upoints

Release 0.12.2

Dec 13, 2017

Contents

Warning: At this point `upoints` only exists to assist the users who have been using it for years, I *absolutely* do **not** recommend its use to new users.

upoints is a collection of GPL v3 licensed modules for working with points on Earth, or other near spherical objects. It allows you to calculate the distance and bearings between points, mangle xearth/xplanet data files, work with online UK trigpoint databases, NOAA's weather station database and other such location databases.

Previous versions of *upoints* were called `earth_distance`, but the name was changed as it no longer reflected the majority of uses the packages was targeted at.

CHAPTER 1

Contents

1.1 Geolocation and path cross

Spurred on by Seemant's voyage in to geoip for deciding on the location of users writing comments on his website I decided to have a look in to MaxMind's library with the intent of using it in a semi-private pathcross-type application. Unfortunately, it turns out it isn't all that simple but there is some fun to be had along the way.

If, like Seemant, you're looking to simply infer the country a specific IP address originates from then the accuracy of the results from the geoip library are generally quite good. Out of the fifty requests I tried the MaxMind database managed to return the correct results for all but three, and each of those incorrect results are because of interesting proxying games being played by their service providers. The accuracy would likely be much higher than 94% using a more random selection of IPs, but I went out of my way to find ones I expected to return incorrect data(large multinational ISPs, mobile providers and backbone infrastructure owners). That being said, the accuracy of the results drops considerably as you zoom in from country-wide geolocation.

My initial idea had been to use the city data to populate an automatically updating database that could be queried to find people in the local area[1]. Much like some of those oh-so-cool Web 2.0 buzzword laden sites do but without the manual updating, Javascript, lack of privacy and continual spam. Me and a few friends already do such a thing using our published hCalendar entries, a heap of nifty Haskell code and some dirty hack infested XSLT. It works well, but it could do so much better given more data. Unfortunately, the geoip solution wouldn't work as I envisaged because the precision of the city data isn't what I naïvely hoped for.

All that aside, and with the failed plan in tatters on the floor, it did leave a few interesting artefacts to be mulled over instead of doing Real Work™.

1.1.1 How inaccurate?

Using MaxMind's Locate my IP service, which presumably queries their largest and most current database to attract customers, I'm reported as being:

[1] By "automatically updating" I mean simply a ping-and-forget service that listens for a user ID and location and updates the database. My test code was a simple five line Python script, it literally reads a configuration file for the user ID and pings my server.

Attribute	Data
Your IP Address	62.249.253.214
Countries	United Kingdom
Region	O2 (Telford and Wrekin)
Global Cities	Telford
Latitude/Longitude	52.6333/-2.5000
ISP	Telford
Organization	Entanet International Ltd
Netspeed	Dialup
Domain Name	entanet.co.uk

Okay, so at the moment of that query it gets the correct country, net speed, organisation(my ISP UKFSN resells Entanet service) but that is it[2]. Not that we really should be expecting any great accuracy with the data, because of the way IPs are assigned and used.

Assuming that I would be happy with my location being reported as Telford, how inaccurate is the data? In the context of path cross the question is "would I be likely to travel to Telford for a beer?" Time to brush up on spherical trigonometry basics I guess.

The data is reasonably correct in stating a location of N52.6333°; W2.5000° for Telford. My location, assuming the WGS-84 datum, is N52.015°; W0.221°.

Calculating the Great-circle distance between the two coordinates is relatively easy. I've hacked together a really simple Python module called *upoints* that allows you to calculate the distance between two points on Earth(or any other approximately spherical body with a few minor changes). It offers the Law of Cosines and haversine methods for calculating the distance, because they're the two I happen to know.

1.1.2 Too inaccurate?

```
>>> from upoints import point
>>> Home = point.Point(52.015, -0.221)
>>> Telford = point.Point(52.6333, -2.5000)
>>> print("%i kM" % Home.distance(Telford))
169 kM
```

The script above tells us that the distance from my house to Telford is approximately 170 kM (just over 100 miles for those so inclined), given that result what is the answer to my question "would I be likely to travel to Telford for a beer?" Probably not.

The answer isn't that simple though. Whereas I probably wouldn't travel 170 kM for a beer with my good friend Danny(sorry Danny!), I would consider travelling 170 kM to meet up with Roger Beckinsale. It isn't because Danny is bad company(quite the contrary), it is because I live eight kilometres from Danny's house and can pop round for a beer whenever the urge hits me. Roger on the other hand lives on the Isle of Lewis, as far North West as the British Isles go, and I haven't seen him for a year or so.

There is only one conclusion to draw from this: Accuracy is in the eye of the beerholder(sorry!). This conclusion has led me to implement some new features in our manual path cross tool, all based around the idea of relative proximity.

The "average" location of a person is important when calculating whether your paths cross[3]. I'm not really interested in seeing when somebody who works at the same site as me is within twenty kilometres of me as it would clearly happen a lot, but I'd like to see when somebody visits from abroad or heads to a show within perhaps thirty kilometres of my location.

[2] I guess you could argue it gets the US area code, US metro code and zipcode correct as none of them apply here.

[3] The implementation actually considers the mode, and not the average, in calculating "home" locations. It makes it less prone to errors when people only report long distance changes, because the clustering isn't so obvious. If more people hosted a complete hCard, we wouldn't even need to calculate this.

1.1.3 Your proximity alert

I've hacked support for relative proximities in to our Haskell tool, but *upoints* could be used as the basis to implement something similar in Python. Taking Seemant, who lives in Boston, Ma., as an example as it is his fault I'm playing with Python and geoip *upoints* can tell us:

```
>>> Seemant = point.Point(42, -71)
>>> print("%i kM" % Home.distance(Seemant))
5257 kM
```

We now have to make a decision about the range for the proximity alert given that Seemant lives some five thousand kilometres away. Being that I owe him many beers for taking care of a lot of my Gentoo bugs for me, I should perhaps set the range to be quite low and save myself some money.

Without taking in to account my stinginess it seems that a reasonable target range is the square root of the home-to-home distance. From looking at the events I've tagged to meet up with someone in the past year it seems that all of them fall surprisingly evenly within the square root of the distance we live from each other.

Marginally weighted square roots might be more appropriate in reality because there are some anomalies. For example, I travelled from Kensington to West India Dock after LinuxWorld last year to catch up with friends who live a few minutes up the A1 from my house. The reason being for most of last year our schedules seemed to be stopping us meeting up locally, but even that fell within 1.5 times the square root. Adding in a key to show the last face to face meeting, would probably allow one to assign weighting automatically. Continuing the Seemant example would mean increasing his range significantly, being a BTS and email-only contact.

```
>>> import math
>>> math.sqrt(Home.distance(Seemant))
72.51154203831521
```

If we forget about the anomalies, and just take the square root as being correct I can populate the relationship for Seemant with a 73 kM limit. I'm sure each person involved will have their own idea of what a reasonable limit would be, so that should be user defined.

1.1.4 Conclusions

geoip wasn't, and isn't going to become, a viable way to update the path cross database and until more mobile devices come equipped with *GPS* automated updates just aren't going to be usable. If you want to start claiming those owed beers the answer is to publish your schedule in valid hCalendar, and publish a hCard containing your home location so you get the correct range allowance.

If you think of any good uses for *upoints*, drop me a mail. Cool new uses with attached patches are even better!

1.1.5 Bonus

Having already implemented the basic class and distance method, I figured I may as well add bearing calculation too. It's only 4 lines of code, so why not?

```
>>> print("A heading of %i° will find the beers!" % Home.bearing(Telford))
A heading of 294° will find the beers!
```

1.2 The MERLIN system

1.2.1 Introduction

MERLIN, the Multi-Element Radio Linked Interferometer Network, radio telescope array that is spread throughout central England and Wales. The interesting aspect of MERLIN for our uses is high-quality, minimally diverse location identifiers are available publicly. They make a reasonably useful test case for *upoints* objects across small geographic distances.

According to the official MERLIN documentation the locations of the array elements are:

Name	Latitude	Longitude
Cambridge	52°1006.48N	000°0223.25E
Darnhall	53°0838.40N	002°3245.57W
Defford	52°0527.61N	002°0809.62W
Knocking	52°4640.83N	003°0039.55W
Lovell Telescope	53°1410.50N	002°1825.74W
Mark II	53°1351.62N	002°1834.16W
Pickmere	53°1644.42N	002°2641.98W

1.2.2 Using `Point` objects

We can create a Python dictionary containing the locations of the array very simply:

```
>>> from upoints.point import (Point, KeyedPoints)
>>> from upoints import utils
>>> MERLIN = KeyedPoints({
...     'Cambridge': Point((52, 10, 6.48), (0, 2, 23.25)),
...     'Darnhall': Point((53, 8, 38.4), (-2, -32, -45.57)),
...     'Defford': Point((52, 5, 27.61), (-2, -8, -9.62)),
...     'Knocking': Point((52, 46, 40.83), (-3, -0, -39.55)),
...     'Lovell Telescope': Point((53, 14, 10.5), (-2, -18, -25.74)),
...     'Mark II': Point((53, 13, 51.62), (-2, -18, -34.16)),
...     'Pickmere': Point((53, 16, 44.42), (-2, -26, -41.98)),
... })
```

As a simple smoke test the MERLIN website contains a location page which states the longest baseline in the array is Cambridge to Knocking at 217 kM, and also that the shortest baseline is between the Jodrell Bank site and Pickmere at 11.2 kM. The *Point* object's *distance()* method can calculate these distances for us quite simply:

```
>>> "%.3f kM" % MERLIN['Cambridge'].distance(MERLIN['Knocking'])
'217.312 kM'
>>> "%.3f kM" % MERLIN['Lovell Telescope'].distance(MERLIN['Pickmere'])
'10.322 kM'
>>> "%.3f kM" % MERLIN['Mark II'].distance(MERLIN['Pickmere'])
'10.469 kM'
```

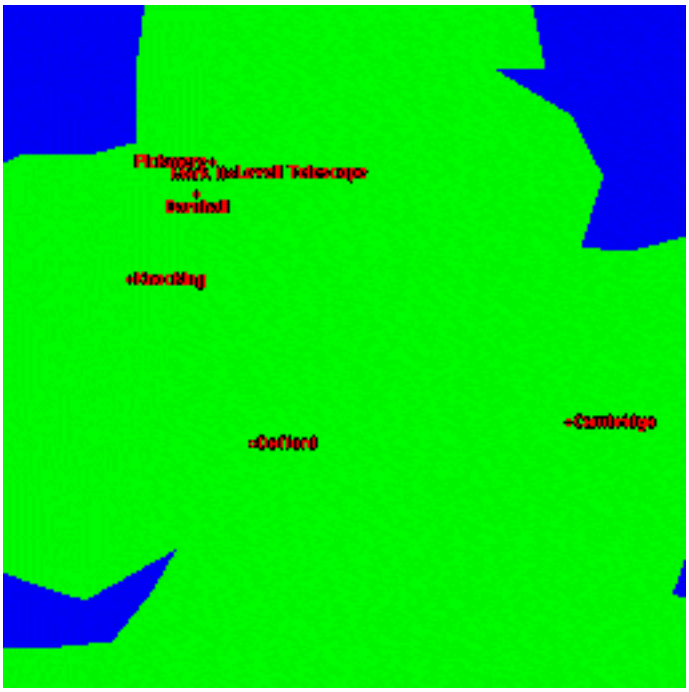
Note: The web page gives the shortest baseline as the distance from Pickmere to Jodrell Bank, but doesn't give a location for Jodrell Bank. However, as can be seen from the example above the two array elements based at Jodrell Bank(Lovell Telescope and the Mark II) are giving a plausible value.

1.2.3 Using `dump_xearth_markers()`

The MERLIN website also contains a layman description page that has a nice map showing the locations of the array elements, we can create a similar image with xplanet or xearth:

```
>>> print("\n".join(utils.dump_xearth_markers(MERLIN)))
52.168467 0.039792 "Cambridge"
53.144000 -2.545992 "Darnhall"
52.091003 -2.136006 "Defford"
52.778008 -3.010986 "Knocking"
53.236250 -2.307150 "Lovell Telescope"
53.231006 -2.309489 "Mark II"
53.279006 -2.444994 "Pickmere"
```

The map on the website contains a few more locations presumably to help the viewer with orientation, but the image below is useful as a good approximation. And, of course, the locations could be supplemented either by hand, or by using one of the other *upoints* supported databases.



1.2.4 Examining local solar time

Imagine the contrived example that we were allowed access to each of the locations and we're hoping to catch the end of an imaginary partial eclipse occurring at 05:45 UTC on 2007-09-20 we can find the best location to view from quite simply. Clearly, the most important factor is whether the Sun will be visible at the given time and this can be calculated very easily:

```
>>> import datetime
>>> for name, rise in MERLIN.sunrise(datetime.date(2007, 9, 20)):
...     if rise > datetime.time(5, 45): continue
...     print(name)
...     print("     - sunrise @ %s UTC" % rise.strftime("%H:%M"))
Cambridge
     - sunrise @ 05:41 UTC
```

This simple code snippet shows us that we should set up our equipment at the Cambridge site, which lucky for me is only a short trip up the road:

```
>>> Home = Point(52.015, -0.221)
>>> print("%i kM" % Home.distance(MERLIN['Cambridge']))
24 kM
```

1.2.5 Comparisons with other `Point`-type objects

In our contrived example above we may wish to travel only if the weather will be warm enough that we're unlikely to freeze to death(that risk is only acceptable for a full eclipse), and we can use the other *upoints* tools to find closest weather station quite easily:

```
>>> from upoints import weather_stations
>>> ICAO_stations_database = urllib.urlopen("http://weather.noaa.gov/data/nsd_cccc.txt
↪")
>>> ICAO_stations = weather_stations.Stations(ICAO_stations_database, "ICAO")
>>> calc_distance = lambda (name, location): MERLIN['Cambridge'].distance(location)
>>> station_id, station_data = sorted(ICAO_stations.items(), key=calc_distance)[0]
>>> print(station_data)
Cambridge (N52.200°; E000.183°)
```

The `calc_distance()` function simply returns the distance from the Cambridge MERLIN station to the provided station, and we use it as the sorting method to discover the closest weather station from the NOAA database. The `station_id` and `station_data` variables are set to the first result from the sorted list of station distances, which thanks to the `calc_distance()` sorting method are the details of the closest weather station.

As we're already using Python we may as well use Python to fetch the weather data for the station using the ever useful pymetar library.

```
>>> report = pymetar.ReportFetcher(station_id).FetchReport()
>>> report_decoded = pymetar.ReportParser().ParseReport(report)
>>> print("%i°C @ %s" % (report_decoded.getTemperatureCelsius(),
...                      report_decoded.getISOTime()))
10°C @ 2007-11-28 19:20:00Z
```

1.3 Cities and cities.py

A colleague pointed me to the GNU miscfiles cities database after I posted geolocation and path cross, suggesting that it would be a useful database to support. Being that it includes five hundred places around the globe, and I already have the database installed, I have to agree.

GNU miscfiles is a package of, well miscellaneous files. It contains, amongst other things a list of world currencies, languages and the file we're looking at today `cities.dat`.

In v1.4.2, the version I have installed, `cities.dat` contains 497 entries. The file is a simple flat Unicode database, with records separated by `//`, a format that would be as well suited to processing with awk as it would with Python.

```
ID          : 315
Type        : City
Population  :
Size        :
Name        : Cambridge
Country    : UK
```

```
Region     : England
Location    : Earth
Longitude  : 0.1
Latitude   : 52.25
Elevation  :
Date        : 19961207
Entered-By  : Rob.Hooft@EMBL-Heidelberg.DE
```

You don't need to hand process the data though, I've added *cities* to the upoints tarball that takes care of importing the data. When you import the entries with *import_locations()* it returns a dictionary of *City* objects that are children of the *Trigpoint* objects defined for Trigpointing and point.py

On my Gentoo desktop the cities database is installed as /usr/share/misc/cities.dat, and can be imported as simply as:

```
>>> from upoints import cities
>>> Cities = cities.Cities(open("/usr/share/misc/cities.dat"))
```

And the imported database can be used in a variety of ways:

```
>>> print("%i cities" % len(Cities))
497 cities
>>> print("Cities larger with more than 8 million people")
Cities larger with more than 8 million people
>>> for city in Cities:
...     if city.population > 8000000:
...         print("  %s - %s" % (city.name, city.population))
   Bombay - 8243405
   Jakarta - 9200000
   Moskwa - 8769000
   Sao Paolo - 10063110
   Tokyo - 8354615
   Mexico - 8831079
>>> print("Mountains")
Mountains
>>> for city in Cities:
...     if city.ptype == "Mountain":
...         print("  %s" % city.name)
   Aconcagua
   Popocatepetl
```

You can recreate the database as a smoke test using the following:

```
>>> f = open("cities.dat", "w")
>>> f.write("\n//\n".join(map(str, Cities)))
>>> f.close()
```

unfortunately the files aren't simply comparable using **diff** because of some unusual formatting in the original file, but visually scanning over the **diff -w** output to ignore the whitespace changes shows that we have a correct export.

The *City* class inherits *Trigpoint* which in turn inherits *Point*, and therefore has all the same methods they do. This allows you to calculate distances and bearings between the class:*~upoints.cities.City* objects or any other derivative object of the parent classes. For example, you could use the *dump_xearth_markers()* function:

```
>>> from upoints.utils import dump_xearth_markers
>>> scottish_markers = dict((x.identifier, x) for x in Cities
...                         if x.region == "Scotland")
>>> print("\n".join(dump_xearth_markers(scottish_markers, "name")))
```

```
57.150000 -2.083000 "Aberdeen" # 1
55.950000 -3.183000 "Edinburgh" # 83
55.867000 -4.267000 "Glasgow" # 92
```

Take a look at the Sphinx generated documentation that is included in the tarball to see what can be done.

1.4 Pythons on a plane

In what is probably the final spin-off from geolocation and path cross we'll be using the *upoints* modules to work with airport locations. This can be useful if you'd like to calculate how far you've travelled in a certain period, or just as a large database for calculating rough distances between other places using the closest airports as locations because of their abundance.

NOAA publishes an enormous amount of world weather information, and often it is keyed to airport location's weather stations. Unlike many of the commercial weather data companies NOAA publish their data in clean, well defined formats, and along with the weather data they also publish extensive location data for the weather stations they monitor. And many thanks to them, because we can use their databases to populate our local geolocation databases.

```
>>> from upoints import (point, weather_stations)
>>> WMO_stations_database = urllib.urlopen("http://weather.noaa.gov/data/nsd_bbsss.txt
↪")
>>> WMO_stations = weather_stations.Stations(WMO_stations_database)
```

The above snippet will import the WMO identifier keyed database available from the meteorological station location information page. They also provide a database keyed with ICAO identifiers, which can also be imported:

```
>>> ICAO_stations_database = urllib.urlopen("http://weather.noaa.gov/data/nsd_cccc.txt
↪")
>>> ICAO_stations = weather_stations.Stations(ICAO_stations_database, "ICAO")
```

The WMO indexed database contains 11548 entries and the ICAO keyed database contains 6611 entries as of 2007-05-30. Unfortunately, the WMO database isn't a superset of the ICAO data so you either have to choose one, work with duplicates or import both and filter the duplicates.

Another thing to consider because of the size of the database is whether you need to operate on all the entries at once. Maybe you only want to work with entries in the UK:

```
>>> UK_locations = dict(x for x in ICAO_stations.items()
...                     if x[1].country == "United Kingdom")
```

Let us imagine for a minute that next month you're flying from London Luton to our office in Toulouse, then dropping by Birmingham for GUADEC, and returning to Stansted. If we assume that the planes fly directly along Great Circles and don't get stuck in holding patterns waiting to land then we can calculate the distance for the whole journey quite easily.

```
>>> Europe = dict(x for x in ICAO_stations.items() if x[1].wmo == 6)
>>> del(ICAO_stations)
>>> print(len(Europe))
1130
```

First we can see that the trip is entirely based in Europe, and according to the station location page all the European stations are located within WMO region 6. If we only work with the region 6 locations then our operating database need only contain 1130 entries, and if we wished we could release the full database containing 10000 entries we don't

need from memory using code similar to the snippet above[1].

```
>>> Trip = point.Points([Europe[i] for i in ('EGGW', 'LFBO', 'EGBB', 'EGSS')])
>>> legs = list(Trip.inverse())

>>> print("%i legs" % (len(Trip) - 1))
3 legs
>>> for i in range(len(Trip) - 1):
...     print("  * %s to %s" % (Trip[i].name, Trip[i+1].name))
...     print("    - %i kilometres on a bearing of %i degrees" % (legs[i][1],␣
↪legs[i][0]))
  * Luton Airport  to Toulouse / Blagnac
    - 923 kilometres on a bearing of 171 degrees
  * Toulouse / Blagnac to Birmingham / Airport
    - 1006 kilometres on a bearing of 347 degrees
  * Birmingham / Airport to Stansted Airport
    - 148 kilometres on a bearing of 114 degrees
>>> print("For a total of %i kilometres" % sum(i[1] for i in legs))
For a total of 2078 kilometres
```

The *Station* class inherits from *Trigpoint* and as such you can use the functions and methods defined for it with `Station` objects. You could, for example, create a nice graphical view of your trip with xplanet:

```
>>> Trip = dict(zip(("2007-06-29", "2007-06-30", "2007-07-12",
...                  "2007-07-14"),
...                 Trip))
>>> f = open("trip.txt", "w")
>>> from upoints import utils
>>> f.write("\n".join(utils.dump_xearth_markers(Trip, "name")))
>>> f.close()
```

The code above will create a file named `trip.txt` that can be used with xplanet or xearth. It actually produces a reasonably accurate, and quite useful graphical representation of a trip. An example of the output with xplanet can be seen on the right.

If you'd prefer to see locations marked up with dates, perhaps as an aid to your own path cross suite, simply don't set the `name` parameter in your call to *dump_xearth_markers()*. Also, as the function only requires a dictionary of *Trigpoint*-style objects you could apply `filter()` and `map()` expressions to the objects to generate your own labels for the markers.

[1] I've personally taken to creating and using `cPickle` dumps of the database, where each WMO region is stored in a separate file. If you do this you end up with some interesting results including the 123 locations from the Antarctic, and the 8 obviously classifiable locations missing an WMO region in the data file. I personally found it quite interesting that the list of entries by region is Europe(30%), Asia(30%), North and Central America(12%). I'd expected it be more along the lines of one third Asia and one quarter each for Europe and North America with the rest split reasonably evenly.

There is a wealth of Sphinx generated HTML output in the tarball, including documentation and usage examples. If you still have any questions after reading the documentation, drop me a mail and I'll do my best to answer your questions. Also, I'd love to hear from you if come up with any clever uses for for the modules in *upoints*.

1.5 Trigpointing and point.py

One interesting email I received after posting geolocation and path cross asked if the module could be used for quick visualisation when trigpointing. Now that I've found out what trigpointing is I believe it can, and I've added a couple of extra features to make it easier for trigpointers to use the *upoints* module.

Firstly, for those who don't know, trigpointing is the activity of tracking down trigpoints and recording them. I guess you could make a parallel to trainspotting, but with a navigational slant. Some people apparently use *GPS* units to track down the trigpoints, that I'd suggest makes it just hiking with trigpoints as waypoints. And some people, like Robert Johnson who mailed me, prefer to do the navigation with just an ordnance survey map and a compass which in my eyes makes it a little more interesting. Also, a few sites I've found with Google seem to suggest that many trigpointers like to use triangulation, although I suspect some mean trilateration, to travel between trigpoints as a navigational challenge.

Note: Robert tells me that TrigpointingUK is a popular website among trigpointers in the UK. It contains information about many of the trigpoints you can find, such as the one closest to me at Bygrave

---

Anybody who knows me well will attest that that I'm quite the navigation geek, mostly just as a curiosity being that what we're really talking about is just applications of specific branches of math. As such, I actually find the concept of trigpointing by hand quite intriguing. That being said technology is here to assist us, and with that let me introduce *trigpoints* a simple extension over the original `edist.py` script.

```
>>> from upoints import trigpoints
>>> database_location = urllib.urlopen("http://www.haroldstreet.org.uk/waypoints/
↪alltrigs-wgs84.txt")
>>> Trigpoints = trigpoints.Trigpoints(database_location)
>>> print(len(Trigpoints))
6557
```

Thanks to the online database we now have the locations of all the Ordnance Survey trigpoints in an easy to use format – a Python dictionary.

If I'd like to see trigpoints close to me, say within 20kM, and less than 60m above sea level I could tap the following in to my IPython session:

```
>>> Home = trigpoints.point.Point(52.015, -0.221)
>>> for identifier, trigpoint in sorted(Trigpoints.items()):
...     if Home.__eq__(trigpoint, 20) and trigpoint.altitude < 60:
...         print("%s - %s" % (identifier, trigpoint))
500936 - Broom Farm (52°03'57"N, 000°16'53"W alt 37m)
501822 - Crane Hill (52°11'10"N, 000°14'51"W alt 58m)
503750 - Limlow Hill (52°03'31"N, 000°04'20"W alt 59m)
505681 - Sutton (52°06'24"N, 000°11'57"W alt 55m)
```

Or we can display all the trigpoints within a given region. For example, to show trigpoints within the region from 51°52‘15”N, 000°28‘29”W to 52°09‘07”N, 000°01‘52”W.

```
>>> latitude_min = trigpoints.utils.to_dd(51, 52, 15)
>>> longitude_min = trigpoints.utils.to_dd(0, -28, -29)
>>> latitude_max = trigpoints.utils.to_dd(52, 9, 7)
>>> longitude_max = trigpoints.utils.to_dd(0, -1, -52)
>>> for identifier, trigpoint in sorted(Trigpoints.items()):
...     if latitude_min < trigpoint.latitude < latitude_max \
...       and longitude_min < trigpoint.longitude < longitude_max:
...         print("%s - %s" % (identifier, trigpoint))
500928 - Bromley Common (51°52'17"N, 000°06'14"W alt 118m)
500936 - Broom Farm (52°03'57"N, 000°16'53"W alt 37m)
501097 - Bygrave (52°00'38"N, 000°10'24"W alt 97m)
501417 - Cherrys Green (51°55'13"N, 000°01'52"W alt 126m)
501428 - Chicksands North Radio Mast (52°02'46"N, 000°22'17"W alt 62m)
501928 - Croydon Hill (52°07'37"N, 000°05'26"W alt 78m)
502034 - Deacon Hill (51°57'19"N, 000°21'46"W alt 173m)
502908 - Hammer Hill Farm (52°04'32"N, 000°24'05"W alt 89m)
503138 - Higham Gobion (51°58'48"N, 000°23'55"W alt 75m)
503750 - Limlow Hill (52°03'31"N, 000°04'20"W alt 59m)
503774 - Little Easthall Farm (51°53'23"N, 000°15'23"W alt 140m)
504024 - Marsh Farm Mh (51°55'24"N, 000°27'39"W alt 152m)
505392 - Sish Lane (51°54'39"N, 000°11'11"W alt 136m)
505681 - Sutton (52°06'24"N, 000°11'57"W alt 55m)
505852 - Therfield (52°01'03"N, 000°03'38"W alt 168m)
506163 - Warden Hill (51°55'20"N, 000°24'53"W alt 195m)
506165 - Warden Tunnel (52°05'15"N, 000°22'30"W alt 84m)
```

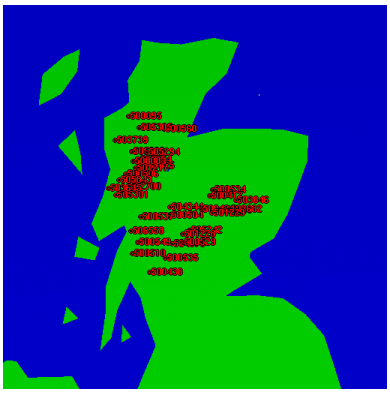
Or we could generate a file to use with xearth that contains all the trigpoints above 1000m above sea level:

```
>>> from upoints.utils import dump_xearth_markers
>>> high_markers = {}
>>> for identifier, trigpoint in Trigpoints.items():
...     if trigpoint.altitude > 1000:
...         high_markers[identifier] = trigpoint
>>> f = open("high_markers.txt", "w")
>>> f.write("\n".join(dump_xearth_markers(high_markers)))
>>> f.close()
```

Now we can use xearth, or xplanet, to visualise the trigpoints that are higher than 1000m. If you start xearth with the command **`xearth -pos "fixed 57 -4" -mag 25 -noroot -markerfile high_markers.txt`** you will see an image similar to the one on the right.

You could, of course, use *`dump_xearth_markers()`* to dump the entire trigpoint database, but with over 6000 locations the result is just going to be a sea of blurred text when rendered.

And it is possible to fold the generation of the `high_markers` dictionary in to a single operation using lambda expressions and `filter()` such as:



```
>>> high_markers = dict(filter(lambda x: x[1].altitude > 1000,
...                            Trigpoints.items()))
```

However, you opinion on whether this is cleaner or not depends a lot on your background. If only you could run `filter()` on a dictionary directly, this would definitely be the better solution. I'm going to continue using the unrolled version on this page because it seems more people are comfortable with them in spite of me favouring the `filter()` and `lambda()` version, but it is just a matter of taste and yours may vary.

Using *`trigpoints`* you could generate marker file for locations with an altitude of between 900m and 910m using their location names as labels.

```
>>> display_markers = {}
>>> for identifier, trigpoint in Trigpoints.items():
...     if 900 < trigpoint.altitude < 910:
...         display_markers[identifier] = trigpoint
>>> f = open("display_markers.txt", "w")
>>> f.write("\n".join(dump_xearth_markers(display_markers,
...                                       "name")))
>>> f.close()
```

The result of how that query could be shown with xplanet can be found to the right.

The *`Trigpoint`* class inherits from the *`Point`* class, and therefore has all the same methods it does. You can calculate distances and bearings between trigpoints. I suggest reading the HTML files generated by Sphinx that are included in the tarball to see how it all works, including some more examples.

Note: And on a slight tangent, in my mind one of the best reasons for using Python is now evident, Nokia provide

Python builds for some of their "smartphone" handsets. This means it is possible to use *trigpoints* on the move using only the mobile phone in your pocket, and it makes for a fun diversion from Snake 3D. Even as a simple database it can be surprisingly useful, especially given the difficulty of finding the minuscule trigpoint symbol on Ordnance Survey's Explorer series maps.

We're on a journey now, so if you can think of any cool uses for any of the classes and functions in the *upoints* tarball drop me a mail.

1.6 xearth and path cross

As a final comment in the geolocation and path cross entry I wrote:

> If you think of any good uses for *upoints*, drop me a mail. Cool new uses with attached patches are even better!

The first chunk of feedback I received was from a co-worker, Kelly Turner, who asked how difficult it would be to use *upoints* with xearth's marker files. The answer is not too difficult, not too difficult at all.

As a little background the reason for wanting access to the data in marker files is that our internal contact database allows us to export a file containing a subset of our contact's known current locations[1]. The original reasoning behind the feature was to allow simple visualisation of a team's members, for example to quickly locate somebody who is close to a customer's site.

I've reworked the original `edist.py` script in to something a little more generic, and it also now includes a new module, *xearth*, that can import locations from an xearth marker file. Unfortunately, I can't use an example generated from our system because I don't have permission to publish the data but I'll give an example of its usage with a public file:

```
>>> from upoints import xearth
>>> earth_markers = urllib.urlopen("http://xplanet.sourceforge.net/Extras/earth-
↪markers-schaumann")
>>> markers = xearth.Xearths(earth_markers)
>>> print(repr(markers['Cairo']))
Xearth(30.05, 31.25, 'Egypt')
>>> print(markers['Warsaw'])
Poland (N52.250°; E021.000°)
```

There are plenty of comments in the *xearth* file, but what you get from the *Xearths* is a dictionary with the location name as a key, and value consisting of a tuple of a *Point* object and any associated comments from the source file.

You can use all the methods, such as *distance()* and *bearing()*, that are defined in the *Point* class on *Xearth* objects.

```
>>> print("Suva to Tokyo is %i kM" % markers['Suva'].distance(markers['Tokyo']))
Suva to Tokyo is 7253 kM
>>> print("Vienna to Brussels on %i°" % markers['Vienna'].bearing(markers['Brussels
↪']))
Vienna to Brussels on 293°
```

With the original purpose of the marker file export feature being finding people local to a customer it would be nice to use *xearth* to do the same in a more programmatic way, and of course that is possible too.

[1] All thanks to Simon Woods according to the source code repository, so a big thanks to him!

```
>>> Customer = xearth.point.Point(52.015, -0.221)
>>> for marker in markers:
...       distance = Customer.distance(markers[marker])
...       if distance < 300:
...           print("%i kM - %s, %s" % (distance, marker, markers[marker]))
57 kM - London, United Kingdom (N51.500°; W000.170°)
```

Imagining for a second the customer lives in my house, the only marker within 300 kilometres of me in the city marker file we've imported is London.

I'll end this entry with similar text to that which created it: If you think of any good uses for *upoints*, drop me a mail. Cool new uses with attached patches are even better!

1.7 edist

1.7.1 Simple command line coordinate processing

1.7.2 SYNOPSIS

edist [option]... <command> <location...>

1.7.3 DESCRIPTION

edist operates on one, or more, locations specified in various formats. For example, a location string of "52.015;-0.221" would be interpreted as 52.015 degrees North by 0.221 degrees West, as would "52d0m54s N 000d13m15s W". Positive values can be specified with a "+" prefix, but it isn't required.

It is possible to use Maidenhead locators, such as "IO92" or "IO92va", for users who are accustomed to working with them.

Users can maintain a local configuration file that lists locations with assigned names, and then use the names on the command line. This makes command lines much easier to read, and also makes reusing locations at a later date simpler. See *CONFIGURATION FILE*.

1.7.4 OPTIONS

--version Show the version and exit.

-v, –verbose / –quiet Change verbosity level of output.

--config <file> Config file to read custom locations from.

--csv-file <file> CSV file (gpsbabel format) to read route/locations from.

-o, --format <format> Produce output in dms, dm or dd format.

-u <units>, --units <units> Display distances in kilometres, statute miles or nautical miles.

-l, --location <location> Location to operate on.

-h, --help Show this message and exit.

1.7.5 COMMANDS

bearing

Calculate the initial bearing between locations bearing

g, –string Display named bearings.

-h, --help Show this message and exit.

destination

Calculate destination from locations.

-l <accuracy>, --locator <accuracy> Accuracy of Maidenhead locator output.

-h, --help Show this message and exit.

display

Pretty print the locations.

-l <accuracy>, --locator <accuracy> Accuracy of Maidenhead locator output.

-h, --help Show this message and exit.

distance

Calculate distance between locations.

-h, --help Show this message and exit.

final-bearing

Calculate final bearing between locations.

-g, --string Display named bearings.

-h, --help Show this message and exit.

flight-plan

Calculate flight plan for locations.

-s <speed>, --speed <speed> Speed to calculate elapsed time.

-t <format>, --time <format> Display time in hours, minutes or seconds.

-h, --help Show this message and exit.

range

Check locations are within a given range.

sunrise

Calculate the sunrise time for locations.

-h, --help Show this message and exit.

sunset

Calculate the sunset time for locations.

-h, --help Show this message and exit.

1.7.6 CONFIGURATION FILE

The configuration file, by default `~/.edist.conf`, is a simple ``INI` format file, with sections headers defining the name of the location and their data defining the actual position. You can define locations by either their latitude and longitude, or with a Maidenhead locator string. Any options that aren't handled will simply ignored. For example:

```
[Home]
latitude = 52.015
longitude = -0.221

[Cambridge]
latitude = 52.200
longitude = 0.183

[Pin]
locator = IO92
```

With the above configuration file one could find the distance from `Home` to `Cambridge` using `edist -l Home -l Cambridge distance`.

1.7.7 BUGS

None known.

1.7.8 AUTHOR

Written by James Rowe

1.7.9 RESOURCES

Home page: https://github.com/JNRowe/upoints

1.7.10 COPYING

1.8 API documentation

upoints - Modules for working with points on Earth

`upoints` is a collection of GPL v3 licensed modules for working with points on Earth, or other near spherical objects. It allows you to calculate the distance and bearings between points, mangle xearth/xplanet data files, work with online UK trigpoint databases, NOAA's weather station database and other such location databases.

The *upoints.point* module is the simplest interface available, and is mainly useful as a naïve object for simple calculation and subclassing for specific usage. An example of how to use it follows:

```
>>> from upoints import point
>>> Home = point.Point(52.015, -0.221)
>>> Telford = point.Point(52.6333, -2.5000)
>>> int(Home.distance(Telford))
169
>>> int(Home.bearing(Telford))
294
>>> int(Home.final_bearing(Telford))
293
>>> import datetime
>>> Home.sun_events(datetime.date(2007, 6, 28))
(datetime.time(3, 42), datetime.time(20, 24))
>>> Home.sunrise(datetime.date(2007, 6, 28))
datetime.time(3, 42)
>>> Home.sunset(datetime.date(2007, 6, 28))
datetime.time(20, 24)
```

1.8.1 Contents

baken

baken - Imports baken data files.

class `upoints.baken.`**Baken**(*latitude*, *longitude*, *antenna=None*, *direction=None*, *frequency=None*, *height=None*, *locator=None*, *mode=None*, *operator=None*, *power=None*, *qth=None*)

Bases: *upoints.point.Point*

Class for representing location from baken data files.

New in version 0.4.0.

Initialise a new `Baken` object.

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **antenna** (*str*) – Location's antenna type
- **direction** (*tuple of int*) – Antenna's direction
- **frequency** (*float*) – Transmitter's frequency
- **height** (*float*) – Antenna's height
- **locator** (*str*) – Location's Maidenhead locator string

- **mode** (*str*) – Transmitter's mode
- **operator** (*tuple of str*) – Transmitter's operator
- **power** (*float*) – Transmitter's power
- **qth** (*str*) – Location's qth

Raises LookupError – No position data to use

class upoints.baken.**Bakens**(*baken_file=None*)

Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Baken* objects.

New in version 0.5.1.

Initialise a new *Bakens* object.

import_locations(*baken_file*)

Import baken data files.

import_locations() returns a dictionary with keys containing the section title, and values consisting of a collection *Baken* objects.

It expects data files in the format used by the baken amateur radio package, which is Windows INI style files such as:

```
[Abeche, Chad]
latitude=14.460000
longitude=20.680000
height=0.000000

[GB3BUX]
frequency=50.000
locator=IO93BF
power=25 TX
antenna=2 x Turnstile
height=460
mode=A1A
```

The reader uses the configparser module, so should be reasonably robust against encodings and such. The above file processed by import_locations() will return the following dict object:

```
{"Abeche, Chad": Baken(14.460, 20.680, None, None, None, 0.000,
                       None, None, None, None, None),
 "GB3BUX": : Baken(None, None, "2 x Turnstile", None, 50.000,
                   460.000, "IO93BF", "A1A", None, 25, None)}
```

Args:: baken_file (iter): Baken data to read

Returns Named locations and their associated values

Return type dict

cellid

cellid - Imports OpenCellID data files.

class upoints.cellid.**Cell**(*ident*, *latitude*, *longitude*, *mcc*, *mnc*, *lac*, *cellid*, *crange*, *samples*, *created*, *updated*)

Bases: *upoints.point.Point*

Class for representing a cellular cite from OpenCellID.org.

New in version 0.11.0.

Initialise a new `Cell` object.

Parameters

- **ident** (*int*) – OpenCellID database identifier
- **latitude** (*float*) – Cell's latitude
- **longitude** (*float*) – Cell's longitude
- **mcc** (*int*) – Cell's country code
- **mnc** (*int*) – Cell's network code
- **lac** (*int*) – Cell's local area code
- **cellid** (*int*) – Cell's identifier
- **crange** (*int*) – Cell's range
- **samples** (*int*) – Number of samples for the cell
- **created** (*datetime.datetime*) – Date the cell was first entered
- **updated** (*datetime.datetime*) – Date of the last update

class upoints.cellid.**Cells**(*cells_file=None*)

Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Cell* objects.

New in version 0.11.0.

Initialise a new `Cells` object.

import_locations(*cells_file*)

Parse OpenCellID.org data files.

`import_locations()` returns a dictionary with keys containing the OpenCellID.org database identifier, and values consisting of a `Cell` objects.

It expects cell files in the following format:

```
22747,52.0438995361328,-0.2246370017529,234,33,2319,647,0,1,
2008-04-05 21:32:40,2008-04-05 21:32:40
22995,52.3305015563965,-0.2255620062351,234,10,20566,4068,0,1,
2008-04-05 21:32:59,2008-04-05 21:32:59
23008,52.3506011962891,-0.2234109938145,234,10,10566,4068,0,1,
2008-04-05 21:32:59,2008-04-05 21:32:59
```

The above file processed by `import_locations()` will return the following `dict` object:

```
{23008: Cell(23008, 52.3506011963, -0.223410993814, 234, 10, 10566,
             4068, 0, 1, datetime.datetime(2008, 4, 5, 21, 32, 59),
             datetime.datetime(2008, 4, 5, 21, 32, 59)),
 22747: Cell(22747, 52.0438995361, -0.224637001753, 234, 33, 2319,
             647, 0, 1, datetime.datetime(2008, 4, 5, 21, 32, 40),
             datetime.datetime(2008, 4, 5, 21, 32, 40)),
```

```
 22995: Cell(22995, 52.3305015564, -0.225562006235, 234, 10, 20566,
             4068, 0, 1, datetime.datetime(2008, 4, 5, 21, 32, 59),
             datetime.datetime(2008, 4, 5, 21, 32, 59))}
```

Parameters **cells_file** (*iter*) – Cell data to read

Returns Cell data with their associated database identifier

Return type dict

cities

cities - Imports GNU miscfiles cities data files.

class upoints.cities.**Cities**(*data=None*)

Bases: *upoints.point.Points*

Class for representing a group of *City* objects.

New in version 0.5.1.

Initialise a new Cities object.

import_locations(*data*)

Parse GNU miscfiles cities data files.

import_locations() returns a list containing *City* objects.

It expects data files in the same format that GNU miscfiles provides, that is:

```
ID          : 1
Type        : City
Population  : 210700
Size        :
Name        : Aberdeen
 Country    : UK
 Region     : Scotland
Location    : Earth
 Longitude  : -2.083
 Latitude   :   57.150
 Elevation  :
Date        : 19961206
Entered-By  : Rob.Hooft@EMBL-Heidelberg.DE
//
ID          : 2
Type        : City
Population  : 1950000
Size        :
Name        : Abidjan
 Country    : Ivory Coast
 Region     :
Location    : Earth
 Longitude  : -3.867
 Latitude   :    5.333
 Elevation  :
Date        : 19961206
Entered-By  : Rob.Hooft@EMBL-Heidelberg.DE
```

When processed by import_locations() will return list object in the following style:

```
[City(1, "City", 210700, None, "Aberdeen", "UK", "Scotland",
      "Earth", -2.083, 57.15, None, (1996, 12, 6, 0, 0, 0, 4,
      341, -1), "Rob.Hooft@EMBL-Heidelberg.DE"),
 City(2, "City", 1950000, None, "Abidjan", "Ivory Coast", "",
      "Earth", -3.867, 5.333, None, (1996, 12, 6, 0, 0, 0, 4,
      341, -1), "Rob.Hooft@EMBL-Heidelberg.DE")])
```

Parameters **data** (*iter*) – NOAA (National Oceanographic and Atmospheric Administration) station data to read

Returns Places as `City` objects

Return type list

Raises `TypeError` – Invalid value for data

class upoints.cities.**City**(*identifier*, *name*, *ptype*, *region*, *country*, *location*, *population*, *size*, *latitude*, *longitude*, *altitude*, *date*, *entered*)

Bases: *upoints.trigpoints.Trigpoint*

Class for representing an entry from the GNU miscfiles cities data file.

New in version 0.2.0.

Initialise a new `City` object.

Parameters

- **identifier** (*int*) – Numeric identifier for object
- **name** (*str*) – Place name
- **ptype** (*str*) – Type of place
- **region** (*str*) – Region place can be found
- **country** (*str*) – Country name place can be found
- **location** (*str*) – Body place can be found
- **population** (*int*) – Place's population
- **size** (*int*) – Place's area
- **latitude** (*float*) – Station's latitude
- **longitude** (*float*) – Station's longitude
- **altitude** (*int*) – Station's elevation
- **date** (*time.struct_time*) – Date the entry was added
- **entered** (*str*) – Entry's author

upoints.cities.**TEMPLATE = 'ID : %s\nType : %s\nPopulation : %s\nSize : %s\nName : %s\n Country : %s\n Region : '**

GNU miscfiles cities.dat template

edist

class upoints.edist.**LocationsError**(*function=None*, *data=None*)

Bases: `exceptions.ValueError`

Error object for data parsing error.

New in version 0.6.0.

function
Function where error is raised.

data
Location number and data

Initialise a new `LocationsError` object.

Parameters

- **function** (*str*) – Function where error is raised
- **data** (*tuple*) – Location number and data

class `upoints.edist.`**NumberedPoint**(*latitude*, *longitude*, *name*, *units='km'*)
Bases: *upoints.point.Point*

Class for representing locations from command line.

See also:

upoints.point.Point

New in version 0.6.0.

name
A name for location, or its position on the command line

units
Unit type to be used for distances

Initialise a new `NumberedPoint` object.

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **name** (*str*) – Location's name or command line position
- **units** (*str*) – Unit type to be used for distances

class `upoints.edist.`**NumberedPoints**(*locations=None*, *format='dd'*, *verbose=True*, *config_locations=None*, *units='km'*)
Bases: *upoints.point.Points*

Class for representing a group of *NumberedPoint* objects.

New in version 0.6.0.

Initialise a new `NumberedPoints` object.

Parameters

- **locations** (*list of str*) – Location identifiers
- **format** (*str*) – Coordinate formatting system to use
- **verbose** (*bool*) – Whether to generate verbose output
- **config_locations** (*dict*) – Locations imported from user's config file
- **units** (*str*) – Unit type to be used for distances

bearing(*mode*, *string*)
Calculate bearing/final bearing between locations.

Parameters

- **mode** (*str*) – Type of bearing to calculate
- **string** (*bool*) – Use named directions

destination (*distance*, *bearing*, *locator*)
Calculate destination locations for given distance and bearings.

Parameters

- **distance** (*float*) – Distance to travel
- **bearing** (*float*) – Direction of travel
- **locator** (*str*) – Accuracy of Maidenhead locator output

display (*locator*)
Pretty print locations.

Parameters **locator** (*str*) – Accuracy of Maidenhead locator output

distance ()
Calculate distances between locations.

flight_plan (*speed*, *time*)
Output the flight plan corresponding to the given locations.

Todo

Description

Parameters

- **speed** (*float*) – Speed to use for elapsed time calculation
- **time** (*str*) – Time unit to use for output

import_locations (*locations*, *config_locations*)
Import locations from arguments.

Parameters

- **locations** (*list of str*) – Location identifiers
- **config_locations** (*dict*) – Locations imported from user's config
- **file** –

range (*distance*)
Test whether locations are within a given range of the first.

Parameters **distance** (*float*) – Distance to test location is within

sun_events (*mode*)
Calculate sunrise/sunset times for locations.

Parameters **mode** (*str*) – Sun event to display

upoints.edist.**read_locations** (*filename*)
Pull locations from a user's config file.

Parameters **filename** (*str*) – Config file to parse

Returns List of locations from config file

Return type dict

upoints.edist.**read_csv**(*filename*)
Pull locations from a user's CSV file.

Read gpsbabel's CSV output format

Parameters **filename** (*str*) – CSV file to parse

Returns List of locations as `str` objects

Return type tuple of dict and list

upoints.edist.**main**()
Main script handler.

Returns 0 for success, >1 error code

Return type int

geonames

geonames - Imports geonames.org data files.

class upoints.geonames.**Location**(*geonameid*, *name*, *asciiname*, *alt_names*, *latitude*, *longitude*, *feature_class*, *feature_code*, *country*, *alt_country*, *admin1*, *admin2*, *admin3*, *admin4*, *population*, *altitude*, *gtopo30*, *tzname*, *modified_date*, *timezone=None*)
Bases: *upoints.trigpoints.Trigpoint*

Class for representing a location from a geonames.org data file.

All country codes are specified with their two letter ISO-3166 country code.

New in version 0.3.0.

Variables **__TIMEZONES** – `dateutil.gettz` cache to speed up generation

Initialise a new `Location` object.

Parameters

- **geonameid** (*int*) – ID of record in geonames database
- **name** (*unicode*) – Name of geographical location
- **asciiname** (*str*) – Name of geographical location in ASCII encoding
- **alt_names** (*list of unicode*) – Alternate names for the location
- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **feature_class** (*str*) – Location's type
- **feature_code** (*str*) – Location's code
- **country** (*str*) – Location's country
- **alt_country** (*str*) – Alternate country codes for location
- **admin1** (*str*) – FIPS code (subject to change to ISO code), ISO code for the US and CH
- **admin2** (*str*) – Code for the second administrative division, a county in the US
- **admin3** (*str*) – Code for third level administrative division

- **admin4** (*str*) – Code for fourth level administrative division
- **population** (*int*) – Location's population, if applicable
- **altitude** (*int*) – Location's elevation
- **gtopo30** (*int*) – Average elevation of 900 square metre region, if available
- **tzname** (*str*) – The timezone identifier using POSIX timezone names
- **modified_date** (*datetime.date*) – Location's last modification date in the geonames databases
- **timezone** (*int*) – The non-DST timezone offset from UTC in minutes

class upoints.geonames.**Locations**(*data=None*, *tzfile=None*)

Bases: *upoints.point.Points*

Class for representing a group of *Location* objects.

New in version 0.5.1.

Initialise a new `Locations` object.

import_locations(*data*)

Parse geonames.org country database exports.

`import_locations()` returns a list of `trigpoints.Trigpoint` objects generated from the data exported by geonames.org.

It expects data files in the following tab separated format:

```
2633441     Afon Wyre       Afon Wyre       River Wayrai,River Wyrai,Wyre
↪52.3166667      -4.1666667      H       STM     GB      GB      00
↪                   0               -9999   Europe/London   1994-01-13
2633442     Wyre    Wyre    Viera   59.1166667      -2.9666667      T
↪ISL     GB      GB      V9                                      0
↪    Europe/London   2004-09-24
2633443     Wraysbury       Wraysbury       Wyrardisbury    51.45   -0.55   P
↪       PPL     GB              P9                                      0
↪  28      Europe/London   2006-08-21
```

Files containing the data in this format can be downloaded from the geonames.org site in their database export page.

Files downloaded from the geonames site when processed by `import_locations()` will return `list` objects of the following style:

```
[Location(2633441, "Afon Wyre", "Afon Wyre",
          ['River Wayrai', 'River Wyrai', 'Wyre'],
          52.3166667, -4.1666667, "H", "STM", "GB", ['GB'], "00",
          None, None, None, 0, None, -9999, "Europe/London",
          datetime.date(1994, 1, 13)),
 Location(2633442, "Wyre", "Wyre", ['Viera'], 59.1166667,
          -2.9666667, "T", "ISL", "GB", ['GB'], "V9", None, None,
          None, 0, None, 1, "Europe/London",
          datetime.date(2004, 9, 24)),
 Location(2633443, "Wraysbury", "Wraysbury", ['Wyrardisbury'],
          51.45, -0.55, "P", "PPL", "GB", None, "P9", None, None,
          None, 0, None, 28, "Europe/London",
          datetime.date(2006, 8, 21))]
```

Parameters **data** (*iter*) – geonames.org locations data to read

Returns geonames.org identifiers with *Location* objects

Return type list

Raises FileFormatError – Unknown file format

import_timezones_file(*data*)

Parse geonames.org timezone exports.

import_timezones_file() returns a dictionary with keys containing the timezone identifier, and values consisting of a UTC offset and UTC offset during daylight savings time in minutes.

It expects data files in the following format:

```
Europe/Andorra      1.0     2.0
Asia/Dubai  4.0     4.0
Asia/Kabul  4.5     4.5
```

Files containing the data in this format can be downloaded from the geonames site in their database export page

Files downloaded from the geonames site when processed by import_timezones_file() will return dict object of the following style:

```
{"Europe/Andorra": (60, 120),
 "Asia/Dubai": (240, 240),
 "Asia/Kabul": (270, 270)}
```

Parameters **data** (*iter*) – geonames.org timezones data to read

Returns geonames.org timezone identifiers with their UTC offsets

Return type list

Raises FileFormatError – Unknown file format

upoints.geonames.**tz** = None

dateutil module reference if available

gpx

gpx - Imports GPS eXchange format data files.

class upoints.gpx.**Routepoint**(*latitude*, *longitude*, *name=None*, *description=None*, *elevation=None*, *time=None*)

Bases: upoints.gpx._GpxElem

Class for representing a rtepoint element from GPX data files.

New in version 0.10.0.

See also:

_GpxElem

Initialise a new _GpxElem object.

Parameters

- **latitude** (*float*) – Element's latitude
- **longitude** (*float*) – Element's longitude

- **name** (*str*) – Name for Element
- **description** (*str*) – Element's description
- **elevation** (*float*) – Element's elevation
- **time** (utils.Timestamp) – Time the data was generated

class upoints.gpx.**Routepoints**(*gpx_file=None*, *metadata=None*)

Bases: upoints.gpx._SegWrap

Class for representing a group of *Routepoint* objects.

New in version 0.10.0.

Initialise a new _SegWrap object.

export_gpx_file()

Generate GPX element tree from *Routepoints*.

Returns

GPX element tree depicting *Routepoints* objects

Return type etree.ElementTree

import_locations(*gpx_file*)

Import GPX data files.

import_locations() returns a series of lists representing track segments with *Routepoint* objects as contents.

It expects data files in GPX format, as specified in GPX 1.1 Schema Documentation, which is XML such as:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<gpx version="1.1" creator="upoints/0.12.2"
xmlns="http://www.topografix.com/GPX/1/1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.
↪com/GPX/1/1/gpx.xsd">
  <rte>
    <rtept lat="52.015" lon="-0.221">
      <name>Home</name>
      <desc>My place</desc>
    </rtept>
    <rtept lat="52.167" lon="0.390">
      <name>MSR</name>
      <desc>Microsoft Research, Cambridge</desc>
    </rtept>
  </rte>
</gpx>
```

The reader uses the ElementTree module, so should be very fast when importing data. The above file processed by import_locations() will return the following list object:

```
[[Routepoint(52.015, -0.221, "Home", "My place"),
  Routepoint(52.167, 0.390, "MSR", "Microsoft Research, Cambridge")], ]
```

Parameters **gpx_file** (*iter*) – GPX data to read

Returns Locations with optional comments

Return type list

class upoints.gpx.**Trackpoint**(*latitude*, *longitude*, *name=None*, *description=None*, *elevation=None*, *time=None*)

Bases: upoints.gpx._GpxElem

Class for representing a trackpoint element from GPX data files.

New in version 0.10.0.

See also:

_GpxElem

Initialise a new _GpxElem object.

Parameters

- **latitude** (*float*) – Element's latitude
- **longitude** (*float*) – Element's longitude
- **name** (*str*) – Name for Element
- **description** (*str*) – Element's description
- **elevation** (*float*) – Element's elevation
- **time** (utils.Timestamp) – Time the data was generated

class upoints.gpx.**Trackpoints**(*gpx_file=None*, *metadata=None*)

Bases: upoints.gpx._SegWrap

Class for representing a group of *Trackpoint* objects.

New in version 0.10.0.

Initialise a new _SegWrap object.

export_gpx_file()

Generate GPX element tree from Trackpoints.

Returns

GPX element tree depicting Trackpoints objects

Return type etree.ElementTree

import_locations(*gpx_file*)

Import GPX data files.

import_locations() returns a series of lists representing track segments with *Trackpoint* objects as contents.

It expects data files in GPX format, as specified in GPX 1.1 Schema Documentation, which is XML such as:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<gpx version="1.1" creator="upoints/0.12.2"
xmlns="http://www.topografix.com/GPX/1/1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.
↪com/GPX/1/1/gpx.xsd">
  <trk>
    <trkseg>
      <trkpt lat="52.015" lon="-0.221">
```

```
        <name>Home</name>
        <desc>My place</desc>
      </trkpt>
      <trkpt lat="52.167" lon="0.390">
        <name>MSR</name>
        <desc>Microsoft Research, Cambridge</desc>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

The reader uses the `ElementTree` module, so should be very fast when importing data. The above file processed by `import_locations()` will return the following `list` object:

```
[[Trackpoint(52.015, -0.221, "Home", "My place"),
  Trackpoint(52.167, 0.390, "MSR", "Microsoft Research, Cambridge")], ]
```

Parameters **gpx_file** (*iter*) – GPX data to read

Returns Locations with optional comments

Return type list

class upoints.gpx.**Waypoint**(*latitude*, *longitude*, *name=None*, *description=None*, *elevation=None*, *time=None*)

Bases: upoints.gpx._GpxElem

Class for representing a waypoint element from GPX data files.

New in version 0.8.0.

See also:

_GpxElem

Initialise a new `_GpxElem` object.

Parameters

- **latitude** (*float*) – Element's latitude
- **longitude** (*float*) – Element's longitude
- **name** (*str*) – Name for Element
- **description** (*str*) – Element's description
- **elevation** (*float*) – Element's elevation
- **time** (utils.Timestamp) – Time the data was generated

class upoints.gpx.**Waypoints**(*gpx_file=None*, *metadata=None*)

Bases: upoints.point.TimedPoints

Class for representing a group of `Waypoint` objects.

New in version 0.8.0.

Initialise a new `Waypoints` object.

export_gpx_file()

Generate GPX element tree from `Waypoints` object.

Returns GPX element tree depicting `Waypoints` object

Return type etree.ElementTree

import_locations(*gpx_file*)

Import GPX data files.

`import_locations()` returns a list with `Waypoint` objects.

It expects data files in GPX format, as specified in GPX 1.1 Schema Documentation, which is XML such as:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<gpx version="1.1" creator="PocketGPSWorld.com"
xmlns="http://www.topografix.com/GPX/1/1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.
↪com/GPX/1/1/gpx.xsd">

  <wpt lat="52.015" lon="-0.221">
    <name>Home</name>
    <desc>My place</desc>
  </wpt>
  <wpt lat="52.167" lon="0.390">
    <name>MSR</name>
    <desc>Microsoft Research, Cambridge</desc>
  </wpt>
</gpx>
```

The reader uses the `ElementTree` module, so should be very fast when importing data. The above file processed by `import_locations()` will return the following `list` object:

```
[Waypoint(52.015, -0.221, "Home", "My place"),
 Waypoint(52.167, 0.390, "MSR", "Microsoft Research, Cambridge")]
```

Parameters **gpx_file** (*iter*) – GPX data to read

Returns Locations with optional comments

Return type list

kml

kml - Imports KML data files.

class upoints.kml.**Placemark**(*latitude*, *longitude*, *altitude=None*, *name=None*, *description=None*)

Bases: *upoints.trigpoints.Trigpoint*

Class for representing a Placemark element from KML data files.

New in version 0.6.0.

Initialise a new `Placemark` object.

Parameters

- **latitude** (*float*) – Placemarks's latitude
- **longitude** (*float*) – Placemark's longitude
- **altitude** (*float*) – Placemark's altitude
- **name** (*str*) – Name for placemark

- **description** (*str*) – Placemark's description

tokml()
Generate a KML Placemark element subtree.

Returns KML Placemark element

Return type etree.Element

class upoints.kml.**Placemarks**(*kml_file=None*)
Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Placemark* objects.

New in version 0.6.0.

Initialise a new Placemarks object.

export_kml_file()
Generate KML element tree from Placemarks.

Returns KML element tree depicting Placemarks

Return type etree.ElementTree

import_locations(*kml_file*)
Import KML data files.

import_locations() returns a dictionary with keys containing the section title, and values consisting of *Placemark* objects.

It expects data files in KML format, as specified in KML Reference, which is XML such as:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
    <Document>
        <Placemark id="Home">
            <name>Home</name>
            <Point>
                <coordinates>-0.221,52.015,60</coordinates>
            </Point>
        </Placemark>
        <Placemark id="Cambridge">
            <name>Cambridge</name>
            <Point>
                <coordinates>0.390,52.167</coordinates>
            </Point>
        </Placemark>
    </Document>
</kml>
```

The reader uses the ElementTree module, so should be very fast when importing data. The above file processed by import_locations() will return the following dict object:

```
{"Home": Placemark(52.015, -0.221, 60),
 "Cambridge": Placemark(52.167, 0.390, None)}
```

Parameters **kml_file** (*iter*) – KML data to read

Returns Named locations with optional comments

Return type dict

nmea

nmea - Imports GPS NMEA-formatted data files.

class upoints.nmea.**Fix**(*time*, *latitude*, *longitude*, *quality*, *satellites*, *dilution*, *altitude*, *geoid_delta*, *dgps_delta=None*, *dgps_station=None*, *mode=None*)

Bases: *upoints.point.Point*

Class for representing a GPS NMEA-formatted system fix.

New in version 0.8.0.

Initialise a new Fix object.

Parameters

- **time** (*datetime.time*) – Time the fix was taken
- **latitude** (*float*) – Fix's latitude
- **longitude** (*float*) – Fix's longitude
- **quality** (*int*) – Mode under which the fix was taken
- **satellites** (*int*) – Number of tracked satellites
- **dilution** (*float*) – Horizontal dilution at reported position
- **altitude** (*float*) – Altitude above MSL
- **geoid_delta** (*float*) – Height of geoid's MSL above WGS84 ellipsoid
- **dgps_delta** (*float*) – Number of seconds since last DGPS sync
- **dgps_station** (*int*) – Identifier of the last synced DGPS station
- **mode** (*str*) – Type of reading

static parse_elements(*elements*)

Parse essential fix's data elements.

Parameters **elements** (*list*) – Data values for fix

Returns Fix object representing data

Return type *Fix*

quality_string()

Return a string version of the quality information.

Returns:: str: Quality information as string

class upoints.nmea.**Locations**(*gpsdata_file=None*)

Bases: *upoints.point.Points*

Class for representing a group of GPS location objects.

New in version 0.8.0.

Initialise a new Locations object.

import_locations(*gpsdata_file*, *checksum=True*)

Import GPS NMEA-formatted data files.

import_locations() returns a list of *Fix* objects representing the fix sentences found in the GPS data.

It expects data files in NMEA 0183 format, as specified in the official documentation, which is ASCII text such as:

```
$GPGSV,6,6,21,32,65,170,35*48
$GPGGA,142058,5308.6414,N,00300.9257,W,1,04,5.6,1374.6,M,34.5,M,,*6B
$GPRMC,142058,A,5308.6414,N,00300.9257,W,109394.7,202.9,191107,5,E,A*2C
$GPGSV,6,1,21,02,76,044,43,03,84,156,49,06,89,116,51,08,60,184,30*7C
$GPGSV,6,2,21,09,87,321,50,10,77,243,44,11,85,016,49,12,89,100,52*7A
$GPGSV,6,3,21,13,70,319,39,14,90,094,52,16,85,130,49,17,88,136,51*7E
$GPGSV,6,4,21,18,57,052,27,24,65,007,34,25,62,142,32,26,88,031,51*73
$GPGSV,6,5,21,27,64,343,33,28,45,231,16,30,84,198,49,31,90,015,52*7C
$GPGSV,6,6,21,32,65,170,34*49
$GPWPL,5200.9000,N,00013.2600,W,HOME*5E
$GPGGA,142100,5200.9000,N,00316.6600,W,1,04,5.6,1000.0,M,34.5,M,,*68
$GPRMC,142100,A,5200.9000,N,00316.6600,W,123142.7,188.1,191107,5,E,A*21
```

The reader only imports the GGA, or GPS fix, sentences currently but future versions will probably support tracks and waypoints. Other than that the data is out of scope for `upoints`.

The above file when processed by `import_locations()` will return the following `list` object:

```
[Fix(datetime.time(14, 20, 58), 53.1440233333, -3.01542833333, 1,
     4, 5.6, 1374.6, 34.5, None, None),
 Position(datetime.time(14, 20, 58), True, 53.1440233333,
          -3.01542833333, 109394.7, 202.9,
          datetime.date(2007, 11, 19), 5.0, 'A'),
 Waypoint(52.015, -0.221, 'Home'),
 Fix(datetime.time(14, 21), 52.015, -3.27766666667, 1, 4, 5.6,
     1000.0, 34.5, None, None),
 Position(datetime.time(14, 21), True, 52.015, -3.27766666667,
          123142.7, 188.1, datetime.date(2007, 11, 19), 5.0, 'A')]
```

Note: The standard is quite specific in that sentences *must* be less than 82 bytes, while it would be nice to add yet another validity check it isn't all that uncommon for devices to break this requirement in their "extensions" to the standard.

Todo

Add optional check for message length, on by default

Parameters

- **gpsdata_file** (*iter*) – NMEA data to read
- **checksum** (*bool*) – Whether checksums should be tested

Returns Series of locations taken from the data

Return type list

class upoints.nmea.**LoranPosition**(*latitude*, *longitude*, *time*, *status*, *mode=None*)

Bases: *upoints.point.Point*

Class for representing a GPS NMEA-formatted Loran-C position.

Initialise a new `LoranPosition` object.

Parameters

- **latitude** (*float*) – Fix's latitude

- **longitude** (*float*) – Fix's longitude
- **time** (*datetime.time*) – Time the fix was taken
- **status** (*bool*) – Whether the data is active
- **mode** (*str*) – Type of reading

mode_string()
Return a string version of the reading mode information.

Returns Quality information as string

Return type str

static parse_elements(*elements*)
Parse position data elements.

Parameters **elements** (*list*) – Data values for fix

Returns Fix object representing data

Return type *Fix*

upoints.nmea.**MODE_INDICATOR = {'A': 'Autonomous', 'E': 'Estimated', 'D': 'Differential', 'M': 'Manual', 'N': 'Invalid',**
NMEA's mapping of code to reading type

class upoints.nmea.**Position**(*time*, *status*, *latitude*, *longitude*, *speed*, *track*, *date*, *variation*, *mode=None*)
Bases: *upoints.point.Point*

Class for representing a GPS NMEA-formatted position.

New in version 0.8.0.

Initialise a new Position object.

Parameters

- **time** (*datetime.time*) – Time the fix was taken
- **status** (*bool*) – Whether the data is active
- **latitude** (*float*) – Fix's latitude
- **longitude** (*float*) – Fix's longitude
- **speed** (*float*) – Ground speed
- **track** (*float*) – Track angle
- **date** (*datetime.date*) – Date when position was taken
- **variation** (*float*) – Magnetic variation
- **mode** (*str*) – Type of reading

mode_string()
Return a string version of the reading mode information.

Returns Quality information as string

Return type str

static parse_elements(*elements*)
Parse position data elements.

Parameters **elements** (*list*) – Data values for position

Returns Position object representing data

Return type *Position*

class upoints.nmea.**Waypoint**(*latitude*, *longitude*, *name*)
Bases: *upoints.point.Point*

Class for representing a NMEA-formatted waypoint.

New in version 0.8.0.

Initialise a new `Waypoint` object.

Parameters

- **latitude** (*float*) – Waypoint's latitude
- **longitude** (*float*) – Waypoint's longitude
- **name** (*str*) – Comment for waypoint

static parse_elements(*elements*)
Parse waypoint data elements.

Parameters **elements** (*list*) – Data values for fix

Returns Object representing data

Return type *nmea.Waypoint*

upoints.nmea.**calc_checksum**(*sentence*)
Calculate a NMEA 0183 checksum for the given sentence.

NMEA checksums are a simple XOR of all the characters in the sentence between the leading "$" symbol, and the "*" checksum separator.

Parameters **sentence** (*str*) – NMEA 0183 formatted sentence

upoints.nmea.**nmea_latitude**(*latitude*)
Generate a NMEA-formatted latitude pair.

Parameters **latitude** (*float*) – Latitude to convert

Returns NMEA-formatted latitude values

Return type tuple

upoints.nmea.**nmea_longitude**(*longitude*)
Generate a NMEA-formatted longitude pair.

Parameters **longitude** (*float*) – Longitude to convert

Returns NMEA-formatted longitude values

Return type tuple

upoints.nmea.**parse_latitude**(*latitude*, *hemisphere*)
Parse a NMEA-formatted latitude pair.

Parameters

- **latitude** (*str*) – Latitude in DDMM.MMMM
- **hemisphere** (*str*) – North or South

Returns Decimal representation of latitude

Return type float

upoints.nmea.**parse_longitude**(*longitude*, *hemisphere*)
Parse a NMEA-formatted longitude pair.

Parameters

- **longitude** (*str*) – Longitude in DDDMM.MMMM
- **hemisphere** (*str*) – East or West

Returns Decimal representation of longitude

Return type float

osm

osm - Imports OpenStreetMap data files..

class upoints.osm.**Node**(*ident*, *latitude*, *longitude*, *visible=False*, *user=None*, *timestamp=None*, *tags=None*)

Bases: *upoints.point.Point*

Class for representing a node element from OSM data files.

New in version 0.9.0.

Initialise a new Node object.

Parameters

- **ident** (*int*) – Unique identifier for the node
- **latitude** (*float*) – Nodes's latitude
- **longitude** (*float*) – Node's longitude
- **visible** (*bool*) – Whether the node is visible
- **user** (*str*) – User who logged the node
- **timestamp** (*str*) – The date and time a node was logged
- **tags** (*dict*) – Tags associated with the node

fetch_area_osm(*distance*)

Fetch, and import, an OSM region.

Parameters **distance** (*int*) – Boundary distance in kilometres

Returns All the data OSM has on a region imported for use

Return type *Osm*

get_area_url(*distance*)

Generate URL for downloading OSM data within a region.

Parameters **distance** (*int*) – Boundary distance in kilometres

Returns

URL that can be used to fetch the OSM data within distance of location

Return type str

static parse_elem(*element*)

Parse a OSM node XML element.

Parameters **element** (*etree.Element*) – XML Element to parse

Returns Object representing parsed element

Return type *Node*

toosm()
Generate a OSM node element subtree.

Returns OSM node element

Return type etree.Element

class upoints.osm.**Osm**(*osm_file=None*)
Bases: *upoints.point.Points*

Class for representing an OSM region.

New in version 0.9.0.

Initialise a new Osm object.

export_osm_file()
Generate OpenStreetMap element tree from Osm.

import_locations(*osm_file*)
Import OSM data files.

import_locations() returns a list of Node and Way objects.

It expects data files conforming to the OpenStreetMap 0.5 DTD, which is XML such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.5" generator="upoints/0.9.0">
  <node id="0" lat="52.015749" lon="-0.221765" user="jnrowe" visible="true"␣
↪timestamp="2008-01-25T12:52:11+00:00" />
  <node id="1" lat="52.015761" lon="-0.221767" visible="true" timestamp="2008-
↪01-25T12:53:00+00:00">
    <tag k="created_by" v="hand" />
    <tag k="highway" v="crossing" />
  </node>
  <node id="2" lat="52.015754" lon="-0.221766" user="jnrowe" visible="true"␣
↪timestamp="2008-01-25T12:52:30+00:00">
    <tag k="amenity" v="pub" />
  </node>
  <way id="0" visible="true" timestamp="2008-01-25T13:00:00+0000">
    <nd ref="0" />
    <nd ref="1" />
    <nd ref="2" />
    <tag k="ref" v="My Way" />
    <tag k="highway" v="primary" />
  </way>
</osm>
```

The reader uses the ElementTree module, so should be very fast when importing data. The above file processed by import_locations() will return the following *Osm* object:

```
Osm([
    Node(0, 52.015749, -0.221765, True, "jnrowe",
         utils.Timestamp(2008, 1, 25, 12, 52, 11), None),
    Node(1, 52.015761, -0.221767, True,
         utils.Timestamp(2008, 1, 25, 12, 53), None,
         {"created_by": "hand", "highway": "crossing"}),
    Node(2, 52.015754, -0.221766, True, "jnrowe",
         utils.Timestamp(2008, 1, 25, 12, 52, 30),
         {"amenity": "pub"}),
    Way(0, [0, 1, 2], True, None,
        utils.Timestamp(2008, 1, 25, 13, 00),
```

```
        {"ref": "My Way", "highway": "primary"})],
    generator="upoints/0.9.0")
```

Parameters **osm_file** (*iter*) – OpenStreetMap data to read

Returns Nodes and ways from the data

Return type *Osm*

class upoints.osm.**Way**(*ident*, *nodes*, *visible=False*, *user=None*, *timestamp=None*, *tags=None*)

Bases: *upoints.point.Points*

Class for representing a way element from OSM data files.

New in version 0.9.0.

Initialise a new Way object.

Parameters

- **ident** (*int*) – Unique identifier for the way
- **nodes** (*list of str*) – Identifiers of the nodes that form this way
- **visible** (*bool*) – Whether the way is visible
- **user** (*str*) – User who logged the way
- **timestamp** (*str*) – The date and time a way was logged
- **tags** (*dict*) – Tags associated with the way

static parse_elem(*element*)

Parse a OSM way XML element.

Parameters **element** (*etree.Element*) – XML Element to parse

Returns *Way* object representing parsed element

Return type *Way*

toosm()

Generate a OSM way element subtree.

Returns OSM way element

Return type etree.Element

upoints.osm.**get_area_url**(*location*, *distance*)

Generate URL for downloading OSM data within a region.

This function defines a boundary box where the edges touch a circle of distance kilometres in radius. It is important to note that the box is neither a square, nor bounded within the circle.

The bounding box is strictly a trapezoid whose north and south edges are different lengths, which is longer is dependant on whether the box is calculated for a location in the Northern or Southern hemisphere. You will get a shorter north edge in the Northern hemisphere, and vice versa. This is simply because we are applying a flat transformation to a spherical object, however for all general cases the difference will be negligible.

Parameters

- **location** (Point) – Centre of the region
- **distance** (*int*) – Boundary distance in kilometres

Returns

URL that can be used to fetch the OSM data within `distance` of `location`

Return type str

`point`

point - Classes for working with locations on Earth.

class `upoints.point.`**`KeyedPoints`**(*points=None*, *parse=False*, *units='metric'*)
Bases: `dict`

Class for representing a keyed group of *`Point`* objects.

New in version 0.2.0.

Initialise a new `KeyedPoints` object.

Parameters

- **`points`** (*`bool`*) – *`Point`* objects to wrap
- **`points`** – Whether to attempt import of `points`
- **`units`** (*`str`*) – Unit type to be used for distances when parsing string locations

`bearing`(*order*, *format='numeric'*)
Calculate bearing between locations.

Parameters

- **`order`** (*`list`*) – Order to process elements in
- **`format`** (*`str`*) – Format of the bearing string to return

Returns Bearing between points in series

Return type list of float

`destination`(*bearing*, *distance*)
Calculate destination locations for given distance and bearings.

Parameters

- **`bearing`** (*`float`*) – Bearing to move on in degrees
- **`distance`** (*`float`*) – Distance in kilometres

`distance`(*order*, *method='haversine'*)
Calculate distances between locations.

Parameters

- **`order`** (*`list`*) – Order to process elements in
- **`method`** (*`str`*) – Method used to calculate distance

Returns Distance between points in `order`

Return type list of float

`final_bearing`(*order*, *format='numeric'*)
Calculate final bearing between locations.

Parameters

- **`order`** (*`list`*) – Order to process elements in
- **`format`** (*`str`*) – Format of the bearing string to return

Returns Bearing between points in series

Return type list of float

forward(*bearing*, *distance*)
Calculate destination locations for given distance and bearings.

Parameters

- **bearing** (*float*) – Bearing to move on in degrees
- **distance** (*float*) – Distance in kilometres

import_locations(*locations*)
Import locations from arguments.

Parameters **locations** (*list of 2-tuple of str*) – Identifiers and locations

inverse(*order*)
Calculate the inverse geodesic between locations.

Parameters **order** (*list*) – Order to process elements in

Returns

Bearing and distance between points in series

Return type list of 2-tuple of float

midpoint(*order*)
Calculate the midpoint between locations.

Parameters **order** (*list*) – Order to process elements in

Returns Midpoint between points in series

Return type list of Point

range(*location*, *distance*)
Test whether locations are within a given range of the first.

Parameters

- **location** (Point) – Location to test range against
- **distance** (*float*) – Distance to test location is within

Returns Objects within specified range

Return type list of Point

sun_events(*date=None*, *zenith=None*)
Calculate sunrise/sunset times for locations.

Parameters

- **date** (*datetime.date*) – Calculate rise or set for given date
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the sunrise and sunset events for each point

Return type list of 2-tuple of datetime.datetime

sunrise(*date=None*, *zenith=None*)
Calculate sunrise times for locations.

Parameters

- **date** (*datetime.date*) – Calculate sunrise for given date
- **zenith** (*str*) – Calculate sunrise events, or end of twilight

Returns The time for the sunrise for each point

Return type list of datetime.datetime

sunset(*date=None*, *zenith=None*)
Calculate sunset times for locations.

Parameters

- **date** (*datetime.date*) – Calculate sunset for given date
- **zenith** (*str*) – Calculate sunset events, or start of twilight

Returns The time for the sunset for each point

Return type list of datetime.datetime

to_grid_locator(*precision='square'*)
Calculate Maidenhead locator for locations.

Parameters **precision** (*str*) – Precision with which generate locator string

Returns Maidenhead locator for each point

Return type list of str

class upoints.point.**Point**(*latitude*, *longitude*, *units='metric'*, *angle='degrees'*, *timezone=0*)
Bases: object

Simple class for representing a location on a sphere.

New in version 0.2.0.

Initialise a new Point object.

Parameters

- **latitude** (*float, tuple or list*) – Location's latitude
- **longitude** (*float, tuple or list*) – Location's longitude
- **angle** (*str*) – Type for specified angles
- **units** (*str*) – Units type to be used for distances
- **timezone** (*int*) – Offset from UTC in minutes

Raises

- ValueError – Unknown value for angle
- ValueError – Unknown value for units
- ValueError – Invalid value for latitude or longitude

bearing(*other*, *format='numeric'*)
Calculate the initial bearing from self to other.

Note: Applying common plane Euclidean trigonometry to bearing calculations suggests to us that the bearing between point A to point B is equal to the inverse of the bearing from Point B to Point A, whereas

spherical trigonometry is much more fun. If the `bearing` method doesn't make sense to you when calculating return bearings there are plenty of resources on the web that explain spherical geometry.

Todo

Add Rhumb line calculation

Parameters

- **other** (Point) – Location to calculate bearing to
- **format** (*str*) – Format of the bearing string to return

Returns Initial bearing from self to other in degrees

Return type float

Raises `ValueError` – Unknown value for `format`

destination(*bearing*, *distance*)

Calculate the destination from self given bearing and distance.

Parameters

- **bearing** (*float*) – Bearing from self
- **distance** (*float*) – Distance from self in `self.units`

Returns Location after travelling `distance` along `bearing`

Return type *Point*

distance(*other*, *method='haversine'*)

Calculate the distance from self to other.

As a smoke test this check uses the example from Wikipedia's Great-circle distance entry of Nashville International Airport to Los Angeles International Airport, and is correct to within 2 kilometres of the calculation there.

Parameters

- **other** (Point) – Location to calculate distance to
- **method** (*str*) – Method used to calculate distance

Returns Distance between self and other in `units`

Return type float

Raises `ValueError` – Unknown value for `method`

final_bearing(*other*, *format='numeric'*)

Calculate the final bearing from self to other.

See also:

bearing

Parameters

- **other** (Point) – Location to calculate final bearing to
- **format** (*str*) – Format of the bearing string to return

Returns Final bearing from self to other in degrees

Return type float

Raises `ValueError` – Unknown value for `format`

forward(*bearing*, *distance*)
Calculate the destination from self given bearing and distance.

Parameters

- **bearing** (*float*) – Bearing from self
- **distance** (*float*) – Distance from self in `self.units`

Returns Location after travelling `distance` along `bearing`

Return type *Point*

inverse(*other*)
Calculate the inverse geodesic from self to other.

Parameters **other** (`Point`) – Location to calculate inverse geodesic to

Returns Bearing and distance from self to other

Return type tuple of float objects

midpoint(*other*)
Calculate the midpoint from self to other.

See also:

bearing

Parameters **other** (`Point`) – Location to calculate midpoint to

Returns Great circle midpoint from self to other

Return type *Point*

sun_events(*date=None*, *zenith=None*)
Calculate the sunrise time for a `Point` object.

See also:

utils.sun_rise_set

Parameters

- **date** (*datetime.date*) – Calculate rise or set for given date
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the given events in the specified timezone

Return type tuple of datetime.datetime

sunrise(*date=None*, *zenith=None*)
Calculate the sunrise time for a `Point` object.

See also:

utils.sun_rise_set

Parameters

- **date** (*datetime.date*) – Calculate rise or set for given date
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the given event in the specified timezone

Return type datetime.datetime

sunset(*date=None*, *zenith=None*)

Calculate the sunset time for a `Point` object.

See also:

utils.sun_rise_set

Parameters

- **date** (*datetime.date*) – Calculate rise or set for given date
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the given event in the specified timezone

Return type datetime.datetime

to_grid_locator(*precision='square'*)

Calculate Maidenhead locator from latitude and longitude.

Parameters **precision** (*str*) – Precision with which generate locator string

Returns Maidenhead locator for latitude and longitude

Return type str

class upoints.point.**Points**(*points=None*, *parse=False*, *units='metric'*)

Bases: `list`

Class for representing a group of *Point* objects.

New in version 0.2.0.

Initialise a new `Points` object.

Parameters

- **points** (*list of Point*) – *Point* objects to wrap
- **parse** (*bool*) – Whether to attempt import of `points`
- **units** (*str*) – Unit type to be used for distances when parsing string locations

bearing(*format='numeric'*)

Calculate bearing between locations.

Parameters **format** (*str*) – Format of the bearing string to return

Returns Bearing between points in series

Return type list of float

destination(*bearing*, *distance*)

Calculate destination locations for given distance and bearings.

Parameters

- **bearing** (*float*) – Bearing to move on in degrees
- **distance** (*float*) – Distance in kilometres

Returns Points shifted by `distance` and `bearing`

Return type list of Point

distance(*method='haversine'*)

Calculate distances between locations.

Parameters **method** (*str*) – Method used to calculate distance

Returns Distance between points in series

Return type list of float

final_bearing(*format='numeric'*)

Calculate final bearing between locations.

Parameters **format** (*str*) – Format of the bearing string to return

Returns Bearing between points in series

Return type list of float

forward(*bearing*, *distance*)

Calculate destination locations for given distance and bearings.

Parameters

- **bearing** (*float*) – Bearing to move on in degrees
- **distance** (*float*) – Distance in kilometres

Returns Points shifted by `distance` and `bearing`

Return type list of Point

import_locations(*locations*)

Import locations from arguments.

Parameters **locations** (*list of str or tuple*) – Location identifiers

inverse()

Calculate the inverse geodesic between locations.

Returns

Bearing and distance between points in series

Return type list of 2-tuple of float

midpoint()

Calculate the midpoint between locations.

Returns Midpoint between points in series

Return type list of Point

range(*location*, *distance*)

Test whether locations are within a given range of `location`.

Parameters

- **location** (`Point`) – Location to test range against

- **distance** (*float*) – Distance to test location is within

Returns Points within range of the specified location

Return type list of Point

sun_events(*date=None*, *zenith=None*)
Calculate sunrise/sunset times for locations.

Parameters

- **date** (*datetime.date*) – Calculate rise or set for given date
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the sunrise and sunset events for each point

Return type list of 2-tuple of datetime.datetime

sunrise(*date=None*, *zenith=None*)
Calculate sunrise times for locations.

Parameters

- **date** (*datetime.date*) – Calculate sunrise for given date
- **zenith** (*str*) – Calculate sunrise events, or end of twilight

Returns The time for the sunrise for each point

Return type list of datetime.datetime

sunset(*date=None*, *zenith=None*)
Calculate sunset times for locations.

Parameters

- **date** (*datetime.date*) – Calculate sunset for given date
- **zenith** (*str*) – Calculate sunset events, or start of twilight

Returns The time for the sunset for each point

Return type list of datetime.datetime

to_grid_locator(*precision='square'*)
Calculate Maidenhead locator for locations.

Parameters **precision** (*str*) – Precision with which generate locator string

Returns Maidenhead locator for each point

Return type list of str

class upoints.point.**TimedPoint**(*latitude*, *longitude*, *units='metric'*, *angle='degrees'*, *timezone=0*, *time=None*)
Bases: *upoints.point.Point*

Class for representing a location with an associated time.

New in version 0.12.0.

Initialise a new `TimedPoint` object.

Parameters

- **latitude** (*float, tuple or list*) – Location's latitude

- **longitude** (*float, tuple or list*) – Location's longitude
- **angle** (*str*) – Type for specified angles
- **units** (*str*) – Units type to be used for distances
- **timezone** (*int*) – Offset from UTC in minutes
- **time** (*datetime.datetime*) – Time associated with the location

trigpoints

trigpoints - Imports trigpoint marker files.

class upoints.trigpoints.**Trigpoint**(*latitude*, *longitude*, *altitude*, *name=None*, *identity=None*)
Bases: *upoints.point.Point*

Class for representing a location from a trigpoint marker file.

> **Warning:** Although this class stores and presents the representation of altitude it doesn't take it in to account when making calculations. For example, consider a point at the base of Mount Everest and a point at the peak of Mount Everest the actual distance travelled between the two would be considerably larger than the reported value calculated at ground level.

New in version 0.2.0.

Initialise a new `Trigpoint` object.

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **altitude** (*float*) – Location's altitude
- **name** (*str*) – Name for location
- **identity** (*int*) – Database identifier, if known

class upoints.trigpoints.**Trigpoints**(*marker_file=None*)
Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Trigpoint* objects.

New in version 0.5.1.

Initialise a new `Trigpoints` object.

import_locations(*marker_file*)
Import trigpoint database files.

`import_locations()` returns a dictionary with keys containing the trigpoint identifier, and values that are *Trigpoint* objects.

It expects trigpoint marker files in the format provided at alltrigs-wgs84.txt, which is the following format:

```
H  SOFTWARE NAME & VERSION
I  GPSU 4.04,
S SymbolSet=0
...
W,500936,N52.066035,W000.281449,    37.0,Broom Farm
```

```
W,501097,N52.010585,W000.173443,    97.0,Bygrave
W,505392,N51.910886,W000.186462,   136.0,Sish Lane
```

Any line not consisting of 6 comma separated fields will be ignored. The reader uses the `csv` module, so alternative whitespace formatting should have no effect. The above file processed by `import_locations()` will return the following `dict` object:

```
{500936: point.Point(52.066035, -0.281449, 37.0, "Broom Farm"),
 501097: point.Point(52.010585, -0.173443, 97.0, "Bygrave"),
 505392: point.Point(51.910886, -0.186462, 136.0, "Sish Lane")}
```

Parameters **marker_file** (*iter*) – Trigpoint marker data to read

Returns Named locations with *Trigpoint* objects

Return type dict

Raises `ValueError` – Invalid value for `marker_file`

tzdata

tzdata - Imports timezone data files from UNIX zoneinfo.

class `upoints.tzdata.`**Zone**(*location*, *country*, *zone*, *comments=None*)

Bases: *upoints.point.Point*

Class for representing timezone descriptions from zoneinfo data.

New in version 0.6.0.

Initialise a new `Zone` object.

Parameters

- **location** (*str*) – Primary location in ISO 6709 format
- **country** (*str*) – Location's ISO 3166 country code
- **zone** (*str*) – Location's zone name as used in zoneinfo database
- **comments** (*list*) – Location's alternate names

class `upoints.tzdata.`**Zones**(*zone_file=None*)

Bases: *upoints.point.Points*

Class for representing a group of *Zone* objects.

New in version 0.6.0.

Initialise a new Zones object.

dump_zone_file()

Generate a zoneinfo compatible zone description table.

Returns zoneinfo descriptions

Return type list

import_locations(*zone_file*)

Parse zoneinfo zone description data files.

`import_locations()` returns a list of *Zone* objects.

It expects data files in one of the following formats:

```
AN  +1211-06900      America/Curacao
AO  -0848+01314      Africa/Luanda
AQ  -7750+16636      Antarctica/McMurdo      McMurdo Station, Ross Island
```

Files containing the data in this format can be found in the `zone.tab` file that is normally found in `/usr/share/zoneinfo` on UNIX-like systems, or from the standard distribution site.

When processed by `import_locations()` a `list` object of the following style will be returned:

```
[Zone(None, None, "AN", "America/Curacao", None),
 Zone(None, None, "AO", "Africa/Luanda", None),
 Zone(None, None, "AO", "Antartica/McMurdo",
      ["McMurdo Station", "Ross Island"])]
```

Parameters **zone_file** (*iter*) – `zone.tab` data to read

Returns Locations as *Zone* objects

Return type list

Raises `FileFormatError` – Unknown file format

utils

utils - Support code for *upoints*.

upoints.utils.**BODIES = {'Mercury': 2440, 'Sun': 696000, 'Neptune': 24622, 'Moon': 1738, 'Mars': 3390, 'Venus': 6052,**
Body radii of various solar system objects

upoints.utils.**BODY_RADIUS = 6367**
Default body radius to use for calculations

exception upoints.utils.**FileFormatError**(*site=None*)
Bases: `exceptions.ValueError`

Error object for data parsing error.

New in version 0.3.0.

Initialise a new `FileFormatError` object.

Parameters **site** (*str*) – Remote site name to display in error message

upoints.utils.**LONGITUDE_FIELD = 20**
Maidenhead locator constants

upoints.utils.**NAUTICAL_MILE = 1.852**
Number of kilometres per nautical mile

upoints.utils.**STATUTE_MILE = 1.609**
Number of kilometres per statute mile

class upoints.utils.**Timestamp**
Bases: `datetime.datetime`

Class for representing an OSM timestamp value.

isoformat()
Generate an ISO 8601 formatted time stamp.

Returns ISO 8601 formatted time stamp

Return type str

static **parse_isoformat**(*timestamp*)
Parse an ISO 8601 formatted time stamp.

Parameters **timestamp** (*str*) – Timestamp to parse

Returns Parsed timestamp

Return type *Timestamp*

class upoints.utils.**TzOffset**(*tzstring*)
Bases: datetime.tzinfo

Time offset from UTC.

Initialise a new `TzOffset` object.

Args:: tzstring (str): ISO 8601 style timezone definition

as_timezone()
Create a human-readable timezone string.

Returns Human-readable timezone definition

Return type str

dst(*dt=None*)
Daylight Savings Time offset.

Note: This method is only for compatibility with the `tzinfo` interface, and does nothing

Parameters **dt** (*any*) – For compatibility with parent classes

utcoffset(*dt=None*)
Return the offset in minutes from UTC.

Parameters **dt** (*any*) – For compatibility with parent classes

upoints.utils.**ZENITH = {'civil': -6, 'astronomical': -18, None: -0.8333333333333334, 'nautical': -12}**
Sunrise/-set mappings from name to angle

upoints.utils.**angle_to_distance**(*angle*, *units='metric'*)
Convert angle in to distance along a great circle.

Parameters

- **angle** (*float*) – Angle in degrees to convert to distance
- **units** (*str*) – Unit type to be used for distances

Returns Distance in `units`

Return type float

Raises `ValueError` – Unknown value for `units`

upoints.utils.**angle_to_name**(*angle*, *segments=8*, *abbr=False*)
Convert angle in to direction name.

Parameters

- **angle** (*float*) – Angle in degrees to convert to direction name
- **segments** (*int*) – Number of segments to split compass in to

- **abbr** (*bool*) – Whether to return abbreviated direction string

Returns Direction name for `angle`

Return type str

`upoints.utils.`**calc_radius**(*latitude*, *ellipsoid='WGS84'*)

Calculate earth radius for a given latitude.

This function is most useful when dealing with datasets that are very localised and require the accuracy of an ellipsoid model without the complexity of code necessary to actually use one. The results are meant to be used as a *BODY_RADIUS* replacement when the simple geocentric value is not good enough.

The original use for `calc_radius` is to set a more accurate radius value for use with trigpointing databases that are keyed on the OSGB36 datum, but it has been expanded to cover other ellipsoids.

Parameters

- **latitude** (*float*) – Latitude to calculate earth radius for
- **ellipsoid** (*tuple of float*) – Ellipsoid model to use for calculation

Returns Approximated Earth radius at the given latitude

Return type float

`upoints.utils.`**distance_to_angle**(*distance*, *units='metric'*)

Convert a distance in to an angle along a great circle.

Parameters

- **distance** (*float*) – Distance to convert to degrees
- **units** (*str*) – Unit type to be used for distances

Returns Angle in degrees

Return type float

Raises `ValueError` – Unknown value for `units`

`upoints.utils.`**dump_xearth_markers**(*markers*, *name='identifier'*)

Generate an Xearth compatible marker file.

`dump_xearth_markers()` writes a simple Xearth marker file from a dictionary of `trigpoints.Trigpoint` objects.

It expects a dictionary in one of the following formats. For support of `Trigpoint` that is:

```
{500936: Trigpoint(52.066035, -0.281449, 37.0, "Broom Farm"),
 501097: Trigpoint(52.010585, -0.173443, 97.0, "Bygrave"),
 505392: Trigpoint(51.910886, -0.186462, 136.0, "Sish Lane")}
```

And generates output of the form:

```
52.066035 -0.281449 "500936" # Broom Farm, alt 37m
52.010585 -0.173443 "501097" # Bygrave, alt 97m
51.910886 -0.186462 "205392" # Sish Lane, alt 136m
```

Or similar to the following if the `name` parameter is set to `name`:

```
52.066035 -0.281449 "Broom Farm" # 500936 alt 37m
52.010585 -0.173443 "Bygrave" # 501097 alt 97m
51.910886 -0.186462 "Sish Lane" # 205392 alt 136m
```

Point objects should be provided in the following format:

```
{"Broom Farm": Point(52.066035, -0.281449),
 "Bygrave": Point(52.010585, -0.173443),
 "Sish Lane": Point(51.910886, -0.186462)}
```

And generates output of the form:

```
52.066035 -0.281449 "Broom Farm"
52.010585 -0.173443 "Bygrave"
51.910886 -0.186462 "Sish Lane"
```

Note: xplanet also supports xearth marker files, and as such can use the output from this function.

See also:

upoints.xearth.Xearths.import_locations

Parameters

- **markers** (*dict*) – Dictionary of identifier keys, with `Trigpoint` values
- **name** (*str*) – Value to use as Xearth display string

Returns List of strings representing an Xearth marker file

Return type list

Raises `ValueError` – Unsupported value for `name`

`upoints.utils.`**`element_creator`**(*namespace=None*)

Create a simple namespace-aware objectify element creator.

Parameters **namespace** (*str*) – Namespace to work in

Returns Namespace-aware element creator

Return type *function*

`upoints.utils.`**`from_grid_locator`**(*locator*)

Calculate geodesic latitude/longitude from Maidenhead locator.

Parameters **locator** (*str*) – Maidenhead locator string

Returns Geodesic latitude and longitude values

Return type tuple of float

Raises

- `ValueError` – Incorrect grid locator length
- `ValueError` – Invalid values in locator string

`upoints.utils.`**`from_iso6709`**(*coordinates*)

Parse ISO 6709 coordinate strings.

This function will parse ISO 6709-1983(E) "Standard representation of latitude, longitude and altitude for geographic point locations" elements. Unfortunately, the standard is rather convoluted and this implementation is incomplete, but it does support most of the common formats in the wild.

The W3C has a simplified profile for ISO 6709 in Latitude, Longitude and Altitude format for geospatial information. It unfortunately hasn't received widespread support as yet, but hopefully it will grow just as the simplified ISO 8601 profile has.

See also:

to_iso6709

Parameters **coordinates** (*str*) – ISO 6709 coordinates string

Returns

A tuple consisting of latitude and longitude in degrees, along with the elevation in metres

Return type tuple

Raises

- ValueError – Input string is not ISO 6709 compliant
- ValueError – Invalid value for latitude
- ValueError – Invalid value for longitude

upoints.utils.**parse_location**(*location*)

Parse latitude and longitude from string location.

Parameters **location** (*str*) – String to parse

Returns Latitude and longitude of location

Return type tuple of float

upoints.utils.**prepare_csv_read**(*data*, *field_names*, **args*, ***kwargs*)

Prepare various input types for CSV parsing.

Parameters

- **data** (*iter*) – Data to read
- **field_names** (*tuple of str*) – Ordered names to assign to fields

Returns CSV reader suitable for parsing

Return type csv.DictReader

Raises TypeError – Invalid value for data

upoints.utils.**prepare_read**(*data*, *method='readlines'*, *mode='r'*)

Prepare various input types for parsing.

Parameters

- **data** (*iter*) – Data to read
- **method** (*str*) – Method to process data with
- **mode** (*str*) – Custom mode to process with, if data is a file

Returns List suitable for parsing

Return type list

Raises TypeError – Invalid value for data

upoints.utils.**prepare_xml_read**(*data*, *objectify=False*)

Prepare various input types for XML parsing.

Parameters

- **data** (*iter*) – Data to read
- **objectify** (*bool*) – Parse using lxml's objectify data binding

Returns Tree suitable for parsing

Return type etree.ElementTree

Raises `TypeError` – Invalid value for data

`upoints.utils.`**repr_assist**(*obj*, *remap=None*)

Helper function to simplify `__repr__` methods.

Parameters

- **obj** – Object to pull argument values for
- **remap** (*dict*) – Argument pairs to remap before output

Returns Self-documenting representation of `value`

Return type str

`upoints.utils.`**sun_events**(*latitude*, *longitude*, *date*, *timezone=0*, *zenith=None*)

Convenience function for calculating sunrise and sunset.

Civil twilight starts/ends when the Sun's centre is 6 degrees below the horizon.

Nautical twilight starts/ends when the Sun's centre is 12 degrees below the horizon.

Astronomical twilight starts/ends when the Sun's centre is 18 degrees below the horizon.

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **date** (*datetime.date*) – Calculate rise or set for given date
- **timezone** (*int*) – Offset from UTC in minutes
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the given events in the specified timezone

Return type tuple of datetime.time

`upoints.utils.`**sun_rise_set**(*latitude*, *longitude*, *date*, *mode='rise'*, *timezone=0*, *zenith=None*)

Calculate sunrise or sunset for a specific location.

This function calculates the time sunrise or sunset, or optionally the beginning or end of a specified twilight period.

Source:

```
Almanac for Computers, 1990
published by Nautical Almanac Office
United States Naval Observatory
Washington, DC 20392
```

Parameters

- **latitude** (*float*) – Location's latitude

- **longitude** (*float*) – Location's longitude
- **date** (*datetime.date*) – Calculate rise or set for given date
- **mode** (*str*) – Which time to calculate
- **timezone** (*int*) – Offset from UTC in minutes
- **zenith** (*str*) – Calculate rise/set events, or twilight times

Returns

The time for the given event in the specified timezone, or `None` if the event doesn't occur on the given date

Return type datetime.time or None

Raises `ValueError` – Unknown value for `mode`

`upoints.utils.`**`to_dd`**(*degrees*, *minutes*, *seconds=0*)

Convert degrees, minutes and optionally seconds to decimal angle.

Parameters

- **degrees** (*float*) – Number of degrees
- **minutes** (*float*) – Number of minutes
- **seconds** (*float*) – Number of seconds

Returns Angle converted to decimal degrees

Return type float

`upoints.utils.`**`to_dms`**(*angle*, *style='dms'*)

Convert decimal angle to degrees, minutes and possibly seconds.

Parameters

- **angle** (*float*) – Angle to convert
- **style** (*str*) – Return fractional or whole minutes values

Returns Angle converted to degrees, minutes and possibly seconds

Return type tuple of int

Raises `ValueError` – Unknown value for `style`

`upoints.utils.`**`to_grid_locator`**(*latitude*, *longitude*, *precision='square'*)

Calculate Maidenhead locator from latitude and longitude.

Parameters

- **latitude** (*float*) – Position's latitude
- **longitude** (*float*) – Position's longitude
- **precision** (*str*) – Precision with which generate locator string

Returns Maidenhead locator for latitude and longitude

Return type str

Raise: ValueError: Invalid precision identifier ValueError: Invalid latitude or longitude value

upoints.utils.**to_iso6709**(*latitude*, *longitude*, *altitude=None*, *format='dd'*, *precision=4*)
Produce ISO 6709 coordinate strings.

This function will produce ISO 6709-1983(E) "Standard representation of latitude, longitude and altitude for geographic point locations" elements.

See also:

from_iso6709

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **altitude** (*float*) – Location's altitude
- **format** (*str*) – Format type for string
- **precision** (*int*) – Latitude/longitude precision

Returns ISO 6709 coordinates string

Return type str

Raises `ValueError` – Unknown value for `format`

upoints.utils.**value_or_empty**(*value*)
Return an empty string for display when value is `None`.

Parameters **value** (*str*) – Value to prepare for display

Returns String representation of `value`

Return type str

weather_stations

weather_stations - Imports weather station data files.

class upoints.weather_stations.**Station**(*alt_id*, *name*, *state*, *country*, *wmo*, *latitude*, *longitude*, *ua_latitude*, *ua_longitude*, *altitude*, *ua_altitude*, *rbsn*)
Bases: *upoints.trigpoints.Trigpoint*

Class for representing a weather station from a NOAA data file.

New in version 0.2.0.

Initialise a new `Station` object.

Parameters

- **alt_id** (*str*) – Alternate location identifier
- **name** (*str*) – Station's name
- **state** (*str*) – State name, if station is in the US
- **country** (*str*) – Country name
- **wmo** (*int*) – WMO region code
- **latitude** (*float*) – Station's latitude
- **longitude** (*float*) – Station's longitude

- **ua_latitude** (*float*) – Station's upper air latitude
- **ua_longitude** (*float*) – Station's upper air longitude
- **altitude** (*int*) – Station's elevation
- **ua_altitude** (*int*) – Station's upper air elevation
- **rbsn** (*bool*) – True if station belongs to RSBN

class upoints.weather_stations.**Stations**(*data=None*, *index='WMO'*)
Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Station* objects.

New in version 0.5.1.

Initialise a new *Stations* object.

import_locations(*data*, *index='WMO'*)
Parse NOAA weather station data files.

`import_locations()` returns a dictionary with keys containing either the WMO or ICAO identifier, and values that are `Station` objects that describes the large variety of data exported by NOAA.

It expects data files in one of the following formats:

```
00;000;PABL;Buckland, Buckland Airport;AK;United States;4;65-58-56N;161-09-
↪07W;;;7;;
01;001;ENJA;Jan Mayen;;Norway;6;70-56N;008-40W;70-56N;008-40W;10;9;P
01;002;----;Grahuken;;Norway;6;79-47N;014-28E;;;;15;
```

or:

```
AYMD;94;014;Madang;;Papua New Guinea;5;05-13S;145-47E;05-13S;145-47E;3;5;P
AYMO;--;---;Manus Island/Momote;;Papua New Guinea;5;02-03-43S;147-25-27E;;;4;;
AYPY;94;035;Moresby;;Papua New Guinea;5;09-26S;147-13E;09-26S;147-13E;38;49;P
```

Files containing the data in this format can be downloaded from the NOAA's site in their station location page.

WMO indexed files downloaded from the NOAA site when processed by `import_locations()` will return `dict` object of the following style:

```
{'00000': Station('PABL', 'Buckland, Buckland Airport', 'AK',
                  'United States', 4, 65.982222. -160.848055, None,
                  None, 7, False),
 '01001'; Station('ENJA', Jan Mayen, None, 'Norway', 6, 70.933333,
                  -7.333333, 70.933333, -7.333333, 10, 9, True),
 '01002': Station(None, 'Grahuken', None, 'Norway', 6, 79.783333,
                  13.533333, None, None, 15, False)}
```

And `dict` objects such as the following will be created when ICAO indexed data files are processed:

```
{'AYMD': Station("94", "014", "Madang", None, "Papua New Guinea",
                 5, -5.216666, 145.783333, -5.216666,
                 145.78333333333333, 3, 5, True,
 'AYMO': Station(None, None, "Manus Island/Momote", None,
                 "Papua New Guinea", 5, -2.061944, 147.424166,
                 None, None, 4, False,
 'AYPY': Station("94", "035", "Moresby", None, "Papua New Guinea",
                 5, -9.433333, 147.216667, -9.433333, 147.216667,
                 38, 49, True}
```

Parameters

- **data** (*iter*) – NOAA station data to read
- **index** (*str*) – The identifier type used in the file

Returns WMO locations with *Station* objects

Return type dict

Raises FileFormatError – Unknown file format

xearth

xearth - Imports xearth-style marker files.

class upoints.xearth.**Xearth**(*latitude*, *longitude*, *comment=None*)

Bases: *upoints.point.Point*

Class for representing a location from a Xearth marker.

New in version 0.2.0.

Initialise a new Xearth object.

Parameters

- **latitude** (*float*) – Location's latitude
- **longitude** (*float*) – Location's longitude
- **comment** (*str*) – Comment for location

class upoints.xearth.**Xearths**(*marker_file=None*)

Bases: *upoints.point.KeyedPoints*

Class for representing a group of *Xearth* objects.

New in version 0.5.1.

Initialise a new Xearths object.

import_locations(*marker_file*)

Parse Xearth data files.

import_locations() returns a dictionary with keys containing the xearth name, and values consisting of a *Xearth* object and a string containing any comment found in the marker file.

It expects Xearth marker files in the following format:

```
# Comment

52.015      -0.221 "Home"           # James Rowe's home
52.6333     -2.5   "Telford"
```

Any empty line or line starting with a '#' is ignored. All data lines are whitespace-normalised, so actual layout should have no effect. The above file processed by import_locations() will return the following dict object:

```
{'Home': point.Point(52.015, -0.221, "James Rowe's home"),
 'Telford': point.Point(52.6333, -2.5, None)}
```

Note: This function also handles the extended xplanet marker files whose points can optionally contain added xplanet specific keywords for defining colours and fonts.

Parameters **marker_file** (*iter*) – Xearth marker data to read

Returns Named locations with optional comments

Return type dict

1.9 Glossary

GPS GPS (Global Positioning System)

Loran LORAN (LOng RAnge Navigation)

1.10 Release HOWTO

1.10.1 Test

In the general case tests can be run via `nose2`:

```
$ nose2 -vv tests
```

When preparing a release it is important to check that *upoints* works with all currently supported Python versions, and that the documentation is correct.

1.10.2 Prepare release

With the tests passing, perform the following steps

- Update the version data in `upoints/_version.py`
- Update `NEWS.rst`, if there are any user visible changes
- Commit the release notes and version changes
- Create a signed tag for the release
- Push the changes, including the new tag, to the GitHub repository

1.10.3 Update PyPI

Create and upload the new release tarballs to PyPI:

```
$ ./setup.py sdist --formats=bztar,gztar register upload --sign
```

Fetch the uploaded tarballs, and check for errors.

You should also perform test installations from PyPI, to check the experience *upoints* users will have.

1.11 Todo

Todo

Description

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/upoints/checkouts/latest/upoints/edist.py:docstring of upoints.edist.NumberedPoints.flight_plan, line 3.)

Todo

Add optional check for message length, on by default

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/upoints/checkouts/latest/upoints/nmea.py:docstring of upoints.nmea.Locations.import_locations, line 47.)

Todo

Add Rhumb line calculation

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/upoints/checkouts/latest/upoints/point.py:docstring of upoints.point.Point.bearing, line 12.)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

u

Index

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z