Skyfield

Astronomi Elegan untuk Python

May 2025

Astronomi Elegan untuk Python

Skyfield menghitung posisi bintang, planet, dan satelit yang mengorbit Bumi. Hasilnya harus sesuai dengan posisi yang dihasilkan oleh United States Naval Observatory dan *Astronomical Almanac* mereka hingga 0,0005 detik busur (setengah "mas" atau milidetik busur).

- Ditulis dengan Python murni.
- Diinstal tanpa kompilasi apa pun.
- Mendukung Python 2.7 dan Python 3.

Menghitung posisi Mars di langit semudah:

```
from skyfield.api import load
# Create a timescale and ask the current time.
ts = load.timescale()
t = ts.now()

# Load the JPL ephemeris DE421 (covers 1900-2050).
planets = load('de421.bsp')
earth, mars = planets['earth'], planets['mars']

# What's the position of Mars, viewed from Earth?
astrometric = earth.at(t).observe(mars)
ra, dec, distance = astrometric.radec()

print(ra)
print(dec)
print(distance)

10h 47m 56.24s
+09deg 03' 23.1"
```

```
2.33251 au
```

Skyfield dapat menghitung koordinat geosentris, seperti yang ditunjukkan pada contoh di atas, atau koordinat toposentris yang spesifik untuk lokasi Anda di permukaan bumi:

```
from skyfield.api import N, W, wgs84

boston = earth + wgs84.latlon(42.3583 * N, 71.0636 * W)

astrometric = boston.at(t).observe(mars)

alt, az, d = astrometric.apparent().altaz()


print(alt)

print(az)


25deg 27' 54.0"

101deg 33' 44.1"
```

Walaupun Skyfield sendiri tidak memiliki ketergantungan pada pustaka AstroPy , ia bersedia menerima objek waktu AstroPy sebagai input dan mengembalikan hasil dalam satuan AstroPy asli:

```
from astropy import units as u

xyz = astrometric.xyz.to(u.au)

altitude = alt.to(u.deg)


print(xyz)

print('{0:0.03f}'.format(altitude))


[-2.19049548  0.71236701  0.36712443] AU

25.465 deg
```

Akademisi dapat mengutip Skyfield sebagai ascl:1907.024 atau 2019ascl.soft07024R (lebih lanjut…)

Dokumentasi

Dokumentasi Skyfield tersedia di sini di situs web utama Skyfield:

- Daftar isi
- Memasang Skyfield
- Referensi API
- Catatan Perubahan

Namun kode sumber dan pelacak masalah tersedia di situs web lain:

- Paket: di Indeks Paket Python
- Sumber: di GitHub
- Diskusi: di GitHub
- Masalah: di GitHub

Lihat Changelog untuk catatan rilis versi saat ini — dan juga untuk pembaruan yang disertakan pada setiap versi sebelumnya!

Daftar isi

Selamat datang di dokumentasi Skyfield, pustaka astronomi Python murni yang dipercepat menggunakan matematika vektor NumPy!

Contoh

Banyak programmer bekerja paling efisien saat mereka dapat memulai dengan contoh yang berfungsi dan mengadaptasinya sesuai kebutuhan mereka. Setiap contoh berikut menggabungkan beberapa fitur Skyfield untuk memecahkan masalah umum, yang seharusnya memberi pembaca dasar untuk memecahkan masalah serupa lainnya.

Berikut adalah daftar contoh yang dapat Anda temukan di bagian di bawah ini:

Dan bagian berikut hanyalah contohnya sendiri.

Pukul berapa tengah hari matahari, saat Matahari melintasi meridian ?

Bumi bergerak paling cepat dalam orbitnya saat paling dekat dengan Matahari pada bulan Januari, dan paling lambat pada bulan Juli saat keduanya paling jauh. Hal ini memperpanjang hari-hari di bulan Januari sekitar 10 detik, memperpendek hari-hari di bulan Juli dengan jumlah yang sama, dan berarti jam tangan Anda tidak akan pernah menunjukkan pukul 12:00 tepat saat Matahari melintasi meridian Anda pada "siang matahari".

Anda dapat menghitung siang hari dengan menanyakan pada jam berapa Matahari melintasi meridian di lokasi Anda. Untuk mengetahui waktu siang hari ini, misalnya, Anda dapat melakukan hal berikut:

```
import datetime as dt

from pytz import timezone
```

```
from skyfield import almanac
from skyfield.api import N, W, wgs84, load

zone = timezone('US/Eastern')
now = zone.localize(dt.datetime.now())
midnight = now.replace(hour=0, minute=0, second=0, microsecond=0)
next_midnight = midnight + dt.timedelta(days=1)

ts = load.timescale()
t0 = ts.from_datetime(midnight)
t1 = ts.from_datetime(next_midnight)
eph = load('de421.bsp')
bluffton = wgs84.latlon(40.8939 * N, 83.8917 * W)

f = almanac.meridian_transits(eph, eph['Sun'], bluffton)
times, events = almanac.find_discrete(t0, t1, f)

# Select transits instead of antitransits.
times = times[events == 1]

t = times[0]
tstr = str(t.astimezone(zone))[:19]
print('Solar noon:', tstr)
Solar noon: 2020-04-19 13:34:33
```

Kapan hari akan gelap malam ini ?

Matahari terbit, matahari terbenam, dan beberapa jenis senja semuanya tersedia melalui modul Almanac Computation . Berikut skrip yang saya gunakan saat ingin tahu kapan hari akan cukup gelap untuk melihat bintang-bintang — atau seberapa pagi saya harus bangun untuk melihat langit pagi:

```
import datetime as dt
from pytz import timezone
```

```
from skyfield import almanac
from skyfield.api import N, W, wgs84, load

# Figure out local midnight.
zone = timezone('US/Eastern')
now = zone.localize(dt.datetime.now())
midnight = now.replace(hour=0, minute=0, second=0, microsecond=0)
next_midnight = midnight + dt.timedelta(days=1)

ts = load.timescale()
t0 = ts.from_datetime(midnight)
t1 = ts.from_datetime(next_midnight)
eph = load('de421.bsp')
bluffton = wgs84.latlon(40.8939 * N, 83.8917 * W)
f = almanac.dark_twilight_day(eph, bluffton)
times, events = almanac.find_discrete(t0, t1, f)

previous_e = f(t0).item()
for t, e in zip(times, events):
    tstr = str(t.astimezone(zone))[:16]
    if previous_e < e:
        print(tstr, ' ', almanac.TWILIGHTS[e], 'starts')
    else:
        print(tstr, ' ', almanac.TWILIGHTS[previous_e], 'ends')
    previous_e = e
2020-04-19 05:09   Astronomical twilight starts
2020-04-19 05:46   Nautical twilight starts
2020-04-19 06:20   Civil twilight starts
2020-04-19 06:49   Day starts
2020-04-19 20:20   Day ends
2020-04-19 20:48   Civil twilight ends
```

```
2020-04-19 21:23   Nautical twilight ends
2020-04-19 22:00   Astronomical twilight ends
```

Seperti yang dapat Anda lihat dari kode di atas, jika tingkat cahaya baru lebih terang, maka kita katakan bahwa tingkat baru tersebut “dimulai”, tetapi jika tingkat baru tersebut lebih gelap, maka kita katakan bahwa tingkat sebelumnya “berakhir” — jadi daripada mengatakan “senja astronomis *dimulai* pada pukul 21:23”, kita katakan “senja bahari *berakhir* pada pukul 21:23.” Itulah sebabnya kode tersebut mengikuti previous_e dan membandingkannya dengan tingkat senja yang baru.

Apa fase Bulan malam ini ?

The *phase* of the Moon is defined as the angle between the Moon and the Sun along the ecliptic. This angle is computed as the difference in the *ecliptic longitude* of the Moon and of the Sun. The result is an angle that is 0° for the New Moon, 90° at the First Quarter, 180° at the Full Moon, and 270° at the Last Quarter. Skyfield also has a method for computing what fraction of a spherical body is illuminated by the sun.

```
from skyfield.api import load
from skyfield.framelib import ecliptic_frame

ts = load.timescale()
t = ts.utc(2019, 12, 9, 15, 36)

eph = load('de421.bsp')
sun, moon, earth = eph['sun'], eph['moon'], eph['earth']

e = earth.at(t)
s = e.observe(sun).apparent()
m = e.observe(moon).apparent()

_, slon, _ = s.frame_latlon(ecliptic_frame)
_, mlon, _ = m.frame_latlon(ecliptic_frame)
phase = (mlon.degrees - slon.degrees) % 360.0

percent = 100.0 * m.fraction_illuminated(sun)
```

```
print('Phase (0°–360°): {0:.1f}'.format(phase))
print('Percent illuminated: {0:.1f}%'.format(percent))
```

```
Phase (0°–360°): 149.4
Percent illuminated: 92.9%
```

What is the angular diameter of a planet, given its radius?

Be careful to select the correct radius when predicting a planet's angular diameter in the sky. Many web sites will quote some kind of "mean radius" that averages between a planet's squat polar radius and its wide equatorial radius. But most astronomers instead want to know the maximum, not average, diameter across a planet's visible face — so you will want to use the planet's equatorial radius in your calculation.

For example, a good current estimate of Neptune's equatorial radius is 24,764 km. We would therefore predicts its angular diameter as:

```
import numpy as np
from skyfield.api import Angle, load

ts = load.timescale()
time = ts.utc(2020, 12, 30)

eph = load('de421.bsp')
earth, neptune = eph['earth'], eph['neptune barycenter']
radius_km = 24764.0

astrometric = earth.at(time).observe(neptune)
ra, dec, distance = astrometric.apparent().radec()
apparent_diameter = Angle(radians=np.arcsin(radius_km / distance.km) * 2.0)
print('{:.6f} arcseconds'.format(apparent_diameter.arcseconds()))
```

```
2.257190 arcseconds
```

This agrees exactly with the output of the NASA HORIZONS system.

When is Venus at its greatest east and west elongations from the Sun?

This example illustrates the several practical steps that are often required to both find events of interest and then to learn more details about them.

- The concept of “elongation from the Sun” is here explained to Skyfield with a function that for any given time t returns the separation between the Sun and Venus in the sky.
- The find_maxima() routine is then set loose to find the moments over the 3 years 2019–2021 at which the elongation of Venus from the Sun reaches its maximum values. Skyfield starts by computing the elongation every step_days = 15 days between the search’s start time and end time, then hones in everywhere it sees a local maximum: a value that’s bigger than either of the two values next to it.
- Finally, a for loop over the results not only displays each maximum but computes and displays an extra fact: whether the elongation is an east or west maximum elongation, which is defined as whether Venus’s apparent ecliptic longitude is ahead of or behind the Sun’s along the great circle of the ecliptic.

This example can serve as a template for many other kinds of custom search:

```
from skyfield.api import load
from skyfield.framelib import ecliptic_frame
from skyfield.searchlib import find_maxima


ts = load.timescale()
t0 = ts.utc(2019)
t1 = ts.utc(2022)


eph = load('de421.bsp')
sun, earth, venus = eph['sun'], eph['earth'], eph['venus']


def elongation_at(t):
    e = earth.at(t)
    s = e.observe(sun).apparent()
    v = e.observe(venus).apparent()
    return s.separation_from(v).degrees


elongation_at.step_days = 15.0


times, elongations = find_maxima(t0, t1, elongation_at)
```

```
for t, elongation_degrees in zip(times, elongations):
    e = earth.at(t)
    _, slon, _ = e.observe(sun).apparent().frame_latlon(ecliptic_frame)
    _, vlon, _ = e.observe(venus).apparent().frame_latlon(ecliptic_frame)
    is_east = (vlon.degrees - slon.degrees) % 360.0 < 180.0
    direction = 'east' if is_east else 'west'
    print('{}  {:4.1f}° {} elongation'.format(
        t.utc_strftime(), elongation_degrees, direction))
```

```
2019-01-06 04:53:35 UTC  47.0° west elongation
2020-03-24 22:13:32 UTC  46.1° east elongation
2020-08-13 00:14:12 UTC  45.8° west elongation
2021-10-29 20:51:56 UTC  47.0° east elongation
```

Are planets separated by 0° at conjunction and 180° at opposition?

It surprises many newcomers to astronomy that the angular separation between two planets never drops all the way to 0° at conjunction nor increases all the way to a full 180° at opposition. The reason is that the planets will still have at least a slight lingering difference in latitude, even when their longitudes have brought them together or have placed them opposite each other in the sky.

We can take as an example the date and time of the conjunction between Mars and the Sun computed in the Opposition and Conjunction section of the Almanac page. How close are they in the sky at that moment? The separation_from() method measures raw angular distance between any two points in the sky:

```
from skyfield.api import load
from skyfield.framelib import ecliptic_frame

ts = load.timescale()
eph = load('de421.bsp')
sun, mars = eph['sun'], eph['mars']

t = ts.utc(2019, 9, 2, 10, 42, 26)
e = earth.at(t)
s = e.observe(sun).apparent()
```

```
m = e.observe(mars).apparent()

print('{:.5f}°'.format(m.separation_from(s).degrees))
```

```
1.08256°
```

They are more than one degree apart! How can that be, if their ecliptic longitudes are at that moment the same? Let's use Skyfield's ecliptic_frame to express their positions in Ecliptic coordinates:

```
print('    Latitude Longitude')


lat, lon, distance = s.frame_latlon(ecliptic_frame)

print('Sun  {:.5f}° {:.5f}°'.format(lat.degrees, lon.degrees))


lat, lon, distance = m.frame_latlon(ecliptic_frame)

print('Mars {:.5f}° {:.5f}°'.format(lat.degrees, lon.degrees))
```

```
   Latitude Longitude
Sun  0.00005° 159.68641°
Mars 1.08260° 159.68641°
```

While the Sun sits very close to the ecliptic — as we would expect, since the ecliptic is defined as the course the Sun takes around the sky each year — the inclination of the orbit of Mars has carried it more than one degree above the ecliptic. That's why the separation_from() method still measured an angle of more than one degree between them.

A similar situation pertains at opposition:

```
t = ts.utc(2020, 10, 13, 23, 25, 55)


e = earth.at(t)

s = e.observe(sun).apparent()

m = e.observe(mars).apparent()


print('Separation: {:.5f}°'.format(m.separation_from(s).degrees))


print('')

print('    Latitude Longitude')
```

```
lat, lon, distance = s.frame_latlon(ecliptic_frame)
print('Sun  {:.5f}° {:.5f}°'.format(lat.degrees, lon.degrees))

lat, lon, distance = m.frame_latlon(ecliptic_frame)
print('Mars {:.5f}° {:.5f}°'.format(lat.degrees, lon.degrees))
Separation: 177.00424°

    Latitude Longitude
Sun  0.00007° 201.07794°
Mars -2.99582° 21.07794°
```

Even though their ecliptic longitudes are 180° apart, the fact that neither the Sun nor Mars is lying exactly on the ecliptic means that the separation_from() method finds that they are not quite 180° apart.

In case you run across the term 'elongation' in discussions of conjunctions and oppositions, it's shorthand for 'the angle between a planet and the Sun' — and so each of the angular separations printed above can, more specifically, be labeled as the 'elongation of Mars' on those dates.

At what angle in the sky is the crescent Moon?

The angle of the crescent Moon changes with the seasons. In the spring, a crescent Moon will stand high above the Sun and appear to be lit from below. In the autumn, the Moon sets farther from the Sun along the horizon and is illuminated more from the side. What if we wanted to know the exact angle?

You can find the answer by asking for the Sun's "position angle" relative to the Moon, an angle you can compute between any two Skyfield positions. The angle will be 90° if the Sun is left of the moon, 180° if the Sun is directly below, and 270° if the Sun is to the right of the Moon.

```
from skyfield.api import N, W, load, wgs84
from skyfield.trigonometry import position_angle_of

ts = load.timescale()
t = ts.utc(2019, 9, 30, 23)

eph = load('de421.bsp')
sun, moon, earth = eph['sun'], eph['moon'], eph['earth']
boston = earth + wgs84.latlon(42.3583 * N, 71.0636 * W)
```

```
b = boston.at(t)

m = b.observe(moon).apparent()

s = b.observe(sun).apparent()

print(position_angle_of(m.altaz(), s.altaz()))

238deg 55' 55.3"
```

The position_angle_of() routine will not only accept the output of altaz(), but also of frame_latlon() if you want a position angle relative to the ecliptic's north pole.

Beware, though, that radec() produces coordinates in the opposite order from what position_angle_of() expects: right ascension is like longitude, not latitude. Try reversing the coordinates, like:

```
print(position_angle_of(m.radec(), s.radec()))

282deg 28' 15.7"
```

Drat, but this angle is backwards, because right ascension increases toward the east whereas the other angles, like azimuth, increase the other way around the circle.

When is a body or fixed coordinate above the horizon?

The following code will determine when the Galactic Center is above the horizon. The Galactic Center is an example of a fixed object, like a star or nebula or galaxy, whose right ascension and declination can be plugged in to a Star() object. The code will also work with a body from an ephemeris, like the Sun, Moon, or one of the planets.

```
from skyfield.api import N, Star, W, wgs84, load

from skyfield.almanac import find_discrete, risings_and_settings

from pytz import timezone


ts = load.timescale()

t0 = ts.utc(2019, 1, 19)

t1 = ts.utc(2019, 1, 21)


moab = wgs84.latlon(38.5725 * N, 109.54972238 * W)

eph = load('de421.bsp')

gc = Star(ra_hours=(17, 45, 40.04), dec_degrees=(-29, 0, 28.1))


f = risings_and_settings(eph, gc, moab)
```

```
tz = timezone('US/Mountain')


for t, updown in zip(*find_discrete(t0, t1, f)):
    print(t.astimezone(tz).strftime('%a %d %H:%M'), 'MST',
          'rises' if updown else 'sets')
```

```
Sat 19 05:51 MST rises
Sat 19 14:27 MST sets
Sun 20 05:47 MST rises
Sun 20 14:23 MST sets
```

At what rate is a target moving across the sky?

If you are interested in the rate at which a target is moving across the sky, you can call Skyfield's frame_latlon_and_rates() method and pass it the frame of reference in which you want the angles measured. First, compute the target's position relative to your geographic location:

```
from skyfield.api import load, wgs84, N,S,E,W


ts = load.timescale()
t = ts.utc(2021, 2, 3, 0, 0)
planets = load('de421.bsp')
earth, mars = planets['earth'], planets['mars']
topos = wgs84.latlon(35.1844 * N, 111.6535 * W, elevation_m=2099.5)


a = (earth + topos).at(t).observe(mars).apparent()
```

In Skyfield, a topocentric location object like topos is also a reference frame oriented to the location's horizon and zenith. So if you pass it to the frame_latlon_and_rates() method, Skyfield will compute the rates at which the altitude and azimuth are changing as the target moves across the sky:

```
(alt, az, distance,
 alt_rate, az_rate, range_rate) = a.frame_latlon_and_rates(topos)


print('Alt: {:+.1f} asec/min'.format(alt_rate.arcseconds.per_minute))
print('Az:  {:+.1f} asec/min'.format(az_rate.arcseconds.per_minute))
```

```
Alt: +548.7 asec/min
```

```
Az:  +1586.4 asec/min
```

You can choose other units besides arcseconds and per_minute. For the possible numerators see AngleRate, and for the possible denominators see Rate.

Computing the range rate-of-change

Both of the calls above return a range_rate that is positive if the body is moving away and negative if the target is moving closer:

```
print('Range rate: {:+.1f} km/s'.format(range_rate.km_per_s))
```

```
Range rate: +16.8 km/s
```

Computing angular speed

You might think that you could compute a target's total angular speed across the sky by simply subjecting the two angular rates of change to the Pythagorean theorem.

But that won't work, because of a subtlety: it turns out that all of the different kinds of longitude — including right ascension, azimuth, and ecliptic longitude — have lines that are far apart at the equator but that draw closer and closer together near the poles. I hope that there is an elegant antique globe sitting near you as you read this in your armchair. Look at the lines on its surface. Down at the equator, the lines of longitude stand far apart, and to move 15° in longitude you would have to travel across very nearly 15° of the Earth's surface. But now look at the poles. The lines of longitude draw so close together that, if you're close enough to the pole, you could cross 15° of longitude by traveling only a very short distance!

Happily, spherical trigonometry gives us a simple correction to apply. Multiplying the longitude rate by the cosine of the latitude gives a bare angular rate of motion across the sky, that can safely be tossed into the Pythagorean theorem:

```
from numpy import cos, sqrt


ralt = alt_rate.degrees.per_minute

raz = az_rate.degrees.per_minute * cos(alt.radians)


degrees_per_minute = sqrt(ralt*ralt + raz*raz)

print('{:.4f}° per minute'.format(degrees_per_minute))
```

```
0.2392° per minute
```

In exactly the same way, if instead you wanted to compute a target's speed against the background of stars, you would multiply the rate at which the right ascension is changing by the cosine of the declination before combining them with the Pythagorean theorem.

What is the right ascension and declination of a point in the sky?

Seorang pengamat sering kali tertarik pada koordinat astronomis dari posisi tertentu di langit di atas mereka. Jika pengamat dapat menentukan posisi menggunakan koordinat ketinggian dan azimuth, maka Skyfield dapat mengembalikan asensio dan deklinasinya yang tepat.

```
from skyfield import api


ts = api.load.timescale()
t = ts.utc(2019, 9, 13, 20)
geographic = api.wgs84.latlon(latitude_degrees=42, longitude_degrees=-87)
observer = geographic.at(t)
pos = observer.from_altaz(alt_degrees=90, az_degrees=0)


ra, dec, distance = pos.radec()
print(ra)
print(dec)
13h 41m 14.65s
+42deg 05' 50.0"
```

Lintang dan bujur berapakah yang berada di bawah asensio rekta dan deklinasi ini ?

Sebagian besar perhitungan Skyfield, seperti pengamatan planet atau satelit Bumi, secara langsung menghasilkan posisi vektor yang berpusat di Bumi. Anda dapat meneruskan vektor tersebut ke metode subpoint()geoid standar untuk menghitung lintang dan bujur.

Namun terkadang asensio dan deklinasi posisi sudah diketahui. Daripada membuat koordinat Stardengan koordinat tersebut dan memintanya untuk menghitung posisinya, ada pendekatan yang lebih sederhana: membuat posisi secara langsung.

```
from skyfield.api import load, wgs84
from skyfield.positionlib import position_of_radec


ts = load.timescale()
t = ts.utc(2020, 1, 3, 12, 45)


earth = 399  # NAIF code for the Earth center of mass
ra_hours = 3.79
dec_degrees = 24.1167
```

```
pleiades = position_of_radec(ra_hours, dec_degrees, t=t, center=earth)
subpoint = wgs84.subpoint(pleiades)


print('Latitude:', subpoint.latitude)
print('Longitude:', subpoint.longitude)
```

Latitude: 24deg 10' 33.5"

Longitude: 123deg 16' 53.9"

Lokasi geografis manakah yang lebih jauh dari pusat Bumi ?

Setelah saya mendaki Gunung Bierstadt di Colorado, seorang teman menyarankan bahwa ketinggiannya yang mencapai 14.000 kaki mungkin telah membawa saya lebih jauh dari pusat Bumi daripada yang pernah saya tempuh sebelumnya. Itu adalah pemikiran yang romantis: bahwa dengan kekuatan saya sendiri, saya telah mendaki lebih jauh dari inti planet asal saya daripada sebelumnya.

Namun ada masalah. Saya tahu bahwa saya pernah mengunjungi sebuah kota yang hanya berjarak beberapa derajat dari ekuator Bumi, dan tonjolan ekuator Bumi mungkin mendorong elevasi yang sederhana pada garis lintang itu lebih jauh dari pusat Bumi daripada puncak gunung di Colorado.

Jadi saya menulis skrip Skyfield cepat untuk membandingkan jarak dari pusat Bumi ke Accra, Ghana, dan puncak Gunung Bierstadt di Colorado.

```
from skyfield.api import N, W, wgs84, load
from skyfield.functions import length_of


ts = load.timescale()
t = ts.utc(2019, 1, 1)


bierstadt = wgs84.latlon(39.5828 * N, 105.6686 * W, elevation_m=4287.012)
m1 = length_of(bierstadt.at(t).xyz.m)
print(int(m1))


accra = wgs84.latlon(5.6037 * N, 0.1870 * W, elevation_m=61)
m2 = length_of(accra.at(t).xyz.m)
print(int(m2))
```

```
assert m2 > m1
print("I was", int(m2 - m1), "meters farther from the Earth's center\n"
    "when I visited Accra, at nearly sea level, than atop\n"
    "Mt. Bierstadt in Colorado.")
6373784
6377995
```

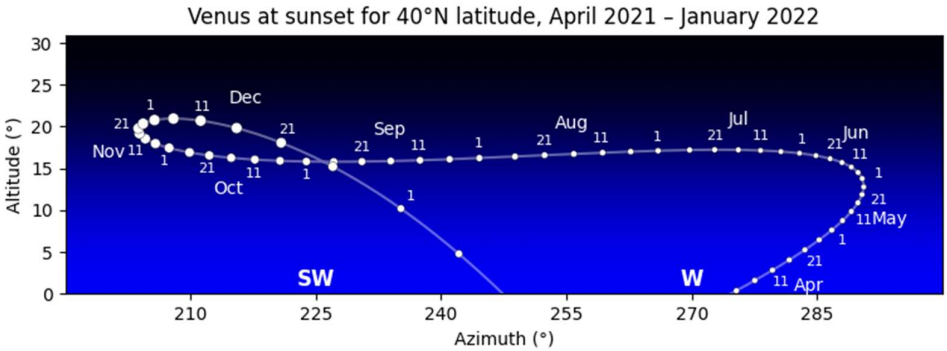
```
I was 4211 meters farther from the Earth's center
when I visited Accra, at nearly sea level, than atop
Mt. Bierstadt in Colorado.
```

Contoh Plot

Bagian dokumentasi ini secara bertahap akan mengumpulkan contoh skrip untuk menghasilkan gambar dari perhitungan Skyfield.

Perhatikan bahwa contoh skrip ini ditulis untuk versi matplotlib yang cukup baru . Jika Anda mencoba menjalankannya pada sistem dengan versi pustaka yang lebih lama, Anda mungkin melihat kesalahan — khususnya pada cara mereka menentukan warna, dalam hal ini Anda dapat mencoba menghilangkan parameter tersebut untuk menjalankan skrip. Bagaimanapun, ini hanya dimaksudkan sebagai titik awal untuk membangun skrip Anda sendiri, baik dengan matplotlib atau pustaka plotting lain yang Anda sukai.

Memetakan penampakan Venus



Langit pagi dan langit sore merupakan subjek yang populer untuk bagan astronomi, tetapi keduanya melibatkan suatu hal yang rumit: kecuali pengamat berada di ekuator, matahari terbit dan terbenam pada waktu yang berbeda setiap harinya. Jadi, apakah bagan menggambarkan "langit sore saat matahari terbenam" atau "45 menit setelah matahari terbenam" atau "satu jam setelah matahari terbenam", garis horizon tunggalnya menggambarkan waktu yang berbeda pada setiap hari berikutnya.

Oleh karena itu, menggambar peta pagi atau sore melibatkan dua langkah. Pertama, hitung semua waktu matahari terbit atau terbenam selama bulan yang Anda minati. Kemudian, buat ketinggian dan azimuth benda yang menjadi target pada waktu tersebut untuk peta Anda.

Contoh skrip ini menghasilkan bagan Venus yang dapat Anda lihat di atas. Beberapa catatan:

1. Pemilihan garis bujur agak sembarangan; lokasi mana pun pada garis lintang +40°N akan menghasilkan bagan yang hampir sama, karena Venus hanya bergerak beberapa piksel setiap hari. Namun, menempatkan pengamat pada meridian utama pada garis bujur 0° berarti kita dapat membanggakan bahwa diagram tersebut adalah untuk waktu Greenwich.

2. Lingkaran-lingkaran tersebut hanya mewakili besarnya Venus secara visual, bukan bentuknya. Bagan yang lebih canggih mungkin menunjukkan bulan sabitnya saat ia membesar dan mengecil.

3. Batas sumbu *x* dikodekan secara kaku agar skrip mudah dibaca, tetapi Anda mungkin dapat mengaturnya secara otomatis. Jika merencanakan adegan di dekat kutub utara,

berhati-hatilah dengan kemungkinan terjadinya lilitan saat azimuth melintasi utara dan kembali ke 0°.

4. Ada satu perubahan desain khusus yang dibuat secara manual: keputusan untuk mengganti label tanggal dari satu sisi kurva ke sisi lainnya pada tanggal 1 Oktober. Namun selain itu, kode tersebut mencoba untuk bersifat umum dan diharapkan dapat menjadi panduan untuk diagram serupa milik Anda sendiri.

```
import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import LinearSegmentedColormap

from skyfield import almanac

from skyfield.api import load, wgs84

from skyfield.magnitudelib import planetary_magnitude


MONTH_NAMES = '0 Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'.split()


# Figure out the times of sunset over our range of dates.


eph = load('de421.bsp')

earth, sun, venus = eph['earth'], eph['sun'], eph['venus']

observer = wgs84.latlon(+40.0, 0.0)


ts = load.timescale()

start, end = ts.utc(2021, 3, 7), ts.utc(2022, 2, 7)


f = almanac.sunrise_sunset(eph, observer)

t, y = almanac.find_discrete(start, end, f)

sunsets = (y == 0)

t = t[sunsets]


# For each moment of sunset, ask Skyfield for the month number, the day

# number, and for Venus's altitude, azimuth, and magnitude.
```

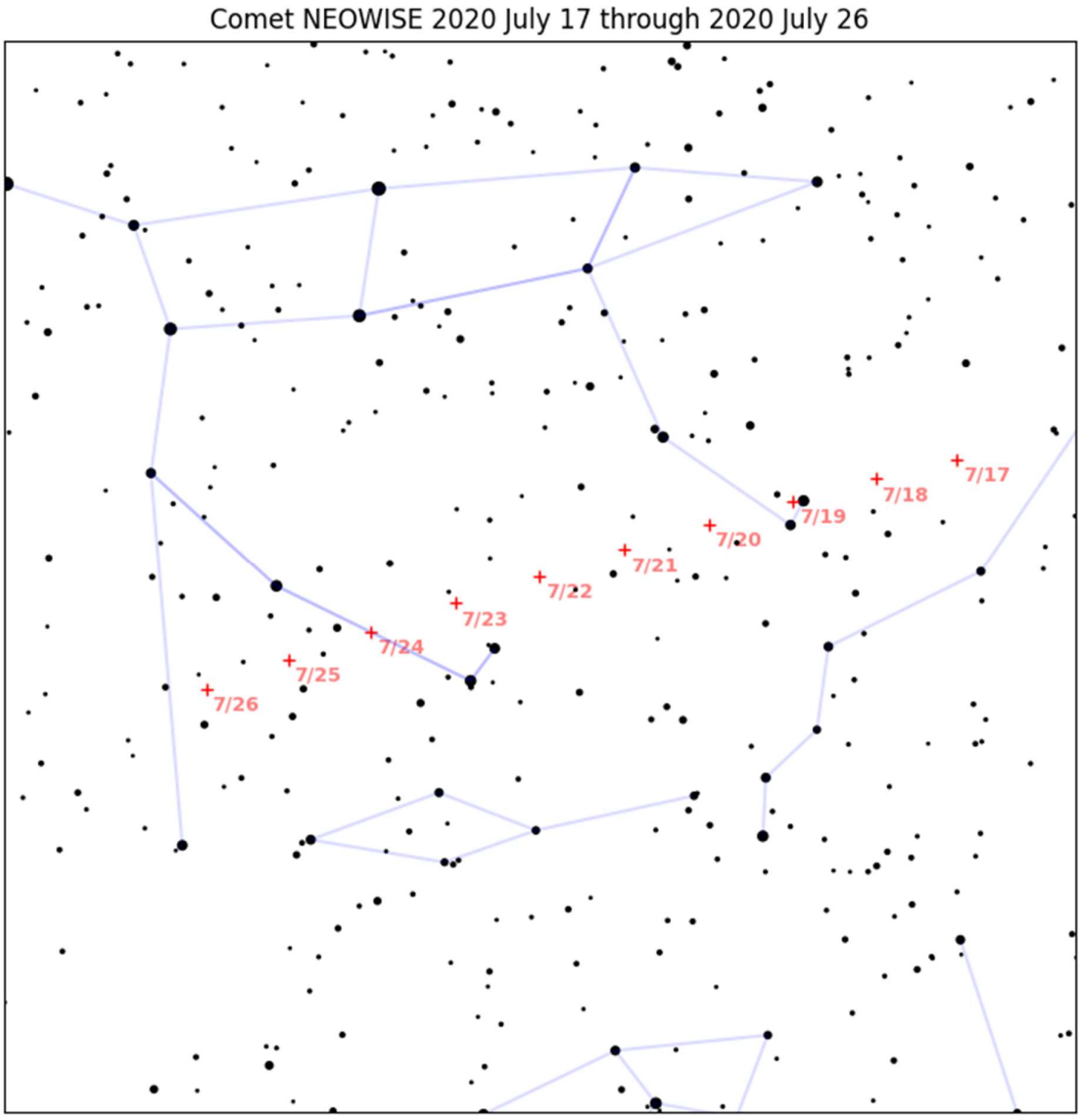
```
year, month, day, hour, minute, second = t.utc
month = month.astype(int)
day = day.astype(int)

apparent = (earth + observer).at(t).observe(venus).apparent()
alt, az, distance = apparent.altaz()
x, y = az.degrees, alt.degrees
m = planetary_magnitude(apparent)

# Convert magnitude to marker size, remembering that smaller magnitude
# numbers mean a brighter Venus (and thus a larger marker).

maxmag = max(m)
minmag = min(m)
size = 40 - 30 * (m - minmag) / (maxmag - minmag)

# Start with a smooth curve tracing Venus's motion.

fig, ax = plt.subplots(figsize=[9, 3])
ax.plot(x, y, c='#fff6', zorder=1)

# Next, put a circle representing Venus on the 1st of the month and on
# every fifth day after that.  (Except for the 30th, which would sit too
# close to the 1st of the following month.)

fives = (day % 5 == 1) & (day < 30)
ax.scatter(x[fives], y[fives], size[fives], 'white',
        edgecolor='black', linewidth=0.25, zorder=2)

# Put day and month labels off to the sides of the curve.
```

```
offset_x, offset_y = 10, 8

for i in np.flatnonzero(fives):
    if i == 0:
        continue  # We can't compute dx/dy with no previous point.

    # Build a unit vector pointing in the direction Venus is traveling.

    day_i = day[i]
    xi = x[i]
    yi = y[i]
    dx = xi - x[i-1]
    dy = yi - y[i-1]
    length = np.sqrt(dx*dx + dy*dy)
    dx /= length
    dy /= length

    # Offset the text at a right angle to the direction of travel.

    side = 'right' if (year[i], month[i]) < (2021, 10) else 'left'
    if side == 'left':
        xytext = - offset_x*dy, offset_y*dx
    else:
        xytext = offset_x*dy, - offset_y*dx

    # Label the dates 1, 11, and 21.

    if day_i in (1, 11, 21):
        ax.annotate(day_i, (xi, yi), c='white', ha='center', va='center',
                    textcoords='offset points', xytext=xytext, size=8)
```

```
    # On the 15th of each month, put the month name.

    if day_i == 16:
        name = MONTH_NAMES[month[i]]
        ax.annotate(name, (xi, yi), c='white', ha='center', va='center',
                textcoords='offset points', xytext=2.2 * np.array(xytext))

# Finally, some decorations.

points = 'N NE E SE S SW W NW'.split()
for i, name in enumerate(points):
    xy = 45 * i, 1
    ax.annotate(name, xy, c='white', ha='center', size=12, weight='bold')

ax.set(
    aspect=1.0,
    title='Venus at sunset for 40°N latitude, April 2021 – January 2022',
    xlabel='Azimuth (°)',
    ylabel='Altitude (°)',
    xlim=(195, 300),
    ylim=(0, max(y) + 10.0),
    xticks=np.arange(210, 300, 15),
)

sky = LinearSegmentedColormap.from_list('sky', ['black', 'blue'])
extent = ax.get_xlim() + ax.get_ylim()
ax.imshow([[0,0], [1,1]], cmap=sky, interpolation='bicubic', extent=extent)

fig.savefig('venus_evening_chart.png')
```

Menggambar peta pencari komet NEOWISE

Berikut adalah skrip mandiri yang menggabungkan empat sumber data berbeda — ephemeris planet, basis data orbit komet, katalog bintang besar, dan diagram konstelasi — untuk memetakan lintasan Komet NEOWISE melintasi Ursa Major selama satu minggu di bulan Juli 2020:



Kodenya mencakup banyak keputusan desain dan penyesuaian presentasi yang mungkin ingin Anda sesuaikan untuk proyek Anda sendiri. Gunakan skrip ini sebagai titik awal:

```
import numpy as np

from matplotlib import pyplot as plt

from matplotlib.collections import LineCollection


from skyfield.api import Star, load
```

```
from skyfield.constants import GM_SUN_Pitjeva_2005_km3_s2 as GM_SUN
from skyfield.data import hipparcos, mpc, stellarium
from skyfield.projections import build_stereographic_projection

# The comet is plotted on several dates `t_comet`.  But the stars only
# need to be drawn once, so we take the middle comet date as the single
# time `t` we use for everything else.

ts = load.timescale()
t_comet = ts.utc(2020, 7, range(17, 27))
t = t_comet[len(t_comet) // 2]  # middle date

# An ephemeris from the JPL provides Sun and Earth positions.

eph = load('de421.bsp')
sun = eph['sun']
earth = eph['earth']

# The Minor Planet Center data file provides the comet orbit.

with load.open(mpc.COMET_URL) as f:
    comets = mpc.load_comets_dataframe(f)

comets = (comets.sort_values('reference')
          .groupby('designation', as_index=False).last()
          .set_index('designation', drop=False))

row = comets.loc['C/2020 F3 (NEOWISE)']
comet = sun + mpc.comet_orbit(row, ts, GM_SUN)

# The Hipparcos mission provides our star catalog.
```

```
with load.open(hipparcos.URL) as f:
    stars = hipparcos.load_dataframe(f)

# And the constellation outlines come from Stellarium.  We make a list
# of the stars at which each edge stars, and the star at which each edge
# ends.

url = ('https://raw.githubusercontent.com/Stellarium/stellarium/master'
       '/skycultures/modern_st/constellationship.fab')

with load.open(url) as f:
    constellations = stellarium.parse_constellations(f)

edges = [edge for name, edges in constellations for edge in edges]
edges_star1 = [star1 for star1, star2 in edges]
edges_star2 = [star2 for star1, star2 in edges]

# We will center the chart on the comet's middle position.

center = earth.at(t).observe(comet)
projection = build_stereographic_projection(center)
field_of_view_degrees = 45.0
limiting_magnitude = 7.0

# Now that we have constructed our projection, compute the x and y
# coordinates that each star and the comet will have on the plot.

star_positions = earth.at(t).observe(Star.from_dataframe(stars))
stars['x'], stars['y'] = projection(star_positions)
```

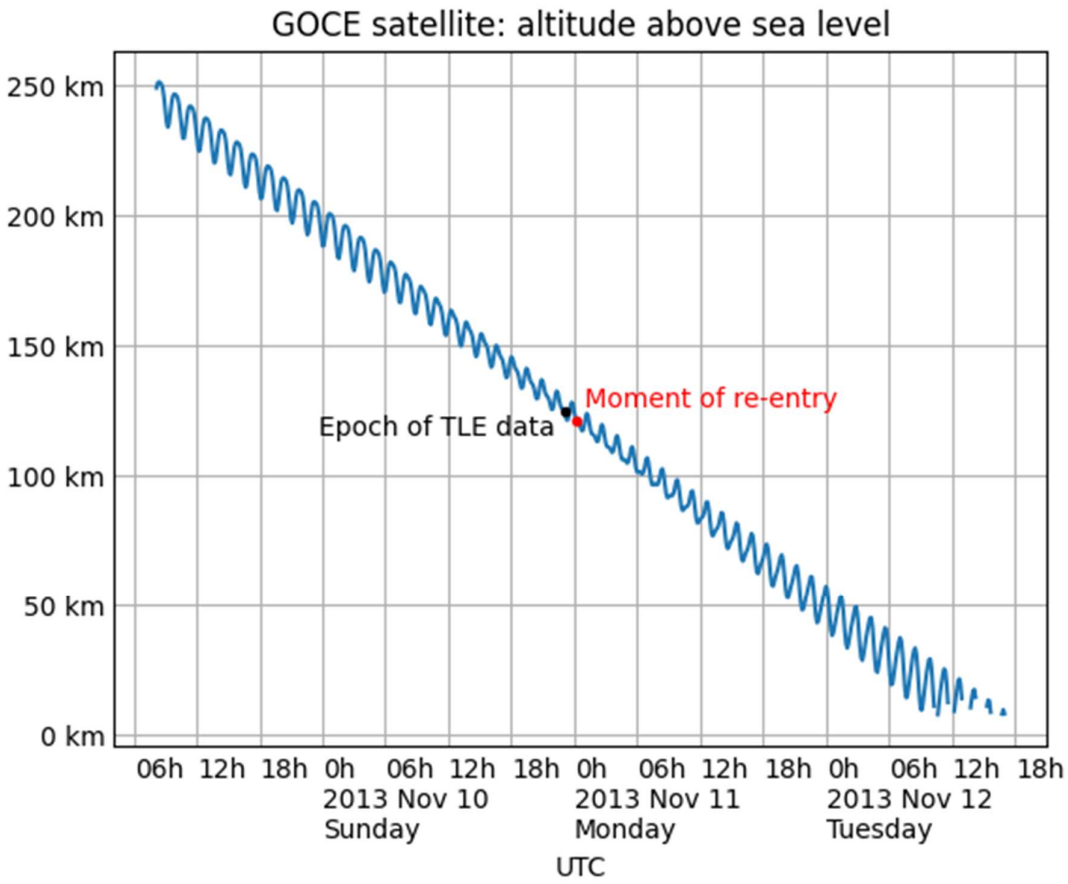
```
comet_x, comet_y = projection(earth.at(t_comet).observe(comet))

# Create a True/False mask marking the stars bright enough to be
# included in our plot.  And go ahead and compute how large their
# markers will be on the plot.

bright_stars = (stars.magnitude <= limiting_magnitude)
magnitude = stars['magnitude'][bright_stars]
marker_size = (0.5 + limiting_magnitude - magnitude) ** 2.0

# The constellation lines will each begin at the x,y of one star and end
# at the x,y of another.  We have to "rollaxis" the resulting coordinate
# array into the shape that matplotlib expects.

xy1 = stars[['x', 'y']].loc[edges_star1].values
xy2 = stars[['x', 'y']].loc[edges_star2].values
lines_xy = np.rollaxis(np.array([xy1, xy2]), 1)

# Time to build the figure!

fig, ax = plt.subplots(figsize=[9, 9])

# Draw the constellation lines.

ax.add_collection(LineCollection(lines_xy, colors='#00f2'))

# Draw the stars.

ax.scatter(stars['x'][bright_stars], stars['y'][bright_stars],
        s=marker_size, color='k')
```

```
# Draw the comet positions, and label them with dates.

comet_color = '#f00'
offset = 0.002

ax.plot(comet_x, comet_y, '+', c=comet_color, zorder=3)

for xi, yi, tstr in zip(comet_x, comet_y, t_comet.utc_strftime('%m/%d')):
    tstr = tstr.lstrip('0')
    text = ax.text(xi + offset, yi - offset, tstr, color=comet_color,
                   ha='left', va='top', fontsize=9, weight='bold', zorder=-1)
    text.set_alpha(0.5)

# Finally, title the plot and set some final parameters.

angle = np.pi - field_of_view_degrees / 360.0 * np.pi
limit = np.sin(angle) / (1.0 - np.cos(angle))

ax.set_xlim(-limit, limit)
ax.set_ylim(-limit, limit)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
ax.set_aspect(1.0)
ax.set_title('Comet NEOWISE {} through {}'.format(
    t_comet[0].utc_strftime('%Y %B %d'),
    t_comet[-1].utc_strftime('%Y %B %d'),
))

# Save.
fig.savefig('neowise-finder-chart.png', bbox_inches='tight')
```

Jika Anda memilih mesin rendering yang berbeda, bukan matplotlib yang sudah lama ada tetapi agak rumit dan rumit , tentu saja panggilan plotting yang Anda buat akan sangat berbeda. Namun, pemuatan dan pemfilteran data dasar akan sama, jadi semoga skrip tersebut tetap membantu Anda memulai menargetkan pustaka plotting yang lebih modern.

Merencanakan ketinggian satelit selama masuk kembali

Berikut ini adalah ketinggian satelit yang semakin menurun seiring dengan berkurangnya orbitnya dan kembali memasuki atmosfer di atas Samudra Pasifik:



Kode untuk menghasilkan diagram menggunakan matplotlib , termasuk tanda centang khusus yang didasarkan pada tanggal, adalah:

```
import numpy as np

from matplotlib import pyplot as plt

from matplotlib.dates import HourLocator, DateFormatter


from skyfield.api import load, EarthSatellite


# Labels for both date and hour on the x axis, and km on y.
```

```
def label_dates_and_hours(axes):
    axes.xaxis.set_major_locator(HourLocator([0]))
    axes.xaxis.set_minor_locator(HourLocator([0, 6, 12, 18]))
    axes.xaxis.set_major_formatter(DateFormatter('0h\n%Y %b %d\n%A'))
    axes.xaxis.set_minor_formatter(DateFormatter('%Hh'))
    for label in ax.xaxis.get_ticklabels(which='both'):
        label.set_horizontalalignment('left')
    axes.yaxis.set_major_formatter('{x:.0f} km')
    axes.tick_params(which='both', length=0)

# Load the satellite's final TLE entry.

sat = EarthSatellite(
    '1 34602U 09013A   13314.96046236  .14220718  20669-5  50412-4 0   930',
    '2 34602 096.5717 344.5256 0009826 296.2811 064.0942 16.58673376272979',
    'GOCE',
)

# Build the time range `t` over which to plot, plus other values.

ts = load.timescale()
t = ts.tt_jd(np.arange(sat.epoch.tt - 2.0, sat.epoch.tt + 2.0, 0.005))
reentry = ts.utc(2013, 11, 11, 0, 16)
earth_radius_km = 6371.0

# Compute geocentric positions for the satellite.

g = sat.at(t)
valid = [m is None for m in g.message]
```

```
# Start a new figure.
fig, ax = plt.subplots()

# Draw the blue curve.
x = t.utc_datetime()
y = np.where(valid, g.distance().km - earth_radius_km, np.nan)
ax.plot(x, y)

# Label the TLE epoch.
x = sat.epoch.utc_datetime()
y = sat.at(sat.epoch).distance().km - earth_radius_km
ax.plot(x, y, 'k.')
ax.text(x, y - 9, 'Epoch of TLE data ', ha='right')

# Label the official moment of reentry.
x = reentry.utc_datetime()
y = sat.at(reentry).distance().km - earth_radius_km
ax.plot(x, y, 'r.')
ax.text(x, y + 6, ' Moment of re-entry', c='r')

# Grid lines and labels.
ax.grid(which='both')
ax.set(title='GOCE satellite: altitude above sea level', xlabel='UTC')
label_dates_and_hours(ax)

# Render the plot to a PNG file.
fig.savefig('goce-reentry.png', bbox_inches='tight')
```

Mengunduh dan Menggunakan File Data

Pertama kali Anda menjalankan program Skyfield, biasanya program tersebut akan mengunduh satu atau beberapa berkas data dari Internet yang menyediakan data tentang orbit planet atau satelit — satu berkas untuk setiap panggilan yang dilakukan program ke load()rutinitas Skyfield. Jika program tersebut terpasang ke terminal, maka bilah kemajuan sederhana akan ditampilkan saat Skyfield mengunduh setiap berkas.

```
from skyfield.api import load

planets = load('de421.bsp')

print('Ready')
```

```
[################################] 100% de421.bsp

Ready
```

Namun, saat kedua kalinya Anda menjalankan program tersebut, program akan menemukan berkas data yang sudah ada di direktori saat ini. Dalam hal ini, program akan menggunakan berkas pada disk tanpa memerlukan akses ke Internet:

```
Ready
```

Sebagian besar program akan berjalan dengan baik menggunakan load()rutin bawaan yang disediakan dalam skyfield.apimodul. Namun, program lain mungkin ingin membuat pemuatnya sendiri sehingga mereka memiliki kesempatan untuk mengesampingkan perilaku bawaannya.

Menentukan direktori unduhan

Objek default load()menyimpan berkas langsung ke direktori kerja Anda saat ini — biasanya folder tempat Anda meluncurkan program Skyfield.

Namun, Anda dapat membuat loader sendiri yang menggunakan direktori berbeda. Cukup buat instance Loaderdengan jalur ke direktori tempat Anda ingin menyimpan file data.

```
from skyfield.api import Loader

load = Loader('~/skyfield-data')
```

Sekarang semua load()operasi Anda akan menargetkan direktori tersebut. Perhatikan bahwa tidak ada batasan berapa banyak Loaderobjek yang dapat Anda buat — silakan buat satu objek untuk menyimpan berkas waktu, satu lagi untuk berkas ephemeris, dan satu lagi untuk TLE satelit Bumi, jika itu memudahkan Anda untuk mengatur semuanya!

Mematikan bilah kemajuan

Jika Anda merasa terganggu dengan adanya bilah kemajuan yang ditampilkan di layar setiap kali Skyfield mengunduh sebuah berkas — yang mungkin menjadi masalah khususnya saat Anda menggunakan Skyfield dalam aplikasi yang lebih besar — Anda dapat menonaktifkan bilah kemajuan tersebut dengan membuat yang Loaderverbositasnya ditetapkan ke salah.

```
from skyfield.api import Loader

load = Loader('~/skyfield-data', verbose=False)
```

Perhitungan Almanak

Untuk membantu Anda menavigasi, berikut adalah topik yang dibahas di halaman ini:

Berikut adalah impor dan objek yang akan menggerakkan contoh di bawah ini:

```
from skyfield import almanac
from skyfield.api import N, S, E, W, load, wgs84


ts = load.timescale()
eph = load('de421.bsp')
sun = eph['Sun']
bluffton = wgs84.latlon(40.8939 * N, 83.8917 * W)
observer = eph['Earth'] + bluffton
```

Perhatikan perbedaannya:

- bluffton— GeographicPosition menghitung vektor dari pusat Bumi ke lokasi geografis tertentu. Kecuali jika ketinggian ditentukan, panjang vektor akan menjadi radius Bumi, yang bervariasi dari 6.357 km di kutub hingga 6.378 km di ekuator.
- observer— ini menghitung vektor yang jauh lebih besar dari pusat Tata Surya ke posisi geografis.

Rutin almanak biasanya perlu melewati vektor penuh ini, bukan hanya posisi geografis. Bergantung pada musim (Bumi paling dekat dengan Matahari pada awal Januari), panjangnya akan mendekati 1 au.

Waspadalah bahwa perhitungan almanak bisa mahal. Karena rutinitas almanak mencari maju mundur dalam rentang waktu tertentu, mereka harus menghitung ulang posisi Bumi dan benda-benda lain secara berulang.

Membulatkan waktu ke menit terdekat

Jika Anda membandingkan hasil almanak dengan sumber resmi seperti Observatorium Angkatan Laut Amerika Serikat , waktu yang dicetak sering kali berbeda karena hasil Observatorium Angkatan Laut dibulatkan ke menit terdekat — waktu apa pun dengan :30atau lebih detik di akhir dibulatkan ke menit berikutnya.

Metode Skyfield utc_strftime() melakukan pembulatan ini secara otomatis jika Anda tidak meminta detik untuk ditampilkan:

```
t = ts.utc(2023, 8, 10, 6, 21, 45.9)


print('Microseconds:', t.utc_strftime('%Y-%m-%d %H:%M:%S.%f'))

print('Nearest second:', t.utc_strftime('%Y-%m-%d %H:%M:%S'))

print('Nearest minute:', t.utc_strftime('%Y-%m-%d %H:%M'))

Microseconds: 2023-08-10 06:21:45.900000

Nearest second: 2023-08-10 06:21:46

Nearest minute: 2023-08-10 06:22
```

Namun perlu diingat bahwa pembulatan tidak akan terjadi secara otomatis jika Anda melakukan pemformatan sendiri menggunakan objek bawaan Python datetime. Misalnya, jika Anda berhenti dengan %M, maka detik diabaikan begitu saja, alih-alih digunakan untuk pembulatan:

```
dt = t.utc_datetime()

print(dt.strftime('%Y-%m-%d %H:%M'))

2023-08-10 06:21
```

Untuk memperbaiki masalah dan membulatkan Python datetimeke menit terdekat, coba tambahkan 30 detik secara manual ke waktu sebelum menampilkannya:

```
from datetime import timedelta


    def nearest_minute(dt):
      return (dt + timedelta(seconds=30)).replace(second=0, microsecond=0)


    dt = nearest_minute(t.utc_datetime())
    print(dt.strftime('%Y-%m-%d %H:%M'))
    2023-08-10 06:22
```

Hasilnya kemudian harus sesuai dengan tabel yang dihasilkan oleh USNO.

Naik dan turun

Skyfield can compute when a given body rises and sets for an observer at the Earth's surface. The routines are designed for bodies at least as far away as Moon, that rise and set about once a day, so it will be caught off-guard if you pass it something fast like an Earth satellite. For that case, see Finding when a satellite rises and sets.

Sunrise and Sunset

Skyfield uses the official definition of sunrise and sunset from the United States Naval Observatory, which defines them as the moment when the center of the sun is 50 arcminutes below the horizon, to account for both the average solar radius of 16 arcminutes and for roughly 34 arcminutes of atmospheric refraction at the horizon. Here's how to ask for the sunrises between a given start and end time:

```
    t0 = ts.utc(2018, 9, 12, 4)
    t1 = ts.utc(2018, 9, 14, 4)


    t, y = almanac.find_risings(observer, sun, t0, t1)
    print(t.utc_iso(' '))
    print(y)
    ['2018-09-12 11:13:12Z', '2018-09-13 11:14:12Z']
    [ True  True]
```

And here's how to ask for the sunsets:

```
    t, y = almanac.find_settings(observer, sun, t0, t1)
    print(t.utc_iso(' '))
    print(y)
    ['2018-09-12 23:49:38Z', '2018-09-13 23:47:56Z']
```

```
[ True  True]
```

Normally every value in the second array will be True, indicating that a rising or setting was successfully detected. See the next section for an example where the value is False.

Detecting polar day and polar night

In the Arctic and Antarctic, there will be summer days when the sun never sets, and winter days when the sun never rises. On such days the second array returned by the rising and setting routines will have the value False instead of True. The time returned will instead be that of transit, whether that takes place above or below the horizon. For example:

```
harra_sweden = wgs84.latlon(67.4066 * N, 20.0997 * E)
harra_observer = eph['Earth'] + harra_sweden


t0 = ts.utc(2022, 12, 18)
t1 = ts.utc(2022, 12, 26)
t, y = almanac.find_risings(harra_observer, sun, t0, t1)


alt, az, dist = harra_observer.at(t).observe(sun).apparent().altaz()


for ti, yi, alti in zip(t.utc_iso(' '), y, alt.degrees):
    print('{} {:5} {:.4f}°'.format(ti, str(yi), alti))
2022-12-18 10:22:54Z True  -0.8333°
2022-12-19 10:29:21Z True  -0.8333°
2022-12-20 10:37:06Z False -0.8387°
2022-12-21 10:37:36Z False -0.8464°
2022-12-22 10:38:06Z False -0.8461°
2022-12-23 10:38:36Z False -0.8380°
2022-12-24 10:31:28Z True  -0.8333°
2022-12-25 10:26:08Z True  -0.8333°
```

This output shows that right around the winter solstice, there are four days on which the Sun never quite reaches the horizon, but is at least a few fractions of a degree below the altitude of -0.8333° that would qualify for the USNO definition of sunrise. So Skyfield instead returns the moment when the Sun is closest to the horizon, with the accompanying value False.

The value False also accompanies a lower transit when the sun is up for an entire 24-hour day and never touches the horizon and sets.

```
utqiagvik_alaska = wgs84.latlon(71.2906 * N, 156.7886 * W)

utqiagvik_observer = eph['Earth'] + utqiagvik_alaska


t0 = ts.utc(2023, 6, 21)

t1 = ts.utc(2023, 6, 22)

t, y = almanac.find_settings(utqiagvik_observer, sun, t0, t1)


alt, az, dist = utqiagvik_observer.at(t).observe(sun).apparent().altaz()


for ti, yi, alti in zip(t.utc_iso(' '), y, alt.degrees):

   print('{} {:5} {:.4f}°'.format(ti, str(yi), alti))

2023-06-21 10:28:55Z False 4.7265°
```

The Sun at this Alaska location doesn't reach the horizon on June 21, and so instead of triumphantly returning the time at which the sun reached -0.8333°, Skyfield returns the moment of anti-transit when the Sun was at its lowest and furthest north — standing a full 4.7° above the horizon.

Moonrise and moonset

Skyfield uses the official definition of moonrise and moonset from the United States Naval Observatory: the moment when the top edge of the Moon is exactly 34 arcminutes below the horizon, to correct for atmospheric refraction.

```
moon = eph['Moon']

t0 = ts.utc(2023, 12, 27)

t1 = ts.utc(2023, 12, 29)


t, y = almanac.find_risings(observer, moon, t0, t1)

print('Moonrises (UTC):', t.utc_iso(' '))


t, y = almanac.find_settings(observer, moon, t0, t1)

print('Moonsets (UTC):', t.utc_iso(' '))

Moonrises (UTC): ['2023-12-27 22:40:11Z', '2023-12-28 23:43:48Z']

Moonsets (UTC): ['2023-12-27 13:54:47Z', '2023-12-28 14:39:33Z']
```

Read the previous section to learn about the Boolean array y.

Planet rising and setting

The rising and setting routines also work for planets. To account for atmospheric refraction under typical conditions, Skyfield will look for the moment when the center of the planet's disc is exactly 34 arcminutes below the horizon.

```
t0 = ts.utc(2020, 2, 1)
t1 = ts.utc(2020, 2, 3)

t, y = almanac.find_risings(observer, eph['Mars'], t0, t1)
print('Mars rises:', t.utc_iso(' '))

t, y = almanac.find_settings(observer, eph['Mars'], t0, t1)
print('Mars sets: ', t.utc_iso(' '))
```

```
Mars rises: ['2020-02-01 09:29:16Z', '2020-02-02 09:28:34Z']
Mars sets:  ['2020-02-01 18:42:57Z', '2020-02-02 18:41:41Z']
```

Read the previous section Detecting polar day and polar night to learn about the Boolean array y.

Computing your own refraction angle

Atmospheric refraction makes bodies at the horizon appear slightly higher than they would be otherwise. That's why Skyfield doesn't wait until a body's altazimuth coordinates have reached 0.0° altitude to proclaim that it has risen. Instead, as explained in the previous sections, Skyfield goes ahead and counts a body as risen as soon as it has reached −0°34′ altitude.

But refraction varies with atmospheric conditions. To supply your own estimate of the altitude of the visible horizon, pass the optional horizon_degrees argument to find_risings() and find_settings() with the target altitude angle you want to use instead.

To help you build an estimate, Skyfield provides a small function that takes an altitude angle, the temperature, and the pressure, and returns a standard United States Naval Observatory estimate for the angle of atmospheric refraction. Here's an example of how to use it:

```
from skyfield.earthlib import refraction

r = refraction(0.0, temperature_C=15.0, pressure_mbar=1030.0)
print('Refraction at the horizon: %.2f arcminutes\n' % (r * 60.0))

t, y = almanac.find_risings(observer, eph['Mars'], t0, t1,
```

```
                    horizon_degrees=-r)
print('Mars rises:', t.utc_iso(' '))


t, y = almanac.find_settings(observer, eph['Mars'], t0, t1,
                    horizon_degrees=-r)
print('Mars sets: ', t.utc_iso(' '))
```

Refraction at the horizon: 34.53 arcminutes

Mars rises: ['2020-02-01 09:29:13Z', '2020-02-02 09:28:30Z']

Mars sets: ['2020-02-01 18:43:00Z', '2020-02-02 18:41:45Z']

If you want to account for both atmospheric refraction and also the radius of the target body, then simply supply an even more negative value for horizon_degrees that combines the correction for refraction with the radius of the body's visible disc in the sky.

Elevated vantage points

Rising and setting predictions usually assume a flat local horizon that does not vary with elevation. Yes, Denver is the Mile High City, but it sees the sun rise against a local horizon that's also a mile high. Since the city's high elevation is matched by the high elevation of the terrain around it, the horizon winds up in the same place it would be for a city at sea level.

But sometimes you need to account for a viewpoint's local prominence above the surrounding terrain. Some observatories, for example, are located on mountaintops that are much higher than the terrain that forms their horizon. And Earth satellites can be hundreds of kilometers above the surface of the Earth that produces their sunrises and sunsets.

You can account for an observer's prominence above their horizon's terrain by setting an artificially negative value for horizon_degrees. If we consider the Earth to be approximately a sphere, then we can use a bit of trigonometry to estimate the position of the horizon for an observer at altitude:

```
from numpy import arccos
from skyfield.units import Angle


# When does the Sun rise in the ionosphere's F-layer, 300km up?


altitude_m = 300e3
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_radius_m + altitude_m)
h = Angle(radians = -arccos(side_over_hypotenuse))
```

```
print('The horizon from 300km up is at %.2f degrees' % h.degrees)


solar_radius_degrees = 0.25
t, y = almanac.find_risings(
   observer, sun, t0, t1,
   horizon_degrees=h.degrees - solar_radius_degrees,
)
print('At', t[0].utc_iso(' '), 'the limb of the Sun crests the horizon')
```

```
The horizon from 300km up is at -17.24 degrees
At 2020-02-01 11:14:57Z the limb of the Sun crests the horizon
```

Beware a possible source of confusion: some people use the word *altitude* for a geographic site's elevation in meters above sea level. And other people — primarily Earth satellite folks — use the term *elevation* for degrees above or below the horizon, which Skyfield instead calls *altitude* (because that's what the syllable *alt-* stands for in the name *altazimuth coordinate system*).

When a right ascension and declination rises and sets

If you are interested in finding the times when a fixed point in the sky rises and sets, simply create a star object with the coordinates of the position you are interested in (see Stars and Distant Objects) and then call the routines described above. Here, for example, is the rising time for the Galactic Center:

```
from skyfield.api import Star


galactic_center = Star(ra_hours=(17, 45, 40.04),
                       dec_degrees=(-29, 0, 28.1))


t, y = almanac.find_risings(observer, galactic_center, t0, t1)
print('The Galactic Center rises at', t[0].utc_iso(' '))
```

```
The Galactic Center rises at 2020-02-01 10:29:00Z
```

The Seasons

Create a start time and an end time to ask for all of the equinoxes and solstices that fall in between.

```
t0 = ts.utc(2018, 1, 1)
t1 = ts.utc(2018, 12, 31)
```

```
t, y = almanac.find_discrete(t0, t1, almanac.seasons(eph))


for yi, ti in zip(y, t):
    print(yi, almanac.SEASON_EVENTS[yi], ti.utc_iso(' '))
```

```
0 Vernal Equinox 2018-03-20 16:15:27Z
1 Summer Solstice 2018-06-21 10:07:18Z
2 Autumnal Equinox 2018-09-23 01:54:06Z
3 Winter Solstice 2018-12-21 22:22:44Z
```

The result t will be an array of times, and y will be 0 through 3 for the Vernal Equinox through the Winter Solstice.

If you or some of your users live in the Southern Hemisphere, you can use the SEASON_EVENTS_NEUTRAL array. Instead of naming specific seasons, it names the equinoxes and solstices by the month in which they occur — so the March Equinox, for example, is followed by the June Solstice.

Phases of the Moon

The phases of the Moon are the same for everyone on Earth, so you don't need to specify the longitude and latitude of your location. Simply ask for the current phase of the Moon. The return value is an angle where 0° is New Moon, 90° is First Quarter, 180° is Full Moon, and 270° is Last Quarter:

```
t = ts.utc(2020, 11, 19)
phase = almanac.moon_phase(eph, t)
print('Moon phase: {:.1f} degrees'.format(phase.degrees))
```

```
Moon phase: 51.3 degrees
```

Or you can have Skyfield search over a range of dates for the moments when the Moon reaches First Quarter, Full Moon, Last Quarter, and New Moon:

```
t0 = ts.utc(2018, 9, 1)
t1 = ts.utc(2018, 9, 10)
t, y = almanac.find_discrete(t0, t1, almanac.moon_phases(eph))


print(t.utc_iso())
print(y)
print([almanac.MOON_PHASES[yi] for yi in y])
```

```
['2018-09-03T02:37:24Z', '2018-09-09T18:01:28Z']
```

```
[3 0]

['Last Quarter', 'New Moon']
```

The result t will be an array of times, and y will be a corresponding array of Moon phases with 0 for New Moon and 3 for Last Quarter. You can use the array MOON_PHASES to retrieve names for each phase.

Lunar Nodes

The Moon's ascending node and descending node are the moments each lunar month when the Moon crosses the plane of Earth's orbit and eclipses are possible.

```
t0 = ts.utc(2020, 4, 22)

t1 = ts.utc(2020, 5, 22)

t, y = almanac.find_discrete(t0, t1, almanac.moon_nodes(eph))


print(t.utc_iso())

print(y)

print([almanac.MOON_NODES[yi] for yi in y])

['2020-04-27T17:54:17Z', '2020-05-10T09:01:42Z']

[1 0]

['ascending', 'descending']
```

Opposition and Conjunction

The moment at which a planet is in opposition with the Sun or in conjunction with the Sun is when their ecliptic longitudes are at 0° or 180° difference.

```
t0 = ts.utc(2019, 1, 1)

t1 = ts.utc(2021, 1, 1)

f = almanac.oppositions_conjunctions(eph, eph['mars'])

t, y = almanac.find_discrete(t0, t1, f)


print(t.utc_iso())

print(y)

['2019-09-02T10:42:26Z', '2020-10-13T23:25:55Z']

[0 1]
```

The result t will be an array of times, and y will be an array of integers indicating which half of the sky the body has just entered: 0 means the half of the sky west of the Sun along the

ecliptic, and 1 means the half of the sky east of the Sun. This means different things for different bodies:

- For the outer planets Mars, Jupiter, Saturn, Uranus, and all other bodies out beyond our orbit, 0 means the moment of conjunction with the Sun and 1 means the moment of opposition.
- Because the Moon moves eastward across our sky relative to the Sun, not westward, the output is reversed compared to the outer planets: 0 means the moment of opposition or Full Moon, while 1 means the moment of conjunction or New Moon.
- The inner planets Mercury and Venus only ever experience conjunctions with the Sun from our point of view, never oppositions, with 0 indicating an inferior conjunction and 1 a superior conjunction.

Meridian Transits

Every day the Earth's rotation swings the sky through nearly 360°, leaving the celestial poles stationary while bringing each star and planet in turn across your *meridian* — the line of right ascension in the sky above you that runs from the South Pole to the North Pole through your local zenith.

You can ask Skyfield for the times at which a body crosses your meridian:

```
t0 = ts.utc(2020, 11, 6)

t1 = ts.utc(2020, 11, 8)

t = almanac.find_transits(observer, eph['Mars'], t0, t1)


print(t.utc_strftime('%Y-%m-%d %H:%M'))

['2020-11-06 03:32', '2020-11-07 03:28']
```

Observers often think of transit as the moment when an object is highest in the sky, but that's only roughly true. At very high precision, if the body has any north or south velocity then its moment of highest altitude will be slightly earlier or later.

Bodies near the poles are exceptions to the general rule that a body is visible at transit but below the horizon at antitransit. For a body that's circumpolar from your location, transit and antitransit are both moments of visibility, when it stands above and below the pole. And objects close to the opposite pole will always be below the horizon, even as they invisibly transit your line of longitude down below your horizon.

Legacy transit routine

Skyfield has an older mechanism for detecting transits that isn't as fast as the function described in the previous section, but it also returns the moments of anti-transit, when a body crosses the line of right ascension that crosses your local nadir:

```
t0 = ts.utc(2020, 11, 6)

t1 = ts.utc(2020, 11, 7)
```

```
f = almanac.meridian_transits(eph, eph['Mars'], bluffton)
t, y = almanac.find_discrete(t0, t1, f)

print(t.utc_strftime('%Y-%m-%d %H:%M'))
print(y)
print([almanac.MERIDIAN_TRANSITS[yi] for yi in y])
['2020-11-06 03:32', '2020-11-06 15:30']
[1 0]
['Meridian transit', 'Antimeridian transit']
```

Some astronomers call these moments “upper culmination” and “lower culmination” instead.

Legacy rising and setting routines

In case you are maintaining older code, versions of Skyfield before 1.47 could only compute sunrises and sunsets with a routine that was much slower than the functions described above. It also tended to miss sunrises and sunsets in the Arctic and Antarctic. Here’s how it was called:

```
f = almanac.sunrise_sunset(eph, bluffton)
t, y = almanac.find_discrete(t0, t1, f)

print(t.utc_iso())
print(y)
['2020-11-06T12:12:46Z', '2020-11-06T22:25:10Z']
[1 0]
```

The result `t` will be an array of times, and `y` will be 1 if the sun rises at the corresponding time and 0 if it sets.

Another old routine [risings_and_settings()](#) worked the same way, but for general targets like planets.

```
f = almanac.risings_and_settings(eph, eph['Mars'], bluffton)
t, y = almanac.find_discrete(t0, t1, f)
print(t.utc_iso())
print(y)
['2020-11-06T09:50:55Z', '2020-11-06T21:08:55Z']
[0 1]
```

Twilight

The routine dark_twilight_day() returns a separate code for each of the phases of twilight:

1. Dark of night.
2. Astronomical twilight.
3. Nautical twilight.
4. Civil twilight.
5. Daytime.

You can find a full example of its use at the When will it get dark tonight?.

Solar terms

The solar terms are widely used in East Asian calendars.

```
from skyfield import almanac_east_asia as almanac_ea


t0 = ts.utc(2019, 12, 1)
t1 = ts.utc(2019, 12, 31)
t, tm = almanac.find_discrete(t0, t1, almanac_ea.solar_terms(eph))


for tmi, ti in zip(tm, t):
    print(tmi, almanac_ea.SOLAR_TERMS_ZHS[tmi], ti.utc_iso(' '))
```

```
17 大雪 2019-12-07 10:18:28Z
18 冬至 2019-12-22 04:19:26Z
```

The result t will be an array of times, and y will be integers in the range 0–23 which are each the index of a solar term. Localized names for the solar terms in different East Asia languages are provided as SOLAR_TERMS_JP for Japanese, SOLAR_TERMS_VN for Vietnamese, SOLAR_TERMS_ZHT for Traditional Chinese, and (as shown above) SOLAR_TERMS_ZHS for Simplified Chinese.

Lunar eclipses

Skyfield can find the dates of lunar eclipses.

```
from skyfield import eclipselib


t0 = ts.utc(2019, 1, 1)
t1 = ts.utc(2020, 1, 1)
t, y, details = eclipselib.lunar_eclipses(t0, t1, eph)
```

```
for ti, yi in zip(t, y):
    print(ti.utc_strftime('%Y-%m-%d %H:%M'),
          'y={}'.format(yi),
          eclipselib.LUNAR_ECLIPSES[yi])
```

```
2019-01-21 05:12 y=2 Total
2019-07-16 21:31 y=1 Partial
```

Note that any eclipse forecast is forced to make arbitrary distinctions when eclipses fall very close to the boundary between the categories “partial”, “penumbral”, and “total”. Skyfield searches for lunar eclipses using the techniques described in the *Explanatory Supplement to the Astronomical Almanac.* Here is its current behavior:

- Skyfield currently finds every one of the 3,642 lunar eclipses listed for the years AD 1000–2500 in NASA’s Five Millennium Canon of Lunar Eclipses by Espenak and Meeus.
- But some slight disagreements are inevitable, because Skyfield uses a modern ephemeris for Earth and Moon positions, while the *Supplement* used the old VSOP87 theory. In 8 cases over the years AD 1000–2500 (around 0.2% of the eclipses listed), Skyfield disagrees with the *Canon* about whether an eclipse was partial or total. And on 1571 July 7 Skyfield finds an eclipse, but the *Canon* judges the Moon to have narrowly missed our shadow on that occasion.
- Skyfield tends to return eclipse times that are a few seconds earlier than those given by the *Canon*. For decades near the present the disagreement rarely exceeds 2 seconds, but for eclipses 2,000 years ago the difference can be as large as 20 seconds.
- Over the full five millennia covered by the *Canon*, Skyfield misses only four eclipses, finds two extra eclipses, and agrees with the *Canon*’s category (partial, penumbral, total) more than 99.8% of the time. Of the two missing eclipses that are closest to the modern day, the *Canon* gives the April 859 eclipse a penumbral magnitude of only 0.0007, and the February 2791 eclipse a penumbral magnitude of only 0.0006 — so the missing eclipses were not exactly major celestial events.

To help you study each eclipse in greater detail, Skyfield returns a details dictionary of extra arrays that provide the dimensions of the Moon and of the Earth’s shadow at the height of the eclipse. The means of each field is hopefully self-explanatory; if any of the terms is unfamiliar, try looking it up online.

```
for name, values in sorted(details.items()):
    print(f'{name:24}  {values}')
```

```
closest_approach_radians  [0.00657921 0.01029097]
moon_radius_radians       [0.00485608 0.00435481]
penumbra_radius_radians   [0.02278213 0.02077108]
```

penumbral_magnitude [2.16831186 1.70327942]

umbra_radius_radians [0.01332129 0.01161176]

umbral_magnitude [1.19418911 0.65164729]

Elemen pertama dalam setiap rangkaian ini berhubungan dengan gerhana pertama yang kami temukan di atas, pada tanggal 21-01-2019, sedangkan elemen kedua berhubungan dengan gerhana pada tanggal 16-07-2019.

Dengan menggabungkan dimensi-dimensi ini dengan posisi Bulan pada puncak gerhana (yang dapat Anda hasilkan menggunakan pendekatan biasa Skyfield untuk menghitung posisi), Anda seharusnya dapat menghasilkan diagram terperinci dari setiap gerhana.

Untuk tinjauan parameter yang berbeda antara prakiraan gerhana, lihat halaman Pembesaran bayangan Bumilunar_eclipses() milik NASA di situs Five Millennium Canon mereka. Jika Anda memerlukan prakiraan gerhana bulan yang dihasilkan oleh serangkaian parameter yang sangat spesifik, cobalah memotong dan menempel fungsi Skyfield ke dalam kode Anda sendiri dan buat penyesuaian di sana — Anda akan memiliki kendali penuh atas hasilnya, dan aplikasi Anda akan kebal terhadap segala penyesuaian yang terjadi di Skyfield di masa mendatang jika ditemukan bahwa akurasi gerhana Skyfield dapat menjadi lebih baik.

Posisi

Skyfield menyimpan lokasi benda langit sebagai vektor (*x,y,z*) yang disebut *posisi.* Ada beberapa jenis benda langit yang dapat dibuatkan posisinya oleh Skyfield. Jika ada jenis benda tertentu yang Anda minati, sebaiknya Anda membaca dokumentasinya terlebih dahulu, lalu kembali ke sini saat Anda siap mempelajari lebih lanjut tentang objek posisi yang dihasilkannya:

- Planet dan bulannya: arsip ephemeris JPL
- Bintang dan Objek Jauh
- Satelit Bumi
- Orbit Kepler (komet dan asteroid)

Anda juga dapat membuat objek posisi sendiri dengan memberikan koordinat (*x, y, z*) ke kelas posisi:

```
from skyfield.positionlib import Barycentric


x = 3141.0
y = 2718.0
z = 5820.0
barycentric = Barycentric([x, y, z])
```

Dokumen ini berfokus pada apa yang dapat Anda lakukan dengan posisi setelah dihitung, dan bagaimana objek posisi digunakan dalam kode Skyfield.

Referensi cepat

Untuk katalog ringkas atribut dan metode yang ditawarkan oleh kelas posisi Skyfield, ikuti tautan ini ke dokumentasi API.

- Posisi astronomi

Semua atribut dan metode kelas posisi inti Skyfield.

- Satuan

Kelas jarak, kecepatan, dan sudut Skyfield, yang menawarkan atribut sederhana seperti km tetapi juga metode mewah seperti dstr() yang memformat sudut sebagai derajat, menit, dan detik.

Sistem referensi ICRS dan J2000

Meskipun skrip Skyfield sering menghasilkan output dalam koordinat bulat — seperti asensio dan deklinasi rektus, atau altitude dan azimuth — Skyfield selalu menyimpan posisi secara internal sebagai vektor Cartesian (*x,y,z*) yang berorientasi sepanjang sumbu Sistem Referensi Langit Internasional (ICRS).

ICRS merupakan pengganti sistem referensi J2000 lama yang memiliki akurasi lebih tinggi. Sistem ini didefinisikan menggunakan posisi quasar yang sangat jauh, sehingga presisinya dapat ditingkatkan setiap dekade karena teleskop radio mengukur posisi quasar dengan lebih akurat.

Jika Anda ingin berinteraksi langsung dengan koordinat ICRS (*x,y,z*), berikut ini adalah titik sumbunya:

- *sumbu x* — Bertujuan pada posisi 1 Januari 2000 dari Vernal Equinox, yang secara teknis didefinisikan sebagai simpul menaik ekliptika pada ekuator langit rata-rata. Para astronom kuno menyebutnya "titik pertama Ares" tetapi presesi telah secara bertahap menggesernya ke konstelasi Pisces.
- *sumbu y* — Ditujukan ke titik 90° timur Vernal Equinox di sepanjang ekuator langit, yang terletak di selatan Betelgeuse dan beberapa derajat timur sabuk Orion.
- *sumbu z* — Menargetkan Kutub Langit Utara.

Sumbu ICRS berada dalam 0,02 detik busur dari sumbu J2000 lama, sehingga banyak skrip hanya memperlakukan koordinat J2000 sebagai koordinat ICRS modern.

Barycentric → Astrometric → Apparent

The most common calculations in Skyfield produce a succession of three (*x,y,z*) positions, one for the observer and two for the body being observed. The three positions can be hard to find in Skyfield code because they are often created and used without ever being given a name. As an example, let's use an ephemeris to compute a geocentric position for Mars:

```
from skyfield.api import load

ts = load.timescale()
t = ts.utc(1980, 1, 1)
planets = load('de421.bsp')
earth, mars = planets['earth'], planets['mars']

# Three positions in a single line of code!
d = earth.at(t).observe(mars).apparent().distance()

print('Mars is {:.2f} au from Earth'.format(d.au))
Mars is 0.97 au from Earth
```

By chaining together four different methods, the line that computes the distance d = … creates and discards three different Skyfield positions in a single line of code! To learn about them, the beginner will probably need to slow up and re-write that line so each object is assigned a separate name:

```
# The "d =" line from the previous example,
# rewritten to give each position a name.

barycentric = earth.at(t)
astrometric = barycentric.observe(mars)
apparent = astrometric.apparent()
d = apparent.distance()
```

This is a common Python pattern. By assigning names to intermediate values, the programmer — without changing the code's result — can pivot between succinct code that fits on a single line and more verbose code that names each intermediate value.

Now that we've given them names, we can discuss the three positions:

- A Barycentric position measures from the Solar System's center of mass. This places its (*x,y,z*) vector in the Barycentric Celestial Reference System (BCRS) — a frame of reference that's inertial enough to support the observe() method.

You'll usually start a Skyfield script by generating a barycentric position for the *center* from which you'll be observing — whether that's the Earth, or a specific location on the Earth's surface, or another body like a satellite, planet, or moon.

- An Astrometric position is returned by the Barycentric.observe() method which, given a target you want to observe, applies the effect of light travel time. For example, on Earth we see the Moon where it was about 1.3 seconds ago, the Sun where it was 8 minutes ago, Jupiter where it was more than half an hour ago, and Neptune where it was about 4 hours ago.
- An Apparent position is computed by calling the Astrometric.apparent() method. This applies two real-world effects that slightly shift everything in the sky: the aberration of light produced by the observer's own motion through space, and the gravitational deflection of light that passes close to masses like the Sun and Jupiter — and, for an observer on the Earth's surface, for deflection produced by the Earth's own gravity. The result is an "apparent" position telling you where the target will really "appear" in tonight's sky; it's the direction you should point your telescope.

When an apparent position is measured from the Earth's center, it can be described more formally as an (*x,y,z*) position in Geocentric Celestial Reference System (GCRS) coordinates.

Position attributes

Five basic attributes are available on each position:

.xyz	An (*x,y,z*) Distance.
.velocity	An (*x,y,z*) Velocity, or None.
.t	The Time of the position, or None.

.center	Body the vector is measured from.
.target	Body the vector is measured to.

The first three attributes listed above are simple instances of Skyfield's distance, velocity, and time classes, which you can learn more about by clicking on their class names. They support operations like:

```
print('Earth x,y,z:', barycentric.xyz.au, 'au')

print('Mars relative velocity:', astrometric.velocity.km_per_s, 'km/s')

print('Time of observation:', apparent.t.utc_strftime())

Earth x,y,z: [-0.16287311  0.88787399  0.38473904] au

Mars relative velocity: [12.66638873 -8.12551301 -3.38356109] km/s

Time of observation: 1980-01-01 00:00:00 UTC
```

Note that the distance unit attributes like au and km and the velocity unit attributes like au_per_d and km_per_s are each three-element NumPy arrays offering (*x,y,z*) coordinates. If the times you're using are Date arrays, then the distance and velocity will each have an additional dimension offering as many (*x,y,z*) coordinates as there are dates in your array.

The .center and .target attributes require a bit more explanation. They specify the origin and destination of the vector, and are displayed if you ask Python to print the vector:

```
print(apparent)

<Apparent GCRS position and velocity at date t center=399 target=499>
```

In this case our positions were generated from a JPL ephemeris file, so the center and target are the simple integers 399 indicating the Earth and 499 indicating Mars. If instead the center or target were defined by a Skyfield object like an Earth satellite or latitude-longitude position, then the .center and .target will be those Skyfield objects themselves.

Astrometric right ascension and declination

The most popular coordinate system for star catalogs treats the night sky as a slowly spinning globe that we view from the inside. Just as we specify position on the Earth's globe by degrees longitude that you would travel along the equator and then degrees latitude that you would travel north or south, coordinates in the sky are measured by two angles. These are called "equatorial coordinates" since they are measured with respect to the equator.

- *Right ascension* ("RA") is the sky's equivalent of longitude, and is measured east along the celestial equator. Since the sky makes roughly one complete turn every day, RA is usually expressed in units of 24 hours rather than 360 degrees. This supports casual inferences, like the fact that a star with an RA of 3^h will climb to your meridian one hour later than a star with an RA of 2^h.

This coordinate presented astronomers with the same challenge that longitude presented geographers: the arbitrary choice of a starting point. In the case of longitude, we measure east

and west from the Airy Transit Circle at the Royal Observatory Greenwich. For RA, astronomers measure east from the Vernal Equinox, the point where the Sun crosses the celestial equator in March as it passes from the southern to the northern half of the sky.

- *Declination* ("Dec") is the sky's equivalent of latitude, measured north and south of the celestial equator in degrees, with north being positive. The North Celestial Pole is at +90° and the South Celestial Pole is at −90°.

There are two common uses for RA/Dec coordinates.

The first common use is as a reference coordinate system for perennial knowledge like star charts, celestial catalogs, and record keeping. This use case presents two complications.

The first complication is that real star positions appear to be in constant motion. The Earth's revolution around the Sun moves the stars in little circles thanks to the aberration of light, and their light can be bent and their images shifted by the gravity of the Sun, Jupiter, and even the Earth. Because these effects are periodic or temporary, they are entirely unsuitable for star charts, which are supposed to be independent of any particular day of the year.

For this reason, always generate reference RA/Dec coordinates from astrometric Skyfield positions, never from apparent positions.

The second complication is that the Earth's poles don't point in a fixed direction but gradually trace 26,000-year circles around the sky. This requires RA/Dec coordinates to specify which year's "equinox" their right ascension is measured from, which will also be the year whose poles and equator they use.

The modern standard for astrometric RA/Dec is the ICRS (described above). With axes that are fixed in the directions of the J2000 equinox and poles, the ICRS is a permanent coordinate system that will never suffer precession. Skyfield returns ICRS coordinates if you simply call radec() without an argument:

```
# Astrometric RA/Dec.

ra, dec, distance = astrometric.radec()


print('RA:', ra)

print('Dec:', dec)

print('Distance:', distance)
```

```
RA: 11h 06m 51.22s

Dec: +09deg 05' 09.2"

Distance: 0.96678 au
```

Here we have printed RA and declination using their default units. See the Angle class documentation for the attributes and methods by which you can retrieve and format an angle's value in units of your own choice.

If your project specifically requires coordinates expressed in the RA/Dec of an older equinox, you can build a time object and pass it to radec():

```
# Astrometric RA/Dec relative to another equinox:

# the J1991.25 epoch used by the Hipparcos catalog.


equinox = ts.J(1991.25)

ra, dec, distance = astrometric.radec(equinox)
```

Apparent right ascension and declination

The other reason that you might generate RA/Dec is practical: you are planning to point a telescope, and after orienting its equatorial mount to the Earth's poles you want a declination that stays fixed all night while your telescope tracks the specified right ascension across the sky.

In this case you aren't likely to be satisfied with RA/Dec coordinates of some other era. You'll want them measured against where the Earth's poles are really pointing tonight.

Skyfield uses the high precision IAU 2000A model to compute both the precession that carries the Earth's poles in their 26,000-year circle around the sky and also the short term wobbles in the Earth's orientation that are called nutation. Together they give Skyfield an accurate assessment of the direction of the poles and equator on a given date.

When pointing a telescope, always use apparent coordinates, since you will want every possible real-world effect accounted for.

While you could pass each position's time to its own radec() method, that would be a bit tedious, so Skyfield provides a shortcut: if you pass the string 'date' then the RA/Dec coordinates use the equinox and poles of the date of the position itself:

```
# Apparent RA/Dec.

ra, dec, distance = apparent.radec('date')


print('RA:', ra)

print('Dec:', dec)

print('Distance:', distance)
```

```
RA: 11h 05m 48.68s

Dec: +09deg 11' 35.7"

Distance: 0.96678 au
```

Note that there are two sources of difference between the astrometric coordinates printed in the previous section and the apparent coordinates printed here. We switched to the RA/Dec

system of a different year, so even the exact same position would have been assigned a different coordinate than before. But we have also asked for the RA/Dec of a whole different point in the sky: the point where Mars will actually appear, not the ideal point where it should sit on a star chart that ignores aberration and deflection.

Azimuth and altitude from a geographic position

The final result that many users seek is the altitude and azimuth of an object above their own local horizon.

- *Altitude* measures the angle above or below the horizon. The zenith is at +90°, an object on the horizon's great circle is at 0°, and the nadir beneath your feet is at −90°.
- *Azimuth* measures the angle around the sky from the north pole: 0° means exactly north, 90° is east, 180° is south, and 270° is west.

Altitude and azimuth are computed by calling the altaz() method on an apparent position. But because the method needs to know which local horizon to use, it does not work on the plain geocentric ("Earth centered") positions that we have been generating so far:

```
alt, az, distance = apparent.altaz()

Traceback (most recent call last):

 ...

ValueError: to compute an apparent position, you must observe from a specific Earth location that you specify using a latitude and longitude
```

You can specify a location on Earth by giving its latitude and longitude to a standard "geodetic system" that models the Earth's shape. The most popular model is WGS84, used by most modern maps and also by the Global Positioning System (GPS). If you are given a longitude and latitude without a datum specified, they are probably WGS84 coordinates.

You can pass the latitude and longitude to a datum like WGS84 by calling its latlon() method. Skyfield provides constants N, S, E, and W that are each positive one or negative one, in case you don't want to remember which directions are positive.

```
from skyfield.api import N,S,E,W, wgs84


# Altitude and azimuth in the sky of a

# specific geographic location


boston = earth + wgs84.latlon(42.3583 * N, 71.0603 * W, elevation_m=43)

astro = boston.at(ts.utc(1980, 3, 1)).observe(mars)

app = astro.apparent()


alt, az, distance = app.altaz()
```

```
print(alt.dstr())

print(az.dstr())

print(distance)

24deg 30' 27.2"

93deg 04' 29.5"

0.678874 au
```

So Mars was more than 24° above the horizon for Bostonians on 1980 March 1 at midnight UTC.

Adjusting altitude for atmospheric refraction

The altitude returned from a plain altaz() call is the ideal position that you would observe if the Earth had no atmosphere. You can also ask Skyfield to estimate where an object might actually appear in the sky after the Earth's atmosphere has *refracted* its image higher. If you know the weather conditions, you can specify them.

```
alt, az, distance = app.altaz(temperature_C=15.0,
                              pressure_mbar=1005.0)

print(alt.dstr())

24deg 32' 34.1"
```

Atau Anda dapat meminta Skyfield untuk menggunakan suhu dan tekanan standar saat membuat simulasi kasar efek refraksi.

```
alt, az, distance = app.altaz('standard')

print(alt.dstr())

24deg 32' 36.4"
```

Perlu diingat bahwa efek refraksi yang dihitung hanyalah perkiraan. Efek atmosfer lokal Anda, dengan banyak lapisan panas dan dingin, angin, dan cuaca, tidak dapat diprediksi dengan presisi tinggi. Dan perlu diingat bahwa refraksi hanya berlaku untuk objek di atas cakrawala. Objek di bawah ketinggian −1,0° tidak disesuaikan untuk refraksi.

Membandingkan posisi

Jika Anda ingin mengetahui sudut antara dua posisi di langit, panggil separation_from() metode salah satu posisi dan berikan posisi lainnya sebagai argumennya. Hasilnya akan berupa Angleobjek.

Jika Anda ingin mengetahui jarak antara dua posisi, kurangi posisi yang ingin Anda gunakan sebagai titik awal dari posisi lainnya. Hasil dari perhitungan vektor ini juga akan menjadi vektor posisi:

```
v = planets['moon'].at(t) - planets['earth'].at(t)

print('The Moon is %d km away' % v.distance().km)
```

```
The Moon is 385010 km away
```

Dua vektor tidak perlu memiliki tanggal dan waktu yang sama untuk dikurangkan. Dengan membandingkan dua posisi dengan waktu yang berbeda, Anda dapat mengukur seberapa jauh sebuah objek telah bergerak:

```
t1 = ts.utc(2015, 10, 11, 10, 30)
t2 = ts.utc(2015, 10, 11, 10, 31)

moon = planets['moon']
p1 = moon.at(t1)
p2 = moon.at(t2)

km = (p2 - p1).distance().km
print('In one minute the Moon moved %d km' % km)
```

```
In one minute the Moon moved 1736 km
```

Koordinat dalam kerangka acuan lain

Anda dapat meminta Skyfield untuk menyatakan posisi dalam sejumlah kerangka acuan lain selain yang diilustrasikan di atas. Untuk detailnya, lihat bab terlampir yang secara khusus didedikasikan untuk menjelaskan Koordinat .

Anda juga dapat membaca bab Kerangka Acuan Planet , yang menjelaskan cara menggunakan kerangka acuan yang tidak disertakan dengan Skyfield, dengan memuatnya dari berkas acuan NASA.

Koordinat

"Koordinat" adalah angka yang kita gunakan untuk menamai posisi astronomi. Untuk posisi apa pun, Skyfield dapat mengembalikan setidaknya setengah lusin set koordinat berbeda yang menamai posisi tersebut.

Melalui pembahasan berikut, ingatlah bahwa koordinat hanyalah nama untuk suatu posisi. Meminta koordinat yang berbeda tidak akan mengubah posisi itu sendiri. Seperti yang dijelaskan dalam bab Posisi , ada perbedaan fisik yang nyata antara, misalnya, posisi astrometris dan posisi semu: keduanya adalah dua titik berbeda di langit. Namun, jika diberikan satu posisi — satu titik di langit — semua koordinat yang dapat Anda hasilkan memiliki arti yang sama persis: arah yang akan Anda tuju untuk meletakkan titik itu di pusatnya.

Koordinat Kartesius versus Koordinat Bola

Sistem koordinat tertentu dapat digunakan untuk menghasilkan dua jenis koordinat: Cartesian dan bola.

Koordinat Cartesian menentukan posisi dengan menyediakan tiga jarak (*x,y,z*) yang diukur sepanjang sumbu yang tegak lurus satu sama lain.

Koordinat bola menentukan lokasi menggunakan dua sudut ditambah jarak. Sama seperti garis lintang dan garis bujur geografis yang menentukan posisi di Bumi, kedua sudut koordinat bola menentukan titik tertentu di langit — titik tertentu di bola langit — dan jarak menentukan seberapa jauh Anda harus bepergian ke arah itu untuk mencapai target.

Melalui bagian berikut, kita akan melihat cara menanyakan koordinat Cartesian dan bola dalam berbagai sistem koordinat.

ICRS dan asensio rektum dan deklinasi ekuatorial

Koordinat ekuator , yang mengukur sudut dari ekuator dan kutub Bumi, telah dijelaskan dalam bab Posisi :

- Asensio rekta dan deklinasi astrometris
- Asensio dan deklinasi rekta yang tampak

Berikut ini cara sumbu dan sudut didefinisikan:

Koordinat Ekuatorial

bidang xy : ekuator bumi

sumbu *x : ekuinoks Maret*

sumbu *z : kutub utara langit*

↔ Asensio rekta 0 ^{jam} –24 ^{jam} ke arah timur di sekitar ekuator

↕ Deklinasi ±90° dari ekuator ke arah kutub

There's a problem with this coordinate system: it slowly changes. Thanks to precession, the Earth's poles are in constant — if gradual — motion. So equatorial coordinates always have to specify whether they are measuring against the Earth's equator and equinox as of 1950, or 2000, or some other specific year. In the late twentieth century "J2000" coordinates became popular, measured against the Earth's mean equator as of noon Terrestrial Time on 2000 January 1.

But in 1998, astronomers decided it was time to get off of the equatorial treadmill. They defined a new coordinate system: the ICRS (International Celestial Reference System). Its axes point almost exactly along the old J2000 axes, but are permanent and immobile, and can achieve unbounded precision because instead of using the Earth its axes are tied to the locations of very distant quasars.

In the Positions chapter you've already seen how to retrieve ICRS coordinates in both Cartesian and spherical form:

```
# Building an example position.

from skyfield.api import load

ts = load.timescale()
t = ts.utc(2021, 12, 2, 14, 7)
eph = load('de421.bsp')
earth, mars = eph['earth'], eph['mars']
position = earth.at(t).observe(mars)

# Producing coordinates.

print('Cartesian ICRS:')

x, y, z = position.xyz.au

print('  x = {:.3f} au'.format(x))
print('  y = {:.3f} au'.format(y))
print('  z = {:.3f} au'.format(z))
print()
```

```
print('Spherical ICRS:')


ra, dec, distance = position.radec()


print(' ', ra, 'right ascension')
print(' ', dec, 'declination')
print(' ', distance, 'distance')
```

```
Cartesian ICRS:
 x = -1.521 au
 y = -1.800 au
 z = -0.772 au

Spherical ICRS:
 15h 19m 15.08s right ascension
 -18deg 08' 37.0" declination
 2.47983 au distance
```

Note that .xyz is a Distance object and supports not only .au but also several other units of measure.

If instead of using the permanent ICRS you want to measure coordinates against the Earth's axes as they precess, then retrieving spherical coordinates is easy: simply provide a date to radec() and it will return a right ascension and declination relative to the Earth's equator and equinox as of that date.

It's less usual for someone to want (*x,y,z*) coordinates measured against the real equator and equinox, so there's no built-in method to retrieve them. Instead, you'll need to import the reference frame itself and pass it to the position's frame_xyz() method:

```
from skyfield.framelib import \
    true_equator_and_equinox_of_date as of_date


print('Cartesian equinox-of-date coordinates:')


x, y, z = position.frame_xyz(of_date).au


print('  x = {:.3f} au'.format(x))
```

```
print('  y = {:.3f} au'.format(y))
print('  z = {:.3f} au'.format(z))
print()

print('Spherical equinox-of-date coordinates:')

ra, dec, distance = position.radec(position.t)

print(' ', ra, 'right ascension')
print(' ', dec, 'declination')
print(' ', distance, 'distance')
```

```
Cartesian equinox-of-date coordinates:
  x = -1.510 au
  y = -1.808 au
  z = -0.775 au

Spherical equinox-of-date coordinates:
  15h 20m 28.70s right ascension
  -18deg 13' 18.5" declination
  2.47983 au distance
```

Note that the distance is exactly the same as before, because this is exactly the same position — it's merely being measured against a slightly different set of axes.

Altitude and azimuth ('horizonal' coordinates)

Altitude and azimuth have already been explained in the Positions chapter, so you can start reading about them there:

- Azimuth and altitude from a geographic position

The coordinate system is called *horizonal* in the sense of "pertaining to the horizon."

Horizonal Coordinates

xy-plane: Horizon

x-axis: North point on the horizon

y-axis: East point on the horizon (left-handed)

z-axis: Zenith

↕ Altitude ±90° above or below horizon

↔ Azimuth 0°–360° measured clockwise from north

As with the equatorial system, the angles associated with horizontal coordinates are so popular that Skyfield provides a built-in method altaz() to retrieve them, while (*x,y,z*) coordinates require a call to frame_xyz() with the geographic location itself passed as the reference frame:

```
# From the chapter on Positions:

# computing altitude and azimuth.


from skyfield.api import load, wgs84


bluffton = wgs84.latlon(+40.8939, -83.8917)

astrometric = (earth + bluffton).at(t).observe(mars)

position = astrometric.apparent()


print('Cartesian:')


x, y, z = position.frame_xyz(bluffton).au


print('  x = {:.3f} au north'.format(x))

print('  y = {:.3f} au east'.format(y))

print('  z = {:.3f} au up'.format(z))

print()


print('Spherical:')


alt, az, distance = position.altaz()


print('  Altitude:', alt)

print('  Azimuth:', az)

print('  Distance:', distance)
```

Cartesian:

x = -1.913 au north

y = 1.200 au east

z = 1.025 au up

Spherical:

Altitude: 24deg 24' 20.6"

Azimuth: 147deg 54' 28.8"

Distance: 2.47981 au

Note that some astronomers use the term “elevation” for what Skyfield calls “altitude”: the angle at which a target stands above the horizon. Obviously both words are ambiguous, since “elevation” can also mean a site’s vertical distance above sea level, and “altitude” can mean an airplane’s height above either sea level or the ground.

Hour Angle and Declination

If you are pointing a telescope or other instrument, you might be interested in a variation on equatorial coordinates: replacing right ascension with *hour angle,* which measures ±180° from your own local meridian.

```
ha, dec, distance = position.hadec()


print('Hour Angle:', ha)
print('Declination:', dec, )
print('Distance:', distance)
```

```
Hour Angle: -02h 02m 28.88s
Declination: -18deg 13' 16.4"
Distance: 2.47981 au
```

To make the hour angle and declination even more useful for pointing real-world instruments, Skyfield includes the effect of polar motion if you have loaded a polar motion table. In that case the declination you get from hadec() will vary slightly from the declination returned by radec(), which doesn’t include polar motion.

ECI versus ECEF coordinates

Here’s a quick explanation of two acronyms that you’re likely to run across in discussions about coordinates.

ECI stands for *Earth-Centered Inertial* and specifies coordinates that are (a) measured from the Earth’s center and (b) that don’t rotate with the Earth itself. The very first coordinates we computed in this chapter, for example, qualify as ECI coordinates, because the position used

the Earth as its center and because the ICRS system of right ascension and declination stays fixed on the celestial sphere even as the Earth rotates beneath it.

ECEF stands for *Earth-Centered Earth-Fixed* and specifies coordinates that are (a) measured from the Earth's center but (b) which rotate with the Earth instead of staying fixed in space. An example would be the latitude and longitude of the Lowell Observatory, which stays in place on the Earth's surface but from the point of view of the rest of the Solar System is rotating with the Earth. A fixed on the Earth's surface is a good example. We will learn about generating ECEF coordinates in the next section.

Geographic ITRS latitude and longitude

Skyfield uses the standard ITRS reference frame to specify positions that are fixed relative to the Earth's surface.

ITRS Coordinates

xy-plane: Earth's equator

x-axis: 0° longitude on the equator

y-axis: 90° east longitude on the equator

z-axis: North pole

↕ Latitude ±90° from equator toward poles

↔ Longitude ±180° from prime meridian with east positive

A location's latitude will vary slightly depending on whether you model the Earth as a simple sphere or more realistically as a slightly flattened ellipsoid. The most popular choice today is to use the WGS84 ellipsoid, which is the one used by the GPS system.

```
from skyfield.api import wgs84
from skyfield.framelib import itrs


# Important: must start with a position
# measured from the Earth's center.
position = earth.at(t).observe(mars)


print('Cartesian:')


x, y, z = position.frame_xyz(itrs).au


print('  x = {:.3f} au'.format(x))
print('  y = {:.3f} au'.format(y))
```

```
print('  z = {:.3f} au'.format(z))
print()

print('Geographic:')

lat, lon = wgs84.latlon_of(position)
height = wgs84.height_of(position)

print(' {:.4f}° latitude'.format(lat.degrees))
print(' {:.4f}° longitude'.format(lon.degrees))
print(' {:.0f} km above sea level'.format(distance.km))
```

```
Cartesian:
  x = 1.409 au
  y = -1.888 au
  z = -0.775 au

Geographic:
 -18.2218° latitude
 -53.2662° longitude
 370974969 km above sea level
```

Note that height is measured from sea level, not from the center of the Earth.

The code above is slightly inefficient, because height_of() will wind up recomputing several values that were already computed in latlon_of(). If you need both, it's more efficient to call geographic_position_of().

There's also a subpoint_of() method if you want Skyfield to compute the geographic position of the sea-level point beneath a given celestial object.

Ecliptic coordinates

Ecliptic coordinates are measured from the plane of the Earth's orbit. They are useful when making maps and diagrams of the Solar System and when exploring the properties of orbits around the Sun, because they place the orbits of the major planets nearly flat against the *xy*-plane — unlike right ascension and declination, which twist the Solar System up at a 23° angle because of the tilt of the Earth's axis.

You might be tempted to ask why we measure against the plane of the Earth's orbit, instead of averaging together all the planets to compute the "invariable plane" of the whole Solar System (to which the Earth's orbit is inclined by something like 1.57°). The answer is: precision. We know the plane of the Earth's orbit to many decimal places, because the Earth carries all of our highest-precision observatories along with it as it revolves around the Sun. Our estimate of the invariable plane, by contrast, is a mere average that changes — at least slightly — every time we discover a new asteroid, comet, or trans-Neptunian object. So the Earth's own orbit winds up being the most pragmatic choice for a coordinate system oriented to the Solar System.

Ecliptic Coordinates

xy-plane: Ecliptic plane (plane of Earth's orbit)

x-axis: March equinox

z-axis: North ecliptic pole

↕ Latitude ±90° above or below the ecliptic

↔ Longitude 0°–360° measured east from March equinox

```
from skyfield.framelib import ecliptic_frame


print('Cartesian ecliptic coordinates:')


x, y, z = position.frame_xyz(ecliptic_frame).au


print('  x = {:.3f} au'.format(x))
print('  y = {:.3f} au'.format(y))
print('  z = {:.3f} au'.format(z))
print()


print('Spherical ecliptic coordinates:')


lat, lon, distance = position.frame_latlon(ecliptic_frame)


print(' {:.4f} latitude'.format(lat.degrees))
print(' {:.4f} longitude'.format(lon.degrees))
print(' {:.3f} au distant'.format(distance.au))
```

```
Cartesian ecliptic coordinates:
 x = -1.510 au
 y = -1.967 au
 z = 0.007 au

Spherical ecliptic coordinates:
 0.1732 latitude
 232.4801 longitude
 2.480 au distant
```

Note the very small values returned for the ecliptic *z* coordinate and for the ecliptic latitude, because Mars revolves around the Sun in very nearly the same plane as the Earth.

Galactic coordinates

Galactic coordinates are measured against the disc of our own Milky Way galaxy, as measured from our vantage point here inside the Orion Arm:

Galactic Coordinates

xy-plane: Galactic plane

x-axis: Galactic center

z-axis: North galactic pole

↕ Latitude ±90° above galactic plane

↔ Longitude 0°–360° east from galactic center

```
from skyfield.framelib import galactic_frame

print('Cartesian galactic coordinates:')

x, y, z = position.frame_xyz(galactic_frame).au

print('  x = {:.3f} au'.format(x))
print('  y = {:.3f} au'.format(y))
print('  z = {:.3f} au'.format(z))
print()
```

```
print('Spherical galactic coordinates:')


lat, lon, distance = position.frame_latlon(galactic_frame)


print(' {:.4f} latitude'.format(lat.degrees))
print(' {:.4f} longitude'.format(lon.degrees))
print(' {:.3f} au distant'.format(distance.au))
```

```
Cartesian galactic coordinates:
 x = 2.029 au
 y = -0.527 au
 z = 1.324 au


Spherical galactic coordinates:
 32.2664 latitude
 345.4330 longitude
 2.480 au distant
```

Astronomers have generated a series of more and more precise estimates of our galaxy's orientation over the past hundred years. Skyfield uses the IAU 1958 Galactic System II, which is believed to be accurate to within ±0.1°.

Velocity

Jika Anda tidak hanya memerlukan vektor posisi tetapi juga vektor kecepatan relatif terhadap kerangka acuan tertentu, maka beralihlah dari satu frame_xyz()metode ke frame_xyz_and_velocity()metode lainnya. Seperti yang dijelaskan dalam dokumentasinya, nilai pengembaliannya mencakup vektor kecepatan.

Mengubah koordinat menjadi posisi

Semua contoh di atas mengambil posisi Skyfield dan mengembalikan koordinat, tetapi terkadang Anda memulai dengan koordinat dan ingin menghasilkan posisi.

Jika Anda memulai dengan koordinat ICRS (*x,y,z*), maka Anda dapat membuat posisi dengan panggilan fungsi:

```
from skyfield.positionlib import build_position


icrs_xyz_au = [-1.521, -1.800, -0.772]
position = build_position(icrs_xyz_au, t=t)
```

Namun, mungkin lebih umum bagi Anda untuk diberi asensio dan deklinasi target. Salah satu pendekatan umum adalah membuat Star seperti yang dijelaskan dalam bab Bintang dan Objek Jauh , yang akan memberi Anda objek yang dapat Anda lewati .observe() seperti benda Skyfield lainnya. Namun, Anda juga dapat membuat posisi secara langsung dengan menggunakan from_radec()metode yang dibawa oleh setiap kelas posisi. Untuk membuat posisi yang tampak, misalnya:

```
from skyfield.positionlib import Apparent


position = Apparent.from_radec(ra_hours=5.59, dec_degrees=5.45)
```

Situasi umum terakhir adalah Anda mengukur ketinggian dan azimuth relatif terhadap cakrawala dan ingin mempelajari asensio dan deklinasi yang tepat dari posisi tersebut. Solusinya adalah menggunakan from_altaz() metode tersebut, tetapi ada kendalanya: karena koordinat sebenarnya di balik ketinggian dan azimuth tertentu berubah setiap saat saat Bumi berputar, pertama-tama Anda perlu menghitung posisi observatorium Anda .at()saat Anda mengukur ketinggian dan azimuth, dan baru kemudian memanggil metode tersebut.

```
# What are the coordinates of the zenith?


b = bluffton.at(t)
apparent = b.from_altaz(alt_degrees=90.0, az_degrees=0.0)


ra, dec, distance = apparent.radec()
print('Right ascension:', ra)
print('Declination:', dec)
Right ascension: 13h 17m 00.26s
Declination: +41deg 00' 27.7"
```

Jika Anda menemukan diri Anda dalam situasi yang bahkan lebih jarang terjadi, seperti perlu membangun posisi dari koordinat ekliptika atau galaksi, maka — meskipun belum ada contoh terdokumentasi yang dapat Anda ikuti — Anda mungkin dapat menyusun solusi bersama dari potongan-potongan ini:

- Metode konstruktor posisi from_time_and_frame_vectors().
- Metode dalam .from_spherical(r, theta, phi)skyfield/functions.py

Matriks Rotasi

Jika Anda mengerjakan sendiri matematika, Anda mungkin ingin mengakses matriks rotasi 3×3 tingkat rendah yang menentukan hubungan antara setiap kerangka acuan koordinat dan ICRS.

Untuk menghitung matriks rotasi, cukup berikan waktu ke rotation_at()metode kerangka acuan:

```
# 3×3 rotation matrix: ICRS → frame


R = ecliptic_frame.rotation_at(t)

print(R)

[[ 0.99998613 -0.00482998 -0.00209857]

 [ 0.00526619  0.9174892   0.39772583]

 [ 0.00000441 -0.39773137  0.91750191]]
```

Anda akan merasa matriks mudah digunakan jika tmerupakan waktu tunggal, tetapi jika tmerupakan keseluruhan larik waktu , maka matriks yang dihasilkan akan memiliki dimensi ketiga dengan jumlah elemen yang sama dengan vektor waktu. NumPy tidak menyediakan dukungan langsung untuk matriks rotasi dengan dimensi tambahan, jadi hindari penggunaan operator perkalian NumPy. Sebagai gantinya, gunakan fungsi utilitas Skyfield:

```
from skyfield.functions import T, mxm, mxv


T(R)       # reverse rotation matrix: frame → ICRS

mxm(R, R2)  # matrix × matrix: combines rotations

mxv(R, v)   # matrix × vector: rotates a vector
```

Tanggal dan Waktu

Para astronom menggunakan berbagai skala yang berbeda untuk mengukur waktu. Skyfield sering kali harus menggunakan beberapa skala waktu dalam satu perhitungan! TimeKelas ini adalah cara Skyfield merepresentasikan satu momen dalam waktu atau serangkaian momen, dan melacak semua sebutan berbeda yang ditetapkan untuk momen tersebut oleh berbagai skala waktu standar:

```
from skyfield.api import load

ts = load.timescale()

t = ts.tt(2000, 1, 1, 12, 0)


print('TT date and time: ', t.tt_strftime())

print('TAI date and time:', t.tai_strftime())

print('UTC date and time:', t.utc_strftime())

print('TDB Julian date: {:.10f}'.format(t.tdb))

print('Julian century: {:.1f}'.format(t.J))

TT date and time:  2000-01-01 12:00:00 TT

TAI date and time: 2000-01-01 11:59:28 TAI

UTC date and time: 2000-01-01 11:58:56 UTC

TDB Julian date: 2451544.9999999991

Julian century: 2000.0
```

Objek Timescaleyang dikembalikan oleh load.timescale() mengelola konversi antara skala waktu yang berbeda dan juga merupakan cara pemrogram membuat Timeobjek untuk tanggal tertentu. Sebagian besar aplikasi hanya membuat satu Timescaleobjek, yang secara konvensional disebut oleh pemrogram Skyfield ts, dan menggunakannya untuk membuat semua waktu mereka.

Sebagai referensi cepat, berikut adalah rentang waktu yang didukung:

- UTC — Waktu Universal Terkoordinasi (“Waktu Greenwich”)
- UT1 — Waktu Universal
- TAI — Waktu Atom Internasional
- TT — Waktu Terestrial
- *TDB — Waktu Dinamis Barycentric (* T_{eph} milik JPL)

Dan berikut adalah tautan ke dokumentasi API untuk skala waktu dan waktu:

- Skala waktu
- Objek waktu

Tanggal kuno dan modern

Skyfield biasanya menggunakan kalender Gregorian modern, bahkan untuk tanggal-tanggal dalam sejarah sebelum kalender Gregorian diadopsi pada tahun 1582. Penggunaan tanggal Gregorian yang "proleptik" ini membuat perhitungan tanggal menjadi sederhana, kompatibel dengan Python datetime, dan juga merupakan perilaku pustaka United States Naval Observatory yang menjadi dasar banyak rutinitas Skyfield pada awalnya.

Akan tetapi, kalender Gregorian terasa janggal bagi para sejarawan dan mahasiswa astronomi kuno, karena kalender yang benar-benar digunakan sebelum tahun 1582 adalah kalender Julian lama yang ditetapkan oleh reformasi kalender Julius Caesar pada tahun 45 SM. Kedua kalender tersebut sama selama seabad antara hari kabisat tahun 200 M dan hari kabisat tahun 300 M. Akan tetapi, karena kalender Julian tidak sepenuhnya sinkron dengan musim, tanggalnya berjalan di depan kalender Gregorian sebelum abad tersebut dan berjalan di belakang kalender Gregorian setelahnya.

Jika Anda ingin Skyfield beralih ke kalender Julian untuk tanggal historis — baik saat menafsirkan tanggal yang Anda masukkan, maupun saat menghasilkan tanggal kalender sebagai output — cukup berikan Timescaleobjek Anda hari Julian di mana Anda ingin kalender tersebut beralih.

```
from skyfield.api import GREGORIAN_START


ts.julian_calendar_cutoff = GREGORIAN_START


t = ts.tt_jd(range(2299159, 2299163))
for s in t.tt_strftime():
    print(s)
```

```
1582-10-03 12:00:00 TT
1582-10-04 12:00:00 TT
1582-10-15 12:00:00 TT
1582-10-16 12:00:00 TT
```

Seperti yang dapat Anda lihat dari empat hari berturut-turut dalam sejarah ini, Paus Gregorius mengubah kalender secara langsung dari tanggal kalender Julian lama 4 Oktober 1582 ke tanggal kalender Gregorian baru 15 Oktober 1582, sehingga tanggal Paskah kembali sinkron dengan ekuinoks. Skyfield menyediakan dua konstanta untuk tanggal pemotongan yang populer:

- GREGORIAN_START— Hari Julian 2299161, di mana kalender Gregorian baru mulai berlaku di Roma.
- GREGORIAN_START_ENGLAND — Julian day 2361222, on which the new Gregorian calendar went into effect in England in 1752 (the reform having initially been rejected by the English bishops, “Seeing that the Bishop of Rome is Antichrist, therefore we may not communicate with him in any thing”).

You are free to choose your own cutoff Julian day if you are studying astronomy records from a country that adopted the Gregorian calendar on some other date. Russia, for example, did not adopt it until the twentieth century. The default value, that asks the timescale to always use Gregorian dates, is None:

```
ts.julian_calendar_cutoff = None
```

Note that even the Julian calendar becomes anachronistic before its adoption in 45 BC, so all dates generated by Skyfield are “proleptic” before that date. And, of course, the Julian calendar was local to the civilization that ringed the Mediterranean. If you are interested in relating astronomical events to more ancient Roman calendars, or the calendars of other civilizations, try searching for a third-party Python package that supports the calendar you are interested in.

Building and printing UTC

The utc parameter in the examples above specifies Coordinated Universal Time (UTC), the world clock known affectionately as “Greenwich Mean Time” which is the basis for all of the world’s timezones. If you are comfortable dealing directly with UTC instead of your local timezone, you can build and display dates without needing any other library besides Skyfield.

You can build a Time from a calendar date and UTC time using Timescale.utc. Provide only as many parameters as you want — year, month, day, hour, minute, and second — and Skyfield will fill in the rest by defaulting to January first and zero hours, minutes, and seconds.

Feel free to use fractional days, hours, and minutes. Here are several ways to specify the exact same time and date:

```
# Four ways to specify 2014 January 18 01:35:37.5


t1 = ts.utc(2014, 1, 18.06640625)
t2 = ts.utc(2014, 1, 18, 1.59375)
t3 = ts.utc(2014, 1, 18, 1, 35.625)
t4 = ts.utc(2014, 1, 18, 1, 35, 37.5)


assert t1 == t2 == t3 == t4    # True!


# Several ways to print a time as UTC.
```

```
print(tuple(t1.utc))

print(t1.utc_iso())

print(t1.utc_strftime())

print(t1.utc_strftime('On %Y %b %d at %H:%M:%S'))

print(t1.utc_jpl())
```

```
(2014, 1, 18, 1, 35, 37.5)

2014-01-18T01:35:38Z

2014-01-18 01:35:38 UTC

On 2014 Jan 18 at 01:35:38

A.D. 2014-Jan-18 01:35:37.5000 UTC
```

The 6 values returned by utc() can be accessed as the attributes year, month, day, hour, minute, and second.

```
print(t1.utc.year, '/', t1.utc.month, '/', t1.utc.day)

print(t1.utc.hour, ':', t1.utc.minute, ':', t1.utc.second)
```

```
2014 / 1 / 18

1 : 35 : 37.5
```

If you want to use the current time, Skyfield leverages the minimal support for UTC in the Python Standard Library to offer a now() function that reads your system clock and returns the current time as a Time object (assuming that your operating system clock is correct and configured with the correct time zone):

```
from skyfield.api import load


# Asking the current date and time


ts = load.timescale()

t = ts.now()

print(t.utc_jpl())
```

```
A.D. 2015-Oct-11 10:00:00.0000 UTC
```

UTC and your timezone

To move beyond UTC and work with other world timezones, you will need to install a time zone database for your version of Python.

- Every version of Python that Skyfield supports will work with the pytz package described in this section.
- Python 3.6 upgraded the Standard Library datetime type so that the contortions of pytz are no longer necessary, and instead recommends dateutil for working with timezones. Consult its documentation if you are interested in using it.
- Python 3.9 will offer a native zoneinfo module that for the first time brings timezone support into the Python Standard Library.

This documentation will focus on the first approach, which works universally across all Python versions. You can install the third-party pytz library by listing it in the dependencies of your package, adding it to your project's requirements.txt file, or simply installing it manually:

```
pip install pytz
```

Once it is installed, building a Time from a local time is simple. Instantiate a normal Python datetime, pass it to the localize() method of your time zone, and pass the result to Skyfield:

```
from datetime import datetime
from pytz import timezone

eastern = timezone('US/Eastern')

# Converting US Eastern Time to a Skyfield Time.

d = datetime(2014, 1, 16, 1, 32, 9)
e = eastern.localize(d)
t = ts.from_datetime(e)
```

When Skyfield returns a Time at the end of a calculation, you can ask for either a UTC datetime or a datetime in your own timezone:

```
# UTC datetime

dt = t.utc_datetime()
print('UTC: ' + str(dt))

# Converting back to an Eastern Time datetime.
```

```
dt = t.astimezone(eastern)
print('EST: ' + str(dt))
```

```
UTC: 2014-01-16 06:32:09+00:00
EST: 2014-01-16 01:32:09-05:00
```

As we would expect, 1:32 AM in the Eastern time zone in January is 6:32 AM local time in Greenwich, England, five hours to the east across the Atlantic.

Note that Skyfield's astimezone() method will detect that you are using a pytz timezone and automatically call its normalize() method for you — which makes sure that daylight savings time is handled correctly — to spare you from having to make the call yourself.

If you want a Time to hold an entire array of dates, as discussed below in Date arrays, then you can provide a list of datetime objects to the Timescale.from_datetimes() method. The UTC methods will then return whole lists of values.

UTC and leap seconds

The rate of Earth's rotation is gradually slowing down. Since the UTC standard specifies a fixed length for the second, promises a day of 24 hours, and limits an hour to 60 minutes, the only way to stay within the rules while keeping UTC synchronized with the Earth is to occasionally add an extra leap second to one of the year's minutes.

See The leap second table if you are interested in printing Skyfield's full list of leap seconds.

The International Earth Rotation Service currently restricts itself to appending a leap second to the last minute of June or the last minute of December. When a leap second is inserted, its minute counts 61 seconds numbered 00–60 instead of staying within the usual range 00–59. One recent leap second was in June 2012:

```
# Display 5 seconds around a leap second


five_seconds = [58, 59, 60, 61, 62]
t = ts.utc(2012, 6, 30, 23, 59, five_seconds)
for string in t.utc_jpl():
    print(string)
```

```
A.D. 2012-Jun-30 23:59:58.0000 UTC
A.D. 2012-Jun-30 23:59:59.0000 UTC
A.D. 2012-Jun-30 23:59:60.0000 UTC
A.D. 2012-Jul-01 00:00:00.0000 UTC
A.D. 2012-Jul-01 00:00:01.0000 UTC
```

Note that Skyfield has no problem with a calendar tuple that has hours, minutes, or — as in this case — seconds that are out of range. When we provided a range of numbers 58 through 62 as seconds, Skyfield added exactly the number of seconds we specified to the end of June and let the value overflow cleanly into the beginning of July.

Keep two consequences in mind when using UTC in your calculations.

First, expect an occasional jump or discrepancy if you are striding forward through time using the UTC minute, hour, or day. For example, an hourly plot of planet's position will show the planet moving slightly farther during an hour that was lengthened by a leap second than during other hours of the year. An Earth satellite's velocity will seem higher when you reach the minute that includes 61 seconds. And so forth. Problems like these are the reason that the Time class only uses UTC for input and output, and insists on keeping time internally using the uniform time scales discussed below in Uniform time scales: TAI, TT, and TDB.

Second, leap seconds disqualify the Python datetime from use as a general way to represent time because in many versions of Python the datetime refuses to accept seconds greater than 59:

```
datetime(2012, 6, 30, 19, 59, 60)

Traceback (most recent call last):

 ...

ValueError: second must be in 0..59
```

That limitation is why Skyfield offers a second version of each method that returns a datetime. These fancier methods return a leap-second flag as an additional return value:

- Time.utc_datetime_and_leap_second()
- Time.astimezone_and_leap_second()

The leap-second return value is usually 0 but jumps to 1 when Skyfield is forced to represent a leap second as a datetime with the incorrect time 23:59:59.

```
# Asking for the leap_second flag to learn the whole story


dt, leap_second = t.astimezone_and_leap_second(eastern)

for dt_i, leap_second_i in zip(dt, leap_second):

   print('{0}  leap_second = {1}'.format(dt_i, leap_second_i))

2012-06-30 19:59:58-04:00  leap_second = 0

2012-06-30 19:59:59-04:00  leap_second = 0

2012-06-30 19:59:59-04:00  leap_second = 1

2012-06-30 20:00:00-04:00  leap_second = 0

2012-06-30 20:00:01-04:00  leap_second = 0
```

Using calendar tuples to represent UTC times is more elegant than using Python datetime objects because leap seconds can be represented accurately. If your application cannot avoid using datetime objects, then you will have to decide whether to simply ignore the leap_second value or to somehow output the leap second information.

Date arithmetic

Dates support a few simple math operations:

```
from datetime import timedelta

t - 10                 # 10 days earlier
t + 0.25               # 6 hours later
t + timedelta(hours=12)  # 12 hours later

t2 - t1  # difference between times, in days
```

Raw numbers, like 10 and 0.25 above, specify days of Terrestrial Time — units of exactly 24 hours of 60 minutes of 60 SI seconds, measured in the Earth's relativistic frame of reference. If you increment or decrement a date across a leap second, you will notice that the clock time returned by Skyfield's UTC functions is one second earlier or later than you expect.

Date arrays

If you want to ask where a planet or satellite was across a whole series of times and dates, then Skyfield will work most efficiently if, instead of building many separate Time objects, you build a single Time object that holds the entire array of dates.

There are three techniques for building a Time array.

- Provide tai() or tt() or tdb() or ut1() with a Python list or NumPy array of numbers for one of the six components of the calendar date (year, month, day, hour, minute, or second).
- Provide tai_jd() or tt_jd() or tdb_jd() or ut1_jd() with a list or NumPy array of floating point numbers.
- Provide from_datetimes() with a Python list of datetime objects.

The first possibility is generally the one that is the most fun, because its lets you vary whichever time unit you want while holding the others constant. You are free to provide out-of-range values and leave it to Skyfield to work out the correct result. Here are some examples:

```
ts.utc(range(1900, 1950))     # Fifty years 1900–1949
ts.utc(1980, range(1, 25))    # 24 months of 1980 and 1981
ts.utc(2005, 5, [1, 11, 21])  # 1st, 11th, and 21st of May
```

```
# Negative values work too!  Here are the
# ten seconds crossing the 1974 leap second.
ts.utc(1975, 1, 1, 0, 0, range(-5, 5))
```

The resulting Time object will hold an array of times. As illustrated in the previous section (on leap seconds), you can use a Python for to print each time separately:

```
t = ts.utc(2020, 6, 16, 7, range(4))

for s in t.utc_strftime('%Y-%m-%d %H:%M'):
    print(s)
2020-06-16 07:00
2020-06-16 07:01
2020-06-16 07:02
2020-06-16 07:03
```

When you provide a time array as input to a Skyfield calculation, the output array will have an extra dimension that expands what would normally be a single result into as many results as you provided dates. We can compute the position of the Earth as an example:

```
# Single Earth position

planets = load('de421.bsp')
earth = planets['earth']
t = ts.utc(2014, 1, 1)
pos = earth.at(t).xyz.au
print(pos)
[-0.17461758  0.88567056  0.38384886]
# Whole array of Earth positions
days = [1, 2, 3, 4]
t = ts.utc(2014, 1, days)
pos = earth.at(t).xyz.au
print(pos)
[[-0.17461758 -0.19179872 -0.20891924 -0.22597338]
 [ 0.88567056  0.88265548  0.87936337  0.87579547]
 [ 0.38384886  0.38254134  0.38111391  0.37956709]]
```

Note the shape of the resulting NumPy array. If you unpack this array into three names, then you get three four-element arrays corresponding to the four dates. These four-element arrays are ready to be submitted to matplotlib and other scientific Python tools:

```
x, y, z = pos    # four values each
plot(x, y)       # example matplotlib call
```

If you instead slice along the second axis, then you can retrieve an individual position for a particular date — and the first position is exactly what was returned above when we computed the January 1st position by itself:

```
print(pos[:,0])
```

```
[-0.17461758  0.88567056  0.38384886]
```

You can combine a Python for loop with Python's zip() builtin to print each time alongside the corresponding coordinates. There are two techniques, one of which is less efficient and the other more efficient.

```
# Less efficient: loop over `t`, forcing the creation of
# a separate `Time` object for each iteration of the loop.


for ti, xi, yi, zi in zip(t, x, y, z):
    print('{}  x = {:.2f} y = {:.2f} z = {:.2f}'.format(
        ti.utc_strftime('%Y-%m-%d'), xi, yi, zi,
    ))
```

```
2014-01-01  x = -0.17 y = 0.89 z = 0.38
2014-01-02  x = -0.19 y = 0.88 z = 0.38
2014-01-03  x = -0.21 y = 0.88 z = 0.38
2014-01-04  x = -0.23 y = 0.88 z = 0.38
```

```
# More efficient: loop over the output of a `Time` method,
# which returns an array of the same length as `t`.


t_strings = t.utc_strftime('%Y-%m-%d')


for tstr, xi, yi, zi in zip(t_strings, x, y, z):
    print('{}  x = {:.2f} y = {:.2f} z = {:.2f}'.format(
        tstr, xi, yi, zi,
    ))
```

2014-01-01 x = -0.17 y = 0.89 z = 0.38

2014-01-02 x = -0.19 y = 0.88 z = 0.38

2014-01-03 x = -0.21 y = 0.88 z = 0.38

2014-01-04 x = -0.23 y = 0.88 z = 0.38

Finally, converting an array Time back into a calendar tuple results in the year, month, day, hour, minute, and second each having the same dimension as the array itself:

```
print(t.utc)

[[2014. 2014. 2014. 2014.]
 [   1.    1.    1.    1.]
 [   1.    2.    3.    4.]
 [   0.    0.    0.    0.]
 [   0.    0.    0.    0.]
 [   0.    0.    0.    0.]]
```

Simply slice across the second dimension of the array to pull a particular calendar tuple out of the larger result:

```
print(t.utc[:,2])

[2014.    1.    3.    0.    0.    0.]
```

Slicing in the other direction, the rows can be fetched not only by index but also through the attribute names year, month, day, hour, minute, and second.

```
print(t.utc.year)

print(t.utc.month)

print(t.utc.day)

print(t.utc.hour)

[2014. 2014. 2014. 2014.]

[1. 1. 1. 1.]

[1. 2. 3. 4.]

[0. 0. 0. 0.]
```

Uniform time scales: TAI, TT, and TDB

Date arithmetic becomes very simple as we leave UTC behind and consider completely uniform time scales. Days are always 24 hours, hours always 60 minutes, and minutes always 60 seconds without any variation or exceptions. Such time scales are not appropriate for your morning alarm clock because they are never delayed or adjusted to stay in sync with the

slowing rotation of the earth. But that is what makes them useful for astronomical calculation — because physics keeps up its dance, and the stars and planets move in their courses, whether humanity pauses to observe a UTC leap second or not.

Because they make every day the same length, uniform time scales can express dates as a simple floating-point count of days elapsed. To make all historical dates come out as positive numbers, astronomers traditionally assign each date a "Julian day" number that starts counting at 4713 BC January 1 in the old Julian calendar — the same date as 4714 BC November 24 in our Gregorian calendar. Following a tradition going back to the Greeks and Ptolemy, the count starts at noon, since the sun's transit is an observable event but the moment of midnight is not.

So twelve noon was the moment of Julian date zero:

```
# When was Julian date zero?


bc_4714 = -4713

t = ts.tt(bc_4714, 11, 24, 12)

print(t.tt)

0.0
```

Did you notice how negative years work — that we expressed 4714 BC using the negative number -4713? People still counted by starting at one, not zero, when the scholar Dionysius Exiguus created the eras BC and AD in around the year AD 500. So his scheme has 1 BC followed immediately by AD 1 without a break. To avoid an off-by-one error, astronomers usually ignore BC and count backwards through a year zero and on into negative years. So negative year −*n* is what might otherwise be called either "*n+1* BC" or "*n+1* BCE" in a history textbook.

More than two million days have passed since 4714 BC, so modern dates tend to be rather large numbers:

```
# 2014 January 1 00:00 UTC expressed as Julian dates


t = ts.utc(2014, 1, 1)

print('TAI = %r' % t.tai)

print('TT  = %r' % t.tt)

print('TDB = %r' % t.tdb)

TAI = 2456658.5004050927

TT  = 2456658.5007775924

TDB = 2456658.500777592
```

What are these three different uniform time scales?

International Atomic Time (TAI) is maintained by the worldwide network of atomic clocks referenced by researchers with a need for very accurate time. The official leap second table is actually a table of offsets between TAI and UTC. At the end of June 2012, for example, the TAI−UTC offset was changed from 34.0 to 35.0 which is what generated the leap second in UTC.

Terrestrial Time (TT) differs from TAI only because astronomers were already maintaining a uniform time scale of their own before TAI was established, using a slightly different starting point for the day. For practical purposes, TT is simply TAI plus exactly 32.184 seconds. So it is now more than a minute ahead of UTC.

You can not only ask Skyfield for TT as a Julian date and a calendar date, but as a floating-point number of years of exactly 365.25 days each — a value which is often used as the time parameter in long-term astronomical formulae:

```
print('Julian year = {:.4f}'.format(t.J))
```

```
Julian year = 2014.0000
```

Finally, Barycentric Dynamical Time (TDB) runs at approximately the rate that an atomic clock would run if it were at rest with respect to the Solar System barycenter, and therefore unaffected by the Earth's motion. The acceleration that Earth experiences in its orbit — sometimes speeding up, sometimes slowing down — varies the rate at which our atomic clocks run relative to an outside observer, as predicted by Einstein's theory of General Relativity. So physical simulations of the Solar System use TDB as their clock. It is considered equivalent to the T_{eph} time scale traditionally used for Solar System and spacecraft simulations at the Jet Propulsion Laboratory.

UT1 and downloading IERS data

Finally, UT1 is the least uniform time scale of all because its clock cannot be housed in a laboratory, nor is its rate established by any human convention. It is, rather, the clock whose "hand" is the rotation of the Earth itself! The direction that the Earth is facing determines not only the coordinates of every city and observatory in the world, but also the local directions that each site will designate as their local "up", "north", and "east".

It is hard to predict future values for UT1. The Earth is a young world with a still-molten iron core, a viscous mantle, and ice ages that move water weight into glaciers at the poles then release it back into the ocean. While we think we can predict, for example, Jupiter's position thousands of years from now, predicting the fluid dynamics of the elastic rotating ellipsoid we call home is — at the moment — beyond us. We can only watch with sensitive instruments to see what the Earth does next.

Skyfield relies on the IERS, the International Earth Rotation Service, for accurate measurements of UT1 and for the schedule of leap seconds (discussed above) that keeps UTC from straying more than 0.9 seconds away from UT1.

Each new version of Skyfield carries recent IERS data in internal tables. This data will gradually fall out of date after each Skyfield release, however, with two consequences:

- The next time the IERS declares a new leap second that is not listed in Skyfield's built-in tables, Skyfield's UTC time will be off by 1 second for every date that falls after the leap second.
- As the Earth's rotation speeds up or slows down in the coming years more than was predicted in Skyfield's built-in UT1 tables, Skyfield's idea of where the Earth is pointing will grow less accurate. This will affect both the position and direction of each GeographicPosition — whether used as an observer or a target — and will also affect Earth satellite positions.

You can avoid both of these problems by periodically downloading new data from the IERS. Simply specify that you don't want Skyfield to use its builtin tables. In that case timescale() will instead download finals2000A.all from the IERS:

```
# Download and use the `finals.all` file.


ts = load.timescale(builtin=False)
```

```
[####################################] 100% finals2000A.all
```

As usual with data files, Skyfield will only download the file the first time you need it, then will keep using that same copy of the file that it finds on disk.

Note that the international agencies responsible for the file's distribution sometimes have trouble keeping their servers up. For example, as I write this in May of 2022, the file cannot be fetched from ftp.iers.org because of an Outage of iers.org data servers reported on their website. At Skyfield issue #730 and Skyfield issue #732 you can find links to alternative data sources which various Skyfield users have been able to access in the meantime.

If your script will always have Internet access and you worry about the file falling out of date (and if you can trust the "modify time" file attribute on your filesystem), then you can have Skyfield download a new copy once the file on disk has grown too old (where "too old" for your application must be determined by comparing your accuracy needs with how quickly UT1 diverges without fresh IERS data; this example uses 30 days only as an illustration):

```
if load.days_old('finals2000A.all') > 30.0:
    load.download('finals2000A.all')


ts = load.timescale(builtin=False)
```

But, beware! For compatibility with versions of Skyfield ≤ 1.30, Skyfield will ignore finals2000A.all if the three old files deltat.data, deltat.preds, and Leap_Second.dat exist in the loader's directory, in which case it will use them instead. This is to prevent users who specify builtins=False, but who downloaded the three necessary files long ago, from experiencing an unexpected download attempt. The hope is that all scripts which did not previously need Internet access will continue to run without it.

If you ever want to display or plot the behavior of UT1, astronomers use two common conventions for stating the difference between clock time and UT1. Skyfield supports them both.

```
print('{:+.4f}'.format(t.dut1))

print('{:+.4f}'.format(t.delta_t))

-0.0970

+67.2810
```

The two quantities are:

- DUT1 — The difference between UTC and UT1, which should always be less than 0.9 if the IERS succeeds at its mission. Note that there are two different reasons that this value changes: every day it changes a small amount because of the drift of UT1; and superimposed on this drift is a big jump of 1.0 seconds every time a leap second passes.
- ΔT — The difference between TT and UT1. This is a much more straightforward value than DUT1, without all the ugly discontinuities caused by leap seconds. Because TT is a uniform timescale, ΔT provides a pure and continuous record of how UT1 has changed over the decades that we have been measuring the Earth's rotation to high precision.

Setting a Custom Value For ΔT

If you ever want to specify your own value for ΔT, then provide a delta_t keyword argument when creating your timescale:

```
load.timescale(delta_t=67.2810).utc((2014, 1, 1))
```

Values cached on the Time object

Saat Anda membuat Time , ia akan menghitung ttatribut Waktu Terestrialnya mulai dari argumen waktu apa pun yang Anda berikan. Jika Anda memberikan utcparameter, misalnya, maka tanggal akan dihitung dan ditetapkan terlebih dahulu, tai lalu dihitung dan ditetapkan tt. Setiap atribut waktu lainnya hanya dihitung sekali, saat pertama kali Anda mengaksesnya.

Aturan umumnya adalah bahwa atribut hanya dihitung satu kali, dan dapat diakses berulang kali secara gratis, sementara metode tidak pernah menyimpan hasil mereka dalam cache — anggap ()tanda kurung setelah nama metode sebagai pengingat bahwa "ini akan melakukan perhitungan baru setiap saat."

Selain skala waktu, setiap Timeobjek menyimpan beberapa kuantitas lain yang sering dibutuhkan dalam astronomi. Skyfield hanya menghitung atribut ini sesuai permintaan, saat pertama kali pengguna mencoba mengaksesnya atau memanggil komputasi yang membutuhkan nilainya:

gmst

Waktu Sideris Rata-Rata Greenwich dalam jam, dalam kisaran 0,0 ≤ gmst < 24,0.

gast

Waktu Sideris Tampak Greenwich dalam jam, dalam kisaran 0,0 ≤ gast < 24,0.

MBahasa Indonesia:MT

Matriks 3x3 ini dan inversnya melakukan rotasi lengkap antara vektor dalam ICRF dan vektor dalam sistem referensi dinamis untuk waktu dan tanggal ini.

CBahasa Indonesia:CT

Matriks 3×3 ini dan kebalikannya melakukan rotasi lengkap antara vektor dalam ICRF dan vektor dalam sistem referensi antara langit (CIRS) pada waktu dan tanggal ini.

Anda biasanya tidak perlu mengakses matriks ini sendiri, karena matriks tersebut digunakan secara otomatis oleh radec() metode saat Anda menggunakan epoch=parameternya untuk meminta asensio dan deklinasi rektus dalam sistem referensi dinamis, dan saat Anda menanyakan GeographicPositionposisi suatu objek.

Planet dan bulannya: berkas ephemeris JPL

Untuk planet dan bulannya, Laboratorium Propulsi Jet (JPL) NASA menawarkan tabel posisi dengan akurasi tinggi untuk rentang waktu mulai dari beberapa dekade hingga beberapa abad. Setiap tabel disebut *ephemeris* , dari kata Yunani kuno yang berarti *harian* . Berikut cara mengunduh dan membuka ephemeris DE421 JPL dan menanyakan posisi Mars:

```
from skyfield.api import load


ts = load.timescale()
t = ts.utc(2021, 2, 26, 15, 19)


planets = load('de421.bsp')  # ephemeris DE421
mars = planets['Mars Barycenter']
barycentric = mars.at(t)
```

Atau Anda dapat menghitung posisi Mars dari sudut pandang lain di Tata Surya:

```
earth = planets['Earth']
astrometric = earth.at(t).observe(mars)
```

Untuk rinciannya:

- Lihat Mengunduh dan Menggunakan Berkas Data untuk mempelajari lebih lanjut tentang load()rutinitas dan cara memilih direktori tempat mengunduh berkas. Perhatikan bahwa berkas ephemeris tidak pernah menerima pembaruan, jadi setelah Anda memiliki berkas seperti de421.bspdi disk, Anda tidak perlu mengunduhnya lagi.
- Lihat Posisi untuk mempelajari apa yang dapat Anda lakukan dengan barycentricdan astrometricposisi yang dihitung dalam kode di atas.

Untuk mempelajari lebih lanjut tentang berkas ephemeris itu sendiri, teruslah membaca di sini.

Memilih Ephemeris

Berikut ini adalah berkas ephemeris serbaguna yang paling populer, dari seri Ephemeris Pengembangan “DE” JPL yang terkenal.

Diterbitkan	Pendek	Sedang	Panjang
tahun 1997		de405.bsp 1600 hingga 2200	de406.bsp -3000 hingga 3000

Diterbitkan	Pendek	Sedang	Panjang
		63MB	287MB
	de421.bsp		de422.bsp
Tahun 2008	1900 hingga 2050		-3000 hingga 3000
	17 juta		623MB
	de430_1850-2150.bsp	de430t.bsp	de431t.bsp
Tahun 2013	1850 hingga 2150	1550 hingga 2650	–13200 hingga 17191
	31 juta	128MB	3,5 GB
	de440s.bsp	de440.bsp	de441.bsp
Tahun 2020	1849 hingga 2150	1550 hingga 2650	-13200 hingga 17191
	32 juta	114MB	3,1 GB

Anda dapat menemukan lebih banyak berkas ephemeris di Internet pada berbagai tautan unduhan Ephemeris yang tercantum di dekat bagian bawah halaman ini.

Bagaimana Anda dapat memilih ephemeris yang tepat untuk proyek Anda?

1. Pastikan planet, bulan, atau benda lain yang ingin Anda amati termasuk dalam target yang didukung oleh ephemeris. Lihat Mencantumkan target yang didukung ephemeris di bawah ini jika Anda ingin mengunduh berkas ephemeris dan memeriksa sendiri isinya. Beberapa direktori ephemeris daring menyertakan indeks yang mencantumkan isi setiap ephemeris dalam direktori; cari berkas dengan nama seperti README.txtatau aa_summaries.txt.
2. Pilih ephemeris terpendek yang akan mencakup tanggal yang dibutuhkan proyek Anda. File yang lebih pendek tidak hanya akan menghemat waktu pengunduhan dan ruang disk, tetapi ephemeris jangka pendek terbaru DE430 dan DE440 lebih akurat daripada ephemeris jangka panjang DE431 dan DE441 karena file yang lebih pendek menyertakan efek inti cair Bulan, efek yang "tidak cocok untuk integrasi mundur lebih dari beberapa abad."

Jika Anda memerlukan rentang tanggal kuno tertentu tetapi menemukan ephemeris jangka panjang terlalu besar, baca bagian di bawah ini tentang Membuat kutipan ephemeris — Anda seharusnya dapat membuat file yang lebih kecil yang hanya menyertakan rentang tanggal yang Anda butuhkan.

3. The most recent ephemeris files should be the most accurate because they are built using the highest accuracy data from humankind’s telescopes and spacecraft.

Note that ephemeris files don’t provide numbers for how accurate their planet positions are. Sometimes you can find ballpark estimates of accuracy in an ephemeris file’s official report. The report *The Planetary and Lunar Ephemerides DE430 and DE431*, for example, states that:

- “The orbits of the inner planets are known to subkilometer accuracy”

- “an accuracy of 0”.0002 … is the limiting error source for the orbits of the terrestrial planets, and corresponds to orbit uncertainties of a few hundred meters.”
- “The orbits of Jupiter and Saturn are determined to accuracies of tens of kilometers”
- “Uranus, Neptune, and Pluto … observations … limit position accuracies to several thousand kilometers.”

You can find links to the various ephemeris reports in the Ephemeris bibliography at the bottom of this page. Some ephemeris files also have a built-in text summary you can print to the screen:

```
print(planets.spk.comments())

; de421.bsp LOG FILE

;

; Created 2008-02-12/11:33:34.00.

...
```

The more recent ephemeris files tend to have the most informative comment texts.

I myself use DE421 and DE422 because they’re small, accurate, and cover long enough time periods for all of my projects. When the time comes to upgrade, I’ll probably move next to DE440; its short-term de440s.bsp file is especially attractive because it’s only twice the size of DE421 while delivering higher accuracy plus an extra century of data.

To find out what difference the choice of ephemeris makes, simply run a sample calculation with one ephemeris then again using another. The difference will usually be far below the resolution of your instruments unless (a) you’re doing radio astronomy or (b) you’re planning to place an actual payload in orbit around the target body.

Listing the targets that an ephemeris supports

You can use print() to check whether an ephemeris lists a specific planet or moon. Here, for example, are the targets supported by DE421:

```
print(planets)

SPICE kernel file 'de421.bsp' has 15 segments

 JD 2414864.50 - JD 2471184.50  (1899-07-28 through 2053-10-08)

    0 -> 1    SOLAR SYSTEM BARYCENTER -> MERCURY BARYCENTER

    0 -> 2    SOLAR SYSTEM BARYCENTER -> VENUS BARYCENTER

    0 -> 3    SOLAR SYSTEM BARYCENTER -> EARTH BARYCENTER

    0 -> 4    SOLAR SYSTEM BARYCENTER -> MARS BARYCENTER

    0 -> 5    SOLAR SYSTEM BARYCENTER -> JUPITER BARYCENTER

    0 -> 6    SOLAR SYSTEM BARYCENTER -> SATURN BARYCENTER

    0 -> 7    SOLAR SYSTEM BARYCENTER -> URANUS BARYCENTER
```

```
0 -> 8    SOLAR SYSTEM BARYCENTER -> NEPTUNE BARYCENTER
0 -> 9    SOLAR SYSTEM BARYCENTER -> PLUTO BARYCENTER
0 -> 10   SOLAR SYSTEM BARYCENTER -> SUN
3 -> 301  EARTH BARYCENTER -> MOON
3 -> 399  EARTH BARYCENTER -> EARTH
1 -> 199  MERCURY BARYCENTER -> MERCURY
2 -> 299  VENUS BARYCENTER -> VENUS
4 -> 499  MARS BARYCENTER -> MARS
```

Skyfield can generate positions for any body that an ephemeris links to target zero, the Solar System barycenter. For example, DE421 — as you can see above — provides a segment directly linking the Solar System barycenter with the Sun:

```
sun = planets['Sun']
print(sun)
'de421.bsp' segment 0 SOLAR SYSTEM BARYCENTER -> 10 SUN
```

By contrast, generating a position for the Moon with DE421 requires Skyfield to add together two segments behind the scenes. The first segment provides the position of the Earth-Moon center of gravity, while the second segment provides the offset from there to the Moon.

```
moon = planets['Moon']
print(moon)
Sum of 2 vectors:
 'de421.bsp' segment 0 SOLAR SYSTEM BARYCENTER -> 3 EARTH BARYCENTER
 'de421.bsp' segment 3 EARTH BARYCENTER -> 301 MOON
```

Note that most planets are so massive compared to their moons that you can ignore the difference between the planet and its system barycenter. If you want to observe Mars or Jupiter from elsewhere in the Solar System, just ask for the Mars Barycenter or Jupiter Barycenter position instead. The Earth-Moon system is unusual for featuring a satellite with so much mass — though even in that case, their common barycenter is always inside the Earth. Only Pluto has a satellite so massive and so distant that the Pluto-Charon barycenter is in space between them.

Making an excerpt of an ephemeris

Several of the ephemeris files listed below are very large. While most programmers will follow the example above and use DE421, if you wish to go beyond its 150-year period you will need a larger ephemeris. And programmers interested in the moons of Jupiter will need JUP310, which weighs in at nearly a gigabyte.

What if you need data from a very large ephemeris, but don't require its entire time span?

When you installed Skyfield another library named jplephem will have been installed. When invoked from the command line, it can build an excerpt of a larger ephemeris without needing to download the entire file, thanks to the fact that HTTP supports a Range: header that asks for only specific bytes of a file. For example, let's pull two weeks of data for Jupiter's moons (using a shell variable $u for the URL only to make the command less wide here on the screen and easier to read):

```
$ u=https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/jup310.bsp

$ python -m jplephem excerpt 2018/1/1 2018/1/15 $u jup_excerpt.bsp
```

The resulting file jup_excerpt.bsp weighs in at only 0.2 MB instead of 932 MB but supports all of the same objects as the original JUP310 over the given two-week period:

```
$ python -m jplephem spk jup_excerpt.bsp

File type DAF/SPK and format LTL-IEEE with 13 segments:

2458119.75..2458210.50  Jupiter Barycenter (5) -> Io (501)

2458119.50..2458210.50  Jupiter Barycenter (5) -> Europa (502)

2458119.00..2458210.50  Jupiter Barycenter (5) -> Ganymede (503)

2458119.00..2458210.50  Jupiter Barycenter (5) -> Callisto (504)

...
```

You can load and use it directly off of disk with load_file().

Closing the ephemeris file automatically

If you need to close files as you finish using them instead of waiting until the application exits, each Skyfield ephemeris offers a close() method. You can either call it manually, or use Python's closing() context manager to call close() automatically when a block of code finishes:

```
from contextlib import closing


ts = load.timescale()

t = ts.J2000


with closing(planets):

  planets['venus'].at(t)  # Ephemeris can be used here
```

Type 1 and Type 21 ephemeris formats

If you generate an ephemeris with a tool like NASA's HORIZONS system, it might be in a format not yet natively supported by Skyfield. The first obstacle to opening the ephemeris might be its lack of a recognized suffix:

```
load('wld23593.15')
```

```
Traceback (most recent call last):
  ...
ValueError: Skyfield does not know how to open a file named 'wld23593.15'
```

A workaround for the unusual filename extension is to open the file manually using Skyfield's JPL ephemeris support. The next obstacle, however, will be a lack of support for Type 21 ephemerides in Skyfield:

```
from skyfield.jpllib import SpiceKernel
kernel = SpiceKernel('wld23593.15')
Traceback (most recent call last):
  ...
ValueError: SPK data type 21 not yet supported
```

Older files with a similar format might instead generate the complaint "SPK data type 1 not yet supported."

Happily, thanks to Shushi Uetsuki, a pair of third-party libraries exist that offer preliminary support for Type 1 and Type 21 ephemerides!

- https://pypi.org/project/spktype01/
- https://pypi.org/project/spktype21/

Dokumentasi mereka sudah menyertakan contoh pembuatan koordinat mentah, tetapi banyak pengguna Skyfield ingin menggunakannya bersama dengan metode Skyfield standar seperti observe(). Untuk mengintegrasikannya dengan Skyfield lainnya, Anda perlu menentukan kelas fungsi vektor baru yang memanggil modul pihak ketiga untuk membuat koordinat:

```
from skyfield.constants import AU_KM
from skyfield.vectorlib import VectorFunction
from spktype21 import SPKType21

t = ts.utc(2020, 6, 9)

eph = load('de421.bsp')
earth = eph['earth']

class Type21Object(VectorFunction):
    def __init__(self, kernel, target):
```

```
        self.kernel = kernel
        self.center = 0
        self.target = target

    def _at(self, t):
        k = self.kernel
        r, v = k.compute_type21(0, self.target, t.whole, t.tdb_fraction)
        return r / AU_KM, v / AU_KM, None, None

kernel = SPKType21.open('wld23593.15')
chiron = Type21Object(kernel, 2002060)

ra, dec, distance = earth.at(t).observe(chiron).radec()
print(ra)
print(dec)
00h 27m 38.99s
+05deg 57' 08.9"
```

Mudah-mudahan dukungan pihak ketiga untuk segmen ephemeris SPK Tipe 1 dan Tipe 23 ini akan mencukupi untuk proyek yang membutuhkannya, hingga ada waktu bagi kontributor Skyfield untuk mengintegrasikan dukungan tersebut ke dalam Skyfield itu sendiri.

Tautan unduhan Ephemeris

Skyfield mengetahui URL untuk setiap berkas ephemeris yang paling populer — jadi Anda dapat mengunduhnya dengan mudah load('de421.bsp') dan Skyfield akan tahu di mana mengunduh berkas tersebut jika Anda belum memilikinya di disk. Namun, untuk berkas ephemeris yang kurang dikenal, Anda mungkin perlu memberikan load()URL lengkapnya.

Berikut ini beberapa situs umum tempat Anda dapat mengunduh berkas ephemeris resmi:

- Untuk planet:

ftp://ssd.jpl.nasa.gov/pub/eph/planets/bsp/

https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/

- Untuk planet bulan:

ftp://ssd.jpl.nasa.gov/pub/eph/satellites/bsp/

https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/

Bibliografi ephemeris

DE405 / DE406

- Ephemeris Planet dan Bulan JPL, DE405/LE405 (Standish 1998)
- Periksa JPL DE405 menggunakan pengamatan optik modern (Morrison dan Evans 1998)
- Posisi CCD untuk Planet Luar pada tahun 1996–1997 Ditentukan dalam Kerangka Acuan Ekstragalaksi (Stone 1998)
- Astrometri Pluto dan Saturnus dengan instrumen meridian CCD Bordeaux dan Valinhos (Rapaport, Teixeira, Le Campion, Ducourant1, Camargo, Benevides-Soares 2002)

DE421

- Ephemeris Planet dan Bulan DE421 (Folkner, Williams, Boggs 2009)

DE430 / DE431

- Ephemeris Planet dan Bulan DE430 dan DE431 (Folkner, Williams, Boggs, Park, Kuchynka 2014)
- DE430 Orbit Bulan, Librasi Fisik dan Koordinat Permukaan (Williams, Boggs, Folkner 2013)

DE440 / DE441

- Ephemeris Planet dan Lunar JPL DE440 dan DE441 (Park, Folkner, Williams, dan Boggs 2021)

Analisis menyebutkan beberapa ephemerides

- Pemodelan Ketidakpastian Ephemeris Tata Surya untuk Pencarian Gelombang Gravitasi yang Kuat dengan Pulsar Timing Array (Kolaborasi NANOGrav 2020)

.bsp Dokumentasi format file

- SPICE toolkit: Bacaan Wajib SPK (menjelaskan .bspberkas)
- SPICE toolkit: Double Precision Array Files (DAF) (menjelaskan format biner)

Bintang dan Objek Jauh

Skyfield dapat menghasilkan posisi bintang atau objek jauh lainnya yang Anda muat dari katalog bintang, atau yang koordinat ICRS-nya dapat Anda berikan.

Katalog Hipparcos

Salah satu katalog bintang yang paling populer adalah katalog Hipparcos, yang disusun dari hasil pengamatan teleskop satelit Badan Antariksa Eropa tahun 1989–1993. Ejaannya disengaja; nama astronom Yunani kuno tersebut adalah "Hipparchus" tetapi nama observatorium dan katalog yang dihasilkan merupakan akronim dari "High Precision Parallax Collecting Satellite."

Untuk memuat katalog bintang besar ini dengan Skyfield diperlukan pustaka data Pandas . Jika Anda menggunakan distribusi Python ilmiah Anaconda, Anda dapat menginstal Pandas dengan:

```
conda install pandas
```

Jika tidak, Anda dapat mencoba menginstalnya dengan pip alat instalasi paket Python standar:

```
pip install pandas
```

Apa pun cara Anda menginstalnya, pustaka Pandas memberi Skyfield kemampuan untuk memuat katalog Hipparcos sebagai kerangka data Pandas dengan 118.218 baris. Anda dapat memfilter bintang-bintang individual dari kerangka data menggunakan loc[]operasi Pandas dengan nomor katalog Hipparcos bintang tersebut. Misalnya, Bintang Barnard — bintang yang bergerak paling cepat melintasi langit kita — memiliki sebutan Hipparcos "HIP 87937" dan dapat diakses seperti ini:

```
from skyfield.api import Star, load
from skyfield.data import hipparcos

with load.open(hipparcos.URL) as f:
    df = hipparcos.load_dataframe(f)

barnards_star = Star.from_dataframe(df.loc[87937])
```

Untuk menghasilkan posisi bintang di langit Bumi pada tanggal tertentu, hitung posisi Bumi di Tata Surya dan kemudian observe() bintangnya:

```
planets = load('de421.bsp')
earth = planets['earth']

ts = load.timescale()
```

```
t = ts.now()

astrometric = earth.at(t).observe(barnards_star)

ra, dec, distance = astrometric.radec()

print(ra)

print(dec)

17h 57m 47.77s

+04deg 44' 01.1"
```

Posisi tersebut akan mencerminkan gerak bintang yang sebenarnya sebagaimana diukur oleh misi HIPPARCOS.

Mencari bintang tertentu

Kami mencari Barnard's Star pada contoh di atas menggunakan nomor katalog Hipparcos, HIP 87937. Jika proyek Anda menargetkan beberapa bintang tertentu, Anda dapat mempelajari nomor katalog HIP mereka melalui sejumlah mesin pencari daring. Berikut ini adalah pilihan yang populer:

https://vizier.u-strasbg.fr/viz-bin/VizieR?-source=1239/hip_main

Referensi umum seperti Wikipedia juga cenderung memberikan nomor HIP saat menjelaskan sebuah bintang, jadi pencarian pada halaman bintang dengan akronim “HIP” kemungkinan besar akan mengarahkan Anda ke nomor Hipparcos-nya.

Bintang dengan posisi “nan”

Katalog Hipparcos, sebagaimana ditunjukkan dalam publikasi resminya The Hipparcos and Tycho Catalogues , mencakup 263 bintang target yang tidak dapat dihitung posisinya secara akurat. Bintang-bintang ini memiliki tanda kosong untuk asensio dan deklinasinya dalam teks katalog itu sendiri. Di Skyfield, mereka akan selalu mengembalikan koordinat dengan nilai nan(nilai floating point "bukan angka").

Jika Anda ingin menghindari nankoordinat, Anda dapat memfilternya dari kerangka data Anda dengan:

```
df = df[df['ra_degrees'].notnull()]
```

The result will be the same if you filter by the dec_degrees column instead.

Filtering the star catalog

In addition to selecting individual stars using their HIP number, you can build a Star object that contains all of the stars in a dataframe. You can combine this with the standard Pandas dataframe filtering techniques to select only stars above a certain brightness.

For example, let’s imagine that we wanted to plot the stars in the constellation Orion — but only stars that are at least magnitude 2.5. Pandas will let us filter the Hipparcos catalog dataframe in a single line of code:

```
df = df[df['magnitude'] <= 2.5]
```

```
print('After filtering, there are {} stars'.format(len(df)))
```

After filtering, there are 93 stars

You can use this dataframe to build a Star object that will compute the positions of all 93 stars at once.

```
bright_stars = Star.from_dataframe(df)


t = ts.utc(2018, 9, 3)
astrometric = earth.at(t).observe(bright_stars)
ra, dec, distance = astrometric.radec()


print('There are {} right ascensions'.format(len(ra.hours)))
print('and {} declinations'.format(len(dec.degrees)))
```

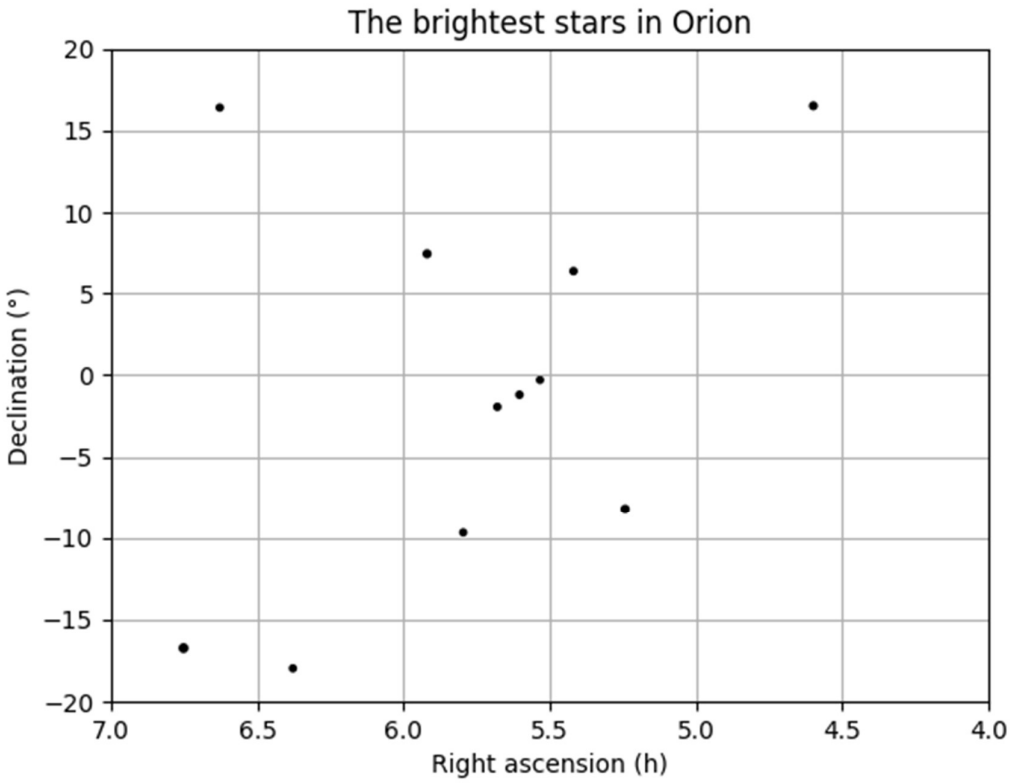
There are 93 right ascensions

and 93 declinations

Each element of the right ascension array and the declination array corresponds to one of the 93 selected stars. Their position can be combined with their magnitude to produce a plot.

```
from matplotlib import pyplot as plt


fig, ax = plt.subplots()
ax.scatter(ra.hours, dec.degrees, 8 - df['magnitude'], 'k')
ax.set_xlim(7.0, 4.0)
ax.set_ylim(-20, 20)
ax.grid(True)
ax.set(title='The brightest stars in Orion',
    xlabel='Right ascension (h)', ylabel='Declination (°)')
fig.savefig('bright_stars.png')
```

The result of the simple filtering and plotting is an (admittedly primitive) rendering of Orion!



For a more complete example of code that draws a star chart, see Drawing a finder chart for comet NEOWISE.

Building a single star from its coordinates

If instead of loading up a whole star catalog you have the coordinates for one particular star you are interested in, you can instantiate a Star directly using keyword arguments. Right ascension and declination can be specified as traditional base-60 coordinates, where the fractions are “minutes” and “seconds”, if you pass tuples instead of floats:

```
from skyfield.api import Star, load

planets = load('de421.bsp')

earth = planets['earth']


barnard = Star(ra_hours=(17, 57, 48.49803),

               dec_degrees=(4, 41, 36.2072))


ts = load.timescale()

t = ts.now()

astrometric = earth.at(t).observe(barnard)
```

```
ra, dec, distance = astrometric.radec()
print(ra)
print(dec)
17h 57m 48.50s
+04deg 41' 36.2"
```

Of course, this has simply returned the same position. More interesting is that we can ask for the position to be expressed in the dynamical reference system of the moving Earth instead of in the same fixed ICRS system in which we provided it:

```
ra, dec, distance = astrometric.radec(epoch=ts.now())
print(ra)
print(dec)
17h 58m 30.80s
+04deg 41' 41.9"
```

Distances to stars

The distance that Skyfield computes to a particular star might or might not be meaningful, depending both on which star catalog you are using and on which particular star you observe.

Skyfield has no way to even guess the distance to a star if it has only been given its right ascension and declination, as is the case for Barnard's Star as we defined it above. To represent this vector of unknown length, Skyfield generates a vector exactly one gigaparsec long:

```
print(distance)  # exactly 1 gigaparsec
2.06265e+14 au
```

This is so very far outside of our galaxy that such positions should be easy for you to tell apart from real distances to stars. Real distances will all be within a few hundred parsecs or less, because our telescopes cannot currently measure the parallax of stars that are any further away than that.

In the next section, we will redefine Barnard's Star and this time supply a real value for its parallax from a recent star catalog. You will see in that its distance switches to a realistic estimate.

Proper motion and parallax

Ada cara lain agar Starobjek dapat menghasilkan posisi yang berbeda dari asensio dan deklinasi ICRS lama yang biasa digunakan untuk menginisialisasinya. Pertama, Skyfield akan mengenali bahwa objek harus mengubah posisi selama bertahun-tahun jika kita menentukan gerak sebenarnya dalam milidetik busur ("mas") per tahun. Kedua, Skyfield akan menggeser posisi tampak relatif terhadap lokasi tempat Anda mengamatinya jika Anda memberikan paralaks yang menunjukkan bahwa bintang tersebut cukup dekat dengan Tata Surya sehingga

kita dapat mengukur jaraknya. Terakhir, bahkan ada efek kecil dari kecepatan radialnya jika diketahui.

Berikut ini ketiga efek yang diterapkan pada bintang Barnard:

```
barnard = Star(ra_hours=(17, 57, 48.49803),
 dec_degrees=(4, 41, 36.2072),
 ra_mas_per_year=-798.71,
 dec_mas_per_year=+10337.77,
 parallax_mas=545.4,
 radial_km_per_s=-110.6)


astrometric = earth.at(t).observe(barnard)
ra, dec, distance = astrometric.radec()
print(ra)
print(dec)
print(distance)
17h 57m 47.76s
+04deg 44' 01.3"
3.77863e+05 au
```

Perhatikan bahwa posisi di atas berbeda dengan masukan asensio rekta dan deklinasi, tetapi bukan karena kami meminta koordinat dinamis. Posisi ini berada dalam koordinat ICRS tetap dan menunjukkan gerakan nyata pada bagian bintang Barnard di langit kita.

Lihat panduan Posisi untuk mempelajari operasi yang dapat Anda lakukan dengan posisi astrometri ini setelah menggunakan Star objek untuk menghasilkannya.

Posisi pada zaman selain J2000

Beberapa katalog bintang menyediakan posisi pada zaman selain J2000. Misalnya, katalog bintang yang diamati oleh teleskop antariksa Hipparcos menyediakan posisi mereka pada J1991.25.

Zaman dapat diberikan sebagai argumen epochuntuk Star. Berikut adalah asensio dan deklinasi yang diberikan untuk Bintang Barnard dalam katalog Hipparcos:

```
hipparcos_epoch = ts.tt(1991.25)
barnard2 = Star(ra_hours=(17, 57, 48.97),
  dec_degrees=(4, 40, 5.8),
  ra_mas_per_year=-797.84,
```

```
    dec_mas_per_year=+10326.93,
    epoch=hipparcos_epoch)

ra, dec, distance = earth.at(t).observe(barnard2).radec()
print(ra)
print(dec)
17h 57m 47.75s
+04deg 44' 01.3"
```

Seperti yang Anda lihat, posisi yang dikembalikan untuk tanggal tersebut tsama, meskipun kami menginisialisasi Starobjek ini dengan posisi untuk Bintang Barnard lebih dari 8 tahun lebih awal daripada posisi yang kami gunakan dalam contoh pertama kami.

Kerangka Acuan Planet

Skyfield memiliki dukungan terbatas untuk kerangka acuan planet sebagaimana didefinisikan dalam berkas data Jet Propulsion Lab. Skyfield mendukung:

- Memuat .tfdan .tpcberkas teks yang mendefinisikan berbagai macam konstanta planet.
- Memuat .bcpsusunan biner yang memprediksi orientasi benda-benda Tata Surya dalam ruang pada rentang tanggal tertentu.
- Diberikan garis lintang dan garis bujur pada benda Tata Surya seperti Bulan yang (a) berbentuk bulat dan (b) orientasinya ditentukan oleh rangkaian data dalam sebuah .bcpberkas, yang menghitung letak posisi permukaan tersebut di ruang angkasa — yang memungkinkan pengamatan baik dari sudut pandang posisi permukaan tersebut, atau pengamatan posisi permukaan tersebut bagi pengamat lain di Tata Surya.

Namun, hal ini membuat beberapa fitur dari file tersebut masih belum didukung. Skyfield:

- *Belum dapat* mendukung kerangka acuan planet yang orientasinya ditetapkan sebagai rangkaian angka pada berkas teks.
- *Belum dapat* menghitung posisi lintang dan bujur pada permukaan objek non-bola, yang ketiga sumbunya dalam berkas teks tidak sama panjang.

Diharapkan dukungan untuk fitur-fitur ini akan ditambahkan suatu hari nanti, menjadikan dukungan Skyfield untuk konstanta planet menjadi lengkap.

Mengamati lokasi Bulan

Beginilah cara Anda memuat cukup data untuk memprediksi di mana di langit Anda akan mengarahkan teleskop untuk melihat lintang dan bujur tertentu di Bulan — dalam contoh ini, kawah Aristarchus yang terkenal.

```
from skyfield.api import PlanetaryConstants, load


ts = load.timescale()

t = ts.utc(2019, 12, 20, 11, 5)


eph = load('de421.bsp')

earth, moon = eph['earth'], eph['moon']


pc = PlanetaryConstants()

pc.read_text(load('moon_080317.tf'))
```

```
pc.read_text(load('pck00008.tpc'))
pc.read_binary(load('moon_pa_de421_1900-2050.bpc'))

frame = pc.build_frame_named('MOON_ME_DE421')
aristarchus = moon + pc.build_latlon_degrees(frame, 26.3, -46.8)

apparent = earth.at(t).observe(aristarchus).apparent()
ra, dec, distance = apparent.radec(epoch='date')
print(ra)
print(dec)
```

```
13h 03m 22.96s
-00deg 55' 27.3"
```

Jika lokasi Bulan Anda memiliki ketinggian bukan nol yang menempatkannya di atas atau di bawah “permukaan laut” Bulan, Anda dapat memberikan

argumen build_latlon_degrees() tambahan elevation_m.

Mengamati dari lokasi Bulan

Anda juga dapat bertanya kepada Skyfield: di bagian langit manakah seorang astronot yang berdiri di Bulan akan melihat benda tertentu di Tata Surya? Jawabannya dapat diberikan dalam koordinat asensio dan deklinasi rektal terhadap latar belakang bintang, atau dalam ketinggian dan azimuth yang diukur terhadap cakrawala astronot.

```
from skyfield.api import PlanetaryConstants, load

ts = load.timescale()
t = ts.utc(2019, 12, 20, 11, 5)

eph = load('de421.bsp')
earth, moon = eph['earth'], eph['moon']

pc = PlanetaryConstants()
pc.read_text(load('moon_080317.tf'))
pc.read_text(load('pck00008.tpc'))
```

```
pc.read_binary(load('moon_pa_de421_1900-2050.bpc'))


frame = pc.build_frame_named('MOON_ME_DE421')
aristarchus = moon + pc.build_latlon_degrees(frame, 26.3, -46.8)


apparent = aristarchus.at(t).observe(earth).apparent()
ra, dec, distance = apparent.radec(epoch='date')
print(ra)
print(dec)


alt, az, distance = apparent.altaz()
print(alt, 'degrees above the horizon')
print(az, 'degrees around the horizon from north')
01h 03m 22.96s
+00deg 55' 27.3"
32deg 27' 09.7" degrees above the horizon
118deg 12' 55.9" degrees around the horizon from north
```

Menghitung titik sub-solar di Bulan

Ini tidak hanya berlaku untuk Matahari, tetapi juga untuk semua objek yang menjadi target. Anda dapat mempelajari garis lintang dan garis bujur Bulan tepat di bawah objek tersebut dengan mengamati objek dari pusat Bulan, lalu menanyakan kerangka acuan bulan tentang garis lintang dan garis bujur.

```
sun = eph['Sun']


p = moon.at(t).observe(sun).apparent()
lat, lon, distance = p.frame_latlon(frame)
lon_degrees = (lon.degrees + 180.0) % 360.0 - 180.0
print('Sub-solar latitude: {:.1f} degrees'.format(lat.degrees))
print('Sub-solar longitude: {:.1f} degrees'.format(lon_degrees))
Sub-solar latitude: 0.3 degrees
Sub-solar longitude: -104.9 degrees
```

Menghitung librasi bulan

Librasi Bulan dinyatakan sebagai lintang dan bujur lokasi Bulan yang saat ini paling dekat dengan Bumi. Konvensi yang berlaku adalah bahwa perbedaan geometris sederhana antara posisi Bumi dan Bulan digunakan, bukan posisi yang tertunda cahaya. Jadi:

```
p = (earth - moon).at(t)

lat, lon, distance = p.frame_latlon(frame)

lon_degrees = (lon.degrees + 180.0) % 360.0 - 180.0

print('Libration in latitude: {:.3f}'.format(lat.degrees))

print('Libration in longitude: {:.3f}'.format(lon_degrees))
```

```
Libration in latitude: -6.749

Libration in longitude: 1.520
```

Satu-satunya kehalusan adalah bahwa garis bujur librasi tidak dinyatakan sebagai angka antara 0° dan 360°, seperti yang biasa terjadi pada garis bujur, tetapi sebagai offset positif atau negatif dari nol, yang diselesaikan kode di atas dengan beberapa pengurangan cepat dan modulo.

Menghitung matriks rotasi mentah

Jika Anda memanipulasi vektor secara langsung, Anda mungkin hanya ingin Skyfield menghitung matriks rotasi NumPy untuk memutar vektor dari ICRF ke dalam kerangka acuan benda Tata Surya. Objek frameyang dikembalikan di atas dapat mengembalikan matriks ini secara langsung. Jika diberikan satu waktu t, hasilnya akan menjadi matriks 3×3 sederhana.

```
from skyfield.api import PlanetaryConstants, load


ts = load.timescale()
t = ts.utc(2019, 12, 20, 11, 5)


pc = PlanetaryConstants()
pc.read_text(load('moon_080317.tf'))
pc.read_binary(load('moon_pa_de421_1900-2050.bpc'))


frame = pc.build_frame_named('MOON_ME_DE421')


R = frame.rotation_at(t)
```

```
print(R.shape)
```

```
(3, 3)
```

Sebaliknya, suatu array waktu akan mengembalikan suatu array matriks yang dimensi terakhirnya sedalam panjang vektor waktu.

```
t = ts.utc(2019, 12, 20, 11, range(5, 15))

R = frame.rotation_at(t)

print(t.shape)

print(R.shape)
```

```
(10,)

(3, 3, 10)
```

Transpos R.Tmatriks rotasi dapat digunakan untuk memutar vektor yang sudah ada dalam kerangka referensi tubuh kembali menjadi vektor ICRF standar.

Orbit Kepler

Skyfield sekarang menawarkan dukungan dasar untuk menghitung posisi komet atau planet minor yang orbit elips, parabola, atau hiperboliknya disediakan sebagai elemen orbit Kepler.

Hati-hati bahwa rutin internal yang mendukung orbit Kepler masih dalam tahap dasar dan dapat berubah — hanya antarmuka yang didokumentasikan di sini yang dijamin berfungsi pada versi Skyfield mendatang.

Skyfield memuat elemen orbital dari berkas teks menggunakan pustaka Pandas. Instal sebelum mencoba salah satu contoh di bawah ini:

pip install pandas

Komet

Pusat Planet Minor IAU mendistribusikan berkas CometEls.txtelemen orbital untuk memprediksi posisi komet. Berkas tersebut berupa teks biasa, jadi jangan ragu untuk membukanya dengan penyunting teks untuk melihat komet yang elemen orbitalnya ditawarkan. Untuk membangun kerangka data komet:

```
from skyfield.api import load
from skyfield.data import mpc

with load.open(mpc.COMET_URL) as f:
    comets = mpc.load_comets_dataframe(f)

print(len(comets), 'comets loaded')
```

```
864 comets loaded
```

Karena berkas komet tidak memiliki tanggal kedaluwarsa yang jelas, load.open()berkas hanya akan diunduh satu kali. Panggilan berikutnya akan membuka kembali salinan berkas yang sudah ada di sistem berkas Anda. Untuk memaksa unduhan baru dan menerima orbit dan komet baru yang diperbarui, berikan reload=True.

Untuk membuat posisi komet, pertama-tama pilih barisnya dari kerangka data. Ada beberapa teknik Pandas untuk memilih baris, tetapi sebagian besar pengguna Skyfield akan mengindeks kerangka data mereka berdasarkan penunjukan komet.

```
# Keep only the most recent orbit for each comet,
# and index by designation for fast lookup.
comets = (comets.sort_values('reference')
          .groupby('designation', as_index=False).last()
          .set_index('designation', drop=False))
```

```
# Sample lookups.

row = comets.loc['1P/Halley']

row = comets.loc['C/1995 O1 (Hale-Bopp)']
```

Saat menghitung posisi komet dari Bumi, ada komplikasi: orbit komet tidak diukur dari barisentrum Tata Surya, tetapi berpusat pada Matahari. Oleh karena itu, Anda perlu menambahkan vektor *barisentrum→Matahari* ke vektor *Matahari→komet* untuk menghasilkan posisi yang dapat Anda teruskan ke observe()metode, yang selalu mengukur posisi dari barisentrum Tata Surya.

```
# Generating a position.


from skyfield.constants import GM_SUN_Pitjeva_2005_km3_s2 as GM_SUN


ts = load.timescale()

eph = load('de421.bsp')

sun, earth = eph['sun'], eph['earth']


comet = sun + mpc.comet_orbit(row, ts, GM_SUN)


t = ts.utc(2020, 5, 31)

ra, dec, distance = earth.at(t).observe(comet).radec()

print(ra)

print(dec)

23h 59m 16.85s

-84deg 46' 57.8"
```

Semoga saja Skyfield di masa mendatang mendukung pembuatan posisi untuk seluruh susunan komet dalam satu operasi efisien, tetapi untuk saat ini kode Anda seharusnya hanya beroperasi pada satu komet dalam satu waktu.

Planet Minor

Ada hampir satu juta planet minor dalam basis data unsur orbit Pusat Planet Minor IAU , berkat hasil survei langit otomatis yang luar biasa selama beberapa dekade terakhir.

Basis data dapat diunduh sebagai satu MPCORBberkas — “Orbit Pusat Planet Minor” — yang menawarkan elemen orbital setiap planet minor sebagai teks biasa. Namun, berkas mentah memerlukan sedikit praproses sebelum Skyfield siap memuatnya:

- 43 baris pertama berkas tersebut adalah paragraf yang menjelaskan isinya, menyebutkan ketentuan yang berlaku untuk program perangkat lunak yang dapat menyertakan data, dan menyediakan tautan ke dokumentasi. Skyfield akan mengabaikan atau menghapus baris-baris nondata ini.
- Meskipun versi file yang tidak dikompresi tersedia untuk diunduh, sebagian besar pengguna memilih untuk mengunduh versi terkompresi 55 MB dari Minor Planet Center untuk menghemat bandwidth dan penyimpanan. Mendekompresi seluruh 190 MB data yang tersimpan di dalamnya dapat memerlukan waktu komputasi lebih dari 1 detik, tergantung pada platform dan kecepatan pemrosesan Anda.
- Katalog lengkap mencantumkan hampir 1 juta objek, yang memerlukan waktu beberapa detik untuk dimuat dan diindeks.

Karena semua alasan ini, biasanya paling masuk akal untuk mengunduh, mengekstrak, dan memfilter berkas sebelum memulai aplikasi Anda.

Jika sistem operasi Anda menyediakan alat untuk pencocokan pola, alat tersebut mungkin merupakan alat tercepat untuk memilih orbit tertentu. Berikut cara mengekstrak orbit untuk empat asteroid pertama yang ditemukan — (1) Ceres, (2) Pallas, (3) Juno, dan (4) Vesta — pada sistem Linux:

```
zgrep -P '^(00001|00002|00003|00004) ' MPCORB.DAT.gz > MPCORB.excerpt.DAT
```

Jika sistem operasi Anda tidak memiliki alat tersebut, Anda dapat membuatnya sendiri menggunakan Python. Perhatikan bahwa operasi massal yang diimplementasikan Python dalam C, seperti membaca seluruh isi file dengan read() dan memindai seluruh isi dengan ekspresi reguler findall(), akan jauh lebih cepat daripada menggunakan loop Python untuk membaca setiap baris. Berikut ini contoh skrip untuk melakukan pencarian yang sama seperti zgrepperintah yang ditampilkan di atas:

```
# mpc_make_excerpt.py


"""Search the MPCORB file for minor planets, given their packed designations."""


import argparse

import re

import sys

import zlib


from skyfield.api import load

from skyfield.data import mpc


def main(argv):
```

```
    parser = argparse.ArgumentParser(description='Grep MPCORB.DAT.gz')
    parser.add_argument('designations', nargs='+', help='packed designations'
                        ' of the minor planets whose orbits you need')
    args = parser.parse_args(argv)

    designations = [re.escape(d.encode('ascii')) for d in args.designations]
    pattern = rb'^((?:%s) .*\n)' % rb'|'.join(designations)
    r = re.compile(pattern, re.M)

    data = load.open(mpc.MPCORB_URL).read()
    data = zlib.decompress(data, wbits = zlib.MAX_WBITS | 16)
    lines = r.findall(data)

    sys.stdout.buffer.write(b''.join(lines))

if __name__ == '__main__':
    main(sys.argv[1:])
```

Keempat orbit asteroid yang sama kemudian dapat diekstraksi dengan:

```
python mpc_make_excerpt.py 00001 00002 00003 00004 > MPCORB.excerpt.DAT
```

Perhatikan bahwa berkas minor planets tidak memiliki tanggal kedaluwarsa yang jelas, jadi load.open()dalam skrip di atas, berkas hanya akan diunduh satu kali. Panggilan berikutnya akan membuka kembali salinan berkas yang sudah ada di sistem berkas Anda. Untuk memaksa unduhan baru, berikan reload=True.

Dalam kedua kasus tersebut, berkas yang dihasilkan — tanpa tajuk teks, dan hanya berisi orbit planet minor — siap dimuat oleh Skyfield.

```
with load.open('MPCORB.excerpt.DAT') as f:
    minor_planets = mpc.load_mpcorb_dataframe(f)

print(minor_planets.shape[0], 'minor planets loaded')
```

```
4 minor planets loaded
```

Beberapa pengguna Skyfield telah menemukan berkas Minor Planet Center dengan benda-benda yang elemen orbitnya tidak lengkap , mungkin karena orbitnya masih dalam tahap penentuan. Untuk menghindari EphemerisRangeErrorpengecualian saat Skyfield mencoba menghitung posisi benda-benda ini, Anda dapat meminta Pandas untuk memfilternya dari kerangka data Anda:

```
# Filtering the orbits dataframe to avoid triggering

# an `EphemerisRangeError` on ill-defined orbits.


bad_orbits = minor_planets.semimajor_axis_au.isnull()

minor_planets = minor_planets[~bad_orbits]
```

Seperti yang ditunjukkan pada bagian sebelumnya tentang komet, Anda dapat meminta Pandas untuk mengindeks kerangka data berdasarkan penunjukan planet minor untuk pencarian cepat.

```
# Index by designation for fast lookup.

minor_planets = minor_planets.set_index('designation', drop=False)


# Sample lookups.

row = minor_planets.loc['(1) Ceres']

row = minor_planets.loc['(4) Vesta']
```

Akhirnya, pembuatan posisi melibatkan manuver yang sama yang diperlukan untuk komet: karena orbit planet minor berpusat pada Matahari, vektor posisi Matahari harus ditambahkan ke orbit mereka untuk membangun posisi yang lengkap.

```
ceres = sun + mpc.mpcorb_orbit(row, ts, GM_SUN)

ra, dec, distance = earth.at(t).observe(ceres).radec()

print(ra)

print(dec)

05h 51m 45.85s

+22deg 38' 50.2"
```

Satelit Bumi

Skyfield dapat memprediksi posisi satelit Bumi dengan mengunduh elemen orbit standar SGP4 milik satelit. Elemen orbit diterbitkan oleh organisasi seperti CelesTrak . Waspadalah terhadap batasan berikut:

1. Jangan berharap ada kesesuaian sempurna antara dua perangkat lunak yang mencoba memprediksi posisi satelit. Seperti yang ditunjukkan Vallado, Crawford, dan Hujsak dalam makalah penting mereka Revisiting Spacetrack Report #3 , ada banyak versi berbeda dari algoritme prediksi satelit dasar yang beredar di pasaran. Skyfield menggunakan versi algoritme yang telah diperbaiki dan diperbarui yang mereka terbitkan dalam laporan tersebut.

2. Elemen orbit satelit cepat sekali kedaluwarsa. Seperti dijelaskan di bawah ini dalam Memeriksa epoch set elemen , Anda perlu memperhatikan tanggal "epoch" — tanggal saat set elemen tertentu paling akurat. Set elemen mungkin hanya berguna selama beberapa minggu sebelum atau sesudah epoch-nya. Untuk tanggal yang lebih awal, Anda perlu menarik elemen lama dari arsip; untuk tanggal yang lebih baru, Anda perlu mengunduh set elemen yang baru.

3. Orbit satelit tidak memiliki akurasi yang sempurna. Menurut Laporan Revisiting Spacetrack #3 :

“Keakuratan maksimum untuk TLE dibatasi oleh jumlah tempat desimal di setiap bidang. Secara umum, data TLE akurat hingga sekitar satu kilometer atau lebih pada suatu waktu dan data tersebut cepat menurun.”

4. Mengingat seberapa dekat satelit Bumi dengan Bumi, tidak ada gunanya menggunakan observe()metode Skyfield yang biasa digunakan untuk menghitung waktu tempuh cahaya. Seperti yang ditunjukkan di bawah ini, pengurangan vektor sederhana seharusnya berfungsi dengan baik.

5. Akhirnya, perlu dicatat bahwa meskipun beberapa elemen satelit memiliki nama yang familiar seperti *inklinasi* dan *eksentrisitas* , mereka bukanlah elemen Kepler sederhana, dan rutin SGP4 tidak akan memprediksi posisi yang sama seperti orbit Kepler sederhana.

Format TLE dan pesaingnya

Kebanyakan orang mengunduh set elemen satelit dari CelesTrak:

https://celestrak.org/NORAD/elements/index.php

CelesTrak mendukung beberapa format data. Format Two-Line Element 'TLE' asli menggambarkan orbit satelit menggunakan dua baris teks ASCII yang padat. Whitespace penting karena setiap karakter harus disejajarkan tepat di kolom yang tepat. Berikut ini, misalnya, elemen untuk Stasiun Luar Angkasa Internasional (ISS):

```
ISS (ZARYA)
1 25544U 98067A   24127.82853009  .00015698  00000+0  27310-3 0  9995
```

2 25544 51.6393 160.4574 0003580 140.6673 205.7250 15.50957674452123

Namun CelesTrak juga mendukung format modern. Berikut ini adalah elemen yang sama yang diatur dalam format JSON — khususnya, format CelesTrak FORMAT=json-pretty, yang menambahkan indentasi agar mudah dibaca; gunakan FORMAT=jsonjika Anda tidak memerlukan spasi dan baris baru tambahan:

[{

"OBJECT_NAME": "ISS (ZARYA)",

"OBJECT_ID": "1998-067A",

"EPOCH": "2024-05-06T19:53:04.999776",

"MEAN_MOTION": 15.50957674,

"ECCENTRICITY": 0.000358,

"INCLINATION": 51.6393,

"RA_OF_ASC_NODE": 160.4574,

"ARG_OF_PERICENTER": 140.6673,

"MEAN_ANOMALY": 205.725,

"EPHEMERIS_TYPE": 0,

"CLASSIFICATION_TYPE": "U",

"NORAD_CAT_ID": 25544,

"ELEMENT_SET_NO": 999,

"REV_AT_EPOCH": 45212,

"BSTAR": 0.0002731,

"MEAN_MOTION_DOT": 0.00015698,

"MEAN_MOTION_DDOT": 0

}]

And here are the same elements with FORMAT=CSV (this is only two lines of text, but your browser will probably wrap them to fit your screen):

OBJECT_NAME,OBJECT_ID,EPOCH,MEAN_MOTION,ECCENTRICITY,INCLINATION,RA_OF_ASC_NODE,ARG_OF_PERICENTER,MEAN_ANOMALY,EPHEMERIS_TYPE,CLASSIFICATION_TYPE,NORAD_CAT_ID,ELEMENT_SET_NO,REV_AT_EPOCH,BSTAR,MEAN_MOTION_DOT,MEAN_MOTION_DDOT

ISS (ZARYA),1998-067A,2024-05-06T19:53:04.999776,15.50957674,.000358,51.6393,160.4574,140.6673,205.7250,0,U,25544,999,45212,.2731E-3,.15698E-3,0

Note these differences:

- Even though today the JSON and CSV formats carry exactly the same data as the TLE — for example, all three of the examples above give the inclination as 51.6393°, with four digits past the decimal point — both JSON and CSV will be able to carry greater precision in the future. The old TLE format, by contrast, will never be able to add more decimal places because each element has a fixed-width number of characters.
- The TLE format has run out of simple integer catalog numbers, because it limits the catalog number (in the above example, 25544) to 5 digits. The JSON and CSV formats, by contrast, don't place a limit on the size of the catalog number.
- The JSON and CSV formats are self-documenting. You can tell, even as a first-time reader, which element is the inclination. The old TLE lines provide no hint about which element is which.
- The TLE and CSV formats are both very efficient, and describe an orbit using about 150 characters. Yes, the CSV file has that big 225-character header at the top; but the header line only appears once, no matter how many satellites are listed after it. By contrast, the bulky JSON format requires more than 400 characters per satellite because the element names need to be repeated again every time.

There are more obscure formats in use at CelesTrak, including a 'key-value notation (KVN)' and an unfortunate XML format, but here we will focus on the mainstream formats listed above.

Downloading satellite elements

Whether you choose one of CelesTrak's pre-packaged satellite lists like 'Space Stations' or 'CubeSats', or perform a query for a particular satellite, there are three issues to beware of:

- Be sure to save the data to a file the first time your script runs, so that subsequent runs won't need to download the same data again. This is crucial to reducing the load on the CelesTrak servers and helping them continue to provide CelesTrak as a free service.
- Satellite elements gradually go out of date. Once a file is a few days old, you will probably want to download the file again.
- The CelesTrak data URLs all use the exact same filename. So if you download the 'Space Stations' whose URL looks like gp.php?GROUP=stations, and then the 'CubeSats' with gp.php?GROUP=cubesat, then Skyfield will save them to the same filename gp.php and the CubeSats will wind up overwriting the Space Stations. To avoid this, use the filename= optional argument to create a separate local file for each remote URL.

Here's a useful pattern for downloading element sets with Skyfield:

```
from skyfield.api import load


max_days = 7.0        # download again once 7 days old
name = 'stations.csv'  # custom filename, not 'gp.php'
```

```
    base = 'https://celestrak.org/NORAD/elements/gp.php'
    url = base + '?GROUP=stations&FORMAT=csv'

if not load.exists(name) or load.days_old(name) >= max_days:
        load.download(url, filename=name)
```

The next section will illustrate how to load satellites once you have downloaded the file.

If your project is serious enough that you will need to be able to double-check and replicate old results later, then don't follow this example — every time the file gets too old, this code will overwrite the file with new data. Instead, you will probably want to put the date in the filename, and archive each file along with your project's code.

Loading satellite elements

Once you have downloaded a file of elements, use one of these patterns to load them into Skyfield. For the traditional TLE format:

```
    from skyfield.api import load
    from skyfield.iokit import parse_tle_file

    ts = load.timescale()

    with load.open('stations.tle') as f:
      satellites = list(parse_tle_file(f, ts))

    print('Loaded', len(satellites), 'satellites')
```

Loaded 27 satellites

For the verbose but easy-to-read JSON format:

```
    import json
    from skyfield.api import EarthSatellite, load

    with load.open('stations.json') as f:
      data = json.load(f)

    ts = load.timescale()
```

```
sats = [EarthSatellite.from_omm(ts, fields) for fields in data]
print('Loaded', len(sats), 'satellites')
```

```
Loaded 27 satellites
```

For the more compact CSV format:

```
import csv
from skyfield.api import EarthSatellite, load

with load.open('stations.csv', mode='r') as f:
    data = list(csv.DictReader(f))

ts = load.timescale()
sats = [EarthSatellite.from_omm(ts, fields) for fields in data]
print('Loaded', len(sats), 'satellites')
```

```
Loaded 27 satellites
```

In each case, you are asked to provide a ts timescale. Why? Because Skyfield needs the timescale's knowledge of leap seconds to turn each satellite's epoch date into an .epoch Time object.

Loading satellite data from a string

If your program has already loaded TLE, JSON, or CSV data into memory and doesn't need to read it over again from a file, you can make the string behave like a file by wrapping it in a Python I/O object:

```
# For TLE and JSON:

from io import BytesIO
f = BytesIO(byte_string)

# For CSV:

from io import StringIO
f = StringIO(text_string)
```

You can then use the resulting file object f with the example code in the previous section.

Indexing satellites by name or number

If you want to operate on every satellite in the list that you have loaded from a file, you can use Python's for loop. But if you instead want to select individual satellites by name or number, try building a lookup dictionary using Python's dictionary comprehension syntax:

```
by_name = {sat.name: sat for sat in satellites}

satellite = by_name['ISS (ZARYA)']

print(satellite)

ISS (ZARYA) catalog #25544 epoch 2024-05-09 08:48:20 UTC

by_number = {sat.model.satnum: sat for sat in satellites}

satellite = by_number[25544]

print(satellite)

ISS (ZARYA) catalog #25544 epoch 2024-05-09 08:48:20 UTC
```

Loading a single TLE set from strings

If your program already has the two lines of TLE data for a satellite and doesn't need Skyfield to download and parse a CelesTrak file, you can instantiate an EarthSatellite directly.

```
from skyfield.api import EarthSatellite


ts = load.timescale()

line1 = '1 25544U 98067A   14020.93268519  .00009878  00000-0  18200-3 0  5082'

line2 = '2 25544  51.6498 109.4756 0003572  55.9686 274.8005 15.49815350868473'

satellite = EarthSatellite(line1, line2, 'ISS (ZARYA)', ts)

print(satellite)

ISS (ZARYA) catalog #25544 epoch 2014-01-20 22:23:04 UTC
```

Checking an element set's epoch

The .epoch time of a satellite element set is the date and time on which the element set is most accurate. Before or after that date, the element set will be less accurate. The epoch is a Skyfield Time object:

```
print(satellite.epoch.utc_jpl())

A.D. 2014-Jan-20 22:23:04.0004 UTC
```

If the epoch is too far in the past, you can provide tle_file() with the reload option to force it to download new elements even if the file is already on disk. (Note, though, that there is no guarantee that the new elements will be up-to-date if the source file is not frequently updated for the satellite you are interested in — so this pattern might make you download a new file on each run until the satellite's elements are finally updated.)

```
t = ts.utc(2014, 1, 23, 11, 18, 7)


days = t - satellite.epoch
print('{:.3f} days away from epoch'.format(days))


if abs(days) > 14:
    satellites = load.tle_file(stations_url, reload=True)
2.538 days away from epoch
```

You can read T.S. Kelso on Twitter to follow along with the drama as various satellite element sets go out-of-date each month and await updates from their respective organizations.

Historical satellite element sets

To repeat the warning in the previous section: any particular satellite TLE set is only valid for a couple of weeks to either side of that TLE's epoch.

That limitation unfortunately applies to the past as well as to the future. Just as today's TLE for a satellite can only help you predict its position for a few weeks into the future, it will also be accurate for only a few weeks into the past. Whether the satellite has been performing active maneuvers, or merely coasting amidst the unpredictable magnetic fields and atmospheric drag of the near-Earth environment, a current TLE cannot predict the satellite's position on a date in the distant past.

If you lack access to an archive of old TLE files, try searching the Internet Archive's "Wayback Machine":

https://archive.org/web/

Supply the URL of the current satellite catalog you downloaded and click "Browse History" and the Archive will display a calendar indicating whether any earlier versions of that same satellite catalog are in their archive. If so, then you should be able to download them to your machine and use them when you need historic satellite positions close to the old TLE's epoch date.

Finding when a satellite rises and sets

Skyfield can search between a start time and an end time for each occasion on which a satellite's altitude exceeds a specified number of degrees above the horizon. For example, here is how to determine how many times our example satellite rises above 30° of altitude over the span of a single day:

```
from skyfield.api import wgs84


bluffton = wgs84.latlon(+40.8939, -83.8917)
t0 = ts.utc(2014, 1, 23)
```

```
t1 = ts.utc(2014, 1, 24)
t, events = satellite.find_events(bluffton, t0, t1, altitude_degrees=30.0)
event_names = 'rise above 30°', 'culminate', 'set below 30°'
for ti, event in zip(t, events):
    name = event_names[event]
    print(ti.utc_strftime('%Y %b %d %H:%M:%S'), name)
```

```
2014 Jan 23 06:25:37 rise above 30°
2014 Jan 23 06:26:58 culminate
2014 Jan 23 06:28:19 set below 30°
2014 Jan 23 12:54:56 rise above 30°
2014 Jan 23 12:56:27 culminate
2014 Jan 23 12:57:58 set below 30°
```

The satellite's altitude exceeded 30° twice. For each such occasion, the method find_events() has determined not only the moment of greatest altitude — accurate to within a second or so — but also the time at which the satellite first crested 30° and the moment at which it dipped below it.

Beware that events might not always be in the order rise-culminate-set. Some satellites culminate several times between rising and setting.

By combining these results with the result of the is_sunlit() method (as described below in Find when a satellite is in sunlight), you can determine whether the satellite is in sunlight during these passes, or is eclipsed within the Earth's shadow.

```
eph = load('de421.bsp')
sunlit = satellite.at(t).is_sunlit(eph)


for ti, event, sunlit_flag in zip(t, events, sunlit):
    name = event_names[event]
    state = ('in shadow', 'in sunlight')[sunlit_flag]
    print('{:22} {:15} {}'.format(
        ti.utc_strftime('%Y %b %d %H:%M:%S'), name, state,
    ))
```

```
2014 Jan 23 06:25:37   rise above 30°  in shadow
2014 Jan 23 06:26:58   culminate       in shadow
2014 Jan 23 06:28:19   set below 30°   in shadow
```

2014 Jan 23 12:54:56 rise above 30° in sunlight

2014 Jan 23 12:56:27 culminate in sunlight

2014 Jan 23 12:57:58 set below 30° in sunlight

Finally, you will probably want to check the altitude of the Sun, so that you can ignore passes that happen during the daytime — unless you have some means of observing the satellite (by radio, for example) before it gets dark outside.

Generating a satellite position

Once Skyfield has identified the times at which a particular satellite is overhead, you will probably want to learn more about its position at those times.

The simplest form in which you can generate a satellite position is to call its at() method, which will return an (*x,y,z*) position relative to the Earth's center in the Geocentric Celestial Reference System. (GCRS coordinates are based on even more precise axes than those of the old J2000 system.)

```
# You can instead use ts.now() for the current time

t = ts.utc(2014, 1, 23, 11, 18, 7)

geocentric = satellite.at(t)

print(geocentric.xyz.km)
```

```
[-3918.87650458 -1887.64838745  5209.08801512]
```

Satellite latitude, longitude, and height

Once you have computed a geocentric satellite position, you can use either of several wgs84 object methods to learn the satellite's latitude, longitude, and height:

- latlon_of()
- height_of()
- geographic_position_of()

For example:

```
lat, lon = wgs84.latlon_of(geocentric)

print('Latitude:', lat)

print('Longitude:', lon)
```

```
Latitude: 50deg 14' 37.4"

Longitude: -86deg 23' 23.3"
```

Another wgs84 method computes the subpoint directly below the satellite — the point on the Earth with the same latitude and longitude as the satellite, but with a height above the WGS84 ellipsoid of zero:

- subpoint_of()

If you want the actual position of the ground beneath the satellite, you of course can't assume that the position will be exactly at sea level. You'll need to find a geographic library that lets you load a digital elevation model (DEM), then build a subpoint manually using the elevation returned for the satellite's latitude and longitude.

```
elevation_m = 123.0

subpoint = wgs84.latlon(lat.degrees, lon.degrees, elevation_m)
```

Satellite altitude, azimuth, and distance

You might be most interested in whether the satellite is above or below the horizon from your own position as an observer, and in which direction to look for it. If you build an object to represent your latitude and longitude (as we did when we created the bluffton object above), you can use vector subtraction to ask "where will the satellite be *relative to* my location?"

```
difference = satellite - bluffton
```

Every time you call this vector sum's at() method, it will compute the satellite's position, then your own position, then subtract them. The result will be the position of the satellite relative to you as an observer. If you are interested you can access this relative position as plain (*x,y,z*) coordinates:

```
topocentric = difference.at(t)

print(topocentric.xyz.km)

[ 331.61901192  392.18492744 1049.7597825 ]
```

But the most popular approach is to ask the topocentric position for its altitude and azimuth. The altitude angle runs from 0° at the horizon to 90° directly overhead at the zenith. A negative altitude means the satellite is that many degrees below the horizon.

```
alt, az, distance = topocentric.altaz()


if alt.degrees > 0:
    print('The ISS is above the horizon')


print('Altitude:', alt)

print('Azimuth:', az)

print('Distance: {:.1f} km'.format(distance.km))

The ISS is above the horizon

Altitude: 16deg 16' 32.6"

Azimuth: 350deg 15' 20.4"

Distance: 1168.7 km
```

The azimuth is measured clockwise around the horizon, just like the degrees shown on a compass, from geographic north (0°) through east (90°), south (180°), and west (270°) before returning to the north and rolling over from 359° back to 0°.

Satellite right ascension and declination

If you are interested in where among the stars the satellite will be positioned, then — as with any other Skyfield position object — you can ask for its right ascension and declination, either relative to the fixed axes of the ICRF or else in the dynamical coordinate system of the date you specify.

```
ra, dec, distance = topocentric.radec()  # ICRF ("J2000")


print(ra)

print(dec)

03h 19m 07.97s

+63deg 55' 47.2"

ra, dec, distance = topocentric.radec(epoch='date')


print(ra)

print(dec)

03h 20m 22.42s

+63deg 58' 45.2"
```

See Positions to learn more about these possibilities.

Find a satellite's range rate

If you're interested in the Doppler shift of the radio signal from a satellite, you'll want to know the rate at which the satellite's range to your antenna is changing. To determine the rate, use the position method frame_latlon_and_rates() whose third return value will be the range and whose sixth return value will be the range's rate of change.

Our example satellite culminates at around 20° above the horizon just after 11:20pm UTC. As expected, its range reaches a minimum during that minute and its range rate swaps from negative (drawing closer) to positive (moving away).

```
t = ts.utc(2014, 1, 23, 11, range(17, 23))

pos = (satellite - bluffton).at(t)

_, _, the_range, _, _, range_rate = pos.frame_latlon_and_rates(bluffton)


from numpy import array2string
```

```
print(array2string(the_range.km, precision=1), 'km')

print(array2string(range_rate.km_per_s, precision=2), 'km/s')

[1434.2 1190.5 1064.3 1097.3 1277.4 1553.6] km

[-4.74 -3.24 -0.84  1.9   3.95  5.14] km/s
```

I've chosen here to ask for coordinates in the observer's alt-az frame of reference, but in fact the choice of coordinate system doesn't matter if we're going to ignore everything but the range and range rate: those two quantities should be independent of the orientation of the spherical coordinate system we choose.

Find when a satellite is in sunlight

A satellite is generally only visible to a ground observer when there is still sunlight up at its altitude. The satellite will visually disappear when it enters the Earth's shadow and reappear when it emerges back into the sunlight. If you are planning to observe a satellite visually, rather than with radar or radio, you will want to know which satellite passes are in sunlight. Knowing a satellite's sunlit periods is also helpful when modeling satellite power and thermal cycles as it goes in and out of eclipse.

Skyfield provides a simple geometric estimate for this through the is_sunlit() method. Given an ephemeris with which it can compute the Sun's position, it will return True when the satellite is in sunlight and False otherwise.

```
eph = load('de421.bsp')


two_hours = ts.utc(2014, 1, 20, 0, range(0, 120, 20))

sunlit = satellite.at(two_hours).is_sunlit(eph)

print(sunlit)

[ True  True False False  True  True]
```

As usual, you can use Python's zip() builtin if you want to loop across the times and corresponding values.

```
for ti, sunlit_i in zip(two_hours, sunlit):
    print('{}  {} is in {}'.format(
        ti.utc_strftime('%Y-%m-%d %H:%M'),
        satellite.name,
        'sunlight' if sunlit_i else 'shadow',
    ))

2014-01-20 00:00  ISS (ZARYA) is in sunlight

2014-01-20 00:20  ISS (ZARYA) is in sunlight

2014-01-20 00:40  ISS (ZARYA) is in shadow
```

2014-01-20 01:00 ISS (ZARYA) is in shadow

2014-01-20 01:20 ISS (ZARYA) is in sunlight

2014-01-20 01:40 ISS (ZARYA) is in sunlight

Find whether the Earth blocks a satellite's view

The Earth looms large in the sky of an Earth-orbiting satellite. To plan an observation you may want to know when a given celestial object is blocked by the Earth and not visible from your satellite. Skyfield provides a simple geometric estimate for this through the is_behind_earth() method.

```
eph = load('de421.bsp')
earth, venus = eph['earth'], eph['venus']


two_hours = ts.utc(2014, 1, 20, 0, range(0, 120, 20))
p = (earth + satellite).at(two_hours).observe(venus).apparent()
sunlit = p.is_behind_earth()
print(sunlit)
```

[False False True True False False]

See the previous section for how to associate each of these True and False values with their corresponding time.

Avoid calling the observe method

When computing positions for the Sun, Moon, planets, and stars, Skyfield encourages a far more fussy approach than directly subtracting two vectors. In those cases, the user is encouraged to compute their current location with at() and then call the observe() method on the result so that Skyfield can correctly adjust the object's position for the time it takes light to travel.

1. This turns out to be expensive for Earth satellites, however, because the routines with which Skyfield computes satellite positions are not currently very fast.
2. And it turns out to be useless, because satellites are too close and move far too slowly (at least compared to something like a planet) for the light travel time to make any difference.

How far off will your observations be if you simply subtract your position vector from the satellite's vector, as encouraged above? Let's try the alternative and measure the difference.

To use the observe() method, you need a position measured all the way from the Solar System Barycenter (SSB). To anchor both our observer location and that of the satellite to the SSB, we can use vector addition with an ephemeris that predicts the Solar System position of the Earth:

```
# OVERLY EXPENSIVE APPROACH - Compute both the satellite
```

```
# and observer positions relative to the Solar System
# barycenter ("ssb"), then call observe() to compensate
# for light-travel time.

t = ts.utc(2014, 1, 23, 11, 18, 7)
de421 = load('de421.bsp')
earth = de421['earth']
ssb_bluffton = earth + bluffton
ssb_satellite = earth + satellite
topocentric2 = ssb_bluffton.at(t).observe(ssb_satellite).apparent()
```

What difference has all of that work made? We can subtract the resulting positions to find out the distance between them:

```
# After all that work, how big is the difference, really?

difference_km = (topocentric2 - topocentric).distance().km
print('Difference between the two positions:')
print('{0:.3f} km'.format(difference_km))

difference_angle = topocentric2.separation_from(topocentric)
print('Angle between the two positions in the sky:')
print('{}'.format(difference_angle))
Difference between the two positions:
0.087 km
Angle between the two positions in the sky:
00deg 00' 04.6"
```

And there you have it!

While satellite positions are only accurate to about a kilometer anyway, accounting for light travel time only affected the position in this case by less than an additional tenth of a kilometer. This difference is not meaningful when compared to the uncertainty that is inherent in satellite positions to begin with, so you should neglect it and simply subtract GCRS-centered vectors instead as detailed above.

Detecting Propagation Errors

After building a satellite object, you can examine the *epoch* date and time when the TLE element set's predictions are most accurate. The epoch attribute is a Time, so it supports all of the standard Skyfield date methods:

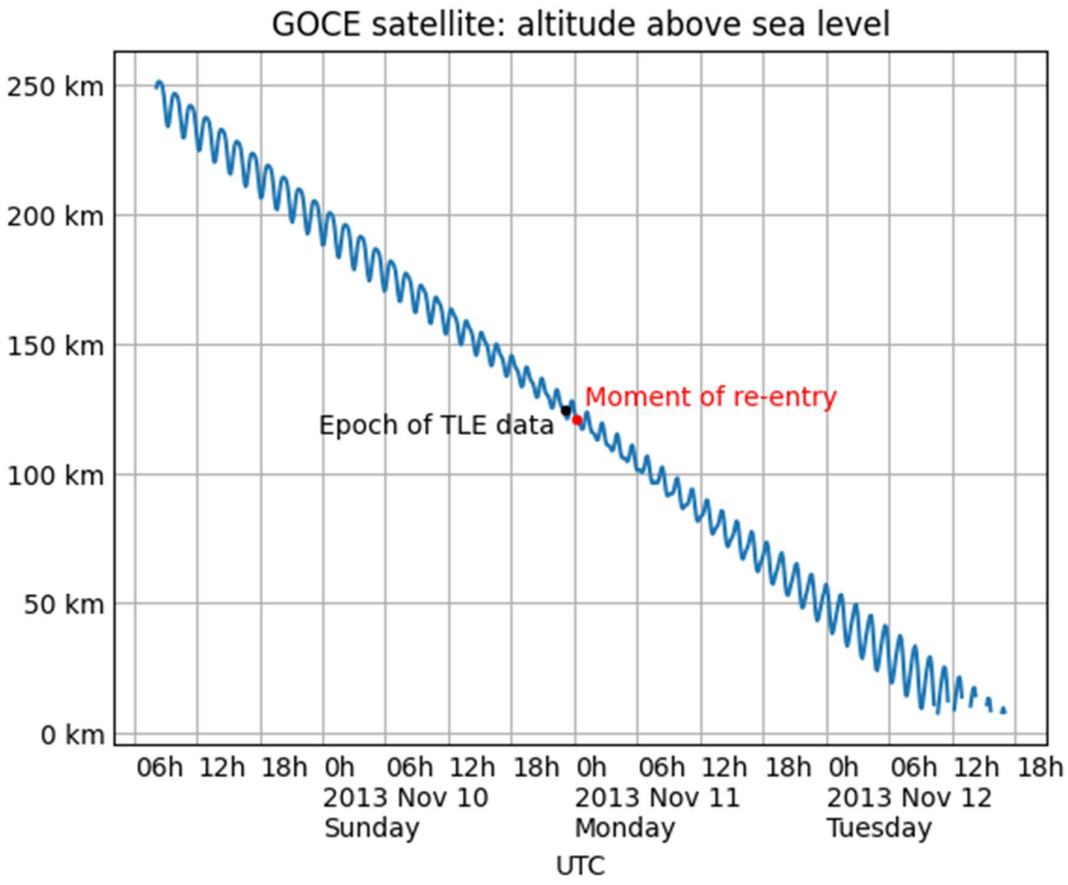
```
from skyfield.api import EarthSatellite


text = """
GOCE
1 34602U 09013A   13314.96046236  .14220718  20669-5  50412-4 0   930
2 34602 096.5717 344.5256 0009826 296.2811 064.0942 16.58673376272979
"""
lines = text.strip().splitlines()


sat = EarthSatellite(lines[1], lines[2], lines[0])
print(sat.epoch.utc_jpl())
```

```
A.D. 2013-Nov-10 23:03:03.9479 UTC
```

Skyfield is willing to generate positions for dates quite far from a satellite's epoch, even if they are not likely to be meaningful. But it cannot generate a position beyond the point where the elements stop making physical sense. At that point, the satellite will return a position and velocity (nan, nan, nan) where all of the quantities are the special floating-point value nan which means *not-a-number*.

When a propagation error occurs and you get nan values, you can examine the message attribute of the returned position to learn the error that the SGP4 propagator encountered.

We can take as an example the satellite elements above. They are the last elements ever issued for GOCE, just before the satellite re-entered the atmosphere after an extended and successful mission. Because of the steep decay of its orbit, the elements are valid over an unusually short period — from just before noon on Saturday to just past noon on Tuesday:



By asking for GOCE's position just before or after this window, we can learn about the propagation errors that are limiting this TLE set's predictions:

```
geocentric = sat.at(ts.utc(2013, 11, 9))
print('Before:')
print(geocentric.xyz.km)
print(geocentric.message)


geocentric = sat.at(ts.utc(2013, 11, 13))
print('\nAfter:')
print(geocentric.xyz.km)
print(geocentric.message)
```

```
Before:
[nan nan nan]
mean eccentricity is outside the range 0.0 to 1.0
```

After:

[-5021.82658191   742.71506112 3831.57403957]

mrt is less than 1.0 which indicates the satellite has decayed

If you use a Time array to ask about an entire range of dates, then message will be a sequence filled in with None whenever the SGP4 propagator was successful and otherwise recording the propagator error:

```
from pprint import pprint


geocentric = sat.at(ts.utc(2013, 11, [9, 10, 11, 12, 13]))

pprint(geocentric.message)

['mean eccentricity is outside the range 0.0 to 1.0',

 None,

 None,

 None,

 'mrt is less than 1.0 which indicates the satellite has decayed']
```

Build a satellite with a specific gravity model

If your satellite elements are designed for another gravity model besides the default WGS72 model, then use the underlying sgp4 module to build the satellite. It will let you customize the choice of gravity model:

```
from sgp4.api import Satrec, WGS84

satrec = Satrec.twoline2rv(line1, line2, WGS84)

sat = EarthSatellite.from_satrec(satrec, ts)
```

Build a satellite from orbital elements

Jika Anda memulai dengan parameter orbit satelit mentah dan bukan teks TLE, Anda akan ingin berinteraksi langsung dengan pustaka sgp4 yang digunakan Skyfield untuk perhitungan satelit tingkat rendah.

Pustaka yang mendasarinya menyediakan akses ke konstruktor tingkat rendah yang membangun model satelit langsung dari parameter orbit numerik:

```
from sgp4.api import Satrec, WGS72


satrec = Satrec()

satrec.sgp4init(

   WGS72,          # gravity model
```

```
    'i',           # 'a' = old AFSPC mode, 'i' = improved mode
    5,             # satnum: Satellite number
    18441.785,     # epoch: days since 1949 December 31 00:00 UT
    2.8098e-05,    # bstar: drag coefficient (/earth radii)
    6.969196665e-13, # ndot: ballistic coefficient (radians/minute^2)
    0.0,           # nddot: second derivative of mean motion (radians/minute^3)
    0.1859667,     # ecco: eccentricity
    5.7904160274885, # argpo: argument of perigee (radians)
    0.5980929187319, # inclo: inclination (radians)
    0.3373093125574, # mo: mean anomaly (radians)
    0.0472294454407, # no_kozai: mean motion (radians/minute)
    6.0863854713832, # nodeo: right ascension of ascending node (radians)
)
```

Jika Anda memerlukan detail lebih lanjut, sgp4initmetode ini didokumentasikan di bagian Menyediakan elemen Anda sendiri pada dokumentasi pustaka sgp4 pada Indeks Pengemasan Python.

Untuk membungkus model satelit tingkat rendah ini dalam objek Skyfield, panggil konstruktor khusus ini:

```
sat = EarthSatellite.from_satrec(satrec, ts)
print('Satellite number:', sat.model.satnum)
print('Epoch:', sat.epoch.utc_jpl())
```

```
Satellite number: 5
Epoch: A.D. 2000-Jun-27 18:50:24.0000 UTC
```

Hasilnya seharusnya berupa objek satelit yang berperilaku persis seperti yang dimuat dari garis TLE.

Mencari tanggal kejadian astronomi

Saat Anda merencanakan pengamatan atau misi Anda sendiri, Anda mungkin tertarik untuk mencari tanggal dan waktu keadaan yang belum ada solusi siap pakai yang tercantum di halaman Perhitungan Almanak Skyfield .

Ada dua jenis penelusuran yang dapat Anda lakukan. Salah satunya adalah penelusuran "peristiwa" yang mencari momen posisi atau penyelarasan yang sesuai dengan definisi yang Anda cari. Yang lainnya adalah penelusuran "ekstrem" yang ingin Anda ketahui saat suatu nilai mencapai nilai maksimum atau minimumnya.

Menemukan kejadian diskrit

Lakukan pencarian kejadian diskret ketika Anda ingin mengetahui tanggal di mana pengukuran berkelanjutan, seperti sudut atau jarak, melebihi nilai tertentu.

Misalnya, Anda mungkin sedang membaca teks astronomi tradisional dan menemukan definisi “kuadratur” — momen ketika dari sudut pandang Bumi, elongasi sebuah planet dari Matahari adalah 90°.

Bagaimana Anda menghitung tanggal kuadratur?

Selalu mulai dengan mencari sendiri kejadiannya. Ini akan memberi Anda gambaran tentang bagaimana fenomena itu berperilaku sebelum Anda mencoba melakukan pencarian otomatis. Dalam kasus kuadratur, kita dapat mulai dengan memilih Mars dan mencoba menghitung elongasinya hari ini. Kamus, ensiklopedia, atau referensi daring akan menjelaskan kepada kita bahwa "elongasi" planet adalah pemisahan sudutnya dari Matahari, jadi mari kita hitung posisi Mars dan Matahari lalu sudut antara posisi tersebut:

```
from skyfield import api


ts = api.load.timescale()

t = ts.utc(2020, 6, 2)


eph = api.load('de421.bsp')

earth, sun, mars = eph['earth'], eph['sun'], eph['mars']

e = earth.at(t)

s = e.observe(sun)

m = e.observe(mars)


print('%.4f' % s.separation_from(m).degrees)

88.5752
```

Sudahkah kita menghitung elongasi dengan benar? Kita harus selalu memeriksa ulang pekerjaan kita terhadap otoritas lain jika memungkinkan. Mengingat tanggal yang sama, situs NASA JPL HORIZONS dapat menghasilkan tabel dengan kolom "SOT", yang (seperti yang dijelaskan dalam definisi di bawah tabel) adalah elongasi matahari:

```
 Date__(UT)__HR:MN   R.A._____(ICRF)_____DEC    S-O-T /r
*******************************************************

 2020-Jun-02 00:00   23 01 20.93 -08 51 51.6  88.5698 /L
```

Wah, sial.

Sistem HORIZONS memberikan angka “88.5698” sebagai elongasi pada tengah malam UTC. Itu bukan angka yang sama yang kami hitung, meskipun mendekati. Bagaimana Skyfield dan HORIZONS bisa menghasilkan angka yang berbeda untuk elongasi? Mungkin kami menghasilkan asensio rektum dan deklinasi yang sedikit berbeda untuk Mars. Mari kita periksa:

```
ra, dec, distance = m.radec()

print(ra, '/', dec)

23h 01m 20.93s / -08deg 51' 51.6"
```

Tidak, bukan itu perbedaannya — angka-angka Skyfield ini sama persis dengan asensio dan deklinasi rektum pada keluaran HORIZONS yang ditunjukkan di atas.

Selalu ada baiknya untuk mempelajari setiap detail keluaran HORIZONS saat menyelidiki perbedaan hasil. Berikut ini, misalnya, paragraf pertama definisi S-O-Tbidang tersebut:

```
 S-O-T /r =

   Sun-Observer-Target angle; target's apparent SOLAR ELONGATION seen from
```

the observer location at print-time. Angular units: DEGREES

So that’s the difference! We computed the angle between the *astrometric* positions of the Sun and Mars, whereas the elongation is more properly an angular difference between *apparent* positions. (The Positions page explains the difference.) Thus:

```
s = e.observe(sun).apparent()

m = e.observe(mars).apparent()


print('%.4f' % s.separation_from(m).degrees)

88.5698
```

Much better! We now have a perfect match with HORIZONS which gives us high confidence that we are computing the elongation correctly.

Next let’s search for a moment of quadrature. I did not deliberately plan the example this way, but it looks like Mars is very close to quadrature as I type this! To determine whether

quadrature was just reached or is a few days in the future, let's compute the value over a few days and see whether it's growing or shrinking:

```
def mars_elongation_degrees(t):
    e = earth.at(t)
    s = e.observe(sun).apparent()
    m = e.observe(mars).apparent()
    return s.separation_from(m).degrees


t = ts.utc(2020, 6, range(2 - 3, 2 + 3))


for ti, ei in zip(t, mars_elongation_degrees(t)):
    print('%s %.4f' % (ti.utc_strftime('%b %d'), ei))
```

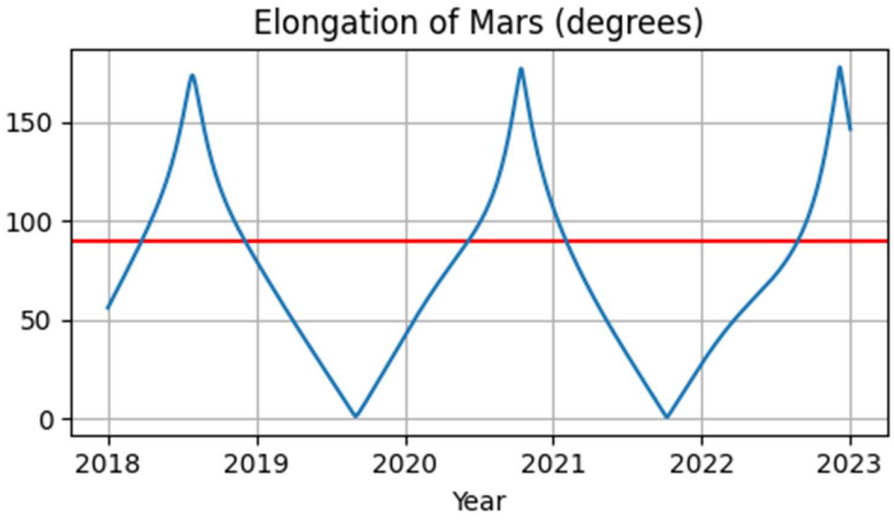
```
May 30 87.6881
May 31 87.9810
Jun 01 88.2749
Jun 02 88.5698
Jun 03 88.8657
Jun 04 89.1626
```

We see that the elongation of Mars is growing slowly right now, at a rate of less than a degree per day, but is very nearly at our target value of 90°. Does is always grow slowly? Does it wane at the same rate? Are there periods during which its change is quick and others during which it is slow?

I always recommend plotting any value on which you are planning to perform a search. It can help develop an intuition around how the value changes through time.

```
from matplotlib import pyplot as plt


fig, ax = plt.subplots(figsize=(5, 3))


t = ts.utc(2018, 1, range(366 * 5))
ax.axhline(90, color='r')  # Red line at 90°
ax.plot(t.J, mars_elongation_degrees(t))
ax.set(title='Elongation of Mars (degrees)', xlabel='Year')
```

```
    ax.grid(True)


    fig.tight_layout()
    fig.savefig('mars-elongation.png')
```



The dates of quadrature are where the elongation intersects the red 90° line that we have drawn across the figure. Mars seems to spend most of its time with an elongation of less than 90° — over on the same side of the sky as the Sun — and spends only a few months at a greater elongation.

Once we have learned to compute the value we are interested in and have plotted its behavior, there are only three tasks involved in launching a search for the dates on which it occurs:

1. Define a function of time returning an integer that changes each time the circumstance occurs. In a very simple case like this one, you can simply use the values False and True because in Python those are the integers zero and one.
2. Give the function a step_days attribute telling the search routine how far apart to space its test dates when it first searches for where your function switches values.
3. Pass the function to the same find_discrete() routine that you would use for a search with the standard almanac functions.

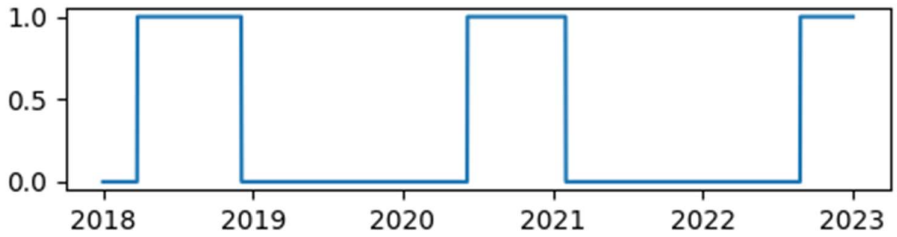
The first task is quite easy in this case. We simply need to compare the elongation with 90°. This transforms the continuous angle measurement into a discrete function that jumps instantly between zero and one.

```
    def mars_quadrature(t):
        e = earth.at(t)
        s = e.observe(sun).apparent()
        m = e.observe(mars).apparent()
```

```
    return s.separation_from(m).degrees >= 90


mars_quadrature.step_days = 90  # Explained below
```

Since the Python values False and True are really the integers 0 and 1, a plot of this function shows a square wave whose positive excursions identify the periods of time during which Mars is more than 90° from the Sun — as we can verify by comparing this plot with our earlier plot.

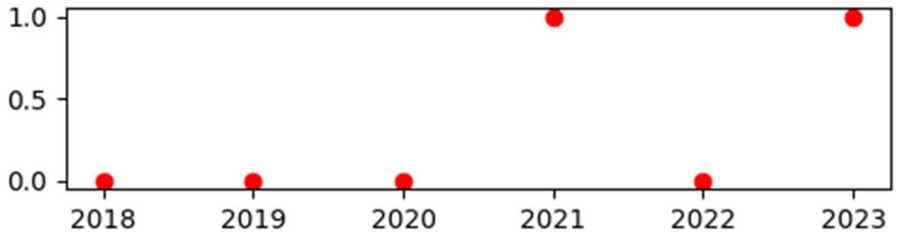
```
fig, ax = plt.subplots(figsize=(5, 1.5))

ax.plot(t.J, mars_quadrature(t))

fig.tight_layout()

fig.savefig('mars-quadrature.png')
```



The second task is to specify the step_days interval over which the search routine should sample our function. If the samples are too far apart, some events could be skipped. But generating too many samples will waste time and memory.

In this example, it is clearly not sufficient to sample our quadrature routine once a year, because the samples would be so far apart that they might skip an entire cycle. Here's our function sampled at the beginning of each calendar year:

```
t_annual = ts.utc(range(2018, 2024))

fig, ax = plt.subplots(figsize=(5, 1.5))

ax.plot(t_annual.J, mars_quadrature(t_annual), 'ro')

fig.tight_layout()

fig.savefig('mars-quadrature-undersampled.png')
```



If you compare this with the previous plot, you will recognize this as our square wave sampled on January 1st of each year.

While a search launched with these data points would find the quadratures of 2021 and 2022, it would entirely miss the Mars opposition of 2018 — because the search routine does not dive in to search between data points that have the same value, as the points for 2018 and 2019 do here. So step_days must always be a smaller time period than the briefest of the events you are trying to detect. (If you have ever studied signal processing, you will recognize that this is the same problem as undersampling an audio signal.)

Mars quadrature events appear to be separated by at least a half-year. For safety let's ask for data points twice as often as that, which gives us the step_days setting that we saw above:

```
mars_quadrature.step_days = 90  # Every ninety days
```

Finally, we are ready to unleash find_discrete():

```
from skyfield.searchlib import find_discrete


t1 = ts.utc(2018)

t2 = ts.utc(2023)

t, values = find_discrete(t1, t2, mars_quadrature)


print(t)

print(values)
```

```
<Time tt=[2458202.1729387594 ... 2459818.7282241164] len=5>

[1 0 1 0 1]
```

The result is a pair of arrays. The first provides the dates and times of quadrature, and the second provides the value that our function switches to on each date. The Python built-in function zip() can iterate across both arrays at once to pair up the dates with the values:

```
for ti, vi in zip(t, values):
    print(ti.utc_strftime('%Y-%m-%d %H:%M '), vi)
```

```
2018-03-24 16:08  1

2018-12-03 00:34  0

2020-06-06 19:11  1

2021-02-01 10:34  0

2022-08-27 05:27  1
```

And we are done! Those are the UTC dates on which Mars reaches western quadrature (when our discrete routine has just changed to 1) and eastern quadrature (when our routine has

changed to 0), as can be confirmed by comparing these dates with those in a standard reference.

Finding extrema

Sometimes you are not interested in when a continuous function of time passes a threshold like 90°, but when it reaches a minimum or maximum value — the two possibilities are collectively called a function's "extrema" — whose exact value you might not be able to predict beforehand.

For example, one challenge of observing Venus is that from Earth's point of view Venus's smaller orbit always keeps it within a few dozen degrees of the Sun. Even when Venus is not so close to the Sun that it's hidden in the Sun's glare, it will be an evening star that's already setting by the time we can see it or a morning star that is soon followed by sunrise.

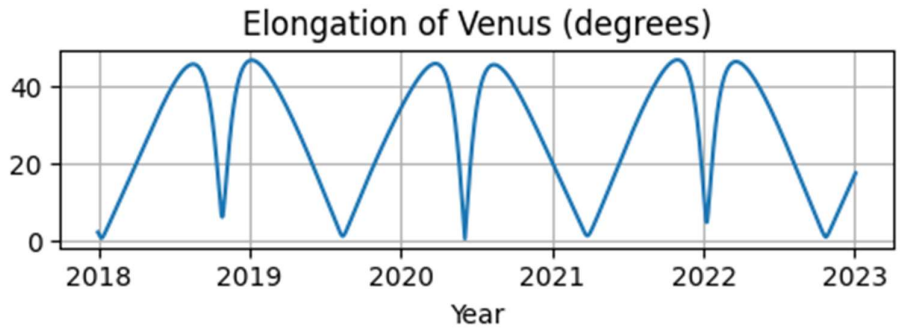
This leads observers to be interested in when Venus is farthest from the Sun — when its elongation is greatest.

The steps are similar to those outlined in the previous section. First, we define a function.

```
venus = eph['venus']


def venus_elongation_degrees(t):
    e = earth.at(t)
    s = e.observe(sun).apparent()
    v = e.observe(venus).apparent()
    return s.separation_from(v).degrees
```

Kemudian kami menghitung perkiraan kasar seberapa sering Venus mencapai elongasi terbesarnya. Pendekatan terbaik adalah membuat plot, yang juga akan memberi kita gambaran tentang perilaku elongasi Venus.

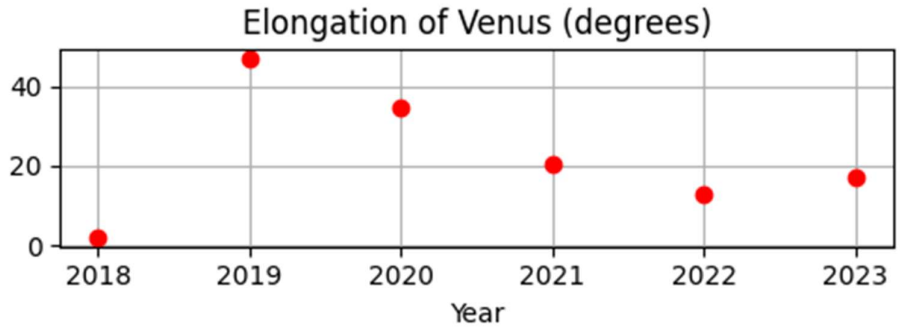
```
fig, ax = plt.subplots(figsize=(5, 2))


t = ts.utc(2018, 1, range(366 * 5))
ax.plot(t.J, venus_elongation_degrees(t))
ax.set(title='Elongation of Venus (degrees)', xlabel='Year')
ax.grid()


fig.tight_layout()
fig.savefig('venus-elongation.png')
```



Anda mungkin terkejut dengan asimetri antara minimum alternatif — antara, katakanlah, minimum bertahap yang lebar yang dicapai pada pertengahan 2019 versus minimum cepat yang tajam yang terjadi berikutnya pada pertengahan 2020. Tetapi jika Anda menyelidiki lebih lanjut dan memetakan Venus dan Bumi dalam orbitnya, alasannya akan menjadi jelas: Venus, pada orbitnya yang lebih cepat, menghabiskan sebagian besar waktunya di sisi lain Matahari secara bertahap mengejar kita, menciptakan minimum yang lebar seperti itu pada pertengahan 2019. Kemudian Venus akhirnya mengejar dan — seperti mobil balap yang melaju kencang di bagian dalam tikungan — melewati dengan sangat cepat antara planet kita dan Matahari, menghasilkan "v" yang lebih tajam dalam plot kita.

Seperti halnya kuadratur Mars, sampel yang jarang — misalnya, setahun sekali — tidak akan memberikan data yang cukup bagi rutinitas pencarian:

```
fig, ax = plt.subplots(figsize=(5, 2))


t = ts.utc(range(2018, 2024))
ax.plot(t.J, venus_elongation_degrees(t), 'ro')
ax.set(title='Elongation of Venus (degrees)', xlabel='Year')
ax.grid()


fig.tight_layout()
fig.savefig('venus-elongation-undersampled.png')
```



Dengan sampel-sampel ini, rutinitas pencarian akan sepenuhnya kehilangan dua titik maksimum tahun 2020 karena sampel-sampel ini kebetulan menangkap fungsi pada dua momen yang membuatnya tampak seolah-olah seluruh tahun 2020 dihabiskan untuk menurun dari titik maksimum pada tahun 2019 menuju titik minimum pada tahun 2022. Rutin pencarian hanya menyelidiki sampel yang lebih tinggi daripada sampel-sampel di kedua sisinya.

Jika Anda bereksperimen dengan sampel yang ditempatkan lebih berdekatan, Anda akan menemukan bahwa bentuk keseluruhan fungsi — termasuk nilai maksimum dan minimumnya — menjadi jelas setelah sampel ditempatkan sekitar satu bulan terpisah. Ukuran langkah ini kemudian dapat digunakan untuk meluncurkan penelusuran:

```
from skyfield.searchlib import find_maxima


venus_elongation_degrees.step_days = 30  # about a month


t1 = ts.utc(2018)

t2 = ts.utc(2023)

t, values = find_maxima(t1, t2, venus_elongation_degrees)


print(len(t), 'maxima found')

6 maxima found
```

Dengan menggunakan zip() bawaan Python, Anda dapat melakukan perulangan di kedua larik, mencetak waktu dan sudut setiap perpanjangan maksimum:

```
for ti, vi in zip(t, values):
   print(ti.utc_strftime('%Y-%m-%d %H:%M '), '%.2f' % vi,
       'degrees elongation')

2018-08-17 17:31  45.93 degrees elongation

2019-01-06 04:54  46.96 degrees elongation

2020-03-24 22:14  46.08 degrees elongation

2020-08-13 00:14  45.79 degrees elongation

2021-10-29 20:52  47.05 degrees elongation

2022-03-20 09:25  46.59 degrees elongation
```

Menemukan nilai minimum

Skyfield menyediakan find_minima()rutinitas yang simetris dengan find_maxima() fungsi yang dijelaskan di bagian sebelumnya. Untuk mengetahui kapan Venus berada paling dekat dengan matahari:

```
from skyfield.searchlib import find_minima
t, values = find_minima(t1, t2, venus_elongation_degrees)

for ti, vi in zip(t, values):
    print(ti.utc_strftime('%Y-%m-%d %H:%M:%S '), '%.2f' % vi,
          'degrees elongation')
2018-01-08 20:15:14  0.76 degrees elongation
2018-10-27 00:48:08  6.22 degrees elongation
2019-08-13 23:03:20  1.27 degrees elongation
2020-06-03 18:48:01  0.48 degrees elongation
2021-03-26 13:47:02  1.35 degrees elongation
2022-01-08 15:16:27  4.81 degrees elongation
2022-10-23 07:32:47  1.05 degrees elongation
```

Akurasi dan Efisiensi Skyfield

Dokumen ini masih dalam tahap pengerjaan, yang akan dikembangkan menjadi panduan lengkap. Saat ini, panduan ini hanya mencakup satu topik.

Presesi dan Nutasi

Seperti yang dijelaskan dalam bagian tentang Asensio Rekta dan Deklinasi Semu , Skyfield menggunakan model presesi-nutasi IAU 2000A untuk menentukan orientasi sumbu Bumi pada tanggal tertentu. Ini digunakan dalam:

- Asensio rektus dan deklinasi nyata sebagaimana dikembalikan oleh radec().
- Posisi geografis yang dihasilkan oleh elipsoid Bumi seperti wgs84model.
- Kerangka true equator and equinox of date acuan yang dapat digunakan untuk menghasilkan koordinat (*x, y, z*) seperti dijelaskan dalam bab Koordinat .
- Kerangka acuan ITRS ditawarkan melalui itrsobjek, yang menggabungkan orientasi kutub dan ekuinoks dengan rotasi harian Bumi.

Jika Anda perlu melakukan perhitungan matematika tingkat rendah sendiri, setiap objek waktu menawarkan matriks t.M yang memutar koordinat dari ICRS ke sistem koordinat ekuator Bumi pada tanggal dan waktu tersebut. Lihat bagian tentang Matriks Rotasi untuk panduan penggunaan matriks rotasi.

Gerakan Kutub

Telah ditemukan lebih dari satu abad yang lalu bahwa kerak Bumi memiliki sedikit kebebasan untuk bergoyang sehubungan dengan sumbu rotasi Bumi di ruang angkasa, karena benua dan dasar samudra terikat pada massa planet hanya melalui kopling fluida mantel kental planet kita. Di Skyfield Anda dapat melihat ukuran efeknya dengan memuat berkas data resmi dari International Earth Rotation Service (IERS) dan mengukur ekskursi maksimum parameter gerakan kutub *x* dan *y* :

```
from skyfield.api import load
from skyfield.data import iers

url = load.build_url('finals2000A.all')
with load.open(url) as f:
    finals_data = iers.parse_x_y_dut1_from_finals_all(f)

ts = load.timescale()
iers.install_polar_motion_table(ts, finals_data)
```

```
tt, x_arcseconds, y_arcseconds = ts.polar_motion_table

print('x:', max(abs(x_arcseconds)), 'arcseconds')

print('y:', max(abs(y_arcseconds)), 'arcseconds')
```

```
x: 0.32548 arcseconds

y: 0.596732 arcseconds
```

Dalam perhitungan Skyfield jenis apa posisi pasti kerak Bumi ikut berperan?

Pada tingkat paling dasar, dua rotasi gerakan kutub diterapkan langsung ke kerangka acuan ITRS di skyfield.framelib.itrs, melengkapi perubahan orientasi Bumi yang sudah diperkirakan Skyfield berkat model presesi dan nutasi IAU 2000A. Hal ini, pada gilirannya, sedikit memengaruhi:

- Posisi pengamat di permukaan bumi.
- Posisi relatif suatu target terhadap pengamat di permukaan bumi.
- Semua koordinat alt-azimuth, karena sudut kutub *x* dan *y* memiringkan titik zenith dan horizon lokal yang menjadi dasar pengukuran ketinggian dan azimuth bagi pengamat tertentu.
- Posisi target observasi apa pun yang berlokasi di permukaan Bumi — misalnya, jika Anda menghitung posisi stasiun darat dari perspektif wahana antariksa.

Agar Skyfield menerapkan gerakan polar saat menghitung posisi dan koordinat, cukup instal tabel IERS pada objek skala waktu seperti yang ditunjukkan pada kode contoh di atas. Gerakan polar akan digunakan di mana pun gerakan tersebut diterapkan.

Tabel detik kabisat

Jika Anda ingin memeriksa ulang bahwa tabel leap second Skyfield sesuai dengan alat atau perangkat lunak lain, Anda dapat dengan mudah mencetaknya. Setiap objek skala waktu menawarkan serangkaian tanggal Julian leap_dates dan serangkaian lain dengan panjang yang sama yang diberi nama leap_offsets yang menawarkan perbedaan antara UTC dan TAI dalam detik:

```
ts = load.timescale()

for jd, offset in zip(ts.leap_dates, ts.leap_offsets):
    ymd = ts.tt_jd(jd).tt_strftime('%Y-%m-%d')
    print(jd, ymd, '{:+}'.format(int(offset)))
```

```
2441499.5 1972-07-01 +11

2441683.5 1973-01-01 +12

2442048.5 1974-01-01 +13

...

2456109.5 2012-07-01 +35
```

2457204.5 2015-07-01 +36

2457754.5 2017-01-01 +37

Perhatikan bahwa setiap detik kabisat terjadi tepat sebelum tanggal Julian yang diberikan dalam tabel. Mengambil baris kedua sebagai contoh, selisih antara TAI dan UTC meningkat menjadi +12 pada saat pertama hari 1973-01-01. Hal ini terjadi karena hari itu langsung didahului oleh detik kabisat yang dikaitkan dengan hari *sebelumnya* sebagai detik terakhirnya. Jadi detik kabisat mengikuti detik normal, bukan kabisat 1972-12-31 12:59:59 dan memiliki sebutan khusus 1972-12-31 12:59:60.

Performa buruk dan menggunakan CPU 100%

Pada beberapa sistem, pengguna melaporkan bahwa Skyfield menghabiskan 100% seluruh CPU dan menyulitkan melakukan pekerjaan lain.

Ini bukan sesuatu yang dapat dikontrol langsung oleh Skyfield. Ini adalah pustaka NumPy yang mendasarinya yang memutuskan cara melakukan setiap operasi matematika yang diminta Skyfield. Dan dalam kasus ini, versi NumPy yang terinstal milik pengguna memutuskan untuk menjalankan operasi vektor secara paralel di semua CPU. (Ironisnya, hal ini membuat operasi menjadi lebih lambat!)

Jika Anda mendapati NumPy berperilaku tidak semestinya dengan cara yang sama di sistem Anda, pengguna melaporkan bahwa mereka dapat memaksakan perilaku single-threaded dengan menetapkan variabel lingkungan ini:

```
export OPENBLAS_NUM_THREADS=1

export MKL_NUM_THREADS=1
```

Jika Anda ingin menerapkan pengaturan ini langsung dalam kode Python Anda, maka Anda harus mengubah pengaturan lingkungan ini *sebelum* mengimpor NumPy:

```
# First set up the environment.


import os

os.environ['OPENBLAS_NUM_THREADS'] = '1'

os.environ['MKL_NUM_THREADS'] = '1'


# And only then import NumPy.

import numpy
```

Solusi yang sama mungkin berhasil pada sistem Anda.

Elemen Orbital yang Berosilasi

Skyfield mampu menghitung elemen orbital yang berosilasi terhadap bidang ekliptika atau bidang ekuator. Data yang dihasilkan oleh Skyfield cocok dengan data yang dihasilkan oleh sistem HORIZONS milik JPL.

Menghasilkan Elemen

Panggilan osculating_elements_of()untuk menghasilkan sebuah OsculatingElementsobjek. Misalnya, berikut ini cara menemukan elemen osilasi bulan yang mengorbit bumi:

```
from skyfield.api import load
from skyfield.elementslib import osculating_elements_of

ts = load.timescale()
t = ts.utc(2018, 4, 22)

planets = load('de421.bsp')
earth = planets['earth']
moon = planets['moon']

position = (moon - earth).at(t)
elements = osculating_elements_of(position)
```

Elemen-elemen tersebut kemudian menjadi atribut dari objek Elemen:

```
i = elements.inclination.degrees
e = elements.eccentricity
a = elements.semi_major_axis.km

print('Inclination: {0:.2f} degrees'.format(i))
print('Eccentricity: {0:.5f}'.format(e))
print('Semimajor axis: {0:.0f} km'.format(a))
```

```
Inclination: 20.46 degrees
Eccentricity: 0.03104
Semimajor axis: 380577 km
```

Perhatikan bahwa satu elemen, waktu periapsis, belum tentu merupakan nilai unik: jika suatu orbit bersifat periodik, maka benda akan mencapai periapsis berulang kali pada serangkaian tanggal yang dipisahkan oleh period_in_days.

```
print('Periapsis:', elements.periapsis_time.utc_strftime())

print('Period: {0:.2f} days'.format(elements.period_in_days))
```

```
Periapsis: 2018-04-20 16:09:42 UTC

Period: 26.88 days
```

Anda dapat menambahkan atau mengurangi periode untuk menghasilkan serangkaian tanggal periapsis yang sama validnya untuk kumpulan elemen orbital tersebut.

```
next = elements.periapsis_time + elements.period_in_days

print('Next periapsis:', next.utc_strftime())
```

```
Next periapsis: 2018-05-17 13:14:56 UTC
```

Atribut objek OsculatingElements

Berikut adalah daftar atribut objek Elemen dan jenisnya:

Objek OscatingElements

| **Elemen yang menggambarkan bentuk orbit:**

├— eksentrisitas → numpy.ndarray

|

| **Elemen yang menggambarkan kemiringan bidang orbit:**

├— kemiringan → Sudut objek

|

| **Elemen yang menggambarkan arah kemiringan bidang orbit:**

├— longitude_of_ascending_node → Objek sudut

|

| **Elemen yang menggambarkan arah periapsis:**

├— argument_of_periapsis → Objek sudut

├— longitude_of_periapsis → Objek sudut

├— periapsis_time → Objek waktu

|

| **Elemen yang menggambarkan ukuran orbit:**

├— apoapsis_distance → Objek jarak

├── mean_motion_per_day → Objek sudut

├── periapsis_distance → Objek jarak

├── period_in_days → numpy.ndarray

├── semi_latus_rectum → Objek jarak

├── semi_major_axis → Objek jarak

├── semi_minor_axis → Objek jarak

|

| **Elemen yang menggambarkan posisi sekunder di orbit:**

├── argument_of_latitude → Objek sudut

├── eccentric_anomaly → Objek sudut

├── mean_anomaly → Objek sudut

├── mean_longitude → Objek sudut

├── true_anomaly → Objek sudut

├── true_longitude → Objek sudut

├── (posisi sekunder dapat tersirat dalam periapsis_time

| karena pada periapsis semua anomali adalah 0)

|

| **Atribut lainnya:**

└── waktu → Objek waktu

Untuk sepenuhnya menentukan lokasi dan orbit suatu objek, diperlukan satu elemen dari setiap kategori di atas.

Bidang Referensi

Secara default, elements()metode ini menghasilkan elemen menggunakan bidang *xy* ICRF sebagai bidang referensi. Ini setara dengan bidang ekuatorial J2000.0 dalam toleransi J2000.0. Jika Anda menginginkan elemen menggunakan ekliptika J2000.0 sebagai bidang referensi, berikan sebagai argumen kedua:

```
from skyfield.data.spice import inertial_frames

ecliptic = inertial_frames['ECLIPJ2000']


t = ts.utc(2018, 4, 22, range(0,25))

position = (moon - earth).at(t)

elements = osculating_elements_of(position, ecliptic)
```

Menggunakan Skyfield dengan AstroPy

Proyek AstroPy merupakan upaya komunitas yang luas untuk menyatukan perangkat lengkap bagi para astronom dan astrofisikawan yang bekerja. Perangkat ini dapat mengonversi antara puluhan unit yang berbeda, membungkus vektor numerik dalam tipe data array dan tabelnya sendiri, menerjemahkan antara sejumlah skala waktu dan kerangka koordinat, menjalankan algoritme pemrosesan gambar pada gambar astronomi, dan mencari katalog langit daring.

Skyfield tidak bergantung pada AstroPy, tetapi dapat menyajikan hasilnya menggunakan satuan AstroPy. Berikut ini adalah titik-titik koneksi yang disediakan Skyfield antara kedua pustaka tersebut:

1. Anda dapat memberikan nilai AstroPy Timedan Skyfield akan menerjemahkannya ke dalam representasinya sendiri.
2. from astropy.time import Time
3. from skyfield.api import load
4.
5. atime = Time('2010-01-01T00:00:00', scale='utc')
6. print(atime)
7.
8. ts = load.timescale()
9. t = ts.from_astropy(atime)
10. print(t.utc_jpl())
11. 2010-01-01T00:00:00.000
12. A.D. 2010-Jan-01 00:00:00.0000 UTC
13. Bidang langit Barycentric, Astrometric, atau Apparent posisi dan kecepatan dapat mengubah dirinya sendiri menjadi besaran AstroPy menggunakan satuan linier dan kecepatan apa pun yang Anda tentukan.
14. import astropy.units as u
15. from skyfield.api import load
16.
17. planets = load('de421.bsp')
18. earth = planets['earth']
19.
20. ts = load.timescale()
21. t = ts.utc(1980, 1, 1)

22. barycentric = earth.at(t)
23.
24. print(barycentric.xyz.to(u.au))
25. print(barycentric.velocity.to(u.au / u.day))
26. [-0.16287311 0.88787399 0.38473904] AU
27. [-0.01721258 -0.00279426 -0.0012121] AU / d
28. Posisi Skyfield juga dapat mengembalikan SkyCoord objek AstroPy lengkap yang menggabungkan vektor posisi dengan kerangka referensinya.
29. from astropy.coordinates import ICRS
30. sc = barycentric.to_skycoord()
31. print(sc)
32. <SkyCoord (ICRS): (x, y, z) in AU
33. (-0.16287311, 0.88787399, 0.38473904)>
34. Sudut Skyfield dapat mengekspresikan dirinya sebagai besaran AstroPy dalam satuan ukuran sudut apa pun yang diminta.
35. ra, dec, distance = barycentric.radec()
36. declination = dec.to(u.deg)
37. print('{0:0.03f}'.format(declination))

23.084 deg

Referensi API

Tautan cepat ke bagian di bawah ini:

Versi

Skyfield menawarkan tuple skyfield.VERSION yang memungkinkan kode Anda menentukan versi Skyfield yang terinstal.

```
import skyfield

print(skyfield.VERSION)
```

Lihat Memeriksa versi Skyfield Anda .

Membuka file

```
# File you already have.


from skyfield.api import load_file
planets = load_file('~/Downloads/de405.bsp')
```

load_file(jalur)	Buka berkas pada drive lokal Anda, gunakan ekstensinya untuk menebak jenisnya.

```
# File you want Skyfield to download automatically.


from skyfield.api import load
ts = load.timescale()
planets = load('de405.bsp')
```

Loader(direktori[, bertele-tele, kadaluarsa])	Alat untuk mengunduh dan membuka berkas data astronomi.
Loader.build_url(nama berkas)	Kembalikan URL yang akan dicoba diunduh Skyfield untuk nama file yang diberikan.
Loader.days_old(nama berkas)	Kembalikan seberapa baru filenamedimodifikasi, diukur dalam hari.
Loader.download(url[, nama file, cadangan])	Unduh berkas, meskipun sudah ada di disk; kembalikan jalurnya.
Loader.path_to(nama berkas)	Kembalikan jalur ke filenamedalam direktori loader ini.
Loader.timescale([delta_t, bawaan])	Kembalikan Timescalebangunan menggunakan data rotasi Bumi resmi.
Loader.tle_file(url[, muat ulang, nama berkas, ts, …])	Memuat dan mengurai berkas TLE, yang menghasilkan daftar satelit Bumi.

Skala waktu

Sebuah skrip biasanya dimulai dengan membangun satu Skyfield Timescale untuk digunakan untuk semua konversi tanggal dan waktu:

```
from skyfield import api
ts = api.load.timescale()
```

Metodenya adalah:

Timescale.now()	Mengembalikan tanggal dan waktu saat ini sebagai Timeobjek.
Timescale.from_datetime(tanggal waktu)	Kembalikan a Timeuntuk Python datetime.
Timescale.from_datetimes(daftar_tanggal_waktu)	Kembalikan a Timeuntuk daftar datetimeobjek Python.
Timescale.utc(tahun[, bulan, tanggal, jam, ...])	Membangun dari tanggal KalenderTime UTC .
Timescale.tai([tahun, bulan, tanggal, jam, ...])	Buat dari tanggal_KalenderTime Waktu Atom Internasional .
Timescale.tai_jd(jd[, pecahan])	Buatlah Timedari tanggal Julian Waktu Atom Internasional.
Timescale.tt([tahun, bulan, tanggal, jam, ...])	Membangun dari tanggal KalenderTime Waktu Terestrial .
Timescale.tt_jd(jd[, pecahan])	Membangun Timedari tanggal Julian Waktu Terestrial.
Timescale.J(tahun)	Membangun Timedari Waktu Terestrial tahun Julian atau array.
Timescale.tdb([tahun, bulan, tanggal, jam, ...])	Membangun dari tanggal KalenderTime Waktu Dinamis Barisentrik .
Timescale.tdb_jd(jd[, pecahan])	Membangun Timedari tanggal Julian Waktu Dinamis Barycentric.
Timescale.ut1([tahun, bulan, tanggal, jam, ...])	Buat dari tanggal_KalenderTime Waktu Universal UT1 .
Timescale.ut1_jd(jd)	Membangun Timedari tanggal Julian Waktu Universal UT1.
Timescale.from_astropy(T)	Bangun Skyfield Timedari objek waktu AstroPy.
Timescale.linspace(t0, t1[, jumlah])	numWaktu pengembalian diberi jarak yang sama antara t0ke t1.

Time objects

The Time class is Skyfield's way of representing either a single time, or a whole array of times. The same time can be represented in several different time scales.

t.tai	International Atomic Time (TAI) as a Julian date.
t.tt	Terrestrial Time (TT) as a Julian date.
t.J	Terrestrial Time (TT) as floating point Julian years.
t.tdb	Barycentric Dynamical Time (TDB) as a Julian date.
t.ut1	Universal Time (UT1) as a Julian date.

A couple of offsets between time scales are also available.

t.delta_t	Difference TT − UT1 in seconds.
t.dut1	Difference UT1 − UTC in seconds.

Other time scales and conversions are available through its methods.

Time.utc_jpl()	Convert to a string like A.D.2014-Jan-18 01:35:37.5000 UTC..
Time.utc_iso([delimiter, places])	Convert to an ISO 8601 string like 2014-01-18T01:35:38Z in UTC.
Time.utc_strftime([format])	Format the UTC time using a Python datetime formatting string.
Time.utc_datetime()	Convert to a Python datetime in UTC.
Time.utc_datetime_and_leap_second()	Convert to a Python datetime in UTC, plus a leap second value.
Time.astimezone(tz)	Convert to a Python datetime in a particular timezone tz.
Time.astimezone_and_leap_second(tz)	Convert to a Python datetime and leap second in a timezone.
Time.toordinal()	Return the proleptic Gregorian ordinal of the UTC date.
Time.tai_calendar()	TAI as a (year, month, day, hour, minute, second) Calendar date.

Time.tt_calendar()	TT as a (year, month, day, hour, minute, second) Calendar date.
Time.tdb_calendar()	TDB as a (year, month, day, hour, minute, second) Calendar date.
Time.ut1_calendar()	UT1 as a (year, month, day, hour, minute, second) Calendar date.
Time.tai_strftime([format])	Format TAI with a datetime strftime() format string.
Time.tt_strftime([format])	Format TT with a datetime strftime() format string.
Time.tdb_strftime([format])	Format TDB with a datetime strftime() format string.
Time.ut1_strftime([format])	Format UT1 with a datetime strftime() format string.
Time.M	3×3 rotation matrix: ICRS → equinox of this date.
Time.MT	3×3 rotation matrix: equinox of this date → ICRS.
Time.gmst	Greenwich Mean Sidereal Time (GMST) in hours.
Time.gast	Greenwich Apparent Sidereal Time (GAST) in hours.
Time.nutation_matrix()	Compute the 3×3 nutation matrix N for this date.
Time.precession_matrix()	Compute the 3×3 precession matrix P for this date.
Time.to_astropy()	Return an AstroPy object representing this time.

Time utilities

compute_calendar_date(jd_integer[, …])	Convert Julian day jd_integer into a calendar (year, month, day).

Vector functions

The common API shared by planets, Earth locations, and Earth satellites.

VectorFunction	Given a time, computes a corresponding position.
VectorFunction.at(t)	At time t, compute the target's position relative to the center.

Either adding two vector functions v1 + v2 or subtracting them v1 - v2 produces a new function of time that, when invoked with .at(t), returns the sum or difference of the vectors returned by the two functions.

Planetary ephemerides

By downloading a SpiceKernel file, Skyfield users can build vector functions predicting the positions of the Moon, Sun, and planets. See Planets and their moons: JPL ephemeris files.

SpiceKernel(path)	Ephemeris file in NASA .bsp format.
SpiceKernel.close()	Close this ephemeris file.
SpiceKernel.names()	Return all target names that are valid with this kernel.
SpiceKernel.decode(name)	Translate a target name into its integer code.

Kernels also support lookup using the Python kernel['Mars'] syntax, in which case they return a function of time that returns vectors from the Solar System barycenter to the named body.

Planetary magnitudes

skyfield.magnitudelib.planetary_magnitude(position)

Given the position of a planet, return its visual magnitude.

```
>>> from skyfield.api import load
>>> from skyfield.magnitudelib import planetary_magnitude
>>> ts = load.timescale()
>>> t = ts.utc(2020, 7, 31)
>>> eph = load('de421.bsp')
>>> astrometric = eph['earth'].at(t).observe(eph['jupiter barycenter'])
>>> print('%.2f' % planetary_magnitude(astrometric))
-2.73
```

The formulae are from Mallama and Hilton "Computing Apparent Planetary Magnitude for the Astronomical Almanac" (2018). Two of the formulae have inherent limits:

- Saturn's magnitude is unknown and the function will return nan (the floating-point value "Not a Number") if the "illumination phase angle" — the angle of the vertex observer-Saturn-Sun — exceeds 6.5°.
- Neptune's magnitude is unknown and will return nan if the illumination phase angle exceeds 1.9° and the position's date is before the year 2000.

And one formula is not fully implemented (though contributions are welcome!):

- Skyfield does not compute which features on Mars are facing the observer, which can introduce an error of ±0.06 magnitude.

Planetary reference frames

PlanetaryConstants	Planetary constants manager.
Frame	Planetary constants frame, for building rotation matrices.

Almanac

Routines to search for events like sunrise, sunset, and Moon phase.

find_risings(observer, target, start_time, …)	Return the times at which a target rises above the eastern horizon.
find_settings(observer, target, start_time, …)	Return the times at which a target sets below the western horizon.
find_transits(observer, target, start_time, …)	Return the times at which a target transits across the meridian.
seasons(ephemeris)	Build a function of time that returns the quarter of the year.
moon_phase(ephemeris, t)	Return the Moon phase 0°–360° at time t, where 180° is Full Moon.
moon_phases(ephemeris)	Build a function of time that returns the moon phase 0 through 3.
moon_nodes(ephemeris)	Build a function of time that identifies lunar nodes.
oppositions_conjunctions(ephemeris, target)	Build a function to find oppositions and conjunctions with the Sun.
meridian_transits(ephemeris, target, topos)	Build a function of time for finding when a body transits the meridian.
sunrise_sunset(ephemeris, topos)	Build a function of time that returns whether the Sun is up.
dark_twilight_day(ephemeris, topos)	Build a function of time returning whether it is dark, twilight, or day.
risings_and_settings(ephemeris, target, topos)	Build a function of time that returns whether a body is up.
lunar_eclipses(start_time, end_time, eph)	Return the lunar eclipses between start_time and end_time.

Geographic locations

Skyfield supports two Earth datums for translating between latitude/longitude and Cartesian coordinates. They each use a slightly different estimate of the Earth's oblateness. The most popular is WGS84, which is used by the world's GPS devices:

- wgs84

- iers2010

Each datum offers a method for taking a latitude and longitude and returning a GeographicPosition that knows its position in space:

Geoid.latlon(latitude_degrees, longitude_degrees)	Return a GeographicPosition for a given latitude and longitude.

Going in the other direction, there are several methods for converting an existing Skyfield position into latitude, longitude, and height:

Geoid.latlon_of(position)	Return the latitude and longitude of a position.
Geoid.height_of(position)	Return the height above the Earth's ellipsoid of a position.
Geoid.geographic_position_of(position)	Return the GeographicPosition of a position.
Geoid.subpoint_of(position)	Return the point on the ellipsoid directly below a position.

Once you have used either of the above approaches to build a GeographicPosition, it offers several methods:

GeographicPosition.at(t)	At time t, compute the target's position relative to the center.
GeographicPosition.lst_hours_at(t)	Return the Local Apparent Sidereal Time, in hours, at time t.
GeographicPosition.refract(altitude_degrees, …)	Predict how the atmosphere will refract a position.
GeographicPosition.rotation_at(t)	Compute rotation from GCRS to this location's altazimuth system.

Kepler orbits

See Kepler Orbits for computing the positions of comets, asteroids, and other minor planets.

Kepler orbit data

load_mpcorb_dataframe(fobj)	Parse a Minor Planet Center orbits file into a Pandas dataframe.
load_comets_dataframe(fobj)	Parse a Minor Planet Center comets file into a Pandas dataframe.
load_comets_dataframe_slow(fobj)	Parse a Minor Planet Center comets file into a Pandas dataframe.

Earth satellites

By downloading TLE satellite element sets, Skyfield users can build vector functions that predict their positions. See Earth Satellites.

EarthSatellite(line1, line2[, name, ts])	An Earth satellite loaded from a TLE file and propagated with SGP4.
EarthSatellite.from_omm(ts, element_dict)	Build an EarthSatellite from OMM text fields.
EarthSatellite.from_satrec(satrec, ts)	Build an EarthSatellite from a raw sgp4 Satrec object.
TEME	The satellite-specific True Equator Mean Equinox frame of reference.

Stars and other distant objects

Star The position in the sky of a star or other fixed object.

Astronomical positions

The ICRF three-dimensional position vector serves as the base class for all of the following position classes. Each class represents an (*x,y,z*) .xyz and .velocity vector oriented to the axes of the International Celestial Reference System (ICRS), an inertial system that's an update to J2000 and that does not rotate with respect to the universe.

ICRF	An (*x,y,z*) position and velocity oriented to the ICRF axes.
Barycentric	An (*x,y,z*) position measured from the Solar System barycenter.
Astrometric	An astrometric (*x,y,z*) position relative to a particular observer.
Apparent	An apparent (*x,y,z*) position relative to a particular observer.
Geocentric	An (*x,y,z*) position measured from the center of the Earth.

You can also generate a position at the Solar System Barycenter.

SSB The Solar System Barycenter.

Positions are usually generated by the at(t) method of a vector function, rather than being constructed manually. But you can also build a position directly from a raw vector, or from right ascension and declination coordinates with position_of_radec().

position_of_radec(ra_hours, dec_degrees[, …])	Build a position object from a right ascension and declination.

All position objects offer five basic attributes:

.xyz	An (*x,y,z*) Distance.
.velocity	An (*x,y,z*) Velocity, or None.

.t	The Time of the position, or None.
.center	Body the vector is measured from.
.target	Body the vector is measured to.

The .xyz attribute used to be named .position. To support older code, Skyfield will always recognize the original name as an alias.

All positions support these methods:

ICRF.distance()	Compute the distance from the origin to this position.
ICRF.speed()	Compute the magnitude of the velocity vector.
ICRF.radec([epoch])	Compute equatorial RA, declination, and distance.
ICRF.hadec()	Compute hour angle, declination, and distance.
ICRF.altaz([temperature_C, pressure_mbar])	Compute (alt, az, distance) relative to the observer's horizon
ICRF.from_altaz([alt, az, alt_degrees, …])	Generate an Apparent position from an altitude and azimuth.
ICRF.separation_from(another_icrf)	Return the angle between this position and another.
ICRF.frame_xyz(frame)	Return this position as an (*x,y,z*) vector in a reference frame.
ICRF.frame_xyz_and_velocity(frame)	Return (*x,y,z*) position and velocity vectors in a reference frame.
ICRF.frame_latlon(frame)	Return latitude, longitude, and distance in the given frame.
ICRF.frame_latlon_and_rates(frame)	Return a reference frame latitude, longitude, range, and rates.
ICRF.from_time_and_frame_vectors(t, frame, …)	Constructor: build a position from two vectors in a reference frame.
ICRF.to_skycoord([unit])	Convert this distance to an AstroPy SkyCoord object.

ICRF.phase_angle(sun)	Return this position's phase angle: the angle Sun-target-observer.
ICRF.fraction_illuminated(sun)	Return the fraction of the target's disc that is illuminated.
ICRF.is_sunlit(ephemeris)	Return whether a position in Earth orbit is in sunlight.

In addition to the methods above, several subclasses of the base position class provide unique methods of their own:

Barycentric.observe(body)	Compute the Astrometric position of a body from this location.
Astrometric.apparent()	Compute an Apparent position for this body.

Reference frames

skyfield.framelib.true_equator_and_equinox_of_date	The dynamical frame of Earth's true equator and true equinox of date.
skyfield.framelib.itrs	The International Terrestrial Reference System (ITRS).
skyfield.framelib.ecliptic_frame	Reference frame of the true ecliptic and equinox of date.
skyfield.framelib.ecliptic_J2000_frame	Reference frame of the true ecliptic and equinox at J2000.
skyfield.framelib.galactic_frame	Galactic System II reference frame.
skyfield.sgp4lib.TEME	The satellite-specific True Equator Mean Equinox frame of reference.

Constellations

skyfield.api.load_constellation_map()

Load Skyfield's constellation boundaries and return a lookup function.

Skyfield carries an internal map of constellation boundaries that is optimized for quick position lookup. Call this function to load the map and return a function mapping position to constellation name.

```
>>> from skyfield.api import position_of_radec, load_constellation_map
>>> constellation_at = load_constellation_map()
>>> north_pole = position_of_radec(0, 90)
>>> constellation_at(north_pole)
```

'UMi'

If you pass an array of positions, you'll receive an array of names.

skyfield.api.load_constellation_names()

Return a list of abbreviation-name tuples, like ('Aql', 'Aquila').

You can pass the list to Python's dict() to build a dictionary that turns a constellation abbreviation into a full name:

```
>>> from skyfield.api import load_constellation_names
>>> d = dict(load_constellation_names())
>>> d['UMa']
'Ursa Major'
```

By swapping the order of the two items, you can map the other way, from a full name back to an abbreviation:

```
>>> f = dict(reversed(item) for item in load_constellation_names())
>>> f['Ursa Major']
'UMa'
```

skyfield.data.stellarium.parse_constellations(lines)

Return a list of constellation outlines.

Each constellation outline is a list of edges, each of which is drawn between a pair of specific stars:

```
[
  (name, [(star1, star2), (star3, star4), ...]),
  (name, [(star1, star2), (star3, star4), ...]),
  ...
]
```

Each name is a 3-letter constellation abbreviation; each star is an integer Hipparcos catalog number. See Drawing a finder chart for comet NEOWISE for an example of how to combine this data with the Hipparcos star catalog to draw constellation lines on a chart.

skyfield.data.stellarium.parse_star_names(lines)

Return the names in a Stellarium star_names.fab file.

Returns a list of named tuples, each of which offers a .hip attribute with a Hipparcos catalog number and a .name attribute with the star name. Do not depend on the tuple having only length two; additional fields may be added in the future.

Searching

skyfield.searchlib.find_discrete()

Find the times at which a discrete function of time changes value.

This routine is used to find instantaneous events like sunrise, transits, and the seasons. See Searching for the dates of astronomical events for how to use it yourself.

skyfield.searchlib.find_maxima()

Find the local maxima in the values returned by a function of time.

This routine is used to find events like highest altitude and maximum elongation. See Searching for the dates of astronomical events for how to use it yourself.

skyfield.searchlib.find_minima()

Find the local minima in the values returned by a function of time.

This routine is used to find events like minimum elongation. See Searching for the dates of astronomical events for how to use it yourself.

Osculating orbital elements

This routine returns osculating orbital elements for an object's instantaneous position and velocity.

osculating_elements_of(position[, …])	Produce the osculating orbital elements for a position.

OsculatingElements.apoapsis_distance	Distance object
OsculatingElements.argument_of_latitude	Angle object
OsculatingElements.argument_of_periapsis	Angle object
OsculatingElements.eccentric_anomaly	Angle object
OsculatingElements.eccentricity	numpy.ndarray
OsculatingElements.inclination	Angle object
OsculatingElements.longitude_of_ascending_node	Angle object
OsculatingElements.longitude_of_periapsis	Angle object
OsculatingElements.mean_anomaly	Angle object
OsculatingElements.mean_longitude	Angle object
OsculatingElements.mean_motion_per_day	Angle object
OsculatingElements.periapsis_distance	Distance object

OsculatingElements.periapsis_time	Time object
OsculatingElements.period_in_days	numpy.ndarray
OsculatingElements.semi_latus_rectum	Distance object
OsculatingElements.semi_major_axis	Distance object
OsculatingElements.semi_minor_axis	Distance object
OsculatingElements.time	Time object
OsculatingElements.true_anomaly	Angle object
OsculatingElements.true_longitude	Angle object

Units

Distance	Distance
Velocity	Velocity
Angle	Angle
AngleRate	Rate at which an angle is changing

All three kinds of quantity support one or more methods.

Distance.au(value)	Astronomical units (the Earth-Sun distance of 149,597,870,700 m).
Distance.km(value)	Kilometers (1,000 meters).
Distance.m(value)	Meters.
Distance.length()	Compute the length when this is an (*x,y,z*) vector.
Distance.light_seconds()	Return the length of this vector in light seconds.
Distance.to(unit)	Convert this distance to the given AstroPy unit.
Velocity.au_per_d(value)	Astronomical units per day.
Velocity.km_per_s(value)	Kilometers per second.
Velocity.m_per_s(value)	Meters per second.

Velocity.to(unit)	Convert this velocity to the given AstroPy unit.
Angle.radians(value)	Radians (τ = 2π in a circle).
Angle.hours	Hours (24^h in a circle).
Angle.degrees	Degrees (360° in a circle).
Angle.arcminutes()	Kembalikan sudut dalam menit busur.
Angle.arcseconds()	Mengembalikan sudut dalam detik busur.
Angle.mas()	Kembalikan sudut dalam milidetik busur.
Angle.to(satuan)	Ubah sudut ini ke satuan AstroPy yang diberikan.
Angle.hms([memperingatkan])	Ubah ke tupel (jam, menit, detik).
Angle.signed_hms([memperingatkan])	Ubah ke tupel (tanda, jam, menit, detik).
Angle.hstr([menempatkan, memperingatkan, memformat])	Mengembalikan string seperti ; lihat Memformat sudut .12h 07m 30.00s
Angle.dms([memperingatkan])	Ubah ke tupel (derajat, menit, detik).
Angle.signed_dms([memperingatkan])	Ubah ke tupel (tanda, derajat, menit, detik).
Angle.dstr([menempatkan, memperingatkan, memformat])	Mengembalikan string seperti ; lihat Memformat sudut .181deg 52' 30.0"
AngleRate.radians	Rateperubahan radian.
AngleRate.degrees	Rateperubahan derajat.
AngleRate.arcminutes	Rateperubahan dalam menit busur.
AngleRate.arcseconds	Rateperubahan dalam detik busur.
AngleRate.mas	Rateperubahan dalam milidetik busur.
Rate.per_day	Satuan Waktu Terestrial per hari.
Rate.per_hour	Satuan per jam Waktu Terestrial.
Rate.per_minute	Satuan per menit Waktu Terestrial.
Rate.per_second	Satuan per detik Waktu Terestrial.

Trigonometri

position angle of(pasangan pasangan sudut2)	sudut1,	Mengembalikan sudut posisi satu posisi terhadap posisi lainnya.

Referensi API — Membuka File

Lihat Mengunduh dan Menggunakan Berkas Data untuk penjelasan tentang bagaimana program Skyfield menggunakan contoh kelas Loaderyang dijelaskan di bawah ini untuk mengunduh dan membuka berkas data yang dibutuhkan agar dapat beroperasi.

Kelas pemuat

kelas skyfield.iokit.Loader(direktori , verbose=Benar , kadaluarsa=Salah)

Alat untuk mengunduh dan membuka berkas data astronomi.

Default Loaderyang menyimpan file data ke direktori kerja saat ini dapat diimpor langsung dari API Skyfield:

from skyfield.api import load

Namun, pengguna juga dapat membuat direktori Loadermereka sendiri, jika ada direktori lain yang ingin mereka gunakan untuk menyimpan file data, atau jika mereka ingin menentukan opsi yang berbeda. Direktori tersebut dibuat secara otomatis jika belum ada:

from skyfield.api import Loader

load = Loader('~/skyfield-data')

Pilihannya adalah:

verbose

Jika diatur ke False, maka loader tidak akan mencetak bilah kemajuan ke layar setiap kali mengunduh berkas. (Jika keluaran standar bukan TTY, maka tidak ada bilah kemajuan yang dicetak.)

expire

(Opsi ini tidak lagi didukung.)

Setelah a Loaderdibuat, ia dapat dipanggil seperti fungsi untuk membuka, atau mengunduh dan membuka, file yang namanya dikenali:

planets = load('de405.bsp')

Setiap loader juga mendukung atribut dan beberapa metode.

directoryBahasa Indonesia:

Direktori tempat pemuat ini mencari saat mencoba membuka suatu berkas, dan tempat mengunduh berkas yang belum diunduh.

path_to(nama berkas)

Kembalikan jalur ke filenamedalam direktori loader ini.

days_old(nama berkas)

Kembalikan seberapa baru filenamedimodifikasi, diukur dalam hari.

build_url(nama berkas)

Kembalikan URL yang akan dicoba diunduh Skyfield untuk nama file yang diberikan.

Timbul ValueErrorjika Skyfield tidak mengetahui di mana mendapatkan berkas berdasarkan namanya.

tle_file(url , reload=Salah , filename=Tidak Ada , ts=Tidak Ada , skip_names=Salah)

Memuat dan mengurai berkas TLE, yang menghasilkan daftar satelit Bumi.

Diberikan URL atau jalur lokal ke berkas teks ASCII, ini memuat serangkaian set “Elemen Dua Baris” TLE dan mengembalikan daftar EarthSatelliteobjek untuk set tersebut. Lihat Satelit Bumi .

Lihat open()metode untuk mengetahui arti parameter reloaddan filename.

Lihat parse_tle_file()fungsi untuk arti parameter tsdan skip_names.

download(url , namafile=Tidak ada , cadangan=Salah)

Unduh berkas, meskipun sudah ada di disk; kembalikan jalurnya.

You can specify the local filename to which the file will be saved; the default is to use the final component of url. Set backup to True if you want an already-existing file moved out of the way instead of overwritten.

Your operating system may raise any of several errors during a download: hostname lookup failure (this is the usual symptom if you are disconnected from the Internet); the server refusing the connection; and the connection closing mid-download. Skyfield makes no attempt to intercept or interpret these errors — which vary by operating system — so your application itself should catch network errors if it needs to avoid printing raw Python exceptions, or if you want to retry failed downloads.

open(url, mode='rb', reload=False, filename=None, backup=False)

Open a file, downloading it first if it does not yet exist.

Unlike when you call a loader directly like my_loader(), this my_loader.open() method does not attempt to parse or interpret the file; it simply returns an open file object.

The url can be either an external URL, or else the path to a file on the current filesystem. A relative path will be assumed to be relative to the base directory of this loader object.

If a URL was provided and the reload parameter is true, then any existing file will be removed before the download starts.

The filename parameter lets you specify an alternative local filename instead of having the filename extracted from the final component of the URL.

timescale(delta_t=None, builtin=True)

Return a Timescale built using official Earth rotation data.

delta_t — Lets you override the standard ΔT tables by providing your own ΔT offset in seconds. For details, see Setting a Custom Value For ΔT.

builtin — By default, Skyfield uses ΔT and leap second tables that it carries internally; to instead load this data from files, set this option to False. For compatibility with Skyfield ≤ 1.30, if you have on disk the three files deltat.data, deltat.preds, and Leap_Second.dat, then Skyfield will load them. Otherwise, Skyfield will download and use finals2000A.all from the International Earth Rotation Service. For details, see UT1 and downloading IERS data.

Standalone functions

skyfield.iokit.load_file(path)

Open a file on your local drive, using its extension to guess its type.

This routine only works on .bsp ephemeris files right now, but will gain support for additional file types in the future.

from skyfield.api import load_file

planets = load_file('~/Downloads/de421.bsp')

skyfield.iokit.parse_tle_file(lines, ts=None, skip_names=False)

Menguraikan baris data satelit TLE, menghasilkan urutan satelit.

Diberikan rangkaian linesstring byte (yang dapat berupa berkas biner terbuka, yang berfungsi seperti rangkaian baris dalam Python), rutin ini menghasilkan EarthSatelliteuntuk setiap pasangan baris yang berdekatan yang dimulai dengan dan dan memiliki 69 karakter atau lebih masing-masing. Jika baris sebelum TLE bukan bagian dari TLE lain, maka digunakan sebagai satelit ."1 ""2 ".name

Jika Anda meneruskan tsskala waktu, Skyfield akan menggunakannya untuk membangun .epochatribut tanggal pada setiap satelit; jika tidak, skala waktu yang berasal dari file detik kabisat bawaan Skyfield akan digunakan.

Jika untuk file tertentu Anda melihat baris teks acak yang ditafsirkan sebagai nama satelit, atur skip_nameske Truedan Skyfield tidak akan mencoba menyimpan nama satelit.

Lihat Satelit Bumi untuk detailnya. Pengecualian diajukan jika upaya untuk mengurai sepasang garis kandidat sebagai garis TLE gagal.

Referensi API — Waktu

Lihat panduan Tanggal dan Waktu untuk pembahasan cermat dan terperinci mengenai beberapa skala waktu yang digunakan oleh para astronom, dan tentang cara mengonversi waktu ke dan dari skala waktu yang umum seperti UTC dan zona waktu di seluruh dunia yang diadaptasi darinya.

Tanggal kalender

1. Saat membuat Timedari tanggal kalender, Anda tidak hanya dapat menetapkan satu momen berdasarkan waktu dan tanggalnya, tetapi Anda juga dapat membuat yang Timemewakili seluruh rangkaian momen dengan menyediakan daftar Python atau rangkaian NumPy untuk tahun, bulan, hari, jam, menit, atau detik. Lihat Rangkaian tanggal .
2. Secara default, Skyfield menggunakan kalender Gregorian modern untuk semua tanggal — bahkan tanggal sebelum kalender Gregorian diperkenalkan pada tahun 1582.
3. Anda dapat memilih untuk menggunakan kalender Julian lama untuk tanggal-tanggal kuno, yang merupakan praktik yang paling umum di kalangan sejarawan. Lihat Tanggal-tanggal kuno dan modern .

Skala waktu, untuk membangun dan mengubah waktu

kelas skyfield.timelib.Timescale(delta_t_recent , leap_dates , leap_offsets)

Data yang diperlukan untuk menyatakan tanggal dalam skala waktu yang berbeda.

A Timescalemenyediakan objek waktu dengan tabel data yang mereka butuhkan untuk menerjemahkan antar skala waktu yang berbeda: jadwal detik kabisat UTC, dan nilai ΔT dari waktu ke waktu. Sebagian besar program membuat satu tabel Timescaleyang mereka gunakan untuk membangun Timeobjek mereka:

```
>>> from skyfield.api import load
>>> ts = load.timescale()
>>> t = ts.utc(1980, 3, 1, 9, 30)
>>> t
<Time tt=2444299.896425741>
```

Lihat UT1 dan mengunduh data IERS jika Anda tertarik untuk memeriksa seberapa baru data dalam file yang dimuat berdasarkan skala waktu.

now( )

Mengembalikan tanggal dan waktu saat ini sebagai Timeobjek.

from_datetime(tanggal waktu)

Kembalikan a Timeuntuk Python datetime.

Harus datetime“mengetahui zona waktu”: harus memiliki objek zona waktu sebagai tzinfoatributnya, bukan None.

Baru di versi 1.24.

from_datetimes(daftar_tanggal_waktu)

Kembalikan a Timeuntuk daftar datetimeobjek Python.

Setiap objek datetimeharus “mengetahui zona waktu”: setiap objek harus memiliki objek zona waktu sebagai tzinfoatributnya, bukan None.

Baru di versi 1.24.

utc(tahun , bulan = 1 , hari = 1 , jam = 0 , menit = 0 , detik = 0.0)

Membangun dari tanggal KalenderTime UTC .

New in version 1.24: Passing a Python datetime or a list of datetimes as the first argument has been deprecated (and was never supported for the other time scale methods). Instead, use the methods from_datetime() and from_datetimes().

tai(year=None, month=1, day=1, hour=0, minute=0, second=0.0, jd=None)

Build a Time from an International Atomic Time Calendar date.

New in version 1.6: Passing a Julian date with jd= has been deprecated; instead, use tai_jd().

tai_jd(jd, fraction=None)

Build a Time from an International Atomic Time Julian date.

tt(year=None, month=1, day=1, hour=0, minute=0, second=0.0, jd=None)

Build a Time from a Terrestrial Time Calendar date.

New in version 1.6: Passing a Julian date with jd= has been deprecated; instead, use tt_jd().

tt_jd(jd, fraction=None)

Build a Time from a Terrestrial Time Julian date.

J(year)

Build a Time from a Terrestrial Time Julian year or array.

Julian years are convenient uniform periods of exactly 365.25 days of Terrestrial Time, centered on 2000 January 1 12h TT = Julian year 2000.0.

tdb(year=None, month=1, day=1, hour=0, minute=0, second=0.0, jd=None)

Build a Time from a Barycentric Dynamical Time Calendar date.

New in version 1.6: Passing a Julian date with jd= has been deprecated; instead, use tdb_jd().

tdb_jd(jd, fraction=None)

Build a Time from a Barycentric Dynamical Time Julian date.

ut1(year=None, month=1, day=1, hour=0, minute=0, second=0.0, jd=None)

Build a Time from a UT1 Universal Time Calendar date.

New in version 1.6: Passing a Julian date with jd= has been deprecated; instead, use ut1_jd().

ut1_jd(jd)

Build a Time from a UT1 Universal Time Julian date.

from_astropy(t)

Build a Skyfield Time from an AstroPy time object.

linspace(t0, t1, num=50)

Return num times spaced uniformly between t0 to t1.

This routine is named after, and powered by, the NumPy routine linspace().

The Time object

class skyfield.timelib.Time(ts, tt, tt_fraction=None)

A single moment in history, or an array of several moments.

Skyfield programs don't usually instantiate this class directly, but instead build time objects using one of the timescale methods listed at Time scales. If you do attempt the low-level operation of building a time object yourself, either leave tt_fraction at its default value of None — in which case Skyfield will assume the fraction is zero — or provide a tt_fraction array that has exactly the same dimensions as your tt array.

Four basic floating-point values can be directly accessed as attributes:

tai

International Atomic Time (TAI) as a Julian date.

tt

Terrestrial Time (TT) as a Julian date.

J

Terrestrial Time (TT) as a floating point number of Julian years.

tdb

Barycentric Dynamical Time (TDB) as a Julian date.

ut1

Universal Time (UT1) as a Julian date.

Two standard differences between time scales are also available as attributes:

delta_t

The difference TT − UT1 measured in seconds.

dut1

The difference UT1 − UTC measured in seconds.

All of the other ways of expressing the time and converting it to typical human systems like UTC and world time zones are offered through methods:

astimezone(tz)

Convert to a Python datetime in a particular timezone tz.

If this time is an array, then an array of datetimes is returned instead of a single value.

astimezone_and_leap_second(tz)

Convert to a Python datetime and leap second in a timezone.

Convert this time to a Python datetime and a leap second:

dt, leap_second = t.astimezone_and_leap_second(tz)

The argument tz should be a datetime compatible timezone.

The leap second value is provided because a Python datetime can only number seconds 0 through 59, but leap seconds have a designation of at least 60. The leap second return value will normally be 0, but will instead be 1 if the date and time are a UTC leap second. Add the leap second value to the second field of the datetime to learn the real name of the second.

If this time is an array, then an array of datetime objects and an array of leap second integers is returned, instead of a single value each.

toordinal()

Return the proleptic Gregorian ordinal of the UTC date.

This method makes Skyfield Time objects compatible with Python datetime objects, which also provide a toordinal() method. Thanks to this method, a Time can often be used directly as a coordinate for a plot.

utc_datetime()

Convert to a Python datetime in UTC.

If this time is an array, then a list of datetimes is returned instead of a single value.

utc_datetime_and_leap_second()

Convert to a Python datetime in UTC, plus a leap second value.

Convert this time to a datetime object and a leap second:

dt, leap_second = t.utc_datetime_and_leap_second()

The leap second value is provided because a Python datetime can only number seconds 0 through 59, but leap seconds have a designation of at least 60. The leap second return value will normally be 0, but will instead be 1 if the date and time are a UTC leap

second. Add the leap second value to the second field of the datetime to learn the real name of the second.

If this time is an array, then an array of datetime objects and an array of leap second integers is returned, instead of a single value each.

utc_iso(delimiter='T', places=0)

Convert to an ISO 8601 string like 2014-01-18T01:35:38Z in UTC.

If this time is an array of dates, then a sequence of strings is returned instead of a single string.

utc_jpl()

Convert to a string like A.D. 2014-Jan-18 01:35:37.5000 UTC.

Returns a string for this date and time in UTC, in the format used by the JPL HORIZONS system. If this time is an array of dates, then a sequence of strings is returned instead of a single string.

utc_strftime(format='%Y-%m-%d %H:%M:%S UTC')

Format the UTC time using a Python datetime formatting string.

Ini memanggil Python time.strftime()untuk memformat tanggal dan waktu. Satu string dikembalikan atau seluruh array string, tergantung pada apakah objek waktu ini adalah array. Format yang paling umum digunakan adalah:

- %Ytahun empat digit, %ytahun dua digit
- %mnomor bulan, %Bnama, %bsingkatan
- %dhari dalam bulan
- %Hjam
- %Mmenit
- %SKedua
- %Ahari dalam seminggu, %asingkatannya

Format %Zdan %ztidak didukung; sebagai gantinya, cukup gunakan karakter literal 'UTC'dalam string format Anda.

Jika satuan waktu terkecil dalam format Anda adalah menit atau detik, maka waktu dibulatkan ke menit atau detik terdekat. Jika tidak, nilainya dipotong alih-alih dibulatkan.

tai_calendar( )

TAI sebagai tanggal Kalender (tahun, bulan, hari, jam, menit, detik) .

tt_calendar( )

TT sebagai tanggal Kalender (tahun, bulan, hari, jam, menit, detik) .

tdb_calendar( )

TDB sebagai tanggal Kalender (tahun, bulan, hari, jam, menit, detik) .

ut1_calendar( )

UT1 sebagai tanggal Kalender (tahun, bulan, hari, jam, menit, detik) .

tai_strftime(format='%Y-%m-%d %H:%M:%S TAI')

Format TAI dengan string format datetime strftime().

tt_strftimeBahasa Indonesia: (format='%Y-%m-%d %H:%M:%S TT')

Format TT dengan string format datetime strftime().

tdb_strftime(format='%Y-%m-%d %H:%M:%S TDB')

Format TDB dengan string format datetime strftime().

ut1_strftimeBahasa Indonesia: (format='%Y-%m-%d %H:%M:%S UT1')

Format UT1 dengan string format datetime strftime().

MBahasa Indonesia:

Matriks rotasi 3×3: ICRS → ekuinoks tanggal ini.

MTBahasa Indonesia:

Matriks rotasi 3×3: ekuinoks tanggal ini → ICRS.

J

Mengembalikan tahun Julian titik mengambang atau array tahun untuk tanggal ini.

Tahun Julian merupakan periode seragam yang tepat 365,25 hari Waktu Terestrial, berpusat pada 1 Januari 2000 12h TT = tahun Julian 2000.0.

utcBahasa Indonesia:

Sebuah tupel dalam UTC.(year, month, day, hour, minute, second)

gmstBahasa Indonesia:

Waktu Sideris Rata-Rata Greenwich (GMST) dalam jam.

gastBahasa Indonesia:

Waktu Sideris Tampak Greenwich (GAST) dalam jam.

nutation_matrix( )

Hitunglah matriks nutasi 3×3 N untuk tanggal ini.

precession_matrix( )

Hitunglah matriks presesi 3×3 P untuk tanggal ini.

to_astropy( )

Kembalikan objek AstroPy yang mewakili waktu ini.

Utilitas waktu

skyfield.timelib.compute_calendar_date(jd_integer , julian_before=Tidak Ada)

Ubah hari Julian jd_integermenjadi kalender (tahun, bulan, hari).

Menggunakan kalender Gregorian proleptik kecuali julian_beforejika ditetapkan pada hari Julian tertentu, dalam hal ini kalender Julian digunakan untuk tanggal yang lebih tua dari itu.

Referensi API — Fungsi Vektor

API Skyfield dibangun di atas *fungsi vektor* yang menggunakan waktu sebagai input dan menghasilkan vektor posisi. Anda dapat membuat fungsi vektor untuk Bumi, Bulan, Matahari, planet, dan satelit Bumi, dan dapat menggabungkannya menggunakan penjumlahan dan pengurangan.

kelas skyfield.vectorlib.VectorFunction

Diberi waktu, hitung posisi yang sesuai.

centerBahasa Indonesia:

Objek Tata Surya yang menjadi dasar pengukuran vektor ini. Sering kali ini berupa kode bilangan bulat seperti 399untuk Bumi, 3untuk pusat gravitasi sistem Bumi-Bulan, atau 0untuk pusat Tata Surya itu sendiri, meskipun bisa juga berupa objek tertentu seperti GeographicPosition di permukaan Bumi atau EarthSatellite di orbit di sekitarnya.

targetBahasa Indonesia:

Dengan menggunakan kumpulan nilai yang mungkin sama seperti center, atribut ini menamai target yang ditunjuk oleh vektor. Jadi, vektor adalah perbedaan tiga dimensi antara posisi pusat dan target.

vf1 + vf2

Mengembalikan fungsi vektor baru yang at(t), ketika dipanggil, menghitung jumlah vektor asli vf1dan vf2. Ini akan menimbulkan kesalahan kecuali tempat targetsalah satu dari dua vektor berakhir sama dengan centertempat vektor lainnya dimulai.

vf1 - vf2

Kembalikan fungsi vektor baru yang at(t), saat dipanggil, menghitung di mana , targetakan vf1diposisikan relatif terhadap , targetyang dikurangi vf2. Perhatikan bahwa ini akan menjadi vektor sesaat, tidak dikoreksi untuk jumlah waktu yang dibutuhkan cahaya untuk bergerak dari satu target ke target lainnya. Ini menimbulkan kesalahan kecuali kedua vektor berbagi yang sama center.

at(T)

Pada waktu t, hitung posisi target relatif terhadap pusat.

Jika tmerupakan array waktu, maka objek posisi yang dikembalikan akan menentukan posisi sebanyak jumlah waktu yang ada. Jenis posisi yang dikembalikan bergantung pada nilai atribut center :

- Barisenter Tata Surya:Barycentric
- Pusat Bumi:Geocentric
- Ada lagi:ICRF

Referensi API — Ephemeris Planet

Lihat Planet dan bulannya: berkas ephemeris JPL untuk panduan penggunaan objek ephemeris di bawah ini untuk menghitung posisi planet, bulan, dan Matahari.

Berkas ephemeris JPL .bsp

kelas skyfield.jpllib.SpiceKernel(jalur)

Berkas Ephemeris dalam format NASA .bsp.

File “Spacecraft and Planet Kernel” (SPK) dari NASA menyediakan koordinat (*x,y,z*) untuk benda-benda di Tata Surya seperti Matahari, planet, bulan, dan pesawat ruang angkasa.

Anda dapat mengunduh sendiri file .bsp dan menggunakan kelas ini untuk membukanya, atau menggunakan load()fungsi Skyfield untuk mengunduh ephemeris populer secara otomatis. Setelah dimuat, Anda dapat mencetak objek ini ke layar untuk melihat laporan mengenai segmen yang disertakan:

```
>>> planets = load('de421.bsp')
>>> print(planets)
SPICE kernel file 'de421.bsp' has 15 segments
  JD 2414864.50 - JD 2471184.50  (1899-07-28 through 2053-10-08)
    0 -> 1    SOLAR SYSTEM BARYCENTER -> MERCURY BARYCENTER
    0 -> 2    SOLAR SYSTEM BARYCENTER -> VENUS BARYCENTER
    0 -> 3    SOLAR SYSTEM BARYCENTER -> EARTH BARYCENTER
    0 -> 4    SOLAR SYSTEM BARYCENTER -> MARS BARYCENTER
    0 -> 5    SOLAR SYSTEM BARYCENTER -> JUPITER BARYCENTER
    0 -> 6    SOLAR SYSTEM BARYCENTER -> SATURN BARYCENTER
    0 -> 7    SOLAR SYSTEM BARYCENTER -> URANUS BARYCENTER
    0 -> 8    SOLAR SYSTEM BARYCENTER -> NEPTUNE BARYCENTER
    0 -> 9    SOLAR SYSTEM BARYCENTER -> PLUTO BARYCENTER
    0 -> 10   SOLAR SYSTEM BARYCENTER -> SUN
    3 -> 301  EARTH BARYCENTER -> MOON
    3 -> 399  EARTH BARYCENTER -> EARTH
    1 -> 199  MERCURY BARYCENTER -> MERCURY
    2 -> 299  VENUS BARYCENTER -> VENUS
    4 -> 499  MARS BARYCENTER -> MARS
```

Untuk mengambil satu atau lebih vektor yang diperlukan guna menghitung posisi suatu benda relatif terhadap barisentrum Tata Surya, cari benda tersebut berdasarkan namanya atau bilangan bulat pengenal resmi SPICE:

```
>>> planets['earth']
<VectorSum of 2 vectors:
 'de421.bsp' segment 0 SOLAR SYSTEM BARYCENTER -> 3 EARTH BARYCENTER
 'de421.bsp' segment 3 EARTH BARYCENTER -> 399 EARTH>
>>> planets[499]
<VectorSum of 2 vectors:
 'de421.bsp' segment 0 SOLAR SYSTEM BARYCENTER -> 4 MARS BARYCENTER
 'de421.bsp' segment 4 MARS BARYCENTER -> 499 MARS>
```

Hasilnya akan menjadi VectorFunction contoh di mana Anda dapat meminta posisi pada waktu input tertentu.

close( )

Tutup berkas ephemeris ini.

names( )

Kembalikan semua nama target yang valid dengan kernel ini.

```
>>> pprint(planets.names())
{0: ['SOLAR_SYSTEM_BARYCENTER', 'SSB', 'SOLAR SYSTEM BARYCENTER'],
 1: ['MERCURY_BARYCENTER', 'MERCURY BARYCENTER'],
 2: ['VENUS_BARYCENTER', 'VENUS BARYCENTER'],
 3: ['EARTH_BARYCENTER',
     'EMB',
 ...
```

Hasilnya adalah kamus dengan kunci kode target dan daftar nama sebagai nilai. Nama belakang dalam setiap daftar adalah nama yang digunakan Skyfield saat mencetak informasi tentang suatu badan.

decode(nama)

Terjemahkan nama target ke dalam kode integernya.

```
>>> planets.decode('Venus')
299
```

Dimunculkan ValueErrorjika Anda memberikan nama yang tidak diketahui, atau KeyErrorjika target tidak ada dalam kernel ini. Anda dapat memberikan kode integer jika Anda sudah memilikinya dan hanya ingin memeriksa apakah kode tersebut ada dalam kernel ini.

Referensi API — Lokasi Geografis

skyfield.toposlib.wgs84= <objek skyfield.toposlib.Geoid>

Sistem Geodetik Dunia 1984 Geoid.

Ini adalah geoid standar yang digunakan oleh sistem GPS, dan kemungkinan merupakan standar yang dimaksudkan jika Anda diberikan lintang dan bujur yang tidak menentukan geoid alternatif.

skyfield.toposlib.iers2010= <objek skyfield.toposlib.Geoid>

Layanan Rotasi Bumi Internasional 2010 Geoid.

kelas skyfield.toposlib.Geoid(nama , radius_m , perataan_terbalik)

Elipsoid Bumi: memetakan lintang dan bujur ke posisi (*x, y, z*).

Alih-alih membuat objek geoid mereka sendiri, sebagian besar pengguna Skyfield cukup menggunakan wgs84objek yang sudah ada di dalamnya.

Matematika untuk mengubah posisi menjadi lintang dan bujur didasarkan pada artikel Dr. TS Kelso yang cukup bermanfaat, Sistem Koordinat Orbital, Bagian III .

subpoint(…)

Tidak digunakan lagi sejak versi 1.40: Diganti namanya menjadi geographic_position_of().

polar_radiusBahasa Indonesia:

Jari-jari kutub Bumi, sebagai Distance.

latlon(derajat_lintang , derajat_bujur , elevasi_m=0.0 , cls=<class 'skyfield.toposlib.GeographicPosition'>)

Mengembalikan a GeographicPositionuntuk lintang dan bujur tertentu.

Bujur dan lintang harus ditentukan dalam derajat. Jika tidak ada ketinggian dalam meter yang diberikan, posisi yang dikembalikan akan berada di permukaan elipsoid. Bujur positif ke arah timur, jadi berikan angka negatif untuk barat:

```
from skyfield.api import wgs84

observatory = wgs84.latlon(37.3414, -121.6429)  # 121.6° West
```

Anda dapat menghindari mengingat arah mana yang negatif dengan menggunakan konstanta arah kompas Skyfield, yang memiliki nilai +1 dan −1:

```
from skyfield.api import N, S, E, W

observatory = wgs84.latlon(37.3414 * N, 121.6429 * W)
```

latlon_of(posisi)

Mengembalikan lintang dan bujur suatu position.

Posisinya .centerharus 399, yaitu pusat Bumi. Lintang dan bujur geodetik dikembalikan sebagai sepasang Angleobjek.

height_of(posisi)

Kembalikan ketinggian di atas elipsoid Bumi dari position.

Posisinya .centerharus 399, yaitu pusat Bumi. A Distancedikembalikan dengan memberikan ketinggian geodetik posisi di atas permukaan Bumi.

geographic_position_of(posisi)

Kembalikan GeographicPositiondari position.

Posisinya .centerharus 399, pusat Bumi. A GeographicPositiondikembalikan dengan memberikan posisi geodetik latitudedan longitude, dan elevation di atas atau di bawah permukaan elipsoid.

subpoint_of(posisi)

Kembalikan titik pada ellipsoid tepat di bawah a position.

The position's .center must be 399, the center of the Earth. Returns a GeographicPosition giving the geodetic latitude and longitude that lie directly below the input position, and an elevation above the ellipsoid of zero.

class skyfield.toposlib.GeographicPosition(model, latitude, longitude, elevation, itrs_xyz)

A latitude-longitude-elevation position on Earth.

Each instance of this class holds an (*x,y,z*) vector for a geographic position on, above, or below the Earth's surface, in the ITRS reference frame: the international standard for an Earth-centered Earth-fixed (ECEF) reference frame. Instead of instantiating this class directly, Skyfield users usually give a reference geoid the longitude and latitude they are interested in:

from skyfield.api import wgs84

topos = wgs84.latlon(37.3414, -121.6429)

Once a geographic position has been created, here are its attributes and methods:

model

The Geoid, like WGS84 or IERS2010, that this position uses to map latitude, longitude, and elevation to a three-dimensional Cartesian position.

latitude

An Angle specifying latitude; the north pole has latitude +90°.

longitude

An Angle specifying longitude; east is positive, west is negative.

elevation

A Distance specifying elevation above (positive) or below (negative) the surface of the Earth ellipsoid specified by this position's model.

itrs_xyz

A Distance object giving the spatial (*x,y,z*) coordinates of this location in the ITRS Earth-centered Earth-fixed ("ECEF") reference frame.

center

The integer 399, which identifies this position as geocentric: its (*x,y,z*) coordinates are measured from the Earth's center.

at(t)

Return the position of this Earth location at time t.

lst_hours_at(t)

Return the Local Apparent Sidereal Time, in hours, at time t.

This location's Local Apparent Sidereal Time (LAST) is the right ascension of the zenith at the time t, as measured against the "true" Earth equator and equinox (rather than the fictional "mean" equator and equinox, which ignore the Earth's nutation).

refract(altitude_degrees, temperature_C, pressure_mbar)

Predict how the atmosphere will refract a position.

Given a body that is standing altitude_degrees above the true horizon, return an Angle predicting its apparent altitude given the supplied temperature and pressure, either of which can be the string 'standard' to use 10°C and a pressure of 1010 mbar adjusted for the elevation of this geographic location.

rotation_at(t)

Compute rotation from GCRS to this location's altazimuth system.

class skyfield.toposlib.ITRSPosition(itrs_xyz)

Posisi (*x,y,z*) dalam kerangka ITRS tetap Bumi yang berpusat pada Bumi (ECEF).

Vektor (*x, y, z*) ini tidak memiliki pengetahuan tentang geoid standar, lintang, atau bujur, tetapi berguna jika Anda sudah mengetahui koordinat persegi panjang lokasi target:

from skyfield.api import Distance

from skyfield.toposlib import ITRSPosition

d = Distance(km=[-3918, -1887, 5209])

p = ITRSPosition(d)

at(T)

Kembalikan posisi GCRS dari koordinat ITRS ini pada waktu t.

Referensi API — Almanak

Lihat Perhitungan Almanak untuk panduan penggunaan rutin ini untuk mencari waktu matahari terbit, matahari terbenam, dan fase bulan.

skyfield.almanac.find_risings(pengamat , target , waktu_mulai , waktu_berakhir , derajat_horison=Tidak_Ada)

Kembalikan waktu saat target terbit di atas cakrawala timur.

Dengan adanya pengamat di permukaan Bumi, target seperti Matahari atau Bulan atau planet, dan Time objek awal dan akhir, ini akan mengembalikan dua larik yang memiliki panjang yang sama. Yang pertama adalah Timedaftar saat target naik. Larik kedua berisi Truesetiap kali target benar-benar melintasi cakrawala, dan Falsesaat target hanya melintas tanpa benar-benar menyentuh cakrawala.

Lihat Risings and settings untuk contoh, dan Computing your own angle refraction untuk cara menggunakan horizon_degreesargumen.

Baru di versi 1.47.

skyfield.almanac.find_settings(pengamat , target , waktu_mulai , waktu_berakhir , derajat_horison=Tidak_Ada)

Kembalikan waktu saat target terbenam di bawah cakrawala barat.

Dengan adanya pengamat di permukaan Bumi, target seperti Matahari atau Bulan atau planet, dan Time objek awal dan akhir, ini akan mengembalikan dua larik yang memiliki panjang yang sama. Yang pertama adalah Timedaftar saat target terbenam. Larik kedua berisi Truesetiap kali target benar-benar melintasi cakrawala, dan Falsesaat target hanya melintas tanpa benar-benar menyentuh cakrawala.

Lihat Risings and settings untuk contoh, dan Computing your own angle refraction untuk cara menggunakan horizon_degreesargumen.

Baru di versi 1.47.

skyfield.almanac.find_transits(pengamat , target , waktu mulai , waktu berakhir)

Mengembalikan waktu saat target transit melintasi meridian.

Mengingat adanya pengamat di permukaan Bumi, suatu target seperti Matahari atau Bulan atau sebuah planet, dan Time objek awal dan akhir, ini mengembalikan suatu Timesusunan yang mencantumkan momen saat target transit melintasi meridian.

Lihat Meridian Transits untuk contoh kode.

Baru di versi 1.47.

skyfield.almanac.find_discrete(waktu_mulai , waktu_akhir , f , epsilon=1.157407407407407e-08 , num=12)

Temukan waktu di mana fungsi waktu diskrit berubah nilai.

Rutin ini digunakan untuk menemukan peristiwa sesaat seperti matahari terbit, transit, dan musim. Lihat Mencari tanggal peristiwa astronomi untuk mengetahui cara menggunakannya sendiri.

skyfield.almanac.seasons(bahasa gaul yang mudah diingat)

Bangun fungsi waktu yang mengembalikan kuartal dalam setahun.

The function that this returns will expect a single argument that is a Time and will return 0 through 3 for the seasons Spring, Summer, Autumn, and Winter.

skyfield.almanac.moon_phase(ephemeris, t)

Return the Moon phase 0°–360° at time t, where 180° is Full Moon.

More precisely: this returns an Angle giving the difference between the geocentric apparent ecliptic longitudes of the Moon and Sun, constrained to the interval 0°–360° (0–τ radians) where 0° is New Moon and 180° is Full Moon.

skyfield.almanac.moon_phases(ephemeris)

Build a function of time that returns the moon phase 0 through 3.

The function that this returns will expect a single argument that is a Time and will return the phase of the moon as an integer. See the accompanying array MOON_PHASES if you want to give string names to each phase.

skyfield.almanac.moon_nodes(ephemeris)

Build a function of time that identifies lunar nodes.

This returns a function taking a Time and returning True if the Moon is above the ecliptic else False. See Lunar Nodes for how to use this routine.

skyfield.almanac.oppositions_conjunctions(ephemeris, target)

Build a function to find oppositions and conjunctions with the Sun.

See Opposition and Conjunction for how to call this routine and interpret the results.

skyfield.almanac.meridian_transits(ephemeris, target, topos)

Build a function of time for finding when a body transits the meridian.

The returned function accepts a Time argument and returns True if the target body is west of the observer's meridian at that time, and otherwise returns False. See Meridian Transits for how to use this to search for a body's meridian transits and antimeridian transits.

skyfield.almanac.sunrise_sunset(ephemeris, topos)

Build a function of time that returns whether the Sun is up.

The function that is returned will expect a single argument that is a Time, and will return True if the sun is up, else False.

Skyfield uses the same definition as the United States Naval Observatory: the Sun is up when its center is 0.8333 degrees below the horizon, which accounts for both its apparent radius of

around 16 arcminutes and also for the 34 arcminutes by which atmospheric refraction on average lifts the image of the Sun.

If you need to provide a custom value for refraction, adjust the estimate of the Sun's radius, or account for a vantage point above the Earth's surface, see Risings and settings to learn about the more versatile risings_and_settings() routine.

skyfield.almanac.dark_twilight_day(ephemeris, topos)

Build a function of time returning whether it is dark, twilight, or day.

Fungsi yang dikembalikan ini akan mengharapkan satu argumen yaitu Timedan akan mengembalikan:

0 — Gelapnya malam.

1 — Senja astronomi.

2 — Senja bahari.

3 — Senja sipil.

4 — Matahari terbit.

skyfield.almanac.risings_and_settings(ephemeris , target , topos , derajat_horison=-0.5666666666666667 , derajat_radius=0)

Bangun fungsi waktu yang mengembalikan apakah suatu benda sedang naik.

Ini mengembalikan fungsi yang mengambil Time argumen yang mengembalikan Truejika sudut ketinggian altazimuth benda ditambah radius_degreeslebih besar dari horizon_degrees, jika tidak False. Lihat Risings dan pengaturan untuk mempelajari tentang cara mencari risings dan pengaturan, dan untuk melihat lebih lanjut tentang penggunaan parameter horizon_degreesdan radius_degrees.

skyfield.eclipselib.lunar_eclipses(waktu_mulai , waktu_berakhir , eph)

Kembalikan gerhana bulan antara start_timedan end_time.

Mengembalikan tupel tiga item:

- A Timememberikan tanggal masing-masing gerhana.
- Serangkaian kode integer yang mengidentifikasi seberapa lengkap setiap gerhana.
- Kamus berisi rincian tambahan tentang setiap gerhana.

Rutin ini diadaptasi dari Explanatory Supplement to the Astronomical Almanac 11.2.3. Lihat Lunar eclipses untuk rincian tentang cara memanggil fungsi ini.

skyfield.almanac.phase_angle(ephemeris , tubuh , t)

Tidak digunakan lagi sejak versi 1.42: Gunakan phase_angle()metode posisi sebagai gantinya.

skyfield.almanac.fraction_illuminated(ephemeris , tubuh , t)

Tidak digunakan lagi sejak versi 1.42: Gunakan fraction_illuminated() metode posisi sebagai gantinya.

Referensi API — Satelit Bumi

Lihat Satelit Bumi untuk pengenalan cara mengunduh data satelit Bumi dan menghitung posisinya dengan Skyfield.

kelas skyfield.sgp4lib.EarthSatellite(baris1 , baris2 , nama=None , ts=None)

Satelit Bumi yang dimuat dari berkas TLE dan disebarkan dengan SGP4.

Objek satelit bumi adalah fungsi vektor Skyfield, jadi Anda dapat memanggil at()metodenya untuk menghasilkan posisinya di langit atau menggunakan penjumlahan dan pengurangan untuk menggabungkannya dengan vektor lain.

Parameter satelit umumnya hanya akurat selama satu atau dua minggu sekitar *periode* parameter, tanggal saat parameter tersebut dibuat, yang tersedia sebagai atribut:

epoch

Skyfield Timeyang memberikan momen epoch yang tepat untuk parameter orbit satelit ini.

name

Nama satelit

Saat membuat satelit, gunakan argumen line1dan line2 untuk menyediakan dua baris data dari file TLE sebagai string terpisah. Opsional namememungkinkan Anda memberi nama pada satelit, yang dapat diakses nanti melalui nameatribut. tsadalah Timescaleobjek, yang digunakan untuk menghasilkan epochnilai; jika tidak disediakan, satelit akan menggunakan objek bawaan Timescale.

Jika Anda tertarik dengan detail entri katalog, parameter model SGP4 untuk satelit tertentu dapat diakses melalui modelatributnya:

model.satnum

Nomor katalog satelit NORAD unik yang diberikan dalam berkas TLE.

model.classification

Klasifikasi satelit, atau yang lainnya 'U'“Tidak Diketahui”

model.intldesg

Penunjuk internasional

model.epochyr

Tahun empat digit penuh dari momen zaman himpunan elemen ini.

model.epochdays

Pecahan hari dalam tahun momen zaman.

model.jdsatepoch

Tanggal Julian dari zaman tersebut (dihitung dari epochyrdan epochdays).

model.ndot

Turunan pertama dari gerak rata-rata (diabaikan oleh SGP4).

model.nddot

Turunan waktu kedua dari gerak rata-rata (diabaikan oleh SGP4).

model.bstar

Koefisien hambatan balistik B* dalam jari-jari bumi terbalik.

model.ephtype

Jenis Ephemeris (diabaikan oleh SGP4 karena penentuan sekarang otomatis)

model.elnum

Nomor elemen

model.inclo

Kemiringan dalam radian.

model.nodeo

Asensio rektum dari simpul menaik dalam radian.

model.ecco

Keanehan.

model.argpo

Argumen perigee dalam radian.

model.mo

Anomali rata-rata dalam radian.

model.no_kozai

Gerak rata-rata dalam radian per menit.

model.revnum

Nomor revolusi pada zaman [Revs]

metodekelas from_satrec(satrec , ts)

Bangun EarthSatellite dari objek Satrec sgp4 mentah.

Ini memungkinkan Anda menyediakan elemen orbital numerik mentah, bukan teks dari set TLE. Lihat Membangun satelit dengan model gravitasi spesifik untuk detailnya.

metodekelas from_omm(ts , elemen_dict)

Bangun EarthSatellite dari bidang teks OMM.

Berikan tsobjek skala waktu, dan kamus Python berisi nama dan nilai bidang OMM. Skala waktu digunakan untuk membuat waktu satelit .epoch.

ITRF_position_velocity_error(T)

Tidak digunakan lagi: gunakan objek bingkai TEME dan ITRS sebagai gantinya.

find_events(topos , t0 , t1 , derajat_ketinggian=0.0)

Kembalikan waktu saat satelit terbit, berpuncak, dan terbenam.

Pencarian antara t0dan t1, yang masing-masing seharusnya berupa Timeobjek Skyfield, untuk lintasan satelit ini di atas lokasi toposyang mencapai setidaknya altitude_degreesdi atas cakrawala.

Mengembalikan tuple yang elemen pertamanya adalah array dan elemen keduanya adalah array kejadian:(t, events)Time

- 0 — Satelit naik di atas altitude_degrees.
- 1 — Satelit mencapai puncaknya dan mulai turun lagi.
- 2 — Satelit jatuh di bawah altitude_degrees.

Perhatikan bahwa beberapa kulminasi berturut-turut mungkin terjadi ketika, tanpa pengaturan, satelit mencapai ketinggian puncak kedua setelah turun sebagian langit dari yang pertama.

kelas skyfield.sgp4lib.TEME

Kerangka acuan True Equator Mean Equinox spesifik satelit.

Kerangka TEME ini digunakan untuk mengukur asensio rektum dan deklinasi, dan merupakan kerangka acuan model orbit satelit Bumi SGP4. Kerangka ini agak aneh. Alih-alih mengukur asensio rektum dari titik ekuinoks vernal yang sebenarnya, kerangka ini menggunakan ekuinoks 'rata-rata' yang hanya memperhitungkan presesi tetapi tidak nutasi (ekuinoks yang sama yang digunakan untuk Greenwich Mean Sidereal Time). Hal ini membuat kerangka acuan lebih mudah dipahami oleh komputer tahun 1970-an yang pertama kali menerapkan SGP4.

Didefinisikan dalam AIAA 2006-6753 Lampiran C. Lihat Koordinat dalam kerangka referensi lain untuk panduan penggunaan kerangka referensi Skyfield seperti ini.

Referensi API — Bintang dan objek jauh lainnya

kelas skyfield.starlib.Star(ra=None , dec=None , ra_hours=None , dec_degrees=None , ra_mas_per_year=0.0 , dec_mas_per_year=0.0 , parallax_mas=0.0 , radial_km_per_s=0.0 , names=() , epoch=2451545.0)

Posisi bintang atau objek tetap lainnya di langit.

Setiap Starobjek menentukan posisi objek yang jauh. Anda harus menyediakan asensio dan deklinasi yang tepat relatif terhadap ICRS (perbaikan terbaru pada J2000). Anda dapat menentukan koordinat menggunakan jam dan derajat floating point, atau tuple yang menentukan pecahan jam dan derajat sebagai menit dan detik, atau bahkan Angleobjek Skyfield penuh (yang dapat diinisialisasi sendiri menggunakan jam, derajat, atau radian):

```
>>> barnard = Star(ra_hours=17.963471675, dec_degrees=4.69339088889)
>>> barnard = Star(ra_hours=(17, 57, 48.49), dec_degrees=(4, 41, 36.20))
>>> barnard = Star(ra=Angle(hours=17.963471675),
...                dec=Angle(degrees=4.69339088889))
```

Untuk objek yang gerak dirinya melintasi langit telah terdeteksi, Anda dapat memberikan kecepatan dalam milidetik busur (mas) per tahun, dan bahkan kecepatan paralaks dan radial jika keduanya diketahui:

```
>>> barnard = Star(ra_hours=(17, 57, 48.49803),
...                dec_degrees=(4, 41, 36.2072),
...                ra_mas_per_year=-798.71,
...                dec_mas_per_year=+10337.77,
...                parallax_mas=545.4,
...                radial_km_per_s=-110.6)
```

Lihat Bintang dan Objek Jauh untuk panduan penggunaan Starsetelah Anda membuatnya.

Referensi API — Posisi Astronomi

Lihat Posisi untuk panduan terperinci mengenai berbagai jenis posisi yang dapat dihitung oleh Skyfield, dan untuk pemilihan sistem koordinat yang dapat digunakan untuk mengekspresikannya.

Posisi umum ICRF

kelas skyfield.positionlib.ICRF(posisi_au , kecepatan_au_per_d = Tidak Ada , t = Tidak Ada , pusat = Tidak Ada , target = Tidak Ada)

Posisi dan kecepatan (*x, y, z*) berorientasi pada sumbu ICRF.

Kerangka Acuan Langit Internasional (ICRF) adalah kerangka acuan permanen yang menggantikan J2000. Sumbu-sumbunya saling mendekati dalam 0,02 detik busur. Kerangka ini juga menggantikan sistem lama yang berbasis ekuinoks seperti B1900 dan B1950.

Setiap contoh kelas ini menyediakan .xyzvektor dan .velocityvektor yang menentukan koordinat (*x,y,z*) sepanjang sumbu ICRF. Waktu tertentu .tdapat ditentukan atau dapat berupa None.

tBahasa Indonesia:

Koordinat Timeposisi ini.

positionBahasa Indonesia:

Suatu Distanceobjek yang menawarkan koordinat posisi (*x,y,z*).

velocityBahasa Indonesia:

Suatu Velocityobjek yang menawarkan koordinat kecepatan (*dx/dt,dy/dt,dz/dt*).

Atribut ini akan memiliki nilai Nonejika tidak ada kecepatan yang ditentukan untuk posisi ini.

classmethod from_time_and_frame_vectors(t , frame , jarak , kecepatan)

Konstruktor: membangun posisi dari dua vektor dalam kerangka referensi.

- t— TimePosisi tersebut.
- frame— Kerangka acuan yang tercantum di Koordinat dalam kerangka acuan lainnya .
- distance— Sebuah vektor Distance(*x,y,z*) dalam kerangka yang diberikan.
- velocity— VelocityVektor ẋ,ẏ,ż dalam bingkai yang diberikan.

distance( )

Hitunglah jarak dari titik asal ke posisi ini.

Nilai yang dikembalikan adalah Distanceyang mencetak dirinya sendiri dalam satuan astronomi (au) tetapi juga menawarkan atribut au, km, dan mjika Anda ingin mengakses besarnya sebagai angka.

>>> v = ICRF([1, 1, 0])

>>> print(v.distance())

1.41421 au

speed( )

Hitunglah besarnya vektor kecepatan.

>>> v = ICRF([0, 0, 0], [1, 2, 3])

>>> print(v.speed())

3.74166 au/day

light_timeBahasa Indonesia:

Panjang vektor ini dalam hari waktu tempuh cahaya.

radec(epoch=Tidak ada)

Hitung RA ekuatorial, deklinasi, dan jarak.

Bila dipanggil tanpa parameter, ini mengembalikan asensio rektum dan deklinasi ICRF standar:

>>> from skyfield.api import load

>>> ts = load.timescale()

>>> t = ts.utc(2020, 5, 13, 10, 32)

>>> eph = load('de421.bsp')

>>> astrometric = eph['earth'].at(t).observe(eph['sun'])

>>> ra, dec, distance = astrometric.radec()

>>> print(ra, dec, sep='\n')

03h 21m 47.67s

+18deg 28' 55.3"

Sebaliknya, jika Anda menginginkan koordinat yang direferensikan ke sistem dinamis yang ditentukan oleh ekuator dan ekuinoks Bumi yang sebenarnya, berikan waktu periode tertentu.

>>> ra, dec, distance = astrometric.apparent().radec(epoch='date')

>>> print(ra, dec, sep='\n')

03h 22m 54.73s

+18deg 33' 04.5"

hadec( )

Hitung sudut jam, deklinasi, dan jarak.

Returns a tuple of two Angle objects plus the Distance to the target. The angles are the hour angle (±12 hours) east or west of your meridian along the ITRS celestial equator, and the declination (±90 degees) above or below it. This only works for positions whose center is a

geographic location; otherwise, there is no local meridian from which to measure the hour angle.

Because this declination is measured from the plane of the Earth's physical geographic equator, it will be slightly different than the declination returned by radec() if you have loaded a Polar Motion file.

The coordinates are not adjusted for atmospheric refraction near the horizon.

altaz(temperature_C=None, pressure_mbar='standard')

Compute (alt, az, distance) relative to the observer's horizon

The altitude returned is an Angle measured in degrees above the horizon, while the azimuth Angle measures east along the horizon from geographic north (so 0 degrees means north, 90 is east, 180 is south, and 270 is west).

By default, Skyfield does not adjust the altitude for atmospheric refraction. If you want Skyfield to estimate how high the atmosphere might lift the body's image, give the argument temperature_C either the temperature in degrees centigrade, or the string 'standard' (in which case 10°C is used).

When calculating refraction, Skyfield uses the observer's elevation above sea level to estimate the atmospheric pressure. If you want to override that value, simply provide a number through the pressure_mbar parameter.

separation_from(another_icrf)

Return the angle between this position and another.

```
>>> from skyfield.api import load
>>> ts = load.timescale()
>>> t = ts.utc(2020, 4, 18)
>>> eph = load('de421.bsp')
>>> sun, venus, earth = eph['sun'], eph['venus'], eph['earth']
>>> e = earth.at(t)
>>> s = e.observe(sun)
>>> v = e.observe(venus)
>>> print(s.separation_from(v))
43deg 23' 23.1"
```

You can also compute separations across an array of positions.

```
>>> t = ts.utc(2020, 4, [18, 19, 20])
>>> e = earth.at(t)
>>> print(e.observe(sun).separation_from(e.observe(venus)))
```

3 values from 43deg 23' 23.1" to 42deg 49' 46.6"

cirs_xyz(epoch)

Compute cartesian CIRS coordinates at a given epoch (*x,y,z*).

Calculate coordinates in the Celestial Intermediate Reference System (CIRS), a dynamical coordinate system referenced to the Celestial Intermediate Origin (CIO). As this is a dynamical system it must be calculated at a specific epoch.

cirs_radec(epoch)

Get spherical CIRS coordinates at a given epoch (ra, dec, distance).

Calculate coordinates in the Celestial Intermediate Reference System (CIRS), a dynamical coordinate system referenced to the Celestial Intermediate Origin (CIO). As this is a dynamical system it must be calculated at a specific epoch.

frame_xyz(frame)

Return this position as an (*x,y,z*) vector in a reference frame.

Returns a Distance object giving the (*x,y,z*) of this position in the given frame. See Coordinates in other reference frames.

frame_xyz_and_velocity(frame)

Return (*x,y,z*) position and velocity vectors in a reference frame.

Returns two vectors in the given coordinate frame: a Distance providing an (*x,y,z*) position and a Velocity giving (xdot,ydot,zdot) velocity. See Coordinates in other reference frames.

frame_latlon(frame)

Return latitude, longitude, and distance in the given frame.

Returns a 3-element tuple giving the latitude and longitude as Angle objects and the range to the target as a Distance. See Coordinates in other reference frames.

frame_latlon_and_rates(frame)

Return a reference frame latitude, longitude, range, and rates.

Return a 6-element tuple of 3 coordinates and 3 rates-of-change for this position in the given reference frame:

- Latitude Angle from +90° north to −90° south
- Longitude Angle 0°–360° east
- Radial Distance
- Latitude AngleRate
- Longitude AngleRate
- Radial Velocity

If the reference frame is the ICRS, or is J2000, or otherwise involves the celestial equator and pole, then the latitude and longitude returned will measure what are more commonly called “declination” and “right ascension”. Note that right ascension is usually expressed as hours (24 in a circle), rather than in the degrees that this routine will return.

to_skycoord(unit=None)

Convert this distance to an AstroPy SkyCoord object.

Currently, this will only work with Skyfield positions whose center is the Solar System barycenter or else the geocenter.

phase_angle(sun)

Return this position’s phase angle: the angle Sun-target-observer.

Given a Sun object (which you can build by loading an ephemeris and looking up eph['Sun']), return the Angle from the body’s point of view between light arriving from the Sun and the light departing toward the observer. This angle is 0° if the observer is in the same direction as the Sun and sees the body as fully illuminated, and 180° if the observer is behind the body and sees only its dark side.

New in version 1.42.

fraction_illuminated(sun)

Return the fraction of the target’s disc that is illuminated.

Given a Sun object (which you can build by loading an ephemeris and looking up eph['Sun']), compute what fraction from 0.0 to 1.0 of this target’s disc is illuminated, under the assumption that the target is a sphere.

New in version 1.42.

is_sunlit(ephemeris)

Return whether a position in Earth orbit is in sunlight.

Returns True or False, or an array of such values, to indicate whether this position is in sunlight or is blocked by the Earth’s shadow. It should work with positions produced either by calling at() on a satellite object, or by calling at() on the relative position sat - topos of a satellite with respect to an Earth observer’s position. See Find when a satellite is in sunlight.

is_behind_earth()

Return whether the Earth blocks the view of this object.

For a position centered on an Earth-orbiting satellite, return whether the target is in eclipse behind the disc of the Earth. See Find whether the Earth blocks a satellite’s view.

from_altaz(alt=None, az=None, alt_degrees=None, az_degrees=None, distance=<Distance 0.1 au>)

Generate an Apparent position from an altitude and azimuth.

The altitude and azimuth can each be provided as an Angle object, or else as a number of degrees provided as either a float or a tuple of degrees, arcminutes, and arcseconds:

alt=Angle(...), az=Angle(...)

alt_degrees=23.2289, az_degrees=142.1161

alt_degrees=(23, 13, 44.1), az_degrees=(142, 6, 58.1)

The distance should be a Distance object, if provided; otherwise a default of 0.1 au is used.

Position measured from the Solar System barycenter

class skyfield.positionlib.Barycentric(position_au, velocity_au_per_d=None, t=None, center=None, target=None)

An (*x,y,z*) position measured from the Solar System barycenter.

Skyfield generates a Barycentric position measured from the gravitational center of the Solar System whenever you ask a body for its location at a particular time:

>>> t = ts.utc(2003, 8, 29)

>>> mars.at(t)

<Barycentric BCRS position and velocity at date t center=0 target=499>

This class's .xyz and .velocity are (*x,y,z*) vectors in the Barycentric Celestial Reference System (BCRS), the modern replacement for J2000 coordinates measured from the Solar System Barycenter.

This class inherits the methods of is parent class ICRF as well as the orientation of its axes in space.

observe(body)

Compute the Astrometric position of a body from this location.

To compute the body's astrometric position, it is first asked for its position at the time t of this position itself. The distance to the body is then divided by the speed of light to find how long it takes its light to arrive. Finally, the light travel time is subtracted from t and the body is asked for a series of increasingly exact positions to learn where it was when it emitted the light that is now reaching this position.

>>> earth.at(t).observe(mars)

<Astrometric ICRS position and velocity at date t center=399 target=499>

Astrometric position relative to an observer

class skyfield.positionlib.Astrometric(position_au, velocity_au_per_d=None, t=None, center=None, target=None)

An astrometric (*x,y,z*) position relative to a particular observer.

The astrometric position of a body is its position relative to an observer, adjusted for light-time delay. It is the position of the body back when it emitted (or reflected) the light that is

now reaching the observer's eye or telescope. Astrometric positions are usually generated in Skyfield by calling the Barycentric method observe(), which performs the light-time correction.

Both the .xyz and .velocity are (*x,y,z*) vectors oriented along the axes of the ICRF, the modern replacement for the J2000 reference frame.

It is common to either call .radec() (with no argument) on an astrometric position to generate an *astrometric place* right ascension and declination with respect to the ICRF axes, or else to call .apparent() to generate an Apparent position.

This class inherits the methods of is parent class ICRF as well as the orientation of its axes in space.

altaz()

Compute (alt, az, distance) relative to the observer's horizon

The altitude returned is an Angle measured in degrees above the horizon, while the azimuth Angle measures east along the horizon from geographic north (so 0 degrees means north, 90 is east, 180 is south, and 270 is west).

By default, Skyfield does not adjust the altitude for atmospheric refraction. If you want Skyfield to estimate how high the atmosphere might lift the body's image, give the argument temperature_C either the temperature in degrees centigrade, or the string 'standard' (in which case 10°C is used).

When calculating refraction, Skyfield uses the observer's elevation above sea level to estimate the atmospheric pressure. If you want to override that value, simply provide a number through the pressure_mbar parameter.

apparent()

Compute an Apparent position for this body.

This applies two effects to the position that arise from relativity and shift slightly where the other body will appear in the sky: the deflection that the image will experience if its light passes close to large masses in the Solar System, and the aberration of light caused by the observer's own velocity.

>>> earth.at(t).observe(mars).apparent()

<Apparent GCRS position and velocity at date t center=399 target=499>

These transforms convert the position from the BCRS reference frame of the Solar System barycenter and to the reference frame of the observer. In the specific case of an Earth observer, the output reference frame is the GCRS.

Apparent position relative to an observer

class skyfield.positionlib.Apparent(position_au, velocity_au_per_d=None, t=None, center=None, target=None)

An apparent (*x,y,z*) position relative to a particular observer.

This class's vectors provide the position and velocity of a body relative to an observer, adjusted to predict where the body's image will really appear (hence "apparent") in the sky:

- Light-time delay, as already present in an Astrometric position.
- Deflection: gravity bends light, and thus the image of a distant object, as the light passes massive objects like Jupiter, Saturn, and the Sun. For an observer on the Earth's surface or in Earth orbit, the slight deflection by the gravity of the Earth itself is also included.
- Aberration: incoming light arrives slanted because of the observer's motion through space.

These positions are usually produced in Skyfield by calling the apparent() method of an Astrometric object.

Both the .xyz and .velocity are (*x,y,z*) vectors oriented along the axes of the ICRF, the modern replacement for the J2000 reference frame. If the observer is at the geocenter, they are more specifically GCRS coordinates. Two common coordinates that this vector can generate are:

- *Proper place:* call .radec() without arguments to compute right ascension and declination with respect to the fixed axes of the ICRF.
- *Apparent place,* the most popular option: call .radec('date') to generate right ascension and declination with respect to the equator and equinox of date.

This class inherits the methods of is parent class ICRF as well as the orientation of its axes in space.

Geocentric position relative to the Earth

class skyfield.positionlib.Geocentric(position_au, velocity_au_per_d=None, t=None, center=None, target=None)

An (*x,y,z*) position measured from the center of the Earth.

A geocentric position is the difference between the position of the Earth at a given instant and the position of a target body at the same instant, without accounting for light-travel time or the effect of relativity on the light itself.

Its .xyz and .velocity vectors have (*x,y,z*) axes that are those of the Geocentric Celestial Reference System (GCRS), an inertial system that is an update to J2000 and that does not rotate with the Earth itself.

This class inherits the methods of is parent class ICRF as well as the orientation of its axes in space.

itrf_xyz()

Deprecated; instead, call .frame_xyz(itrs). See Coordinates in other reference frames.

subpoint()

Deprecated; instead, call either iers2010.subpoint(pos) or wgs84.subpoint(pos).

Building a position from right ascension and declination

skyfield.positionlib.position_of_radec(ra_hours, dec_degrees, distance_au=2062648062470
96.38, epoch=None, t=None, center=None, target=None)

Build a position object from a right ascension and declination.

If a specific distance_au is not provided, Skyfield returns a position vector a gigaparsec in length. This puts the position at a great enough distance that it will stand at the same right ascension and declination from any viewing position in the Solar System, to very high precision (within a few hundredths of a microarcsecond).

If an epoch is specified, the input coordinates are understood to be in the dynamical system of that particular date. Otherwise, they will be assumed to be ICRS (the modern replacement for J2000).

Yang baru dalam versi 1.21: Ini menggantikan fungsi yang sudah tidak digunakan lagi position_from_radec() yang distanceargumennya tidak dirancang dengan baik.

Posisi Barysentrum Tata Surya

kelas skyfield.positionlib.SSB

Barisenter Tata Surya.

statis at( )

Kembalikan posisi Barycenter Tata Surya pada waktu t.

Referensi API — Kerangka Referensi

Modul ini skyfield.framelibmenyediakan sejumlah kerangka acuan standar, yang penggunaannya dijelaskan dalam Koordinat dalam kerangka acuan lainnya .

kelas skyfield.framelib.mean_equator_and_equinox_of_date

Kerangka koordinat ekuator dan ekuinoks rata-rata Bumi.

Kerangka ini digunakan untuk mengukur asensio dan deklinasi rektum. Kerangka ini melacak ekuator dan ekuinoks Bumi yang bergeser perlahan di langit akibat presesi, tetapi mengabaikan efek nutasi yang lebih kecil.

kelas skyfield.framelib.true_equator_and_equinox_of_date

Kerangka dinamis ekuator Bumi yang sebenarnya dan ekuinoks Bumi yang sebenarnya pada saat itu.

Kerangka ini digunakan untuk mengukur asensio dan deklinasi rektum. Tidak seperti kerangka acuan tetap J2000 dan ICRS, kerangka 'TETE' ini berputar perlahan saat presesi dan nutasi Bumi menggeser titik ekuinoks. Tidak seperti kerangka TEME acuan, kerangka ini tidak mengabaikan nutasi.

Ini diberikan sebagai kerangka acuan eksplisit jika Anda menginginkan koordinat (*x,y,z*); jika Anda menginginkan sudut, lebih baik menggunakan metode posisi standar radec(epoch='date')karena itu akan mengembalikan satuan konvensional jam asensio-rektum, bukan derajat-bujur yang frame_latlon()akan dikembalikan.

Kerangka acuan ini menggabungkan teori terkini mengenai presesi dan nutasi Bumi dengan sedikit pergeseran antara sistem ITRS dan J2000 untuk menghasilkan asensio rektus dan deklinasi pada tanggal tertentu relatif terhadap sumbu Bumi dan ekuator rotasi.

skyfield.framelib.itrs= <objek skyfield.framelib.itrs>

Sistem Referensi Terestrial Internasional (ITRS).

Ini adalah standar IAU untuk sistem koordinat tetap Bumi yang berpusat di Bumi (ECEF), yang ditambatkan ke kerak Bumi dan benua. Kerangka acuan ini menggabungkan tiga kerangka acuan lainnya: ekuator Bumi yang sebenarnya dan ekuinoks tanggal, rotasi Bumi terhadap bintang-bintang, dan (jika Anda Timescalememiliki offset kutub yang dimuat) goyangan kutub kerak terhadap kutub rotasi Bumi.

Baru di versi 1.34.

skyfield.framelib.ecliptic_frame= <objek skyfield.framelib.ecliptic_frame>

Kerangka acuan ekliptika dan ekuinoks sebenarnya pada saat ini.

skyfield.framelib.ecliptic_J2000_frame= <objek skyfield.framelib.InertialFrame>

Kerangka acuan ekliptika dan ekuinoks sejati pada J2000.

skyfield.framelib.galactic_frame= <objek skyfield.framelib.InertialFrame>

Kerangka acuan Sistem Galaksi II.

Referensi API — Kerangka referensi planet

kelas skyfield.planetarylib.PlanetaryConstants

Manajer konstanta planet.

Anda dapat menggunakan kelas ini untuk membangun model kerja benda-benda Tata Surya dengan memuat file konstanta planet dan kernel orientasi biner. Untuk deskripsi lengkap tentang cara menggunakan ini, lihat Kerangka Acuan Planet .

read_text(berkas)

Membaca variabel bingkai dari berkas KPL/FK.

File yang sesuai biasanya memiliki ekstensi .tfatau .tpcdan akan mendefinisikan serangkaian nama dan nilai yang akan dimuat ke dalam .variableskamus objek ini.

>>> from skyfield.api import load

>>> pc = PlanetaryConstants()

>>> pc.read_text(load('moon_080317.tf'))

>>> pc.variables['FRAME_31006_NAME']

'MOON_PA_DE421'

read_binary(berkas)

Membaca deskripsi segmen biner dari file DAF/PCK.

Segmen biner terdapat dalam .bpcberkas dan memprediksi bagaimana suatu benda seperti planet atau bulan akan berorientasi pada tanggal tertentu.

build_frame_named(nama)

Diberi nama bingkai, kembalikan sebuah Frameobjek.

build_frame(integer , _segment=Tidak Ada)

Diberikan kode bilangan bulat bingkai, kembalikan sebuah Frameobjek.

build_latlon_degrees(bingkai , derajat_lintang , derajat_bujur , elevasi_m=0.0)

Membangun suatu objek yang mewakili suatu lokasi pada permukaan tubuh.

kelas skyfield.planetarylib.Frame(pusat , segmen , matriks)

Kerangka konstanta planet, untuk membangun matriks rotasi.

rotation_at(T)

Kembalikan matriks rotasi untuk bingkai ini pada waktu t.

rotation_and_rate_at(T)

Mengembalikan matriks rotasi dan laju untuk bingkai ini pada waktu t.

Matriks laju yang dikembalikan dalam satuan gerak sudut per hari.

Referensi API — Orbit Kepler

Skyfield menyertakan beberapa kode yang disumbangkan yang dapat menghitung posisi yang diprediksi untuk planet minor dan komet berdasarkan elemen orbit Kepler yang sederhana. Perhitungannya belum terlalu cepat, dan belum dapat menghitung beberapa posisi benda secara efisien menggunakan operasi array NumPy—jika Anda meminta sepuluh posisi komet, maka Skyfield harus menjalankan loop internal yang menghitung masing-masing dari sepuluh posisi secara terpisah.

Pustaka ini belum sepenuhnya didokumentasikan; lihat Kepler Orbits untuk pengenalan fitur-fitur yang sudah didukungnya.

Di sini, kami hanya mendokumentasikan rutinitas untuk memuat data, karena rutinitas tersebut paling stabil.

skyfield.data.mpc.load_mpcorb_dataframe(fobj)

Menguraikan berkas orbit Pusat Planet Minor ke dalam kerangka data Pandas.

Lihat Orbit Kepler . Format berkas MPCORB didokumentasikan di: https://minorplanetcenter.net/iau/info/MPOrbitFormat.html

skyfield.data.mpc.load_comets_dataframe(fobj)

Menguraikan berkas komet Pusat Planet Minor ke dalam kerangka data Pandas.

Ini hanya mengimpor bidang-bidang yang penting untuk menghitung orbit komet. Lihat load_comets_dataframe_slow()untuk rutinitas yang lebih lambat yang mencakup setiap bidang data komet.

Lihat Orbit Kepler . Format berkas komet didokumentasikan di: https://www.minorplanetcenter.net/iau/info/CometOrbitFormat.html

skyfield.data.mpc.load_comets_dataframe_slow(fobj)

Menguraikan berkas komet Pusat Planet Minor ke dalam kerangka data Pandas.

Rutin ini membaca setiap bidang dari berkas data komet. Lihat load_comets_dataframe()untuk rutinitas yang lebih cepat yang menghilangkan beberapa bidang komet yang lebih mahal.

Lihat Orbit Kepler . Format berkas komet didokumentasikan di: https://www.minorplanetcenter.net/iau/info/CometOrbitFormat.html

Referensi API — Elemen Orbital

skyfield.elementslib.osculating_elements_of(posisi , kerangka acuan = Tidak Ada , gm_km3_s2 = Tidak Ada)

Menghasilkan elemen orbital oskulasi untuk suatu posisi.

positionadalah contoh dari ICRF. Ini biasanya dikembalikan oleh at()metode benda Tata Surya mana pun. reference_frameadalah argumen opsional dan merupakan array numpy 3x3. Kerangka acuan secara default adalah ICRF. Kerangka acuan yang umum digunakan ditemukan di skyfield.data.spice.inertial_frames. gm_km3_s2adalah argumen float opsional yang mewakili parameter gravitasi (G*M) dalam satuan km^3/s^2, yang merupakan jumlah massa benda yang mengorbit dan benda yang sedang diorbit. Jika tidak ditentukan, ini dihitung untuk Anda.

Fungsi ini mengembalikan contohOsculatingElements

kelas skyfield.elementslib.OsculatingElements(posisi , kecepatan , waktu , mu_km_s)

Satu atau lebih set elemen orbital yang berosilasi.

Suatu OsculatingElementsobjek dapat diinisialisasi dengan parameter berikut:

posisi : Objek jarak

Vektor posisi dengan bentuk (3,) atau (3, n)

kecepatan : Kecepatan benda

Vektor kecepatan dengan bentuk (3,) atau (3, n)

waktu: Objek waktu

Waktu vektor posisi dan kecepatan

mu_km_s: mengapung

Parameter gravitasi (G*M) dalam satuan km^3/s^2

Referensi API — Unit

Saat Anda meminta posisi untuk mengembalikan jarak, kecepatan, atau sudut, mereka mengembalikan contoh kelas berikut.

kelas skyfield.units.Distance(au=Tidak Ada , km=Tidak Ada , m=Tidak Ada)

Jarak, disimpan secara internal sebagai au dan tersedia di unit lain.

Anda dapat menginisialisasi Distancedengan menyediakan satu float atau array float sebagai parameter au=, km=, atau m=.

Anda dapat mengakses besaran jarak dengan tiga atributnya .au, .km, dan .m. Secara default jarak akan dicetak dalam satuan astronomi (au), tetapi Anda dapat mengendalikan sendiri pemformatan dan pilihan satuan menggunakan pemformatan numerik Python standar:

>>> d = Distance(au=1)

>>> print(d)

1.0 au

>>> print('{:.2f} km'.format(d.km))

149597870.70 km

au(nilai)

Satuan astronomi (jarak Bumi-Matahari 149.597.870.700 m).

km(nilai)

Kilometer (1.000 meter).

m(nilai)

Meter.

length( )

Hitunglah panjangnya jika ini adalah vektor (*x, y, z*).

Panjang vektor Euklides dari vektor ini dikembalikan sebagai Distanceobjek baru.

```
>>> from skyfield.api import Distance
>>> d = Distance(au=[1, 1, 0])
>>> d.length()
<Distance 1.41421 au>
```

light_seconds( )

Kembalikan panjang vektor ini dalam detik cahaya.

to(satuan)

Ubah jarak ini ke satuan AstroPy yang diberikan.

kelas skyfield.units.Velocity(au_per_d=Tidak Ada , km_per_s=Tidak Ada)

Kecepatan, disimpan secara internal sebagai au/hari dan tersedia dalam satuan lain.

Anda dapat menginisialisasi Velocitydengan memberikan float atau array float ke au_per_d=parameternya.

au_per_d(nilai)

Satuan astronomi per hari.

km_per_s(nilai)

Kilometer per detik.

m_per_s(nilai)

Meter per detik.

to(satuan)

Ubah kecepatan ini ke satuan AstroPy yang diberikan.

kelas skyfield.units.Angle(sudut = Tidak Ada , radian = Tidak Ada , derajat = Tidak Ada , jam = Tidak Ada , preferensi = Tidak Ada , ditandatangani = Salah)

radians(nilai)

Radian (τ = 2π dalam lingkaran).

hoursBahasa Indonesia:

Jam (24 jam dalam lingkaran).

degreesBahasa Indonesia:

Derajat (360° dalam lingkaran).

arcminutes( )

Kembalikan sudut dalam menit busur.

arcseconds( )

Mengembalikan sudut dalam detik busur.

mas( )

Kembalikan sudut dalam milidetik busur.

hms(peringatan=Benar)

Ubah ke tupel (jam, menit, detik).

Ketiga besaran tersebut akan memiliki tanda yang sama dengan sudut itu sendiri.

signed_hms(peringatan=Benar)

Ubah ke tupel (tanda, jam, menit, detik).

Akan signbernilai +1 atau -1, dan besaran lainnya akan bernilai positif.

hstr(tempat=2 , peringatan=Benar , format='{0}{1:02}j {2:02}m {3:02}.{4:0{5}}d')

Mengembalikan string seperti ; lihat Memformat sudut .12h 07m 30.00s

New in version 1.39: Added the format= parameter.

dms(warn=True)

Convert to a tuple (degrees, minutes, seconds).

All three quantities will have the same sign as the angle itself.

signed_dms(warn=True)

Convert to a tuple (sign, degrees, minutes, seconds).

The sign will be either +1 or -1, and the other quantities will all be positive.

dstr(places=1, warn=True, format=None)

Return a string like 181deg 52' 30.0"; see Formatting angles.

New in version 1.39: Added the format= parameter.

to(unit)

Convert this angle to the given AstroPy unit.

class skyfield.units.AngleRate

The rate at which an angle is changing.

radians

Rate of change in radians.

degrees

Rate of change in degrees.

arcminutes

Rate of change in arcminutes.

arcseconds

Rate of change in arcseconds.

mas

Rate of change in milliarcseconds.

class skyfield.units.Rate

Measurement whose denominator is time.

per_day

Units per day of Terrestrial Time.

per_hour

Units per hour of Terrestrial Time.

per_minute

Units per minute of Terrestrial Time.

per_second

Units per second of Terrestrial Time.

Formatting angles

To display an angle as decimal degrees or hours, ask the angle for its .hours or .degrees attribute and then use any normal Python mechanism for formatting a float. For example:

ra, dec = Angle(hours=5.5877286), Angle(degrees=-5.38731536)

print('RA {:.8f} hours'.format(ra.hours))

print('Dec {:+.8f} degrees'.format(dec.degrees))

RA 5.58772860 hours

Dec -5.38731536 degrees

If you let Skyfield do the formatting instead, then hours are split into 60 minutes of 60 seconds each, and degrees are split into 60 arcminutes of 60 arcseconds each:

print('RA', ra)

print('Dec', dec)

RA 05h 35m 15.82s

Dec -05deg 23' 14.3"

If you want more control over the display of minutes and seconds, you can call an angle's "hours as a string" method hstr() or "degrees as a string" method dstr(). The simplest adjustment you can make is to specify the number of decimal places that will be shown in the seconds field.

print('RA', ra.hstr(places=4))

print('Dec', dec.dstr(places=4))

RA 05h 35m 15.8230s

Dec -05deg 23' 14.3353"

In each of these examples you can see that Skyfield marks arcminutes with the ASCII apostrophe ' and arcseconds with the ASCII quotation mark ". Using plain ASCII lets Skyfield

support as many operating systems and output media as possible. But it would be more correct to denote arcseconds and arcminutes with the Unicode symbols PRIME and DOUBLE PRIME, and to use the Unicode DEGREE SIGN to mark the whole number:

−5°23′14.3″

If you want to override Skyfield's default notation to create either the string above, or any other notation, then give hstr() or dstr() a format= string of your own. It should use the syntax of Python's str.format() method. For example, here's the exact string you would use to format an angle in degrees, arcminutes, and arcseconds using the traditional typographic symbols discussed above:

print(dec.dstr(format=u'{0}{1}°{2:02}′{3:02}.{4:0{5}}″'))

-5°23′14.3″

(Note that the leading u, for "Unicode", is only mandatory in Python 2, not Python 3.)

Skyfield will call your string's format method with these six arguments:

{0}

Tanda hubung ASCII '-'jika sudutnya negatif, selain itu string kosong. Jika Anda ingin sudut positif dihiasi dengan tanda tambah, coba gunakan {0:+>1}yang memberi tahu Python, "tampilkan parameter posisi 0, tambahkan bidang ke lebar satu karakter jika sudah kurang dari satu karakter, dan gunakan +karakter tersebut untuk melakukan penambahan."

{1}

Jumlah jam atau derajat penuh.

{2}

Jumlah menit penuh.

{3}

Jumlah detik penuh.

{4}

Pecahan detik. Pastikan untuk mengisi kolom ini dengan angka yang places=Anda minta, atau pecahan seperti .0012akan diformat secara salah sebagai .12. Jika Anda meminta places=3, misalnya, Anda akan ingin menampilkan kolom ini sebagai {4:03}. (Lihat juga item berikutnya.)

{5}

Jumlah tempat places=yang Anda minta, yang mungkin akan Anda gunakan seperti {4:0{5}} saat memformat kolom 4. Anda dapat menggunakan ini jika Anda mungkin tidak mengetahui jumlah tempat sebelumnya, misalnya jika jumlah tempat dikonfigurasikan oleh pengguna Anda.

Akan lebih baik jika format=string menjadi pilihan pertama hstr()atau dstr() nama kata kuncinya dapat dihilangkan tetapi, sayangnya, string baru ditambahkan di Skyfield 1.39, yang mana saat itu pilihan lain telah mengambil alih tempat pertama.

Referensi API — Trigonometri

skyfield.trigonometry.position_angle_of(pasangan sudut1 , pasangan sudut2)

Mengembalikan sudut posisi satu posisi terhadap posisi lainnya.

Setiap argumen harus berupa tupel yang dua item pertamanya adalah Angleobjek, seperti tupel yang dikembalikan oleh radec(), frame_latlon(), dan altaz().

Jika salah satu dari dua item sudut bertanda (jika atributnya .signed benar), maka sudut tersebut digunakan sebagai lintang dan yang lainnya sebagai bujur; jika tidak, argumen pertama diasumsikan sebagai lintang.

Jika bujur memiliki .preferenceatribut 'hours', diasumsikan sebagai asensio rekta yang positif ke arah timur. Sudut posisi yang dikembalikan akan menjadi 0 derajat jika posisi #2 berada tepat di utara posisi #1 pada bola langit, 90 derajat jika timur, 180 jika selatan, dan 270 jika barat.

Jika tidak, garis bujur diasumsikan sebagai azimuth, yang diukur dari utara ke timur di sekitar horizon. Sudut posisi yang dikembalikan akan menjadi 0 derajat jika posisi #2 berada tepat di atas posisi #1 di langit, 90 derajat ke kiri, 180 derajat jika di bawah, dan 270 derajat jika di kanan.

```
>>> from skyfield.trigonometry import position_angle_of
>>> from skyfield.units import Angle
>>> a = Angle(degrees=0), Angle(degrees=0)
>>> b = Angle(degrees=1), Angle(degrees=1)
>>> position_angle_of(a, b)
<Angle 315deg 00' 15.7">
```

Catatan Desain

Dokumen ini mengumpulkan berbagai catatan tentang desain API pustaka Skyfield, serta contoh kode yang mengilustrasikannya dan memastikannya tetap berfungsi.

Kelas posisi dengan metode koordinat

Salah satu sumber kebingungan terbesar dalam PyEphem adalah karena ia hanya menawarkan satu rutin untuk menghasilkan koordinat asensio dan deklinasi eksakta, yang argumennya memilih jenis posisi yang akan dihitung — astrometris atau semu — dan juga sistem koordinat yang akan digunakan untuk merepresentasikan posisi-posisi berbeda tersebut di langit.

Pengguna terus-menerus bingung tentang posisi mana yang mereka minta, dan jenis koordinat apa yang digunakan untuk memberi nama posisi tersebut.

Oleh karena itu Skyfield menjaga kedua konsep ini tetap terpisah, dengan mengikuti aturan berikut dalam desain API-nya:

- Posisi adalah hal yang penting. Lagi pula, dalam kehidupan nyata, dua posisi yang berbeda adalah dua tempat yang berbeda di langit. Jika di bawah langit malam Anda menunjuk dengan lengan Anda di satu posisi, dan kemudian di posisi lain yang tidak sama dengan yang pertama, Anda harus secara fisik menggerakkan lengan Anda dari yang pertama ke yang kedua.
- Koordinat bukanlah masalah besar. Koordinat hanyalah nama. Jika diberi posisi, Anda dapat menamainya dengan koordinat ICRS atau koordinat ekuinoks; dengan koordinat ekuatorial atau ekliptika atau galaksi; tetapi bahkan saat Anda melangkah melalui semua pilihan koordinat tersebut, menyebutkan angka-angka yang berbeda untuk audiens Anda, lengan Anda akan tetap berada di posisi yang sama persis. Jari Anda akan menunjuk tepat pada satu tempat di langit yang ditunjuk oleh semua koordinat yang berbeda tersebut.

Oleh karena itu, Skyfield menganggap setiap posisi cukup substansial untuk memiliki objek Python-nya sendiri. Jika Anda mengubah posisi astrometris menjadi posisi nyata, Anda akan mendapatkan objek baru untuk mewakili tempat yang berbeda di langit:

apparent = astrometric.apparent()

Namun, koordinat hanyalah nama belaka dan karenanya tidak menghasilkan objek terpisah untuk setiap pilihan koordinat. Sebaliknya, setiap jenis koordinat hanyalah pemanggilan metode yang berbeda pada posisi yang telah Anda buat:

astrometric.radec()

astrometric.ecliptic_latlon()

astrometric.galactic_latlon()

Pendekatan ini berbeda dengan PyEphem dan satu metodenya yang menghasilkan semua kombinasi posisi dan koordinat, dan dengan AstroPy, yang menghargai koordinat yang berbeda dengan objek yang berbeda bahkan jika kedua koordinat tersebut menunjukkan

tempat yang sama persis di langit. Skyfield malah mencoba menggunakan perbedaan objek versus metode dalam Python untuk memberikan sinyal tambahan kepada amatir tentang kapan sebuah program berbicara tentang lokasi yang benar-benar berbeda di langit, dibandingkan ketika program tersebut hanya menghasilkan angka yang berbeda untuk menunjukkan satu lokasi.

Mengimpor NumPy

Code examples in the Skyfield documentation that need NumPy will always import it as np as that is the standard practice in the wider SciPy community. Examples will then use NumPy features through fully qualified names like np.array since that is how users — especially users new to the scientific Python ecosystem — should be advised to structure their own code.

However, because Skyfield code itself is presumed to always use NumPy in preference to built-in Python numerics, the hundreds of np. prefixes would add only noise. As a consequence, Skyfield's modules themselves simply do a from numpy import of any names that they need.

Skyfield strives to support old versions of both Python and of NumPy, because many users in industry and government cannot upgrade their supporting libraries whenever they want. So the unit tests in CI are run against NumPy 1.11.3 to maintain compatibility with that older version.

Rotation matrices or state transformation matrices?

Instead of keeping position and velocity in separate 3-vectors of floats, the SPICE library from the JPL concatenates them both into a single 6-vector. It can then express the transform of one reference frame into another by a single 6×6 matrix. This is clever, because a transformed velocity is the sum of both the frame's native velocity and also a contribution from the angle through which the position vector is swept. This is very cleverly captured by the 6×6 matrix; the comments in frmchg illustrate its structure:

```
-        -
|          |
|  R     0  |
|          |
|          |
|  dR        |
|  --     R  |
|  dt        |
|          |
-        -
```

The top rows say "the position is simply rotated, with no contribution from the velocity," while the bottom rows say "the velocity is rotated, then added to the position × the rate the frame is rotating."

Since an aggregate frame transform can then be constructed by simply multiplying a series of these 6×6 matrices, a temptation arises: if Skyfield frame objects adopted the same convention, they would only have to carry a single transformation matrix.

The answer is no. Skyfield does not use this technique.

To understand why, observe the waste that happens when using the above matrix: fully one-quarter of the multiplies and something like one-half the adds always create zeros. The SPICE system corrects this by using a one-off implementation of matrix multiplication zzmsxf that in fact treats the operation as smaller 3×3 operations. Its comments note:

```
-       -  -       -
|   |    | |   |    |
| R2 |  0  | |  R1 |  0  |
|   |    | |   |    |
| -----+------ | | -----+------ | =
|   |    | |   |    |
| D2 |  R2 | |  D1 |  R1 |
|   |    | |   |    |
-       -  -       -


-                 -
|         |       |
| R2*R1     |   0    |
|         |       |
| ----------------+----------- |
|         |       |
| D2*R1 + R2*D1 |  R2*R1   |
|         |       |
-                 -
```

If the cost of efficiency is the additional cost and complication of breaking down the 6×6 so as to discard one-quarter of it and do pairwise operations between the remaining three quarters, then Skyfield chooses to not perform the aggregation in the first place.

So in Skyfield let's keep the matrices R and dR/dt in the first diagram always separate. Then we can perform the exact 3×3 operations that SPICE does but without what in Skyfield would be a disaggregation step beforehand plus an aggregation step after.

Produk silang

Bagaimana kita dapat menghitung perkalian silang sambil tetap agnostik tentang apakah kedua vektor yang telah diberikan kepada kita memiliki dimensi kedua?

```
>>> from numpy import array, cross
>>> a = array([1, 2, 3])
>>> b = array([6, 4, 5])
>>> p = array([7, 8, 9])
>>> q = array([1, 0, 2])
```

Produk silang sederhana antara 3 vektor kita:

```
>>> cross(a, p, 0, 0).T
array([-6, 12, -6])
>>> cross(b, q, 0, 0).T
array([ 8, -7, -4])
```

Sekarang kita gabungkan vektor kita ke dalam tumpukan nilai di seluruh dimensi akhir matriks:

```
>>> ab = array([a, b]).T
>>> ab
array([[1, 6],
       [2, 4],
       [3, 5]])
>>> pq = array([p, q]).T
>>> pq
array([[7, 1],
       [8, 0],
       [9, 2]])
>>> cross(ab, pq, 0, 0).T
array([[-6,  8],
       [12, -7],
       [-6, -4]])
```

(Ingin tahu cara melakukan fundamental_arguments() tanpa reshape:)

```
>>> from numpy import array, cross, matrix, float64
>>> m = array([4.0,5.0,6.0])
>>> arg1 = float64(2.0)
```

```
>>> argn = array([2.0, 3.0])
>>> arg1 * m.reshape(3,)
array([ 8., 10., 12.])
>>> argn * m.reshape(3, 1)
array([[ 8., 12.],
       [10., 15.],
       [12., 18.]])
```

Daftar Pustaka

Bacaan lebih lanjut tentang konsep dan teknologi di balik Skyfield.

Surat Edaran USNO 179 oleh George H. Kaplan, Oktober 2005

https://www.usno.navy.mil/USNO/astrnomical-applications/publications/Circular_179.pdf/at_download/file

Pengantar favorit saya tentang bagaimana Persatuan Astronomi Internasional menanggapi masalah sistem J2000 lama pada tahun 1997 dan 2000 dengan mengadopsi ICRS dan model rotasi Bumi yang baru. Berkat penjelasan Kaplan, saya akhirnya memahami manuver pustaka NOVAS USNO yang tersedia secara gratis (yang dilengkapi dengan antarmuka Python!) saat menangani tanggal dan sistem koordinat, dan saya dapat melakukan perhitungan yang sama dengan benar di Skyfield. Buku ini juga memberikan pengantar yang berguna tentang ephemeris JPL yang digunakan NOVAS.

Panduan Pengguna NOVAS Versi F3.1, Maret 2011

https://aa.usno.navy.mil/downloads/novas/NOVAS_C3.1_Guide.pdf

Ada juga beberapa informasi menarik yang berguna dalam panduan United States Naval Observatory tentang penggunaan NOVAS. Panduan ini cenderung lebih praktis dan berorientasi pada pengguna daripada Circular, dan tentu saja juga menjadi panduan saya tentang konsep perpustakaan saat saya menggunakannya sebagai model untuk rutinitas astrometri internal Skyfield.

Dokumen Bacaan Wajib SPICE

https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/kernel.html https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/spk.html https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/daf.html

Saat mengimplementasikan pustaka saudara Skyfield jplephem , dokumen-dokumen ini sangat berharga karena deskripsi terperinci tentang format file biner .bsp serta deskripsi tentang bagaimana sistem SPICE menggabungkan beberapa segmen ephemeris untuk mempelajari perpindahan antara dua objek Tata Surya.

Direktori Dinamika Tata Surya JPL berisi ephemeris PDF

ftp://ssd.jpl.nasa.gov/pub/eph/planets/ioms/

Saya tidak selalu dapat menemukan dokumentasi resmi yang menjelaskan ephemeris JPL tertentu, tetapi direktori ini selalu menjadi titik awal yang baik, karena mengumpulkan beberapa PDF di satu tempat.

Rotasi Bumi — Perubahan Panjang Hari dan Plot ΔT yang menunjukkan suhu dari -2000 hingga +2500

http://astro.ukho.gov.uk/nao/lvm/

Halaman ini menawarkan tabel ΔT terbaru dari Morrison, Stephenson, Hohenkerk, dan Zawilski, yang digunakan Skyfield untuk memprediksi orientasi Bumi selama beberapa tahun

sebelum dan sesudah angka-angka yang lebih rinci yang diterbitkan dari tahun 1973 hingga saat ini oleh IERS (lihat di bawah). Halaman ini juga menyediakan tautan ke makalah akademis mereka, yang menyediakan banyak informasi tentang peristiwa sejarah — terutama gerhana — yang memberi kita bukti ke arah mana Bumi menunjuk pada abad-abad sebelumnya.

Nilai historis kesalahan jam bumi ΔT dan perhitungan gerhana oleh Morrison & Stephenson, 2004

http://adsabs.harvard.edu/full/2004JHA....35..327M

PDF Lengkap

Pembahasan terperinci tentang fakta bahwa ΔT berbentuk parabola sepanjang sejarah karena, dengan mengabaikan variasi jangka pendek, panjang hari Bumi yang semakin pendek membuat perbedaan antara waktu jam modern dan matahari terbit dan terbenam yang sebenarnya semakin lama semakin miring selama berabad-abad karena kesalahan kecil yang diperkenalkan setiap tahun secara bertahap terakumulasi. Gerhana kuno yang catatannya masih ada adalah satu-satunya sumber data kita tentang sejauh mana kesalahan telah terakumulasi setiap abad.

Halaman "Data orientasi bumi" IERS

Dulu ada halaman web Observatorium Angkatan Laut Amerika Serikat yang didedikasikan untuk orientasi Bumi, tetapi akhir-akhir ini saya mencari berkas data di sini:

http://www.iers.org/IERS/EN/DataProducts/EarthOrientationData/eop.html