takes a set of features and returns a bounding box containing of all input features.

Parameters **`features`** – any GeoJSON feature or feature collection

Returns bounding box extent in [minX, minY, maxX, maxY] order

2.1.4 bbox-polygon

`turf.bbox_polygon`(*bbox*, *options=None*)

Takes a bounding box and returns an equivalent Polygon feature.

Parameters

- **`bbox`** – bounding box extent in [minX, minY, maxX, maxY] order
- **`options`** – optional parameters [options[“properties”]={}] Translate GeoJSON Properties to Point [options[“id”]={}] Translate GeoJSON Id to Point

Returns a Polygon representation of the bounding box

2.1.5 bearing

`turf.bearing(start, end, options=None)`

Takes two points and finds the geographic bearing between them, i.e. the angle measured in degrees from the north line (0 degrees)

Parameters

- **`start`** – starting point [lng, lat] or Point feature
- **`end`** – ending point [lng, lat] or Point feature
- **`options`** – dictionary with options: [options[“final”]] - calculates the final bearing if true

Returns bearing in decimal degrees, between -180 and 180 (positive clockwise)

2.1.6 center

`turf.center(features, options=None)`

Takes a Feature or FeatureCollection and returns the absolute center point of all features.

Parameters

- **`features`** – features or collection of features
- **`options`** – optional parameters [options[“properties”]={}] Translate GeoJSON Properties to Point [options[“bbox”]={}] Translate GeoJSON BBox to Point [options[“id”]={}] Translate GeoJSON Id to Point

Returns a Point feature at the absolute center point of all input features

2.1.7 centroid

`turf.centroid(features, options=None)`

Takes one or more features and calculates the centroid using the mean of all vertices. This lessens the effect of small islands and artifacts when calculating the centroid of a set of polygons.

Parameters

- **`features`** – GeoJSON features to be centered
- **`options`** – optional parameters [options[“properties”]={}] Translate GeoJSON Properties to Point

Returns a Point feature corresponding to the centroid of the input features

2.1.8 destination

`turf.destination(origin, distance, bearing, options=None)`

Takes a Point and calculates the location of a destination point given a distance in degrees, radians, miles, or kilometers; and bearing in degrees. This uses the [Haversine formula](http://en.wikipedia.org/wiki/Haversine_formula) to account for global curvature.

Parameters

- **`origin`** – starting point
- **`distance`** – distance from the origin point
- **`bearing`** – bearing ranging from -180 to 180

- **options** – optional parameters [options[“units”]=’kilometers’] miles, kilometers, degrees, or radians [options[“properties”]={}] Translate properties to Point

Returns destination GeoJSON Point feature

2.1.9 distance

`turf.distance` (*start*, *end*, *options=None*)

Calculates the distance between two Points in degrees, radians, miles, or kilometers. This uses the [Haversine formula](http://en.wikipedia.org/wiki/Haversine_formula) to account for global curvature.

Parameters

- **start** – starting point [lng, lat] or Point feature
- **end** – ending point [lng, lat] or Point feature
- **options** – dictionary with units as an attribute. Can be degrees, radians, miles, or kilometers

Returns distance between the 2 points

2.1.10 envelope

`turf.envelope` (*features*, **args*)

Takes any number of features and returns a rectangular Polygon that encompasses all vertices.

Parameters **features** – any GeoJSON feature or feature collection

Returns bounding box extent in [minX, minY, maxX, maxY] order

2.1.11 length

`turf.length` (*features*, *options=None*)

Calculates the total length of the input Feature / FeatureCollection in the specified units.

Parameters

- **features** – a Feature / FeatureCollection of types LineString, MultiLineString, Polygon or MultiPolygon
- **options** – optional parameters [options[“units”]=kilometers] can be degrees, radians, miles, or kilometers

Returns the measured distance

2.1.12 midpoint

`turf.midpoint` (*point1*, *point2*)

Takes two point features and returns a point midway between them. The midpoint is calculated geodesically, meaning the curvature of the earth is taken into account.

Parameters

- **point1** – first point
- **point2** – second point

Returns a point midway between point 1 and point 2

2.1.13 nearest-point

`turf.nearest_point`(*target: Union[Sequence, Dict, turf.helpers._features.Feature]*, *features: GeoJson*) → turf.helpers._features.Point

Calculates the closest reference point from a feature collection towards a target point This calculation is geodesic.

Parameters

- **`target`** – targetPoint the reference point
- **`features`** – points against input point set

Returns the closest point in the features set to the reference point

2.1.14 point-on-feature

`turf.point_on_feature`(*features: GeoJSON*) → turf.helpers._features.Point

Takes a Feature or FeatureCollection and returns a {Point} guaranteed to be on the surface of the feature.

Given a {Polygon}, the point will be in the area of the polygon Given a {LineString}, the point will be along the string Given a {Point}, the point will the same as the input

Parameters **`features`** – any GeoJSON feature or feature collection

Returns Point GeoJSON Feature on the surface of *input*

2.1.15 point-to-line-distance

`turf.point_to_line_distance`(*point: Union[Sequence, Dict, turf.helpers._features.Feature]*, *line: Union[Sequence, Dict, turf.helpers._features.Feature]*, *options: Dict = None*) → float

Returns the minimum distance between a {Point} and a {LineString}, being the distance from a line the minimum distance between the point and any segment of the *LineString*

http://geomalgorithms.com/a02-_lines.html

Parameters

- **`point`** – Point GeoJSON Feature or Geometry
- **`line`** – LineString GeoJSON Feature or Geometry
- **`options`** – Optional parameters [options["units"]]: any supported unit (e.g. degrees, radians, miles…) [options["method"]]: geodesic or 'planar for distance calculation

Returns distance between point and line

2.1.16 polygon-tangents

`turf.polygon_tangents`(*start_point: PointFeature*, *polygon: PolygonFeature*) → turf.helpers._features.FeatureCollection

Finds the tangents of a {Polygon or(MultiPolygon} from a {Point}.

more: http://geomalgorithms.com/a15-_tangents.html

Parameters

- **`point`** – point [lng, lat] or Point feature to calculate the tangent points from
- **`polygon`** – polygon to get tangents from

Returns Feature Collection containing the two tangent points

2.1.17 rhumb-bearing

`turf.rhumb_bearing`(*origin: Union[Sequence, Dict, turf.helpers._features.Feature]*, *destination: Union[Sequence, Dict, turf.helpers._features.Feature]*, *options: Dict = None*) → float

Takes two {Point|points} and finds the bearing angle between them along a Rhumb line * i.e. the angle measured in degrees start the north line (0 degrees)

https://en.wikipedia.org/wiki/Rhumb_line

Parameters

- **`start`** – starting point [lng, lat] or Point feature
- **`end`** – ending point [lng, lat] or Point feature
- **`options`** – Optional parameters [options[“final”]]: Calculates the final bearing if True

Returns bearing from north in decimal degrees

2.1.18 rhumb-destination

`turf.rhumb_destination`(*features: Dict*, *options: Dict = None*) → turf.helpers._features.Point

Returns the destination {Point} having travelled the given distance along a Rhumb line from the origin Point with the (varant) given bearing.

# https://en.wikipedia.org/wiki/Rhumb_line

Parameters

- **`features`** – any GeoJSON feature or feature collection
- **`properties`** – specification to calculate the rhumb line [options[“distance”]=100] distance from the starting point [options[“bearing”]=180] varant bearing angle ranging from -180 to 180 degrees from north [options[“units”]=kilometers] units: specifies distance (can be degrees, radians, miles, or kilometers)
- **`options`** – optional parameters also be part of features[“properties”] [options[“units”]={}] can be degrees, radians, miles, or kilometers [options[“properties”]={}] Translate GeoJSON Properties to Point [options[“id”]={}] Translate GeoJSON Id to Point

Returns a FeatureDestination point.

2.1.19 rhumb-distance

turf.**rhumb_distance**(*origin*, *destination*, *options: Dict = None*) → float

Calculates the rhumb distance between two Points. Units are defined in helpers._units

# https://en.wikipedia.org/wiki/Rhumb_line

Parameters

- **start** – starting point [lng, lat] or Point feature
- **end** – ending point [lng, lat] or Point feature
- **options** – dictionary with units as an attribute. Units are defined in helpers._units

Returns distance between the 2 points

2.1.20 square

turf.**square**(*bbox*)

Takes a bounding box and calculates the minimum square bounding box that would contain the input.

Parameters **bbox** – bounding box extent in [minX, minY, maxX, maxY] order

Returns a square surrounding bbox

2.1.21 great-circle

turf.**great_circle**(*start*, *end*, *options=None*)

Returns the great circle route as LineString

Parameters

- **start** – source point feature
- **end** – destination point feature
- **options** – Optional parameters [options["properties"]={}] line feature properties [options.npoints=100] number of points

Returns great circle line feature

2.2 Coordinates Mutation

2.3 Transformation

2.4 Feature Conversion

2.4.1 explode

turf.**explode**(*features: GeoJson*) → turf.helpers._features.FeatureCollection

Takes a feature or set of features and returns all positions as {Point|points}.

Parameters **features** – any GeoJSON feature or feature collection

Returns {FeatureCollection} points representing the exploded input features

2.4.2 Polygon to Line

`turf.`**`polygon_to_line`**`()`

Converts a {Polygon} to a {LineString} or a {MultiPolygon} to a {FeatureCollection} or {MultiLineString}.

Parameters

- **`polygon`** – Feature to convert
- **`options`** – Optional parameters

Returns {Feature Collection|LineString|MultiLineString} of converted (Multi)Polygon to (Multi)LineString

2.5 Misc

2.6 Helper

2.6.1 feature

`turf.`**`feature`**`(`*geom: (typing.Dict, <class 'turf.helpers._features.Point'>, <class 'turf.helpers._features.LineString'>, <class 'turf.helpers._features.Polygon'>, <class 'turf.helpers._features.MultiPoint'>, <class 'turf.helpers._features.MultiLineString'>, <class 'turf.helpers._features.MultiPolygon'>), properties: Dict = None, options: Dict = None, as_geojson: bool = True*`)` → Union[turf.helpers._features.Feature, Dict]

Wraps a GeoJSON Geometry in a GeoJSON Feature.

Parameters

- **`geom`** – input geometry
- **`properties`** – a dictionary of key-value pairs to add as properties
- **`options`** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **`as_geojson`** – whether the return value should be a geojson

Returns a GeoJSON feature

2.6.2 feature-collection

`turf.`**`feature_collection`**`(`*features: Sequence, options: Dict = None, as_geojson: bool = True*`)` → Union[turf.helpers._features.FeatureCollection, Dict]

Takes one or more Feature and creates a FeatureCollection.

Parameters

- **`features`** – input features
- **`options`** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **`as_geojson`** – whether the return value should be a geojson

Returns a FeatureCollection of Features

2.6.3 geometry

`turf.geometry`(*geom_type: str*, *coordinates: Sequence*, *as_geojson: bool = True*) → Union[Dict, turf.helpers._features.Point, turf.helpers._features.LineString, turf.helpers._features.Polygon, turf.helpers._features.MultiPoint, turf.helpers._features.MultiLineString, turf.helpers._features.MultiPolygon]

Creates a GeoJSON {@link Geometry} from a Geometry string type & coordinates. For GeometryCollection type use *helpers.geometryCollection*

Parameters

- **geom_type** – one of "Point" | "LineString" | "Polygon" | "MultiPoint" | "MultiLineString" | "MultiPolygon"
- **coordinates** – array of coordinates [lng, lat]
- **as_geojson** – whether the return value should be a geojson

Returns a GeoJSON geometry

2.6.4 line-string

`turf.line_string`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.LineString, Dict]

Creates a LineString Feature from an Array of Positions.

Parameters

- **coordinates** – a list of Positions - Position[]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns a LineString feature

2.6.5 line-strings

`turf.line_strings`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.FeatureCollection, Dict]

Creates a LineString FeatureCollection from an Array of LineString coordinates.

Parameters

- **coordinates** – a list of a list of Positions - Position[][]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns LineString FeatureCollection

2.6.6 multi-line-string

`turf.multi_line_string`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.MultiLineString, Dict]

Creates a MultiLineString Feature based on a coordinate array. Properties can be added optionally.

Parameters

- **coordinates** – a list of a list of Positions - Position[][]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns a MultiLineString feature

2.6.7 point

`turf.point`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.Point, Dict]

Creates a Point Feature from a Position.

Parameters

- **coordinates** – coordinates longitude, latitude position in degrees - Position
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns a Point Feature

2.6.8 points

`turf.points`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.FeatureCollection, Dict]

Creates a Point FeatureCollection from an Array of Point coordinates.

Parameters

- **coordinates** – a list of Points - Position[]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns Point FeatureCollection

2.6.9 multi-point

`turf.multi_point`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.MultiPoint, Dict]

Creates a MultiPoint Feature based on a coordinate array. Properties can be added optionally.

Parameters

- **coordinates** – a list of Points - Position[]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns a MultiPoint feature

2.6.10 polygon

`turf.polygon`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.Polygon, Dict]

Creates a Polygon Feature from an Array of LinearRings.

Parameters

- **coordinates** – a list of a list of Positions - Position[][]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns a Polygon Feature

2.6.11 polygons

`turf.polygons`(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.FeatureCollection, Dict]

Creates a Polygon FeatureCollection from an Array of Polygon coordinates.

Parameters

- **coordinates** – an array of polygons - Position[][][]
- **properties** – a dictionary of key-value pairs to add as properties
- **options** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **as_geojson** – whether the return value should be a geojson

Returns Polygon FeatureCollection

2.6.12 multi-polygon

`turf.`**`multi_polygon`**(*coordinates: Sequence*, *properties: Dict = None*, *options: Dict = None*, *as_geojson: bool = True*) → Union[turf.helpers._features.MultiPolygon, Dict]

Creates a MultiPolygon Feature based on a coordinate array. Properties can be added optionally.

Parameters

- **`coordinates`** – an array of polygons - Position[][][]
- **`properties`** – a dictionary of key-value pairs to add as properties
- **`options`** – an options dictionary: [options["bbox"] Bounding Box Array [west, south, east, north] associated with the Feature [options["id"] Identifier associated with the Feature
- **`as_geojson`** – whether the return value should be a geojson

Returns a MultiPolygon feature

2.7 Random

2.8 Data

2.9 Interpolation

2.10 Joins

2.11 Grids

2.12 Classification

2.13 Aggregation

2.14 Meta

2.14.1 Get coords from features

`turf.invariant.`**`get_coords_from_features`**(*features: Any*, *allowed_types: Sequence = None*) → List

Retrieves coords from Features. Features must be a GeoJSON, a Feature object or a list of coordinates, otherwise it raises an exception.

Parameters

- **`features`** – Any input value(s)
- **`allowed_types`** – allowed Feature types

Returns list with extracted coords

2.14.2 Get coords from features

`turf.invariant.`**`get_coords_from_geometry`**(*geometry: Any*, *allowed_types: Sequence = None*, *raise_exception: bool = True*) → List

Retrieves coords from a given Geometry. Geometry must be a GeoJSON, a Geometry object or a list of coordinates, otherwise it raises an exception.

Parameters

- **`geometry`** – Any input value(s)
- **`allowed_types`** – allowed Feature types
- **`raise_exception`** – if an exception should be raised or if it should be silent

Returns list with extracted coords

2.14.3 Get geometry from features

`turf.invariant.`**`get_geometry_from_features`**(*features: Any*, *allowed_types: Sequence = None*) → List

Retrieves Geometries from Features. Features must be a GeoJSON, a Feature object or a list of coordinates, otherwise it raises an exception.

Parameters

- **`features`** – Any input value(s)
- **`allowed_types`** – allowed Feature types

Returns list with extracted coords

2.15 Assertion

2.16 Booleans

2.16.1 boolean-point-in-polygon

`turf.`**`boolean_point_in_polygon`**(*point: Union[Sequence, Dict, turf.helpers._features.Feature]*, *polygon: Union[Dict, turf.helpers._features.Feature]*, *options: Dict = None*)

Takes a {@link Point} and a Polygon or MultiPolygon and determines if the point resides inside the polygon. The polygon can be convex or concave. The function accounts for holes.

reference:

http://en.wikipedia.org/wiki/Even%E2%80%93odd_rule modified from: https://github.com/substack/point-in-polygon/blob/master/index.js which was modified from http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html

Parameters

- **`point`** – input Point Feature
- **`polygon`** – input Polygon or MultiPolygon Feature
- **`options`** – optional parameters [options["ignoreBoundary"]] True if polygon boundary should be ignored when determining if

the point is inside the polygon otherwise False.

Returns True if the Point is inside the Polygon; False otherwise

2.16.2 boolean-point-on-line

`turf.boolean_point_on_line`(*point: PointFeature*, *line: LineFeature*, *options: Dict = {}*) → bool

Returns True if a point is on a line else False. Accepts a optional parameter to ignore the start and end vertices of the linestring.

Parameters

- **`point`** – {Point} GeoJSON Point
- **`line`** – {LineString} GeoJSON LineString
- **`options`** – Optional parameters [options["ignoreEndVertices"]=False] whether to ignore the start and end vertices

Returns boolean True/False if point is on line

2.17 Unit Conversion

2.17.1 convert-area

`turf.convert_area`(*area*, *original_unit='meters'*, *final_unit='kilometers'*)

Parameters

- **`area`** – area to be converted
- **`original_unit`** – original unit of the area
- **`final_unit`** – returned unit of area

Returns the converted area

2.17.2 convert-length

`turf.convert_length`(*length*, *original_unit='kilometers'*, *final_unit='kilometers'*)

Parameters

- **`length`** – length to be converted
- **`original_unit`** – original unit of the length
- **`final_unit`** – return unit of the length

Returns the converted length

2.17.3 degrees-to-radians

`turf.degrees_to_radians(degrees)`

Parameters **`degrees`** – degrees angle

Returns angle in radians

2.17.4 length-to-degrees

`turf.length_to_degrees(distance, units='kilometers')`

Parameters

- **`distance`** – distance in real units
- **`units`** – units of the distance. Can be degrees, radians, miles, kilometers, inches, yards, metres, meters, kilometres, kilometers. Defaults to kilometers

Returns degrees

2.17.5 length-to-radians

`turf.length_to_radians(distance, units='kilometers')`

Parameters

- **`distance`** – distance in real units
- **`units`** – units of the distance. Can be degrees, radians, miles, kilometers, inches, yards, metres, meters, kilometres, kilometers. Defaults to kilometers

Returns radians

2.17.6 radians-to-degrees

`turf.radians_to_degrees(radians)`

Parameters **`radians`** – radians angle in radians

Returns degrees between 0 and 360

2.17.7 radians-to-length

`turf.radians_to_length(radians, units='kilometers')`

Parameters

- **`radians`** – radians in radians across the sphere
- **`units`** – units of the distance. Can be degrees, radians, miles, kilometers, inches, yards, metres, meters, kilometres, kilometers. Defaults to kilometers

Returns distance

CHAPTER

THREE

CONTRIBUTOR GUIDE

If you want to contribute to the project, fork this repository and check the contributing guide:

3.1 How To Contribute

As the library follows a modular structure, it is fairly easy to contribute by working on a subpackage and submit a pull request of those atomic changes.

For this, first you should check this repo's issues and check which modules are being worked on by filtering by the `in-progress` label. Then, you can check turf.js github repo, choose a module that you'd like to implement, making sure that it hasn't been already implemented or is currently being worked by someone else.

When you're ready, open a new issue on this repo outlining the module you'll be working on, so that subsequent contributors can follow the same logic outlined here.

Please attend to the following guidelines:

- Open an issue in `diogomatoschaves/pyturf` outlining your plan.
- Always include tests. pytest is used in this project.
- `pyturf` modules are small, containing a single exported function. See below for a typical module structure.
- Export your module function by including it in `turf/<your-module>/__init__.py` and `turf/__init__.py`. See below for details.
- GeoJSON is the *lingua franca* of `pyturf`. It should be used as the data structure for anything that can be represented as geography.
- Keep your commits atomic and each of them passing all checks (linter and tests).
- Add your new module under the `Available Modules` section.
- Add your new module under its appropriate section in docs/source/modules.
- Avoid extra dependencies if you can.
- Run the linter before submitting changes (see below for more details).
- Run `python -m pytest --verbose --cov=./` from the project root folder to run all the tests and make sure they pass with a sufficiently high coverage.
- Rebase your branch with the upstream master before opening the PR: `git rebase upstream/master`

After you open the PR, make sure that the CI pipeline passes all checks (on the `Checks` tab of the PR).

3.1.1 Code Style

To ensure consistent code style, black is used in this project. At the root level run:

```
$ black .
```

This will automatically reformat all files according to `black`'s specification.

3.1.2 Structure of a `pyturf` module

For a new module named `new_module`, the following structure should be adhered to.

```
turf
|
├── ...
├── __init__.py
├── new_module
    ├── __init__.py
    ├── _new_module.py
    |
    ├── tests
        ├── test_new_module.py
        ├── in
        |   ├── points.geojson
        |   ├── ...
        |
        ├── out
            ├── points.geojson
            ├── ...
```

3.1.3 Importing modules in `__init__.py` files

In order for the module function to be imported directly from `turf`, we need to import them on the `__init__.py` files on both the module and at the root level. So for example, for a new module named `new_module`, on `turf/new_module/__init__.py` we would include:

```
from turf.new_module._new_module import new_module
```

The same logic can be applied to `turf/__init__.py`:

```
...
from turf.new_module import new_module
...
```

3.1.4 Adding Tests

Tests setup in this project follows a certain pattern that, even if not being a *one size fits all*, if followed should ensure a good test flow in most cases.

The pattern consists of importing the tests input and output through files in `turf/new_module/tests/in` and `turf/new_module/tests/out` respectively, and then parameterizing these fixtures to be used in individual tests as required.

These guidelines should be followed:

- The file where tests are executed should be under the directory `tests`.
- The directory `tests` should have sub directories `in` and `out`, where input and output files should be kept respectively.
- Files in both `in` and `out` for a specific test must have the same name, although they can have different file extensions (eg: `.json` or `.geojson`).

Fixtures and expected outputs can then be imported by means of the function `get_fixtures` defined in `turf/utils/test_setup.py`, by providing the test file path as input:

```
import os
from turf.utils.test_setup import get_fixtures

current_path = os.path.dirname(os.path.realpath(__file__))

fixtures = get_fixtures(current_path)
```

The returned value `fixtures` becomes a dictionary of fixtures, with the file names in `in` and `out` as keys, and in turn each fixture is a dictionary containing keys `"in"` and `"out"`, representing the input and output of the tests respectively.

If for some reason you only have either `in` or `out` fixtures, then in order to avoid errors running the tests you should pass the argument `keys` to `get_fixtures` as shown below:

```
import os
from turf.utils.test_setup import get_fixtures

current_path = os.path.dirname(os.path.realpath(__file__))

fixtures = get_fixtures(current_path, keys=["in"]) # Only retrieve input fixtures
```

These fixtures can then be parameterized as individual tests, allowing for only one test definition to be used in multiple test cases. This would follow a structure of the kind:

```
import pytest
from turf.new_module import new_module # Don't import your function directly from turf

@pytest.mark.parametrize(
    "fixture",
    [
        pytest.param(fixture, id=fixture_name)
        for fixture_name, fixture in fixtures.items()
    ],
)
def test_new_module(self, fixture):

    # This is an example function call
    assert new_module(fixture["in"]) == fixture["out"]
```

In order to run the tests, from the root directory run:

```
$ python -m pytest --verbose --cov=./
```

3.1.5 Updating The Documentation

In case you add a new module, please update also the documentation. The structure for the documentation follows the `turf` structure. You can check turf.js documentation. According to the `turf.js documentation`, you can pick the same `block` name and update it with your addition.

```
turf
|
├── ...
├── __init__.py
├── docs
    ├── ...
    ├── conf.py
    |
    ├── modules
        ├── aggregation.rst
        ├── assertion.rst
        ├── booleans.rst
        ├── ...
```

As an example, for adding the `length module` you would have to add the following lines to `measurements.rst`.

```
Length
------

.. autofunction:: turf.length
```

INDEX