Dates

The `Dates` module provides two types for working with dates: `Date` and `DateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TimeType`. The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the `Date` type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both `Date` and `DateTime` are basically immutable `Int64` wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second [1]. The `DateTime` type is not aware of time zones (*naive*, in Python parlance), analogous to a *LocalDateTime* in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both `Date` and `DateTime` are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `0000-12-31` is the day before `0001-01-01`, and year `-0001` (yes, negative one for the year) is 2 BC/BCE, year `-0002` is 3 BC/BCE, etc.

Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00
```

```
julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

`Date` or `DateTime` parsing is accomplished by the use of format strings. Format strings work by the notion of defining *delimited* or *fixed-width* "slots" that contain a period to parse and passing the text to parse and format string to a `Date` or `DateTime` constructor, of the form `Date("2015-01-01","y-m-d")` or `DateTime("20150101","yyyymmdd")`.

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so `"y-m-d"` lets the parser know that between the first and second slots in a date string like `"2014-07-16"`, it should find the `-` character. The `y`, `m`, and `d` characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `"yyyymmdd"` would correspond to a date string like `"20140716"`. The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition `"yyyymm"` from one period character to the next.

Support for text-form month parsing is also supported through the `u` and `U` characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so `u` corresponds to "Jan", "Feb", "Mar", etc. And `U` corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions `dayname` and `monthname`, custom locales can be loaded by

passing in the `locale=>Dict{String,Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the `Date(date_string,format_string)` function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a `Dates.DateFormat`, and pass it instead of a raw format string.

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

You can also use the `dateformat""` string macro. This macro creates the `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5
           Date("2015-01-01", dateformat"y-m-d")
       end
```

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned in the number of `Day`, and `DateTime` in the number of `Millisecond`. Similarly, comparing `TimeType` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `Int64` values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
```

```
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 734562

julia> dump(dt2)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 730151

julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)

julia> dt - dt2
4411 days

julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
2000-02-01T00:00:00

julia> dt - dt2
381110400000 milliseconds
```

Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

```
julia> Dates.yearmonth(t)
(2014, 1)

julia> Dates.monthday(t)
(1, 31)

julia> Dates.yearmonthday(t)
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)
Date
```

```
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 735264

julia> t.instant
Dates.UTInstant{Day}(Day(735264))

julia> Dates.value(t)
735264
```

Query Functions

Query functions provide calendrical information about a `TimeType`. They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"

julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the `TimeType`'s year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31
```

```
julia> Dates.quarterofyear(t)
1

julia> Dates.dayofquarter(t)
31
```

The `dayname` and `monthname` methods can also take an optional `locale` keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely `dayabbr` and `monthabbr`. First the mapping is loaded into the `LOCALES` variable:

```
julia> french_months = ["janvier", "février", "mars", "avril", "mai", "juin",
                        "juillet", "août", "septembre", "octobre", "novembre", "déce

julia> french_monts_abbrev = ["janv","févr","mars","avril","mai","juin",
                              "juil","août","sept","oct","nov","déc"];

julia> french_days = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanc

julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months, french_monts_abbrev
```

The above mentioned functions can then be used to perform the queries:

```
julia> Dates.dayname(t;locale="french")
"vendredi"

julia> Dates.monthname(t;locale="french")
"janvier"

julia> Dates.monthabbr(t;locale="french")
"janv"
```

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr` will error.

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Array{String,1} at index [5]
Stacktrace:
[...]
```

TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The `Dates` module approach tries to follow the simple principle of trying to change as little as possible when doing `Period` arithmetic. This approach is also often known as *calendrical* arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the `Dates` module, it gives the result of February 28th. How does it figure that out? I like to think of the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month *first*, where we get 2014-02-29, which adjusts down to 2014-02-28, and *then* add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' *types*, not their value or positional order; this means `Year` will always be added first, then `Month`, then `Week`, etc. Hence the following *does* result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01
```

```
julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

Tricky? Perhaps. What is an innocent `Dates` user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Day(1):Date(2014,2,3)
Date("2014-01-29"):Day(1):Date("2014-02-03")

julia> collect(dr)
6-element Array{Date,1}:
 2014-01-29
 2014-01-30
 2014-01-31
 2014-02-01
 2014-02-02
 2014-02-03

julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
Date("2014-01-29"):Month(1):Date("2014-07-29")

julia> collect(dr)
7-element Array{Date,1}:
 2014-01-29
 2014-02-28
 2014-03-29
 2014-04-29
 2014-05-29
 2014-06-29
 2014-07-29
```

Adjuster Functions

As convenient as date-period arithmetic is, often the kinds of calculations needed on dates take on a *calendrical* or *temporal* nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions deal with rules relative to the calendar, like first or last

of the month, next Tuesday, or the first and third Wednesdays, etc.

The `Dates` module provides the *adjuster* API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single `TimeType` as input and return or *adjust to* the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input to the Monday of th
2014-07-14

julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last day of the input'
2014-07-31

julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the last day of the inpu
2014-09-30
```

The next two higher-order methods, `tonext`, and `toprev`, generalize working with temporal expressions by taking a `DateFunction` as first argument, along with a starting `TimeType`. A `DateFunction` is just a function, usually anonymous, that takes a single `TimeType` as input and returns a `Bool`, `true` indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday; # Returns true if the day

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience method provided fo
2014-07-15
```

This is useful with the do-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x
           # Return true on the 4th Thursday of November (Thanksgiving)
           Dates.dayofweek(x) == Dates.Thursday &&
           Dates.dayofweekofmonth(x) == 4 &&
           Dates.month(x) == Dates.November
       end
2014-11-27
```

The `Base.filter` method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to November
```

```
# Date range from January 1st, 2014 to January 1st, 2015
julia> dr = Dates.Date(2014):Day(1):Dates.Date(2015);

julia> filter(dr) do x
           Dates.dayofweek(x) == Dates.Tue &&
           Dates.April <= Dates.month(x) <= Dates.Nov &&
           Dates.dayofweekofmonth(x) == 2
       end
8-element Array{Date,1}:
 2014-04-08
 2014-05-13
 2014-06-10
 2014-07-08
 2014-08-12
 2014-09-09
 2014-10-14
 2014-11-11
```

Additional examples and tests are available in `stdlib/Dates/test/adjusters.jl`.

Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period-Real` arithmetic is available. You can extract the underlying integer with `Dates.value`.

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years

julia> y1 + y2
3 years

julia> div(y3,y2)
5
```

```
julia> y3 - y2
8 years

julia> y3 % y2
0 years

julia> div(y3,3) # mirrors integer division
3 years

julia> Dates.value(Dates.Millisecond(10))
10
```

Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`:

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)
2016-08-07T00:00:00
```

Unlike the numeric `round` method, which breaks ties toward the even number by default, the `TimeType` `round` method uses the `RoundNearestTiesUp` rounding mode. (It's difficult to guess what breaking ties to nearest "even" `TimeType` would entail.) Further details on the available `RoundingMode`s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g., `Dates.Second(30)`) divides evenly into the next largest period (in this case, `Dates.Minute(1)`). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a `DateTime` to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that `2016-07-17T12:00:00` was chosen is that it is 17,676,660 hours after `0000-01-01T00:00:00`, and 17,676,660 is divisible by 10.

As Julia `Date` and `DateTime` values are represented according to the ISO 8601 standard, `0000-01-01T00:00:00` was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of `Date` s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest `P(2)`, where `P` is a `Period` type? In some cases (specifically, when `P <: Dates.TimePeriod`) the answer is clear:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00

julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because

the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the `Dates` module.

API reference

Dates and Time Types

`Dates.Period` — Type

```
Period
Year
Month
Week
Day
Hour
Minute
Second
Millisecond
Microsecond
Nanosecond
```

`Period` types represent discrete, human representations of time.

`Dates.CompoundPeriod` — Type

```
CompoundPeriod
```

A `CompoundPeriod` is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, "a year and a day" is not a fixed number of days, but can be expressed using a `CompoundPeriod`. In fact, a `CompoundPeriod` is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a `CompoundPeriod` result.

Dates.Instant — Type

```
Instant
```

`Instant` types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

Dates.UTInstant — Type

```
UTInstant{T}
```

The `UTInstant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The `T` is a `Period` parameter that indicates the resolution or precision of the instant.

Dates.TimeType — Type

```
TimeType
```

`TimeType` types wrap `Instant` machine instances to provide human representations of the machine instant. `Time`, `DateTime` and `Date` are subtypes of `TimeType`.

Dates.DateTime — Type

```
DateTime
```

`DateTime` wraps a `UTInstant{Millisecond}` and interprets it according to the proleptic Gregorian calendar.

Dates.Date — Type

```
Date
```

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

`Dates.Time` — Type

```
Time
```

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

Dates Functions

`Dates.DateTime` — Method

```
DateTime(y, [m, d, h, mi, s, ms]) -> DateTime
```

Construct a `DateTime` type by parts. Arguments must be convertible to `Int64`.

`Dates.DateTime` — Method

```
DateTime(periods::Period...) -> DateTime
```

Construct a `DateTime` type by `Period` type parts. Arguments may be in any order. DateTime parts not provided will default to the value of `Dates.default(period)`.

`Dates.DateTime` — Method

```
DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit=10000) -> DateTim
```

Create a `DateTime` through the adjuster API. The starting point will be constructed from the provided `y, m, d...` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied).

Examples

```
julia> DateTime(dt -> Dates.second(dt) == 40, 2010, 10, 20, 10; step = Dates.Se
2010-10-20T10:00:40

julia> DateTime(dt -> Dates.hour(dt) == 20, 2010, 10, 20, 10; step = Dates.Hour
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.DateTime` — Method

```
DateTime(dt::Date) -> DateTime
```

Convert a `Date` to a `DateTime`. The hour, minute, second, and millisecond parts of the new `DateTime` are assumed to be zero.

`Dates.DateTime` — Method

```
DateTime(dt::AbstractString, format::AbstractString; locale="english") -> DateT
```

Construct a `DateTime` by parsing the `dt` date time string following the pattern given in the `format` string.

This method creates a `DateFormat` object each time it is called. If you are parsing many date time strings of the same format, consider creating a `DateFormat` object once and using that as the second argument instead.

`Dates.format` — Method

```
format(dt::TimeType, format::AbstractString; locale="english") -> AbstractStrin
```

Construct a string by using a `TimeType` object and applying the provided `format`. The following character codes can be used to construct the `format` string:

Code	Examples	Comment
`y`	6	Numeric year with a fixed width
`Y`	1996	Numeric year with a minimum width
`m`	1, 12	Numeric month with a minimum width
`u`	Jan	Month name shortened to 3-chars according to the `locale`
`U`	January	Full month name according to the `locale` keyword
`d`	1, 31	Day of the month with a minimum width
`H`	0, 23	Hour (24-hour clock) with a minimum width
`M`	0, 59	Minute with a minimum width
`S`	0, 59	Second with a minimum width
`s`	000, 500	Millisecond with a minimum width of 3
`e`	Mon, Tue	Abbreviated days of the week
`E`	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of `yyyy-mm` specifies that the code `y` should have a width of four while `m` a width of two. Codes that yield numeric digits have an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `format` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use `format`: "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format "yyyy\ymm\m".

`Dates.DateFormat` — Type

```
DateFormat(format::AbstractString, locale="english") -> DateFormat
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Code	Matches	Comment
`y`	1996, 96	Returns year of 1996, 0096
`Y`	1996, 96	Returns year of 1996, 0096. Equivalent to `y`
`m`	1, 01	Matches 1 or 2-digit months
`u`	Jan	Matches abbreviated months according to the `locale` keyword
`U`	January	Matches full month names according to the `locale` keyword
`d`	1, 01	Matches 1 or 2-digit days
`H`	00	Matches hours (24-hour clock)
`I`	00	For outputting hours with 12-hour clock
`M`	00	Matches minutes
`S`	00	Matches seconds
`s`	.500	Matches milliseconds
`e`	Mon, Tues	Matches abbreviated days of the week
`E`	Monday	Matches full name days of the week
`p`	AM	Matches AM/PM (case-insensitive)
`yyyymmdd`	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y\ym\m".

Note that 12:00AM corresponds 00:00 (midnight), and 12:00PM corresponds to 12:00 (noon). When parsing a time with a `p` specifier, any hour (either `H` or `I`) is interpreted as as a 12-hour clock, so the `I` code is mainly useful for output.

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many

times or try the `dateformat""` string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. see `@dateformat_str`.

See `DateTime` and `format` for how to use a DateFormat object to parse and write Date strings respectively.

`Dates.@dateformat_str` — Macro

```
dateformat"Y-m-d H:M:S"
```

Create a `DateFormat` object. Similar to `DateFormat("Y-m-d H:M:S")` but creates the DateFormat object once during macro expansion.

See `DateFormat` for details about format specifiers.

`Dates.DateTime` — Method

```
DateTime(dt::AbstractString, df::DateFormat) -> DateTime
```

Construct a `DateTime` by parsing the `dt` date time string following the pattern given in the `DateFormat` object. Similar to `DateTime(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted date time strings with a pre-created `DateFormat` object.

`Dates.Date` — Method

```
Date(y, [m, d]) -> Date
```

Construct a `Date` type by parts. Arguments must be convertible to `Int64`.

`Dates.Date` — Method

```
Date(period::Period...) -> Date
```

Construct a `Date` type by `Period` type parts. Arguments may be in any order. `Date` parts not provided will default to the value of `Dates.default(period)`.

`Dates.Date` — Method

```
Date(f::Function, y[, m, d]; step=Day(1), limit=10000) -> Date
```

Create a `Date` through the adjuster API. The starting point will be constructed from the provided `y, m, d` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that `f::Function` is never satisfied).

Examples

```
julia> Date(date -> Dates.week(date) == 20, 2010, 01, 01)
2010-05-17

julia> Date(date -> Dates.year(date) == 2010, 2000, 01, 01)
2010-01-01

julia> Date(date -> Dates.month(date) == 10, 2000, 01, 01; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Date` — Method

```
Date(dt::DateTime) -> Date
```

Convert a `DateTime` to a `Date`. The hour, minute, second, and millisecond parts of the `DateTime` are truncated, so only the year, month and day parts are used in construction.

`Dates.Date` — Method

```
Date(d::AbstractString, format::AbstractString; locale="english") -> Date
```

Construct a `Date` by parsing the `d` date string following the pattern given in the `format` string.

This method creates a `DateFormat` object each time it is called. If you are parsing many date strings of the same format, consider creating a `DateFormat` object once and using that as the second argument instead.

`Dates.Date` — Method

```
Date(d::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string `d` using a `DateFormat` object `df`.

`Dates.Time` — Method

```
Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a `Time` type by parts. Arguments must be convertible to `Int64`.

`Dates.Time` — Method

```
Time(period::TimePeriod...) -> Time
```

Construct a `Time` type by `Period` type parts. Arguments may be in any order. `Time` parts not provided will default to the value of `Dates.default(period)`.

`Dates.Time` — Method

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int=10000)
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1), limit::Int=1000
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h, mi, s, ms, us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

Examples

```
julia> Dates.Time(t -> Dates.minute(t) == 30, 20)
20:30:00

julia> Dates.Time(t -> Dates.minute(t) == 0, 20)
20:00:00

julia> Dates.Time(t -> Dates.hour(t) == 10, 3; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Time` — Method

```
Time(dt::DateTime) -> Time
```

Convert a `DateTime` to a `Time`. The hour, minute, second, and millisecond parts of the `DateTime` are used to create the new `Time`. Microsecond and nanoseconds are zero by default.

`Dates.now` — Method

```
now() -> DateTime
```

Return a `DateTime` corresponding to the user's system time including the system timezone locale.

`Dates.now` — Method

```
now(::Type{UTC}) -> DateTime
```

Return a `DateTime` corresponding to the user's system time as UTC/GMT.

`Base.eps` — Function

```
eps(::Type{DateTime}) -> Millisecond
eps(::Type{Date}) -> Day
eps(::Type{Time}) -> Nanosecond
eps(::TimeType) -> Period
```

Return the smallest unit value supported by the `TimeType`.

Examples

```
julia> eps(DateTime)
1 millisecond

julia> eps(Date)
1 day

julia> eps(Time)
1 nanosecond
```

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the *machine epsilon* of the floating point type `T` (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the *relative error* of `T`, it is a "dimensionless" quantity like `one`.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
```

```
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

```
eps(x::AbstractFloat)
```

Return the *unit in last place* (ulp) of `x`. This is the distance between consecutive representable floating point values at `x`. In most cases, if the distance on either side of `x` is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \operatorname{eps}(x)/2.$$

Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf
```

```
julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

Accessor Functions

`Dates.year` — Function

```
year(dt::TimeType) -> Int64
```

The year of a `Date` or `DateTime` as an `Int64`.

`Dates.month` — Function

```
month(dt::TimeType) -> Int64
```

The month of a `Date` or `DateTime` as an `Int64`.

`Dates.week` — Function

```
week(dt::TimeType) -> Int64
```

Return the ISO week date of a `Date` or `DateTime` as an `Int64`. Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, `week(Date(2005, 1, 1))` is the 53rd week of 2004.

Examples

```
julia> Dates.week(Date(1989, 6, 22))
25

julia> Dates.week(Date(2005, 1, 1))
53

julia> Dates.week(Date(2004, 12, 31))
```

```
53
```

Dates.day — Function

```
day(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

Dates.hour — Function

```
hour(dt::DateTime) -> Int64
```

The hour of day of a `DateTime` as an `Int64`.

```
hour(t::Time) -> Int64
```

The hour of a `Time` as an `Int64`.

Dates.minute — Function

```
minute(dt::DateTime) -> Int64
```

The minute of a `DateTime` as an `Int64`.

```
minute(t::Time) -> Int64
```

The minute of a `Time` as an `Int64`.

Dates.second — Function

```
second(dt::DateTime) -> Int64
```

The second of a `DateTime` as an `Int64`.

```
second(t::Time) -> Int64
```

The second of a `Time` as an `Int64`.

`Dates.millisecond` — Function

```
millisecond(dt::DateTime) -> Int64
```

The millisecond of a `DateTime` as an `Int64`.

```
millisecond(t::Time) -> Int64
```

The millisecond of a `Time` as an `Int64`.

`Dates.microsecond` — Function

```
microsecond(t::Time) -> Int64
```

The microsecond of a `Time` as an `Int64`.

`Dates.nanosecond` — Function

```
nanosecond(t::Time) -> Int64
```

The nanosecond of a `Time` as an `Int64`.

`Dates.Year` — Method

```
Year(v)
```

Construct a `Year` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Month` — Method

```
Month(v)
```

Construct a `Month` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Week` — Method

```
Week(v)
```

Construct a `Week` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Day` — Method

```
Day(v)
```

Construct a `Day` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Hour` — Method

```
Hour(dt::DateTime) -> Hour
```

The hour part of a DateTime as a `Hour`.

`Dates.Minute` — Method

```
Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a `Minute`.

Dates.Second — Method

```
Second(dt::DateTime) -> Second
```

The second part of a DateTime as a `Second`.

Dates.Millisecond — Method

```
Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a `Millisecond`.

Dates.Microsecond — Method

```
Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a `Microsecond`.

Dates.Nanosecond — Method

```
Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a `Nanosecond`.

Dates.yearmonth — Function

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a `Date` or `DateTime`.

Dates.monthday — Function

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a `Date` or `DateTime`.

`Dates.yearmonthday` — Function

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a `Date` or `DateTime`.

Query Functions

`Dates.dayname` — Function

```
dayname(dt::TimeType; locale="english") -> String
dayname(day::Integer; locale="english") -> String
```

Return the full day name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> Dates.dayname(Date("2000-01-01"))
"Saturday"

julia> Dates.dayname(4)
"Thursday"
```

`Dates.dayabbr` — Function

```
dayabbr(dt::TimeType; locale="english") -> String
dayabbr(day::Integer; locale="english") -> String
```

Return the abbreviated name corresponding to the day of the week of the `Date` or `DateTime` in

the given `locale`. Also accepts `Integer`.

Examples

```
julia> Dates.dayabbr(Date("2000-01-01"))
"Sat"

julia> Dates.dayabbr(3)
"Wed"
```

`Dates.dayofweek` — Function

```
dayofweek(dt::TimeType) -> Int64
```

Return the day of the week as an `Int64` with `1 = Monday, 2 = Tuesday, etc.`.

Examples

```
julia> Dates.dayofweek(Date("2000-01-01"))
6
```

`Dates.dayofmonth` — Function

```
dayofmonth(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`Dates.dayofweekofmonth` — Function

```
dayofweekofmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return which number it is in `dt`'s month. So if the day of the week of `dt` is Monday, then `1 = First Monday of the month, 2 = Second Monday of the month, etc.` In the range 1:5.

Examples

```
julia> Dates.dayofweekofmonth(Date("2000-02-01"))
1

julia> Dates.dayofweekofmonth(Date("2000-02-08"))
2

julia> Dates.dayofweekofmonth(Date("2000-02-15"))
3
```

`Dates.daysofweekinmonth` — Function

```
daysofweekinmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return the total number of that day of the week in `dt`'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including `dayofweekofmonth(dt) == daysofweekinmonth(dt)` in the adjuster function.

Examples

```
julia> Dates.daysofweekinmonth(Date("2005-01-01"))
5

julia> Dates.daysofweekinmonth(Date("2005-01-04"))
4
```

`Dates.monthname` — Function

```
monthname(dt::TimeType; locale="english") -> String
monthname(month::Integer, locale="english") -> String
```

Return the full name of the month of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> Dates.monthname(Date("2005-01-04"))
"January"
```

```
julia> Dates.monthname(2)
"February"
```

Dates.monthabbr — Function

```
monthabbr(dt::TimeType; locale="english") -> String
monthabbr(month::Integer, locale="english") -> String
```

Return the abbreviated month name of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> Dates.monthabbr(Date("2005-01-04"))
"Jan"

julia> monthabbr(2)
"Feb"
```

Dates.daysinmonth — Function

```
daysinmonth(dt::TimeType) -> Int
```

Return the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

Examples

```
julia> Dates.daysinmonth(Date("2000-01"))
31

julia> Dates.daysinmonth(Date("2001-02"))
28

julia> Dates.daysinmonth(Date("2000-02"))
29
```

Dates.isleapyear — Function

```
isleapyear(dt::TimeType) -> Bool
```

Return `true` if the year of `dt` is a leap year.

Examples

```
julia> Dates.isleapyear(Date("2004"))
true

julia> Dates.isleapyear(Date("2005"))
false
```

Dates.dayofyear — Function

```
dayofyear(dt::TimeType) -> Int
```

Return the day of the year for `dt` with January 1st being day 1.

Dates.daysinyear — Function

```
daysinyear(dt::TimeType) -> Int
```

Return 366 if the year of `dt` is a leap year, otherwise return 365.

Examples

```
julia> Dates.daysinyear(1999)
365

julia> Dates.daysinyear(2000)
366
```

`Dates.quarterofyear` — Function

```
quarterofyear(dt::TimeType) -> Int
```

Return the quarter that `dt` resides in. Range of value is 1:4.

`Dates.dayofquarter` — Function

```
dayofquarter(dt::TimeType) -> Int
```

Return the day of the current quarter of `dt`. Range of value is 1:92.

Adjuster Functions

`Base.trunc` — Method

```
trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

```
julia> trunc(Dates.DateTime("1996-01-01T12:30:00"), Dates.Day)
1996-01-01T00:00:00
```

`Dates.firstdayofweek` — Function

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

```
julia> Dates.firstdayofweek(DateTime("1996-01-05T12:30:00"))
```

```
1996-01-01T00:00:00
```

Dates.lastdayofweek — Function

```
lastdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Sunday of its week.

Examples

```
julia> Dates.lastdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-07T00:00:00
```

Dates.firstdayofmonth — Function

```
firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its month.

Examples

```
julia> Dates.firstdayofmonth(DateTime("1996-05-20"))
1996-05-01T00:00:00
```

Dates.lastdayofmonth — Function

```
lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its month.

Examples

```
julia> Dates.lastdayofmonth(DateTime("1996-05-20"))
1996-05-31T00:00:00
```

Dates.firstdayofyear — Function

```
firstdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its year.

Examples

```
julia> Dates.firstdayofyear(DateTime("1996-05-20"))
1996-01-01T00:00:00
```

Dates.lastdayofyear — Function

```
lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its year.

Examples

```
julia> Dates.lastdayofyear(DateTime("1996-05-20"))
1996-12-31T00:00:00
```

Dates.firstdayofquarter — Function

```
firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its quarter.

Examples

```
julia> Dates.firstdayofquarter(DateTime("1996-05-20"))
1996-04-01T00:00:00

julia> Dates.firstdayofquarter(DateTime("1996-08-20"))
1996-07-01T00:00:00
```

Dates.lastdayofquarter — Function

```
lastdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its quarter.

Examples

```
julia> Dates.lastdayofquarter(DateTime("1996-05-20"))
1996-06-30T00:00:00

julia> Dates.lastdayofquarter(DateTime("1996-08-20"))
1996-09-30T00:00:00
```

Dates.tonext — Method

```
tonext(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc.` Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

Dates.toprev — Method

```
toprev(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the previous day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the previous `dow`, allowing for no adjustment to occur.

Dates.tofirst — Function

```
tofirst(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the first `dow` of its month. Alternatively, `of=Year` will adjust to the first `dow` of the year.

`Dates.tolast` — Function

```
tolast(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the last `dow` of its month. Alternatively, `of=Year` will adjust to the last `dow` of the year.

`Dates.tonext` — Method

```
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false) -> T
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

`Dates.toprev` — Method

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false) ->
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

Periods

`Dates.Period` — Method

```
Year(v)
Month(v)
Week(v)
```

```
Day(v)
Hour(v)
Minute(v)
Second(v)
Millisecond(v)
Microsecond(v)
Nanosecond(v)
```

Construct a `Period` type with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.CompoundPeriod` — Method

```
CompoundPeriod(periods) -> CompoundPeriod
```

Construct a `CompoundPeriod` from a `Vector` of `Periods`. All `Periods` of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
25 hours

julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
-1 hour, 1 minute

julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
1 month, -2 weeks

julia> Dates.CompoundPeriod(Dates.Minute(50000))
50000 minutes
```

`Dates.value` — Function

```
Dates.value(x::Period) -> Int64
```

For a given period, return the value associated with that period. For example, `value(Millisecond(10))` returns 10 as an integer.

`Dates.default` — Function

```
default(p::Period) -> Period
```

Returns a sensible "default" value for the input Period by returning `T(1)` for Year, Month, and Day, and `T(0)` for Hour, Minute, Second, and Millisecond.

Rounding Functions

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`.

`Base.floor` — Method

```
floor(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` less than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `floor(dt, Dates.Hour)` is a shortcut for `floor(dt, Dates.Hour(1))`.

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-06T00:00:00
```

`Base.ceil` — Method

```
ceil(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Dates.Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

`Base.round` — Method

```
round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Return the `Date` or `DateTime` nearest to `dt` at resolution `p`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, `p` may be a type instead of a value: `round(dt, Dates.Hour)` is a shortcut for `round(dt, Dates.Hour(1))`.

```
julia> round(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

Valid rounding modes for `round(::TimeType, ::Period, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Most `Period` values can also be rounded to a specified resolution:

`Base.floor` — Method

```
floor(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` down to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `floor(x, Dates.Hour)` is a shortcut for `floor(x, Dates.Hour(1))`.

```
julia> floor(Dates.Day(16), Dates.Week)
2 weeks

julia> floor(Dates.Minute(44), Dates.Minute(15))
30 minutes

julia> floor(Dates.Hour(36), Dates.Day)
1 day
```

Rounding to a `precision` of `Months` or `Years` is not supported, as these `Periods` are of inconsistent length.

`Base.ceil` — Method

```
ceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` up to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `ceil(x, Dates.Hour)` is a shortcut for `ceil(x, Dates.Hour(1))`.

```
julia> ceil(Dates.Day(16), Dates.Week)
3 weeks

julia> ceil(Dates.Minute(44), Dates.Minute(15))
45 minutes

julia> ceil(Dates.Hour(36), Dates.Day)
2 days
```

Rounding to a `precision` of `Months` or `Years` is not supported, as these `Periods` are of inconsistent length.

`Base.round` — Method

```
round(x::Period, precision::T, [r::RoundingMode]) where T <: Union{TimePeriod,
```

Round `x` to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 90 minutes to the nearest hour) will be rounded up.

For convenience, `precision` may be a type instead of a value: `round(x, Dates.Hour)` is a shortcut for `round(x, Dates.Hour(1))`.

```
julia> round(Dates.Day(16), Dates.Week)
2 weeks

julia> round(Dates.Minute(44), Dates.Minute(15))
45 minutes

julia> round(Dates.Hour(36), Dates.Day)
2 days
```

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Rounding to a `precision` of `Months` or `Years` is not supported, as these `Periods` are of inconsistent length.

The following functions are not exported:

`Dates.floorceil` — Function

```
floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the `floor` and `ceil` of a `Date` or `DateTime` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

```
floorceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> (T
```

Simultaneously return the `floor` and `ceil` of `Period` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

`Dates.epochdays2date` — Function

```
epochdays2date(days) -> Date
```

Take the number of days since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `Date`.

`Dates.epochms2datetime` — Function

```
epochms2datetime(milliseconds) -> DateTime
```

Take the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `DateTime`.

`Dates.date2epochdays` — Function

```
date2epochdays(dt::Date) -> Int64
```

Take the given `Date` and return the number of days since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

`Dates.datetime2epochms` — Function

```
datetime2epochms(dt::DateTime) -> Int64
```

Take the given `DateTime` and return the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

Conversion Functions

Dates.today — Function

```
today() -> Date
```

Return the date portion of `now()`.

Dates.unix2datetime — Function

```
unix2datetime(x) -> DateTime
```

Take the number of seconds since unix epoch `1970-01-01T00:00:00` and convert to the corresponding `DateTime`.

Dates.datetime2unix — Function

```
datetime2unix(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of seconds since the unix epoch `1970-01-01T00:00:00` as a `Float64`.

Dates.julian2datetime — Function

```
julian2datetime(julian_days) -> DateTime
```

Take the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and return the corresponding `DateTime`.

Dates.datetime2julian — Function

```
datetime2julian(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of Julian calendar days since the julian epoch `-4713-11-24T12:00:00` as a `Float64`.

`Dates.rata2datetime` — Function

```
rata2datetime(days) -> DateTime
```

Take the number of Rata Die days since epoch `0000-12-31T00:00:00` and return the corresponding `DateTime`.

`Dates.datetime2rata` — Function

```
datetime2rata(dt::TimeType) -> Int64
```

Return the number of Rata Die days since epoch from the given `Date` or `DateTime`.

Constants

Days of the Week:

Variable	Abbr.	Value (Int)
`Monday`	`Mon`	1
`Tuesday`	`Tue`	2
`Wednesday`	`Wed`	3
`Thursday`	`Thu`	4
`Friday`	`Fri`	5
`Saturday`	`Sat`	6
`Sunday`	`Sun`	7

Months of the Year:

Variable	Abbr.	Value (Int)
`January`	`Jan`	1
`February`	`Feb`	2
`March`	`Mar`	3
`April`	`Apr`	4
`May`	`May`	5
`June`	`Jun`	6
`July`	`Jul`	7
`August`	`Aug`	8
`September`	`Sep`	9
`October`	`Oct`	10
`November`	`Nov`	11
`December`	`Dec`	12

- 1 The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that `Date` and `DateTime` are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

« CRC32c Delimited Files »