ora Documentation

Release 0.8.2

Alex Samuel

May 30, 2023

Contents

CHAPTER 1

Overview

Ora is a complete implementation of dates and times for Python and NumPy.

- A concise, ergonomic API centered around physical times and localization.
- Fast and easy formatting and parsing.
- Batteries included: time zones, calendars.
- High performance, as good or better than other implementations.
- NumPy dtypes for date and time types, with ufunc/vectorized operations.

Ora also provides some advanced and experimental features, such as time and date types in multiple widths; calendars; and C++ library support.

Ora is not a drop-in replacement for *datetime* or other Python time libraries, but supports easy interoperability.

CHAPTER 2

Source

Ora is hosted on GitHub. See the installation instructions.

CHAPTER 3

Tour

```
>>> time = now()
>>> print(time)
2018-03-01T13:07:25.04988400+00:00
```

```
>>> date, daytime = time @ "America/New_York"
>>> print(date)
2018-03-01
```

3.1 Times

Time is the default type for times. An instance represents a *physical instant* independent of time zone.

```
>>> t = now()
```

The standard representation is as a UTC date and daytime.

```
>>> print(t)
2018-03-02T03:57:10.02192398+00:00
```

You may specify a time by date and daytime components. However, you *must* provide a time zone; without this, the components don't specify a physical instant.

```
>>> t = Time(2018, 3, 2, 23, 7, 15, "America/New_York")
>>> print(t)
2018-03-03T04:07:15.00000000+00:00
```

The sixth argument, seconds, may be a *float*; the previous components must be integers. The time zone may be a time zone object or name.

Since a time does not carry a time zone with it, the *repr* is always in terms of UTC components

```
>>> t
Time(2018, 3, 3, 4, 7, 15.00000000, UTC)
```

3.1.1 Time conversion

You can create a *Time* object from a variety of arguments. Remember that an instance represents a physical time, so if you specify a date and daytime representation, you must specify the time zone as well.

- Year, month, day, hour, minute, second, time zone of a local time.
- A date, daytime, and time zone.
- Another time instance, or an *aware datetime.datetime* instance.
- A NumPy *datetime64[s]*, *[ms]*, *[us]*, or *[ns]* instance, assumed UTC.
- An ISO 8601 string, or *"MIN"* or *"MAX"*.

The *std* attribute returns the time represented as closely as possible by a *datetime.datetime* instance. The instance's *tzinfo* is always explicitly set to UTC.

```
>>> time.std
datetime.datetime(2018, 7, 12, 19, 32, 39, 791202, tzinfo=datetime.timezone.utc)
```

3.1.2 Special times

A time class is equipped with special invalid and missing values.

```
>>> Time.INVALID
Time.INVALID
>>> Time.MISSING
Time.MISSING
```

The *valid* property is tre for any time that is not invalid or missing.

```
>>> now().valid
True
```

3.1.3 Class attributes

A time class provides *MIN* and *MAX* attributes, giving the earliest and latest representable times.

```
>>> Time.MIN
Time(1, 1, 1, 0, 0, 0.00000000, UTC)
>>> Time.MAX
Time(9999, 12, 31, 23, 59, 59.99999997, UTC)
```

The *RESOLUTION* attribute is the approximate smallest differnece between time values. For the *Time* class, the resolution is approximately 30 ns.

```
>>> Time.RESOLUTION
2.9802322387695312e-08
```

3.1.4 Arithmetic

Addition with times is always by seconds.

```
>>> print(t)
2018-03-03T04:07:15.00000000+00:00
>>> print(t + 10)
2018-03-03T04:07:25.00000000+00:00
```

The difference between two times is likewise the number of seconds between them.

```
>>> Time(2018, 3, 4, 4, 7, 15, UTC) - t
86400.0
```

3.1.5 Offsets

Internally, Ora represents a time as the number of "ticks" relative to a fixed epoch time. You can access the number of ticks with the *offset* property.

```
>>> print(t)
2018-03-02T12:30:00.00000000+00:00
>>> t.offset
2135927186207539200
```

Each tick is a second or a fraction of a second as given by the *DENOMINATOR* class attribute, and the offset is stored as an unsigned 64-bit integer. You may in fact perform arithmetic directly on these offsets. For example, to add sixty seconds,

```
>>> print(Time.from_offset(t.offset + 60 * t.DENOMINATOR))
2018-03-02T12:31:00.00000000+00:00
```

RESOLUTION is simply the reciprocal of *DENOMINATOR*. The default *Time* type uses ticks of about 30 ns since 0001-01-01T00:00:00+00:00.

3.1.6 Time types

In addition to *Time*, a number of time types are available, each with a different range and resolution.

Type	Size	Resolution	Approx Range (years)
SmallTime	32 bits	1 s	1970-2106
Unix32Time	32 bits	1 s	1902-2038
Unix64Time	64 bits	1 s	0001-9999
Time	64 bits	30 ns	0001-9999
NsTime	64 bits	1 ns	1677-2262
HiTime	64 bits	233 fs	1970-2106
Time128	128 bits	54 zs	0001-9999

These types differ in the epoch, denominator, and integer type used to store the offset. For example, *NsTime* stores a time as signed 64-bit integer nanoseconds since 1970-01-01 UTC midnight. This representation is often used in technical applications, and is also the representation used by NumPy's "datetime64[ns]" dtype.

Convert back and forth using the types themselves.

```
>>> t
Time(2018, 3, 2, 12, 30, 0.00000000, UTC)
>>> NsTime(t)
NsTime(2018, 3, 2, 12, 30, 0.000000000, UTC)
```

If you try to convert a time that doesn't fit, you'll get an *OverflowError*.

```
>>> time = Time(2600, 1, 1, 0, 0, 0, UTC)
>>> NsTime(time)
OverflowError: time out of range
```

Most functions that return a time object accept a *Time* argument, which allows you to specify which time class you want.

```
>>> now(Time=Time128)
Time128(2018, 3, 2, 12, 49, 21.010432000000000, UTC)
```

Note that the precision of *now()* depends on the underlying operating system's clock. For example, on MacOS, the precision is 1 µs, even if the chosen time type can represent additional precision. Keep in mind that the real-time clock on a typical computer is far less accurate than 1 µs, even with clock synchronization enabled.

3.2 Dates

Date is the default type for dates.

```
>>> d = Date(2016, 3, 15)
>>> print(d)
2016-03-15
```

3.2.1 Date parts

The components of date representations are available as attributes. These include the default representation, as well as the ordinal date and week date representations.

```
>>> d.year, d.month, d.day
(2016, Month.Mar, 15)
>>> d.year, d.ordinal
(2016, 75)
>>> d.week_year, d.week, d.weekday
(2016, 10, Weekday.Tue)
```

These components are also accessible in the *ymd* attribute, whose value can be unpacked to produce the ordinary date components.

```
>>> year, month, day = d.ymd
```

There's also a *ymdi* attribute, which contains the date parts encoded in an eight-digit decimal integer.

```
>>> d.ymdi
20160315
```

3.2.2 Date literals

Months and weekdays are both given as enumerations, respectively *Month* and *Weekday*. The enumerals have three-letter abbreviated names.

```
>>> Thu
Weekday.Thu
>>> Oct
Month.Oct
```

The month enumerals also define the *__truediv__* operator to provide this syntactic trick for writing date literals:

```
>>> 2016/Mar/15
Date(2016, Mar, 15)
```

3.2.3 Date conversion

The *Date* constructor makes an effort to convert reasonable date representations to the date type. These include,

- Instances of other Ora date classes.
- Python *datetime.date* instances.
- NumPy *datetime64[D]* instances.
- An integer between 10000000 and 99999999 is interpreted as a YMDI date.
- A three-element sequence is interpreted as a (year, month, day) triplet.
- A two-element sequence is interpreted as a (year, ordinal) pair.

For example,

```
>>> Date(Date16(2016, Mar, 15))
Date(2016, Mar, 15)
>>> Date(datetime.date(2016, 3, 15))
Date(2016, Mar, 15)
>>> Date(np.datetime64("2016-03-15", "D"))
Date(2016, Mar, 15)
>>> Date(20160315)
Date(2016, Mar, 15)
>>> Date((2016, 3, 15))
Date(2016, Mar, 15)
>>> Date([2016, 75])
Date(2016, Mar, 15)
```

Most Ora functions that take a date parameter will accept any of these.

The *std* property produces the corresponding *datetime.date* instance.

```
>>> d.std
datetime.date(2016, 3, 15)
```

3.2.4 Special dates

Each date class provides *MIN* and *MAX* attributes, giving the earliest and latest representable dates.

```
>>> print(Date.MIN, Date.MAX)
0001-01-01 9999-12-31
>>> print(Date16.MIN, Date16.MAX)
1970-01-01 2149-06-04
```

Each class also provides two special date values, distinct from all other dates.

```
>>> Date.INVALID
Date.INVALID
>>> Date.MISSING
Date.MISSING
```

The *valid* property is true for any date that is not invalid or missing.

```
>>> Date(2016, 3, 15).valid
True
>>> Date.MISSING.valid
False
```

3.2.5 Arithemtic

Adding or subtracting from a date shifts the date forward or backward by entire days.

```
>>> print(d + 10)
2016-03-25
>>> print(d - 10)
2016-03-05
```

The difference between two dates is the number of days between them.

```
>>> d - 2016/Jan/1
74
```

Note that this value is one smaller than the date's ordinal, 75, since the ordinal is one-indexed.

3.2.6 Today

The *today()* function returns the current date (based on the system clock) in a specified time zone. *The time zone must be specified*, since at any instant there are always two parts of the earth that are on different dates.

This code was evaluated at approximately 23:00 New York time.

```
>>> today("US/Eastern")
Date(2016, Mar, 15)
>>> today(UTC)
Date(2016, Mar, 16)
```

3.2.7 Other date types

The *Date16* class is similar to *Date*, but uses a 16-bit integer internally, and therefore has a narrower range of dates it can represent.

```
>>> Date16.MIN
Date16(1970, Jan, 1)
>>> Date16.MAX
Date16(2149, Jun, 4)
```

Convert back and forth using the types themselves.

```
>>> d = Date(1973, 12, 3)
>>> Date16(d)
Date16(1973, Dec, 3)
```

If you try to convert a date that doesn't fit, you'll get an *OverflowError*.

```
>>> battle_of_hastings = Date(1066, Oct, 14)
>>> Date16(battle_of_hastings)
OverflowError: date not in range
```

Most functions that return a date object accept a *Date* argument.

```
>>> today("America/New_York", Date=Date16)
Date16(2018, Mar, 1)
```

3.3 Time Zones

TimeZone represents a time zone, using data loaded from zoneinfo files in the tz database. See this link for information about the tz database and its limitations.

3.3.1 Time zone objects

The *TimeZone* type accepts any of the following:

- A time zone name.
- Another *TimeZone* instance.
- A *pytz* time zone object.

For example,

```
>>> tz = TimeZone("America/New_York")
>>> z.name
'America/New_York'
```

Ora also lets you give a time zone name almost anywhere a time zone is required.

```
>>> format_time_iso(now(), "America/New_York")
'2018-03-02T07:54:12-05:00'
```

However, using an explicit time zone object is more efficient.

Ora also provied a *UTC* symbol.

```
>>> UTC
TimeZone('UTC')
```

3.3.2 Localizing

The principle function of a time zone is to localize a time, *i.e.* to convert a time to a date and daytime, or vice versa. The *to_local()* and *from_local()* functions do this, as well as the @ operator (*__matrix_multiply__*) operator. See *Localization*.

3.3.3 Display time zone

Ora stores a "display time zone", which allows you to choose a default time zone for formatting.

```
>>> get_display_time_zone()
TimeZone('America/New_York')
```

The display time zone is initialized based on the *TZ* environment variable, but you may change it with *set_display_time_zone()*. You may also use the string *"display"* to indicate the display time zone in any function that requires a time zone.

```
>>> format_time_iso(now(), "display")
'2018-03-02T07:59:49-05:00'
```

3.3.4 System time zone

Ora also attempts to determine the "system time zone", configured by the host system. If a system time zone cannot be determined, Ora uses UTC.

Use *get_system_time_zone()* to retrieve this, or specify *"system"* as a time zone name. Ora cannot change the system time zone.

3.3.5 Offsets

The *at()* method returns information about a time zone's prescriptions at a given time. The returned structure can be unpacked as a sequence. The offset is the number of seconds to be added to the UTC time to produce the local time.

```
>>> tz.at(now())
TimeZoneParts(offset=-14400, abbreviation='EDT', is_dst=True)
>>> offset, abbrev, is_dst = tz.at(now())
```

To obtain time zone information for a local time (a date and daytime *in that time zone*) instead, use *at_local()*.

```
>>> tz.at_local(2016/Mar/18, MIDNIGHT)
TimeZoneParts(offset=-14400, abbreviation='EDT', is_dst=True)
```

3.3.6 Time zone data

Ora uses a method similar to the standard library's *zoneinfo* module to choose an available zoneinfo database, according to these rules:

1. The first entry of zoneinfo.TZPATH that is an absolute path to a directory, if any. By default, *TZPATH* is initialized from the *PYTHONTZPATH* environment variable, if set.

   If *PYTHONTZPATH* is unset, the path defaults to locations determined by the Python build. If you are using Python distributed with your system, this probably points to your system-supplied zoneinfo directory (often

/usr/share/zoneinfo on UNIX-like systems); use your system's package management tools to update it. If you are using Anaconda Python, this probably points to the installed location of the Conda package *tzdata* (which is distinct from the PyPI package of the same name); use *conda update* to update it.

2. Or, the zoneinfo directory installed with the PyPI tzdata package, if *tzdata* is installed.
3. Or, a copy of the zoneinfo database packaged with Ora itself.

In most Python installations, Ora uses the system-installed zoneinfo by default. To avoid using the system zoneinfo, set *PYTHONTZPATH* to an empty string. Ora will use zoneinfo from the *tzdata* package, if installed, else its own copy.

To instruct Ora to use a specific zoneinfo directory, set the *PYTHONTZPATH* environment variable to the absolute path, or call *set_zoneinfo_dir(path)*. Ora caches loaded time zone objects; any already loaded will not be reloaded if the zoneinfo directory is changed by *set_zoneinfo_dir()*.

Use *get_zoneinfo_dir()* to determine the zoneinfo version of the current zoneinfo directory, or another directory by passing a path. Note that there is no standard for storing the version of a zoneinfo directory; Ora makes an effort to infer this from conventional metadata, but this is not always possible.

3.4 Localization

The central feature of Ora is converting between a time, which is an abstract specification of a particular physical instant, and one of its local representations. A local representation is the date and daytime in a particular time zone which specifies that time.

3.4.1 Time to local

The *from_local()* function converts a (date, daytime) pair and time zone to a time.

```
>>> date = Date(1963, 11, 22)
>>> daytime = Daytime(12, 30, 0)
>>> time = from_local((date, daytime), "America/Chicago")
```

The resulting time is a physical instant, independent of time zone. If you print it, Ora shows the UTC time, as we have no commonly used time representation independent of time zone.

```
>>> time
Time(1963, 11, 22, 18, 30, 0.00000000, UTC)
```

You could, of course, construct the *Time* instance directly, rather than with a date and daytime object. However, this is just a shortcut for *from_local*.

```
>>> time = Time(1963, 11, 22, 12, 30, 0, "America/Chicago")
```

3.4.2 Local to time

The *to_local()* function is the inverse: given a time and a time zone, it computes the local date and daytime representation.

```
>>> time = Time(1963, 11, 22, 18, 30, 0.00000000, UTC)
>>> local = to_local(time, "America/Chicago")
>>> date = local.date
>>> daytime = local.daytime
```

You can unpack resulting object directly into date and daytime components.

```
>>> date, daytime = to_local(time, "America/Chicago")
```

The resulting object also provides accessors for the individual compoennts.

```
>>> to_local(time, "America/Chicago").hour
```

The *from_local* and *to_local* functions and the @ operator will accept time zone names or objects. See *Time Zones*.

```
>>> date, daytime = time @ UTC
>>> date, daytime = time @ "display"
```

DTZ is a synonym for the display time zone.

```
>>> date, daytime = time @ DTZ
```

3.4.3 @ operator

Ora the @ operator ("matrix multiplication") as a shortcut for *from_local* and *to_local*.

If the left-hand argument is a time object, @ is the same as *to_local*.

```
>>> date, daytime = time @ time_zone
```

Since Python operators are implemented on specific types, either the time must be an Ora time object, or the time zone must be an Ora time zone object. The other is converted.

```
>>> date, daytime = time @ "America/Chicago"
>>> date, daytime = "2020-01-19T08:15:00Z" @ UTC
```

The @ operator can also serve as *from_local*, with a *(date, daytime)* pair on the left and the time zone on the right.

```
>>> time_zone = TimeZone("America/Chicago")
>>> time = (date, daytime) @ time_zone
```

You can use the result of *to_local* on the left, for conversion from date, daytime in one time zone to date, daytime in another time zone.

```
>>> date, daytime = (date, daytime) @ z0 @ z1
```

The converts a date and daytime in zone *z0* to a (location-independent) time, then converts this to a date and daytime in zone *z1*.

3.5 Formatting

Ora supports formatting using format specifiers that are a superset of Python's built-in strftime/strptime-style formatting mini language. (Note that locale-specific formatting codes, *%U*/*%W*, and *%Z* are not implemented yet.)

These formatting specifications are available with *format()*, *str.format()*, and "f-string" interpolation.

```
>>> format(date, "%B %d, %Y")
'July 12, 2018'
>>> "Today is {:%A}.".format(date)
```

(continues on next page)

(continued from previous page)

```
'Today is Thursday.'
>>> f"now: {time:%i}"
'now: 2018-07-12T15:36:08+00:00'
```

For times, dates, and daytimes, the default format is the ISO 8601 format, with decimial digits appropriate for the type. This is also the *str* format.

3.5.1 Codes

The following table lists formatting codes.

Code	Example	Description
%A	*Friday*	weekday name
%a	*Fri*	weekday abbreviated name
%B	*March*	the month name
%b	*Mar*	the month abbreviated name
%C	*23:59:59*	ISO 8601 daytime representation
%D	*2018-07-11*	ISO 8601 date (without hyphens, if abbreviated)
%d	*11*	the one-indexed day of the month
%E	*-05:00*	the six-character UTC offset, e.g. +00:00
%e	*Z*	the military time zone letter
%f	*012345*	6 fractional digits (truncated) of seconds
%G	*2018*	ISO 8601 week year
%g	*18*	the last two digits of the week number of the week date
%H	*23*	the 24-hour hour number
%I	*11*	the 12-hour hour number
%i	*2018-07-11T12:30:36+00:00*	ISO 8601 time (without hyphens and colons, if abbreviated)
%j	*092*	the one-indexed ordinal day of the year
%M	*08*	the minute number
%m	*07*	the one-indexed month number
%o	*-18000*	the total signed UTC offset in seconds
%p	*AM*	either “AM”, for hour < 12, or “PM”, otherwise
%S	*08*	seconds of the minute
%T	*2018-07-11T12:30:36Z*	ISO 8601 time with military time zone letter
%u	*3*	ISO 8601 weekday as a decimal number, 1 = Mon through 7 = Sun
%V	*28*	ISO 8601 week as a decimal number
%w	*5*	the weekday number, 0 = Sun through 6 = Sat
%Y	*2018*	the full year
%y	*18*	the last two digits of the year
%z	*-0500*	5-char UTC offset, e.g. +0000

Note that some time zones and UTC offsets cannot be represented with a military time zone letter. Use the numerical UTC offset wherever possible, except possibly when formatting times in UTC only.

3.5.2 Modifiers

Ora understands modifiers, which appear after % and before the code letter.

Digits

A numerical code can be preceded by the number of (integral) digits to show. For example, *%3h* uses three digits for the number of hours,

For *%S*, the number of digits may be followed by a decimal point and number of fractional digits. For example, *%02.3S* shows seconds to ms precision.

For *%i* and *%T*, a decimal point and the number specify the number of fractional digits.

```
>>> format(time, "%i")
'2018-07-12T15:28:53+00:00'
>>> format(time, "%.6i")
'2018-07-12T15:28:53.329243+00:00'
```

Padding

followed by another character sets the pad character. For example, *%#*3H* shows the number of hours with three digits, padded on the left with asterisks.

```
>>> format(time, "%#*4H %#*4M %#*4S")
'**15 **36 ***8'
```

Capitalization

For month and weekday names, ^ specifies all capital letters, and _ specifies all lower-case letters.

```
>>> format(time, "%^A")
'THURSDAY'
>>> format(time, "%_A")
'thursday'
```

Abbreviation

For ISO formats (*%i*, *%T*, *%C*, *%D*), the modifier ~ omits hyphens and colons.

```
>>> format(time, "%i")
'2018-07-12T15:28:53+00:00'
>>> format(time, "%~i")
'20180712T152853+0000'
```

For *%A* and *%B*, ~ selects the abbreviated name, like *%a* and *%b* respectively.

3.5.3 Time zones

By default, times are formatted in UTC. To use a different time zone, follow the format string with @ and a time zone name.

```
>>> format(time, "%i")
'2018-07-12T16:30:20+00:00'
>>> format(time, "%i@America/New_York")
'2018-07-12T12:30:20-04:00'
```

With *str.format()* and interpolated strings, Python allows you to specify the time zone name with another substitution.

```
>>> f"{time:%i@{time_zone}}"
'2018-07-12T12:30:20-04:00'
```

You can ommit the format code entirely and specify only a time zone, if you want the ISO 8601 format.

```
>>> format(time, "@America/New_York")
'2018-07-12T12:30:20-04:00'
```

You can also specify "display" or "system" as the time zone name; see *Display time zone* and *System time zone*. If you omit the time zone name, Ora assumes "display". So, you can format a time in the display time zone rather than UTC by appending @ to the format.

```
>>> format(time)
'2018-07-12T16:30:20+00:00'
>>> format(time, "@")
'2018-07-12T11:30:20-05:00'
>>> get_display_time_zone()
TimeZone('America/Chicago')
```

3.6 Parsing

Ora provides functions to parse times, dates, and daytimes using the same format strings.

```
>>> parse_time("%i", "2018-07-12T11:30:20-05:00")
ora.Time(2018, 7, 12, 16, 30, 20.00000000, UTC)
```

```
>>> parse_date("%B %d, %Y", "July 11, 2018")
Date(2018, Jul, 11)
```

```
>>> parse_daytime("%H:%M", "12:30")
Daytime(12, 30, 00.000000000000000)
```

A time object represents a physical time, and a date and daytime are not sufficient to specify one. To parse a time with a format that does not include a time zone or UTC offset, you must specify the time zone explicitly.

```
>>> parse_time("%DT%C", "2018-07-12T11:30:20", time_zone="America/New_York")
ora.Time(2018, 7, 12, 15, 30, 20.00000000, UTC)
```

The parse functions honor the ~ abbreviation modifier for month and weekday names and ISO formats. The other modifiers are ignored.

3.7 NumPy

Each Ora time, date, and daytime type is also a NumPy scalar, and carries a corresponding dtype. Many Ora functions are also available as ufuncs or vectorized functions, which operate efficiently over NumPy arrays.

3.7.1 Arrays

NumPy will choose the Ora dtype automatically for an array of the corresponding type.

```
>>> arr = np.array([now(), now(), now()])
>>> arr.dtype
dtype(Time)
```

You can also get the dtype from the *dtype* attribute.

```
>>> Time.dtype
dtype(Time)
```

To coerce values to a specific type, specify the dtype explicitly.

```
>>> arr = np.array(["2018-01-01", "2018-07-04", "2018-07-11"], dtype=Date)
>>> arr
array([Date(2018, Jan, 1), Date(2018, Jul, 4), Date(2018, Jul, 11)],
      dtype=Date)
```

You can use arithmetic and NumPy's built-in ufuncs.

```
>>> np.full(8, Date(2018, 1, 1)) + np.arange(8)
array([Date(2018, Jan, 1), Date(2018, Jan, 2), Date(2018, Jan, 3),
       Date(2018, Jan, 4), Date(2018, Jan, 5), Date(2018, Jan, 6),
       Date(2018, Jan, 7), Date(2018, Jan, 8)], dtype=Date)
```

3.7.2 Casting

You can cast a NumPy *datetime64* array with units "s" or smaller to an Ora time type.

```
>>> arr
array(['2000-01-01T00:00:00.000000000', '2019-05-27T18:44:26.000000000'],
      dtype='datetime64[ns]')
>>> arr.astype(Time)
array([ora.Time(2000, 1, 1, 0, 0, 0.00000000, UTC),
       ora.Time(2019, 5, 27, 18, 44, 26.00000000, UTC)], dtype=Time)
```

You can also cast in the ohter direction.

```
>>> arr
array([ora.Time(2019, 5, 27, 18, 45, 24.13385701, UTC),
       ora.Time(2019, 5, 27, 18, 45, 24.13386100, UTC),
       ora.Time(2019, 5, 27, 18, 45, 24.13386100, UTC),
       ora.Time(2019, 5, 27, 18, 45, 24.13386198, UTC)], dtype=Time)
>>> arr.astype("datetime64[ns]")
array(['2019-05-27T18:45:24.133857012', '2019-05-27T18:45:24.133861005',
       '2019-05-27T18:45:24.133861005', '2019-05-27T18:45:24.133861989'],
      dtype='datetime64[ns]')
```

When casting, Ora always assumes that the datetime64 array uses UTC.

You can likewise cast a NumPy *datetime64[D]* array to and from an Ora date dtype.

```
>>> arr = Date(2019, 5, 27) + np.arange(3)
>>> arr
array([Date(2019, May, 27), Date(2019, May, 28), Date(2019, May, 29)],
      dtype=Date)
>>> arr.astype("datetime64[D]")
array(['2019-05-27', '2019-05-28', '2019-05-29'], dtype='datetime64[D]')
```

Ora will not let you cast a *datetime64[D]* array to or from a time array, nor will it let you cast a *datetime64* with unit seconds or smaller to or from a date array. In either case, Ora will set all values in the result to *INVALID* for a date or time dtype, or *NAT* for a *datetime64* dtype.

3.7.3 Functions

The *ora.np* module contains ufuncs and NumPy-specific functions. For instance, to construct a date from components,

```
>>> year = [1988, 2001, 2018]
>>> month = [10, 12, 7]
>>> day = [23, 3, 11]
>>> arr = ora.np.date_from_ymd(year, month, day)
>>> arr
array([Date(1988, Oct, 23), Date(2001, Dec, 3), Date(2018, Jul, 11)],
      dtype=Date)
```

The inverse function returns a structured array.

```
>>> ymd = ora.np.get_ymd(arr)
>>> ymd
array([(1988, 10, 23), (2001, 12,  3), (2018,  7, 11)],
      dtype=[('year', '<i2'), ('month', 'u1'), ('day', 'u1')])
>>> ymd["year"]
array([1988, 2001, 2018], dtype=int16)
```

You can localize an array of times to a time zone.

```
>>> arr = np.array([now(), now(), now()])
>>> date, daytime = ora.np.to_local(arr, "America/New_York")
>>> date
array([Date(2018, Jul, 11), Date(2018, Jul, 11), Date(2018, Jul, 11)],
      dtype=Date)
```

3.7.4 Internals

The NumPy array stores the underlying time, date, or daytime integer offset directly, similar to NumPy's own *datetime64* dtypes.

```
>>> Date.dtype.itemsize
4
>>> Time.dtype.itemsize
8
>>> Daytime.dtype.itemsize
```

Use *ora.np.to_offset()* to obtain an array of the underlying integer offsets.

This function creates a new array containing offsets. Since the offset is the internal representation of a time, you can obtain a similar array, albeit with shared array data, using the ndarray *view()* method and the integer type corresponding to the Ora date, time, or daytime type.

3.7.5 API

This section lists the functions and ufuncs that operate on arrays with Ora dtypes. These are available in the *ora.np* module.

Functions

These functions produce NumPy arrays of Ora objects.

date_from_ordinal_date (*year*, *ordinal*)
Constructs dates from years and ordinal dates, like *Date.from_ordinal_date*.

date_from_week_date (*week_year*, *week*, *weekday*)
Constructs dates from ISO week days, like *Date.from_week_date*.

date_from_ymd (*year*, *month*, *day*)
Contructs dates from day, month, and year components, like *Date.from_ymd*.

date_from_ymdi (*ymdi*)
Constructs dates from YYYYMMDD integers, like *Date.from_ymdi*.

time_from_offset (*offset*)
Constructs times from number of ticks, like *Time.from_offset*. The duration of a tick, and the epoch time from which it's measured, depends on the Ora time type.

to_local (*time*, *time_zone*)
Converts times to local dates and daytimes in a given time zone, like *ora.to_local*. Returns a date array and a daytime array.

from_local (*date*, *daytime*, *time_zone*)
Converts local dates and daytimes to times in a given time zone, like *ora.from_local*. Returns a time array.

The functions above also accept *Time*, *Date*, and/or *Daytime* keyword arguments, to control the dtypes of the resulting arrays.

```
>>> ora.np.time_from_offset(np.arange(4), Time=Unix32Time)
array([ora.Unix32Time(1970, 1, 1, 0, 0, 0., UTC),
       ora.Unix32Time(1970, 1, 1, 0, 0, 1., UTC),
       ora.Unix32Time(1970, 1, 1, 0, 0, 2., UTC),
       ora.Unix32Time(1970, 1, 1, 0, 0, 3., UTC)], dtype=Unix32Time)
```

Ufunc-style broadcasting is applied to the arguments.

```
>>> ora.np.date_from_ymd(2019, [1, 2], [[3, 4], [5, 6]])
array([[Date(2019, Jan, 3), Date(2019, Feb, 4)],
       [Date(2019, Jan, 5), Date(2019, Feb, 6)]], dtype=Date)
```

Ufuncs

is_valid (*obj*)
Returns a boolean array indicating true where the value is valid. Works on time, date, and daytime arrays.

to_offset (*obj*)
Returns the offset (ticks) of the time, date, or daytime array. The offset dtype depends on the dtype of the argument. Each Ora type uses a specific signed or unsigned integer to represent its offset.

get_day (*date*)

get_month (*date*)

get_ordinal_date (*date*)

get_week_date (*date*)

get_weekday (*date*)

get_year (*date*)

get_ymd (*date*)

get_ymdi (*date*)

Dtypes

- ORDINAL_DATE_DTYPE <class 'numpy.dtype'>
- WEEK_DATE_DTYPE <class 'numpy.dtype'>
- YMD_DTYPE <class 'numpy.dtype'>

3.8 Calendars

A calendar is a subset of the dates in a date range. It may represent, for instance, public holidays in a jurisdiction, or dates on which a business is open.

A calendar always carries a range of dates for which it is valid. Calendar queries outside of this date range are invalid, and raise *CalendarRangeError*. Dates outside the range are not implicitly in or not in the calendar.

Calendar represents a calendar. To create one, specify the range as a *(start, stop)* pair, and an iterable of the dates that are in the calendar. The range is half-inclusive; dates on or after *start* and strictly before *stop* are in the range.

```
>>> cal_range = Date(2018, 1, 1), Date(2019, 1, 1)
>>> holidays = Calendar(cal_range, (
...     Date(2018,  1,  1),  # New Years Day
...     Date(2018,  1, 15),  # Martin Luther King Day
...     Date(2018,  5, 28),  # Memorial Day
...     Date(2018,  7,  4),  # Independence Day
...     Date(2018,  9,  3),  # Labor Day
...     Date(2018, 11, 22),  # Thanksgiving Day
...     Date(2018, 12, 25),  # Christmas Day
... ))
```

All the dates in the calendar must be in the range.

Optionally, the calendar can carry a name; use the *name* attribute (initially *None*).

3.8.1 Membership

Use the *in* operator or *contains* method to test membership, for a date or any value that can be converted to a date.

```
>>> Date(2018, 1, 15) in holidays
True
>>> holidays.contains("2018-01-15")
True
```

Remember, the calendar can only test membership for dates in its range.

```
>>> holidays.range
(Date(2018, Jan, 1), Date(2019, Jan, 1))
>>> Date(2017, 12, 31) in holidays
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ora.CalendarRangeError: date not in calendar range
```

3.8.2 Shifts

The *before* method returns the latest calendar date on or before a date.

```
>>> holidays.before(Date(2018, 2, 1))
Date(2018, Jan, 15)
```

The result is that date itself, if it is in the calendar.

```
>>> holidays.before(Date(2018, 1, 15))
Date(2018, Jan, 15)
```

Likewise, *after* returns the earliest calendar date on or after a date.

```
>>> holidays.after(Date(2018, 2, 1))
Date(2018, May, 28)
```

Given a date in the calendar, to find the previous or next calendar date, subtract or add one first.

```
>>> mlk = Date(2018, 1, 15)
>>> holidays.after(mlk + 1)
Date(2018, May, 28)
```

To shift multiple calendar days forward, use the *shift* method. Use a positiveargument to shift forward, negative for backward.

```
>>> holidays.shift(mlk, 3)
Date(2018, Sep, 3)
```

If these methods move past the calendar range, the calendar throws *CalendarRangeError*.

```
>>> holidays.shift(mlk, -3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ora.CalendarRangeError: date not in calendar range
```

3.8.3 Finding calendars

Use the *get_calendar()* function to obtain a calendar. It accepts any of these:

- "none", which contains no dates
- "all", which contains all dates
- a weekday expression such as "Mon", or "Mon,Wed-Fri"
- the name of a calendar file in the global calendar directory

Use *get_calendar_dir()* and *set_calendar_dir()* to retrieve and set global calendar directory. The initial calendar directory is provided with Ora but can be set with the *ORA_CALENDARS* environment variable.

3.8.4 Making calendars

In addition specifying to explicit dates, you can create calendars with these special functions:

make_const_calendar(range, contains) returns a calendar that contains *all* dates in its range, if *contains* is true, or *none* of them otherwise.

make_weekday_calendar(range, weekdays) returns a calendar that contains only the specified weekdays.

```
>>> cal = make_weekday_calendar(cal_range, [Mon, Wed, Fri])
```

3.8.5 Loading calendars

The *load_calendar_file* function takes a path and loads a calendar from a text file in this format:

```
START 2010-01-01
STOP 2021-01-01
2010-01-01
2010-01-18
2010-02-15
```

Blank lines are removed; as is text following each date, which may be used for comments.

```
START 2010-01-01
STOP 2021-01-01

2010-01-01 New Year's Day
2010-01-18 Birthday of Martin Luther King, Jr.
2010-02-15 Washington's Birthday
```

Use *parse_calendar* to parse lines of text directly.

3.8.6 Dumping calendars

Use *format_calendar* to produce the calendar file format. This function returns an interable of lines.

```
>>> for line in format_calendar(cal):
...     print(line)
```

To write this directly to a file, use *dump_calendar_file(cal, path)*.

3.8.7 Arithmetic

A calendar is, in a sense, a boolean mask over the dates in its range. Calendars can be combined using bitwise arithmetic.

The ~ operator returns an inverted calendar, with dates in the range *not* in the original calendar.

The &, |, and ^ operators take two calendars, and return the intersection, union, and symmetric difference, respectively. The range of the combined calendar is always the intersection (overlap) of the two ranges.

```
>>> week_cal = make_weekday_calendar(cal_range, [Mon, Tue, Wed, Thu, Fri])
>>> work_cal = week_cal & ~holidays
```

3.9 Experiments

Ora is also a testbed for a number of experiments, related to temporal programming, API design, and Python extension.

1. Tick-based integer time representations, with heavy use of C++ templates and inlining, for very high performance.

2. An API designed around physical times rather than date-time representations, with support for dates as a distinct concept.
3. Support for time and date types with multiple widths and precisions.
4. Techniques for wrapping C++ template types as Python types. Interoperability is challenging, as C++ templates do not provide virtual dispatch.
5. Batteries-included, zero dependency C++ and Python libraries, including current time zone data.
6. Support for distinct "invalid" and "missing" values. A C++ API that provides function variants that either raise an exception or return an invalid value on error.
7. Full support for both scalar and vector (NumPy) operations. [experimental]
8. Rich calendar support. [planned]

Ora does not meet all these goals! But it is fun to try.

3.10 Limitations

Similar to *datetime*, Ora uses the proleptic Gregorian calendar, for years 1-9999 only. Alternate calendars and B.C.E. dates are not provided. There is no support for leap seconds, relativistic effects, or astronomical times. However, time precision of 1 ns or smaller is supported.

3.10.1 Scope

- Requres C++14 and Python 3.6+.
- Tested on Linux and OSX. Not currently tested on Windows.
- Support for LP64 architectures (so, x86-64 but not x86) only.

CHAPTER 4

Back matter

- genindex
- search

Index