orgparse Documentation

Release 0.1.2dev0

Takafumi Arakaki

Sep 18, 2019

Contents

- Documentation (Read the Docs)
- Repository (at GitHub)
- PyPI
- Travis CI

CHAPTER 1

Install

```
pip install orgparse
```

CHAPTER 2

Usage

There are pretty extensive doctests if you're interested in some specific method. Otherwise here are some example snippets:

2.1 Load org node

```
from orgparse import load, loads

load('PATH/TO/FILE.org')
load(file_like_object)

loads('''
* This is org-mode contents
  You can load org object from string.
** Second header
''')
```

2.2 Traverse org tree

```
>>> root = loads('''
... * Heading 1
... ** Heading 2
... *** Heading 3
... ''')
>>> for node in root[1:]:  # [1:] for skipping root itself
...     print(node)
* Heading 1
** Heading 2
*** Heading 3
>>> h1 = root.children[0]
```

(continues on next page)

(continued from previous page)

```
>>> h2 = h1.children[0]
>>> h3 = h2.children[0]
>>> print(h1)
* Heading 1
>>> print(h2)
** Heading 2
>>> print(h3)
*** Heading 3
>>> print(h2.get_parent())
* Heading 1
>>> print(h3.get_parent(max_level=1))
* Heading 1
```

2.3 Accessing node attributes

```
>>> root = loads('''
... * DONE Heading          :TAG:
...   CLOSED: [2012-02-26 Sun 21:15] SCHEDULED: <2012-02-26 Sun>
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] =>  0:05
...   :PROPERTIES:
...   :Effort:   1:00
...   :OtherProperty:   some text
...   :END:
...   Body texts...
... ''')
>>> node = root.children[0]
>>> node.heading
'Heading'
>>> node.scheduled
OrgDateScheduled((2012, 2, 26))
>>> node.closed
OrgDateClosed((2012, 2, 26, 21, 15, 0))
>>> node.clock
[OrgDateClock((2012, 2, 26, 21, 10, 0), (2012, 2, 26, 21, 15, 0))]
>>> bool(node.deadline)   # it is not specified
False
>>> node.tags == set(['TAG'])
True
>>> node.get_property('Effort')
60
>>> node.get_property('UndefinedProperty')  # returns None
>>> node.get_property('OtherProperty')
'some text'
>>> node.body
'  Body texts...'
```

orgparse.**load**(*path*)
Load org-mode document from a file.

Parameters **path** (*str or file-like*) – Path to org file or file-like object of a org document.

Return type *orgparse.node.OrgRootNode*

orgparse.**loads**(*string*, *filename='<string>'*)
Load org-mode document from a string.

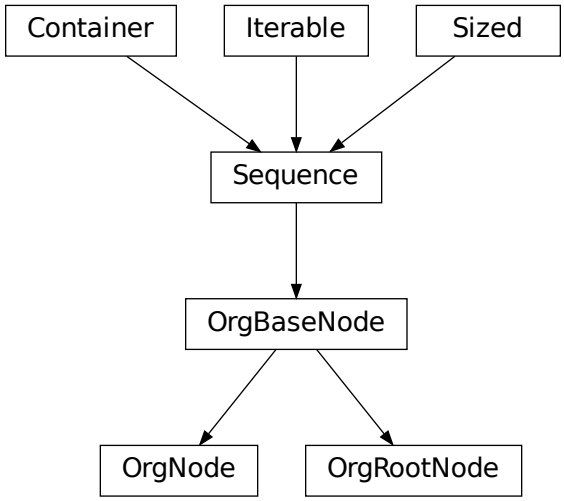
Return type *orgparse.node.OrgRootNode*

orgparse.**loadi**(*lines*, *filename='<lines>'*)
Load org-mode document from an iterative object.

Return type *orgparse.node.OrgRootNode*

CHAPTER 3

Tree structure interface



class orgparse.node.**OrgBaseNode**(*env*, *index=None*)

Base class for *OrgRootNode* and *OrgNode*

env

An instance of *OrgEnv*. All nodes in a same file shares same instance.

OrgBaseNode is an iterable object:

```
>>> from orgparse import loads
>>> root = loads('''
```

(continues on next page)

(continued from previous page)

```
... * Heading 1
... ** Heading 2
... *** Heading 3
... * Heading 4
... ''')
>>> for node in root:
...     print(node)
<BLANKLINE>
* Heading 1
** Heading 2
*** Heading 3
* Heading 4
```

Note that the first blank line is due to the root node, as iteration contains the object itself. To skip that, use slice access `[1:]`:

```
>>> for node in root[1:]:
...     print(node)
* Heading 1
** Heading 2
*** Heading 3
* Heading 4
```

It also support sequence protocol.

```
>>> print(root[1])
* Heading 1
>>> root[0] is root  # index 0 means itself
True
>>> len(root)   # remember, sequence contains itself
5
```

Note the difference between `root[1:]` and `root[1]`:

```
>>> for node in root[1]:
...     print(node)
* Heading 1
** Heading 2
*** Heading 3
```

__init__(*env*, *index=None*)

Create a *OrgBaseNode* object.

Parameters **env** (*OrgEnv*) – This will be set to the *env* attribute.

previous_same_level

Return previous node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... * Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.previous_same_level is None
True
```

(continues on next page)

(continued from previous page)

```
>>> n2.previous_same_level is n1
True
>>> n3.previous_same_level is None  # n2 is not at the same level
True
```

next_same_level

Return next node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... * Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.next_same_level is n2
True
>>> n2.next_same_level is None  # n3 is not at the same level
True
>>> n3.next_same_level is None
True
```

get_parent (*max_level=None*)

Return a parent node.

Parameters **max_level** (*int*) – In the normally structured org file, it is a level of the ancestor node to return. For example, `get_parent(max_level=0)` returns a root node.

In general case, it specify a maximum level of the desired ancestor node. If there is no ancestor node which level is equal to `max_level`, this function try to find an ancestor node which level is smaller than `max_level`.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

For simplicity, accessing *parent* is alias of calling *get_parent()* without argument.

```
>>> n1.get_parent() is n1.parent
True
>>> root.parent is None
True
```

This is a little bit pathological situation – but works.

```
>>> root = loads('''
... * Node 1
```

(continues on next page)

(continued from previous page)

```
... *** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

Now let's play with *max_level*.

```
>>> root = loads('''
... * Node 1 (level 1)
... ** Node 2 (level 2)
... *** Node 3 (level 3)
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n3.get_parent() is n2
True
>>> n3.get_parent(max_level=2) is n2  # same as default
True
>>> n3.get_parent(max_level=1) is n1
True
>>> n3.get_parent(max_level=0) is root
True
```

parent
Alias of *get_parent()* (calling without argument).

children
A list of child nodes.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... *** Node 3
... ** Node 4
... ''')
>>> (n1, n2, n3, n4) = list(root[1:])
>>> (c1, c2) = n1.children
>>> c1 is n2
True
>>> c2 is n4
True
```

Note the difference to `n1[1:]`, which returns the Node 3 also.:

```
>>> (m1, m2, m3) = list(n1[1:])
>>> m2 is n3
True
```

root
The root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> n1 = root[1]
>>> n1.root is root
True
```

level
Level of this node.

Return type int

tags
Tag of this and parents node.

```
>>> from orgparse import loads
>>> n2 = loads('''
... * Node 1    :TAG1:
... ** Node 2   :TAG2:
... ''')[2]
>>> n2.tags == set(['TAG1', 'TAG2'])
True
```

shallow_tags
Tags defined for this node (don't look-up parent nodes).

```
>>> from orgparse import loads
>>> n2 = loads('''
... * Node 1    :TAG1:
... ** Node 2   :TAG2:
... ''')[2]
>>> n2.shallow_tags == set(['TAG2'])
True
```

is_root()
Return `True` when it is a root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> root.is_root()
True
>>> n1 = root[1]
>>> n1.is_root()
False
```

class orgparse.node.**OrgRootNode**(*env*, *index=None*)
Node to represent a file

See *OrgBaseNode* for other available functions.

get_parent(*max_level=None*)
Return a parent node.

Parameters **max_level** (*int*) – In the normally structured org file, it is a level of the ancestor node to return. For example, `get_parent(max_level=0)` returns a root node.

In general case, it specify a maximum level of the desired ancestor node. If there is no ancestor node which level is equal to `max_level`, this function try to find an ancestor node which level is smaller than `max_level`.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

For simplicity, accessing `parent` is alias of calling *`get_parent()`* without argument.

```
>>> n1.get_parent() is n1.parent
True
>>> root.parent is None
True
```

This is a little bit pathological situation – but works.

```
>>> root = loads('''
... * Node 1
... *** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

Now let's play with *max_level*.

```
>>> root = loads('''
... * Node 1 (level 1)
... ** Node 2 (level 2)
... *** Node 3 (level 3)
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n3.get_parent() is n2
True
>>> n3.get_parent(max_level=2) is n2  # same as default
True
>>> n3.get_parent(max_level=1) is n1
True
>>> n3.get_parent(max_level=0) is root
True
```

is_root()
Return `True` when it is a root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> root.is_root()
True
>>> n1 = root[1]
>>> n1.is_root()
False
```

class orgparse.node.**OrgNode**(**args*, ***kwds*)

Node to represent normal org node

See *OrgBaseNode* for other available functions.

get_heading(*format='plain'*)

Return a string of head text without tags and TODO keywords.

```
>>> from orgparse import loads
>>> node = loads('* TODO Node 1').children[0]
>>> node.get_heading()
'Node 1'
```

It strips off inline markup by default (`format='plain'`). You can get the original raw string by specifying `format='raw'`.

```
>>> node = loads('* [[link][Node 1]]').children[0]
>>> node.get_heading()
'Node 1'
>>> node.get_heading(format='raw')
'[[link][Node 1]]'
```

get_body(*format='plain'*)

Return a string of body text.

See also: *get_heading()*.

heading

Alias of `.get_heading(format='plain')`.

body

Alias of `.get_body(format='plain')`.

priority

Priority attribute of this node. It is None if undefined.

```
>>> from orgparse import loads
>>> (n1, n2) = loads('''
... * [#A] Node 1
... * Node 2
... ''').children
>>> n1.priority
'A'
>>> n2.priority is None
True
```

todo

A TODO keyword of this node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('* TODO Node 1')
```

(continues on next page)

(continued from previous page)

```
>>> root.children[0].todo
'TODO'
```

get_property(*key*, *val=None*)
Return property named `key` if exists or `val` otherwise.

Parameters

- **key** (*str*) – Key of property.
- **val** – Default value to return.

properties
Node properties as a dictionary.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   :PROPERTIES:
...   :SomeProperty: value
...   :END:
... ''')
>>> root.children[0].properties['SomeProperty']
'value'
```

scheduled
Return scheduled timestamp

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   SCHEDULED: <2012-02-26 Sun>
... ''')
>>> root.children[0].scheduled
OrgDateScheduled((2012, 2, 26))
```

deadline
Return deadline timestamp.

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   DEADLINE: <2012-02-26 Sun>
... ''')
>>> root.children[0].deadline
OrgDateDeadline((2012, 2, 26))
```

closed
Return timestamp of closed time.

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
```

(continues on next page)

(continued from previous page)

```
... * Node
...   CLOSED: [2012-02-26 Sun 21:15]
... ''')
>>> root.children[0].closed
OrgDateClosed((2012, 2, 26, 21, 15, 0))
```

clock
Return a list of clocked timestamps

Return type a list of a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] =>  0:05
... ''')
>>> root.children[0].clock
[OrgDateClock((2012, 2, 26, 21, 10, 0), (2012, 2, 26, 21, 15, 0))]
```

get_timestamps(*active=False*, *inactive=False*, *range=False*, *point=False*)
Return a list of timestamps in the body text.

Parameters

- **active** (*bool*) – Include active type timestamps.
- **inactive** (*bool*) – Include inactive type timestamps.
- **range** (*bool*) – Include timestamps which has end date.
- **point** (*bool*) – Include timestamps which has no end date.

Return type list of *orgparse.date.OrgDate* subclasses

Consider the following org node:

```
>>> from orgparse import loads
>>> node = loads('''
... * Node
...   CLOSED: [2012-02-26 Sun 21:15] SCHEDULED: <2012-02-26 Sun>
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] =>  0:05
...   Some inactive timestamp [2012-02-23 Thu] in body text.
...   Some active timestamp <2012-02-24 Fri> in body text.
...   Some inactive time range [2012-02-25 Sat]--[2012-02-27 Mon].
...   Some active time range <2012-02-26 Sun>--<2012-02-28 Tue>.
... ''').children[0]
```

The default flags are all off, so it does not return anything.

```
>>> node.get_timestamps()
[]
```

You can fetch appropriate timestamps using keyword arguments.

```
>>> node.get_timestamps(inactive=True, point=True)
[OrgDate((2012, 2, 23), None, False)]
>>> node.get_timestamps(active=True, point=True)
[OrgDate((2012, 2, 24))]
>>> node.get_timestamps(inactive=True, range=True)
```

(continues on next page)

(continued from previous page)

```
[OrgDate((2012, 2, 25), (2012, 2, 27), False)]
>>> node.get_timestamps(active=True, range=True)
[OrgDate((2012, 2, 26), (2012, 2, 28))]
```

This is more complex example. Only active timestamps, regardless of range/point type.

```
>>> node.get_timestamps(active=True, point=True, range=True)
[OrgDate((2012, 2, 24)), OrgDate((2012, 2, 26), (2012, 2, 28))]
```

datelist

Alias of `.get_timestamps(active=True, inactive=True, point=True)`.

Return type list of *orgparse.date.OrgDate* subclasses

```
>>> from orgparse import loads
>>> root = loads('''
... * Node with point dates <2012-02-25 Sat>
...   CLOSED: [2012-02-25 Sat 21:15]
...   Some inactive timestamp [2012-02-26 Sun] in body text.
...   Some active timestamp <2012-02-27 Mon> in body text.
... ''')
>>> root.children[0].datelist      # doctest: +NORMALIZE_WHITESPACE
[OrgDate((2012, 2, 25)),
 OrgDate((2012, 2, 26), None, False),
 OrgDate((2012, 2, 27))]
```

rangelist

Alias of `.get_timestamps(active=True, inactive=True, range=True)`.

Return type list of *orgparse.date.OrgDate* subclasses

```
>>> from orgparse import loads
>>> root = loads('''
... * Node with range dates <2012-02-25 Sat>--<2012-02-28 Tue>
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] => 0:05
...   Some inactive time range [2012-02-25 Sat]--[2012-02-27 Mon].
...   Some active time range <2012-02-26 Sun>--<2012-02-28 Tue>.
...   Some time interval <2012-02-27 Mon 11:23-12:10>.
... ''')
>>> root.children[0].rangelist     # doctest: +NORMALIZE_WHITESPACE
[OrgDate((2012, 2, 25), (2012, 2, 28)),
 OrgDate((2012, 2, 25), (2012, 2, 27), False),
 OrgDate((2012, 2, 26), (2012, 2, 28)),
 OrgDate((2012, 2, 27, 11, 23, 0), (2012, 2, 27, 12, 10, 0))]
```

has_date()

Return `True` if it has any kind of timestamp

repeated_tasks

Get repeated tasks marked DONE in a entry having repeater.

Return type list of *orgparse.date.OrgDateRepeatedTask*

```
>>> from orgparse import loads
>>> node = loads('''
... * TODO Pay the rent
...   DEADLINE: <2005-10-01 Sat +1m>
```

(continues on next page)

(continued from previous page)

```
...   - State "DONE"  from "TODO"  [2005-09-01 Thu 16:10]
...   - State "DONE"  from "TODO"  [2005-08-01 Mon 19:44]
...   - State "DONE"  from "TODO"  [2005-07-01 Fri 17:27]
... ''').children[0]
>>> node.repeated_tasks            # doctest: +NORMALIZE_WHITESPACE
[OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 8, 1, 19, 44, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 7, 1, 17, 27, 0), 'TODO', 'DONE')]
>>> node.repeated_tasks[0].before
'TODO'
>>> node.repeated_tasks[0].after
'DONE'
```

Repeated tasks in `:LOGBOOK:` can be fetched by the same code.

```
>>> node = loads('''
... * TODO Pay the rent
...   DEADLINE: <2005-10-01 Sat +1m>
...   :LOGBOOK:
...   - State "DONE"  from "TODO"  [2005-09-01 Thu 16:10]
...   - State "DONE"  from "TODO"  [2005-08-01 Mon 19:44]
...   - State "DONE"  from "TODO"  [2005-07-01 Fri 17:27]
...   :END:
... ''').children[0]
>>> node.repeated_tasks            # doctest: +NORMALIZE_WHITESPACE
[OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 8, 1, 19, 44, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 7, 1, 17, 27, 0), 'TODO', 'DONE')]
```

See: (info "(org) Repeated tasks")

class orgparse.node.**OrgEnv**(*todos=['TODO'], dones=['DONE'], filename='<undefined>'*)

Information global to the file (e.g, TODO keywords).

nodes

A list of org nodes.

```
>>> OrgEnv().nodes   # default is empty (of course)
[]
```

```
>>> from orgparse import loads
>>> loads('''
... * Heading 1
... ** Heading 2
... *** Heading 3
... ''').env.nodes      # doctest: +ELLIPSIS  +NORMALIZE_WHITESPACE
[<orgparse.node.OrgRootNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>]
```

todo_keys

TODO keywords defined for this document (file).

```
>>> env = OrgEnv()
>>> env.todo_keys
['TODO']
```

done_keys
DONE keywords defined for this document (file).

```
>>> env = OrgEnv()
>>> env.done_keys
['DONE']
```

all_todo_keys
All TODO keywords (including DONEs).

```
>>> env = OrgEnv()
>>> env.all_todo_keys
['TODO', 'DONE']
```

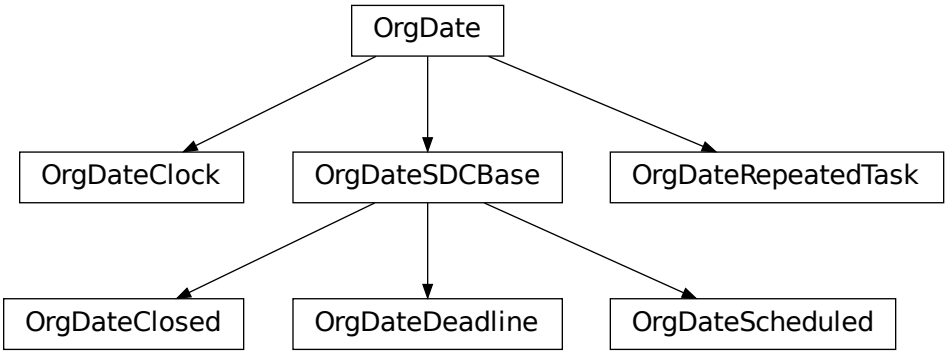
filename
Return a path to the source file or similar information.

If the org objects are not loaded from a file, this value will be a string of the form <SOME_TEXT>.

Return type str

CHAPTER 4

Date interface



class orgparse.date.**OrgDate**(*start*, *end=None*, *active=None*)

__init__(*start*, *end=None*, *active=None*)
Create *OrgDate* object

Parameters

- **start** (*datetime, date, tuple, int, float or None*) – Starting date.
- **end** (*datetime, date, tuple, int, float or None*) – Ending date.
- **active** (*bool or None*) – Active/inactive flag. None means using its default value, which may be different for different subclasses.

```
>>> OrgDate(datetime.date(2012, 2, 10))
OrgDate((2012, 2, 10))
>>> OrgDate((2012, 2, 10))
OrgDate((2012, 2, 10))
>>> OrgDate((2012, 2))  #doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
    ...
ValueError: Automatic conversion to the datetime object
requires at least 3 elements in the tuple.
Only 2 elements are in the given tuple '(2012, 2)'.
>>> OrgDate((2012, 2, 10, 12, 20, 30))
OrgDate((2012, 2, 10, 12, 20, 30))
>>> OrgDate((2012, 2, 10), (2012, 2, 15), active=False)
OrgDate((2012, 2, 10), (2012, 2, 15), False)
```

OrgDate can be created using unix timestamp:

```
>>> OrgDate(datetime.datetime.fromtimestamp(0)) == OrgDate(0)
True
```

start
Get date or datetime object

```
>>> OrgDate((2012, 2, 10)).start
datetime.date(2012, 2, 10)
>>> OrgDate((2012, 2, 10, 12, 10)).start
datetime.datetime(2012, 2, 10, 12, 10)
```

end
Get date or datetime object

```
>>> OrgDate((2012, 2, 10), (2012, 2, 15)).end
datetime.date(2012, 2, 15)
>>> OrgDate((2012, 2, 10, 12, 10), (2012, 2, 15, 12, 10)).end
datetime.datetime(2012, 2, 15, 12, 10)
```

is_active()
Return true if the date is active

has_end()
Return true if it has the end date

has_time()
Return true if the start date has time field

```
>>> OrgDate((2012, 2, 10)).has_time()
False
>>> OrgDate((2012, 2, 10, 12, 10)).has_time()
True
```

has_overlap(*other*)
Test if it has overlap with other *OrgDate* instance

If the argument is not an instance of *OrgDate*, it is converted to *OrgDate* instance by `OrgDate(other)` first.

```
>>> od = OrgDate((2012, 2, 10), (2012, 2, 15))
>>> od.has_overlap(OrgDate((2012, 2, 11)))
```

(continues on next page)

(continued from previous page)

```
True
>>> od.has_overlap(OrgDate((2012, 2, 20)))
False
>>> od.has_overlap(OrgDate((2012, 2, 11), (2012, 2, 20)))
True
>>> od.has_overlap((2012, 2, 11))
True
```

classmethod list_from_str(*string*)
Parse string and return a list of *OrgDate* objects

```
>>> OrgDate.list_from_str("... <2012-02-10 Fri> and <2012-02-12 Sun>")
[OrgDate((2012, 2, 10)), OrgDate((2012, 2, 12))]
>>> OrgDate.list_from_str("<2012-02-10 Fri>--<2012-02-12 Sun>")
[OrgDate((2012, 2, 10), (2012, 2, 12))]
>>> OrgDate.list_from_str("<2012-02-10 Fri>--[2012-02-12 Sun]")
[OrgDate((2012, 2, 10)), OrgDate((2012, 2, 12), None, False)]
>>> OrgDate.list_from_str("this is not timestamp")
[]
>>> OrgDate.list_from_str("<2012-02-11 Sat 10:11--11:20>")
[OrgDate((2012, 2, 11, 10, 11, 0), (2012, 2, 11, 11, 20, 0))]
```

classmethod from_str(*string*)
Parse string and return an *OrgDate* objects.

```
>>> OrgDate.from_str('2012-02-10 Fri')
OrgDate((2012, 2, 10))
>>> OrgDate.from_str('2012-02-10 Fri 12:05')
OrgDate((2012, 2, 10, 12, 5, 0))
```

class orgparse.date.**OrgDateScheduled**(*start*, *end=None*, *active=None*)
Date object to represent SCHEDULED attribute.

class orgparse.date.**OrgDateDeadline**(*start*, *end=None*, *active=None*)
Date object to represent DEADLINE attribute.

class orgparse.date.**OrgDateClosed**(*start*, *end=None*, *active=None*)
Date object to represent CLOSED attribute.

class orgparse.date.**OrgDateClock**(*start*, *end*, *duration=None*, *active=None*)
Date object to represent CLOCK attributes.

```
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] =>  0:30')
OrgDateClock((2010, 8, 8, 17, 0, 0), (2010, 8, 8, 17, 30, 0))
```

duration
Get duration of CLOCK.

```
>>> duration = OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:30'
... ).duration
>>> duration.seconds
1800
>>> total_minutes(duration)
30.0
```

is_duration_consistent()
Check duration value of CLOCK line.

```
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:30'
... ).is_duration_consistent()
True
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:15'
... ).is_duration_consistent()
False
```

classmethod from_str(*line*)
Get CLOCK from given string.

Return three tuple (start, stop, length) which is datetime object of start time, datetime object of stop time and length in minute.

class orgparse.date.**OrgDateRepeatedTask**(*start*, *before*, *after*, *active=None*)
Date object to represent repeated tasks.

before
The state of task before marked as done.

```
>>> od = OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE')
>>> od.before
'TODO'
```

after
The state of task after marked as done.

```
>>> od = OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE')
>>> od.after
'DONE'
```

CHAPTER 5

Further resources

5.1 Internals

5.1.1 orgparse.node

orgparse.node.**parse_heading_level**(*heading*)
Get star-stripped heading and its level

```
>>> parse_heading_level('* Heading')
('Heading', 1)
>>> parse_heading_level('******** Heading')
('Heading', 8)
>>> parse_heading_level('not heading')  # None
```

orgparse.node.**parse_heading_tags**(*heading*)
Get first tags and heading without tags

```
>>> parse_heading_tags('HEADING')
('HEADING', [])
>>> parse_heading_tags('HEADING :TAG1:TAG2:')
('HEADING', ['TAG1', 'TAG2'])
>>> parse_heading_tags('HEADING: this is still heading :TAG1:TAG2:')
('HEADING: this is still heading', ['TAG1', 'TAG2'])
>>> parse_heading_tags('HEADING :@tag:_tag_:')
('HEADING', ['@tag', '_tag_'])
```

Here is the spec of tags from Org Mode manual:

> Tags are normal words containing letters, numbers, _, and `@`. Tags must be preceded and followed by a single colon, e.g., `:work:`.
>
> —(info "(org) Tags")

orgparse.node.**parse_heading_todos**(*heading*, *todo_candidates*)
Get TODO keyword and heading without TODO keyword.

```
>>> todos = ['TODO', 'DONE']
>>> parse_heading_todos('Normal heading', todos)
('Normal heading', None)
>>> parse_heading_todos('TODO Heading', todos)
('Heading', 'TODO')
```

orgparse.node.**parse_heading_priority**(*heading*)

Get priority and heading without priority field..

```
>>> parse_heading_priority('HEADING')
('HEADING', None)
>>> parse_heading_priority('[#A] HEADING')
('HEADING', 'A')
>>> parse_heading_priority('[#0] HEADING')
('HEADING', '0')
>>> parse_heading_priority('[#A]')
('', 'A')
```

orgparse.node.**parse_property**(*line*)

Get property from given string.

```
>>> parse_property(':Some_property: some value')
('Some_property', 'some value')
>>> parse_property(':Effort: 1:10')
('Effort', 70)
```

orgparse.node.**parse_comment**(*line*)

Parse special comment such as `#+SEQ_TODO`

```
>>> parse_comment('#+SEQ_TODO: TODO | DONE')
('SEQ_TODO', 'TODO | DONE')
>>> parse_comment('# not a special comment')  # None
```

orgparse.node.**parse_seq_todo**(*line*)

Parse value part of SEQ_TODO/TODO/TYP_TODO comment.

```
>>> parse_seq_todo('TODO | DONE')
(['TODO'], ['DONE'])
>>> parse_seq_todo(' Fred  Sara   Lucy Mike  |  DONE  ')
(['Fred', 'Sara', 'Lucy', 'Mike'], ['DONE'])
>>> parse_seq_todo('| CANCELED')
([], ['CANCELED'])
>>> parse_seq_todo('REPORT(r) BUG(b) KNOWNCAUSE(k) | FIXED(f)')
(['REPORT', 'BUG', 'KNOWNCAUSE'], ['FIXED'])
```

See also:

- (info "(org) Per-file keywords")
- (info "(org) Fast access to TODO states")

class orgparse.node.**OrgEnv**(*todos=['TODO'], dones=['DONE'], filename='<undefined>'*)

Information global to the file (e.g, TODO keywords).

nodes

A list of org nodes.

```
>>> OrgEnv().nodes   # default is empty (of course)
[]
```

```
>>> from orgparse import loads
>>> loads('''
... * Heading 1
... ** Heading 2
... *** Heading 3
... ''').env.nodes      # doctest: +ELLIPSIS  +NORMALIZE_WHITESPACE
[<orgparse.node.OrgRootNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>,
 <orgparse.node.OrgNode object at 0x...>]
```

todo_keys
TODO keywords defined for this document (file).

```
>>> env = OrgEnv()
>>> env.todo_keys
['TODO']
```

done_keys
DONE keywords defined for this document (file).

```
>>> env = OrgEnv()
>>> env.done_keys
['DONE']
```

all_todo_keys
All TODO keywords (including DONEs).

```
>>> env = OrgEnv()
>>> env.all_todo_keys
['TODO', 'DONE']
```

filename
Return a path to the source file or similar information.

If the org objects are not loaded from a file, this value will be a string of the form <SOME_TEXT>.

Return type str

class orgparse.node.**OrgBaseNode**(*env*, *index=None*)
Base class for *OrgRootNode* and *OrgNode*

env
An instance of *OrgEnv*. All nodes in a same file shares same instance.

OrgBaseNode is an iterable object:

```
>>> from orgparse import loads
>>> root = loads('''
... * Heading 1
... ** Heading 2
... *** Heading 3
... * Heading 4
... ''')
>>> for node in root:
...     print(node)
<BLANKLINE>
* Heading 1
```

(continues on next page)

(continued from previous page)

```
** Heading 2
*** Heading 3
* Heading 4
```

Note that the first blank line is due to the root node, as iteration contains the object itself. To skip that, use slice access `[1:]`:

```
>>> for node in root[1:]:
...     print(node)
* Heading 1
** Heading 2
*** Heading 3
* Heading 4
```

It also support sequence protocol.

```
>>> print(root[1])
* Heading 1
>>> root[0] is root  # index 0 means itself
True
>>> len(root)   # remember, sequence contains itself
5
```

Note the difference between `root[1:]` and `root[1]`:

```
>>> for node in root[1]:
...     print(node)
* Heading 1
** Heading 2
*** Heading 3
```

`_index = None`

Index of *self* in *self.env.nodes*.

It must satisfy an equality:

```
self.env.nodes[self._index] is self
```

This value is used for quick access for iterator and tree-like traversing.

`previous_same_level`

Return previous node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... * Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.previous_same_level is None
True
>>> n2.previous_same_level is n1
True
>>> n3.previous_same_level is None  # n2 is not at the same level
True
```

next_same_level
Return next node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... * Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.next_same_level is n2
True
>>> n2.next_same_level is None  # n3 is not at the same level
True
>>> n3.next_same_level is None
True
```

get_parent(*max_level=None*)
Return a parent node.

Parameters **max_level** (*int*) – In the normally structured org file, it is a level of the ancestor node to return. For example, `get_parent(max_level=0)` returns a root node.

In general case, it specify a maximum level of the desired ancestor node. If there is no ancestor node which level is equal to `max_level`, this function try to find an ancestor node which level is smaller than `max_level`.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

For simplicity, accessing *parent* is alias of calling *get_parent()* without argument.

```
>>> n1.get_parent() is n1.parent
True
>>> root.parent is None
True
```

This is a little bit pathological situation – but works.

```
>>> root = loads('''
... * Node 1
... *** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
```

(continues on next page)

(continued from previous page)

```
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

Now let's play with *max_level*.

```
>>> root = loads('''
... * Node 1 (level 1)
... ** Node 2 (level 2)
... *** Node 3 (level 3)
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n3.get_parent() is n2
True
>>> n3.get_parent(max_level=2) is n2  # same as default
True
>>> n3.get_parent(max_level=1) is n1
True
>>> n3.get_parent(max_level=0) is root
True
```

parent
Alias of *get_parent()* (calling without argument).

children
A list of child nodes.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... *** Node 3
... ** Node 4
... ''')
>>> (n1, n2, n3, n4) = list(root[1:])
>>> (c1, c2) = n1.children
>>> c1 is n2
True
>>> c2 is n4
True
```

Note the difference to `n1[1:]`, which returns the Node 3 also.:

```
>>> (m1, m2, m3) = list(n1[1:])
>>> m2 is n3
True
```

root
The root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> n1 = root[1]
>>> n1.root is root
True
```

level
Level of this node.

Return type int

_get_tags(*inher=False*)
Return tags

Parameters **inher** (*bool*) – Mix with tags of all ancestor nodes if `True`.

Return type set

tags
Tag of this and parents node.

```
>>> from orgparse import loads
>>> n2 = loads('''
... * Node 1    :TAG1:
... ** Node 2   :TAG2:
... ''')[2]
>>> n2.tags == set(['TAG1', 'TAG2'])
True
```

shallow_tags
Tags defined for this node (don't look-up parent nodes).

```
>>> from orgparse import loads
>>> n2 = loads('''
... * Node 1    :TAG1:
... ** Node 2   :TAG2:
... ''')[2]
>>> n2.shallow_tags == set(['TAG2'])
True
```

is_root()
Return `True` when it is a root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> root.is_root()
True
>>> n1 = root[1]
>>> n1.is_root()
False
```

class orgparse.node.**OrgRootNode**(*env*, *index=None*)
Node to represent a file

See *OrgBaseNode* for other available functions.

get_parent(*max_level=None*)
Return a parent node.

Parameters **max_level** (*int*) – In the normally structured org file, it is a level of the ancestor node to return. For example, `get_parent(max_level=0)` returns a root node.

In general case, it specify a maximum level of the desired ancestor node. If there is no ancestor node which level is equal to `max_level`, this function try to find an ancestor node which level is smaller than `max_level`.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node 1
... ** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

For simplicity, accessing `parent` is alias of calling *`get_parent()`* without argument.

```
>>> n1.get_parent() is n1.parent
True
>>> root.parent is None
True
```

This is a little bit pathological situation – but works.

```
>>> root = loads('''
... * Node 1
... *** Node 2
... ** Node 3
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n1.get_parent() is root
True
>>> n2.get_parent() is n1
True
>>> n3.get_parent() is n1
True
```

Now let's play with *max_level*.

```
>>> root = loads('''
... * Node 1 (level 1)
... ** Node 2 (level 2)
... *** Node 3 (level 3)
... ''')
>>> (n1, n2, n3) = list(root[1:])
>>> n3.get_parent() is n2
True
>>> n3.get_parent(max_level=2) is n2  # same as default
True
>>> n3.get_parent(max_level=1) is n1
True
>>> n3.get_parent(max_level=0) is root
True
```

is_root()
Return `True` when it is a root node.

```
>>> from orgparse import loads
>>> root = loads('* Node 1')
>>> root.is_root()
True
>>> n1 = root[1]
>>> n1.is_root()
False
```

class orgparse.node.**OrgNode**(**args*, ***kwds*)

Node to represent normal org node

See *OrgBaseNode* for other available functions.

_parse_pre()

Call parsers which must be called before tree structuring

_iparse_sdc(*ilines*)

Parse SCHEDULED, DEADLINE and CLOSED time tamps.

They are assumed be in the first line.

get_heading(*format='plain'*)

Return a string of head text without tags and TODO keywords.

```
>>> from orgparse import loads
>>> node = loads('* TODO Node 1').children[0]
>>> node.get_heading()
'Node 1'
```

It strips off inline markup by default (`format='plain'`). You can get the original raw string by specifying `format='raw'`.

```
>>> node = loads('* [[link][Node 1]]').children[0]
>>> node.get_heading()
'Node 1'
>>> node.get_heading(format='raw')
'[[link][Node 1]]'
```

get_body(*format='plain'*)

Return a string of body text.

See also: *get_heading()*.

heading

Alias of `.get_heading(format='plain')`.

body

Alias of `.get_body(format='plain')`.

priority

Priority attribute of this node. It is None if undefined.

```
>>> from orgparse import loads
>>> (n1, n2) = loads('''
... * [#A] Node 1
... * Node 2
... ''').children
>>> n1.priority
'A'
```

(continues on next page)

(continued from previous page)

```
>>> n2.priority is None
True
```

_get_tags(*inher=False*)
Return tags

Parameters **inher** (*bool*) – Mix with tags of all ancestor nodes if `True`.

Return type set

todo
A TODO keyword of this node if exists or None otherwise.

```
>>> from orgparse import loads
>>> root = loads('* TODO Node 1')
>>> root.children[0].todo
'TODO'
```

get_property(*key*, *val=None*)
Return property named `key` if exists or `val` otherwise.

Parameters

- **key** (*str*) – Key of property.
- **val** – Default value to return.

properties
Node properties as a dictionary.

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   :PROPERTIES:
...   :SomeProperty: value
...   :END:
... ''')
>>> root.children[0].properties['SomeProperty']
'value'
```

scheduled
Return scheduled timestamp

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   SCHEDULED: <2012-02-26 Sun>
... ''')
>>> root.children[0].scheduled
OrgDateScheduled((2012, 2, 26))
```

deadline
Return deadline timestamp.

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   DEADLINE: <2012-02-26 Sun>
... ''')
>>> root.children[0].deadline
OrgDateDeadline((2012, 2, 26))
```

closed
Return timestamp of closed time.

Return type a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   CLOSED: [2012-02-26 Sun 21:15]
... ''')
>>> root.children[0].closed
OrgDateClosed((2012, 2, 26, 21, 15, 0))
```

clock
Return a list of clocked timestamps

Return type a list of a subclass of *orgparse.date.OrgDate*

```
>>> from orgparse import loads
>>> root = loads('''
... * Node
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] =>  0:05
... ''')
>>> root.children[0].clock
[OrgDateClock((2012, 2, 26, 21, 10, 0), (2012, 2, 26, 21, 15, 0))]
```

get_timestamps(*active=False*, *inactive=False*, *range=False*, *point=False*)
Return a list of timestamps in the body text.

Parameters

- **active** (*bool*) – Include active type timestamps.
- **inactive** (*bool*) – Include inactive type timestamps.
- **range** (*bool*) – Include timestamps which has end date.
- **point** (*bool*) – Include timestamps which has no end date.

Return type list of *orgparse.date.OrgDate* subclasses

Consider the following org node:

```
>>> from orgparse import loads
>>> node = loads('''
... * Node
...   CLOSED: [2012-02-26 Sun 21:15] SCHEDULED: <2012-02-26 Sun>
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] =>  0:05
...   Some inactive timestamp [2012-02-23 Thu] in body text.
...   Some active timestamp <2012-02-24 Fri> in body text.
...   Some inactive time range [2012-02-25 Sat]--[2012-02-27 Mon].
...   Some active time range <2012-02-26 Sun>--<2012-02-28 Tue>.
... ''').children[0]
```

The default flags are all off, so it does not return anything.

```
>>> node.get_timestamps()
[]
```

You can fetch appropriate timestamps using keyword arguments.

```
>>> node.get_timestamps(inactive=True, point=True)
[OrgDate((2012, 2, 23), None, False)]
>>> node.get_timestamps(active=True, point=True)
[OrgDate((2012, 2, 24))]
>>> node.get_timestamps(inactive=True, range=True)
[OrgDate((2012, 2, 25), (2012, 2, 27), False)]
>>> node.get_timestamps(active=True, range=True)
[OrgDate((2012, 2, 26), (2012, 2, 28))]
```

This is more complex example. Only active timestamps, regardless of range/point type.

```
>>> node.get_timestamps(active=True, point=True, range=True)
[OrgDate((2012, 2, 24)), OrgDate((2012, 2, 26), (2012, 2, 28))]
```

datelist

Alias of `.get_timestamps(active=True, inactive=True, point=True)`.

Return type list of *orgparse.date.OrgDate* subclasses

```
>>> from orgparse import loads
>>> root = loads('''
... * Node with point dates <2012-02-25 Sat>
...   CLOSED: [2012-02-25 Sat 21:15]
...   Some inactive timestamp [2012-02-26 Sun] in body text.
...   Some active timestamp <2012-02-27 Mon> in body text.
... ''')
>>> root.children[0].datelist      # doctest: +NORMALIZE_WHITESPACE
[OrgDate((2012, 2, 25)),
 OrgDate((2012, 2, 26), None, False),
 OrgDate((2012, 2, 27))]
```

rangelist

Alias of `.get_timestamps(active=True, inactive=True, range=True)`.

Return type list of *orgparse.date.OrgDate* subclasses

```
>>> from orgparse import loads
>>> root = loads('''
... * Node with range dates <2012-02-25 Sat>--<2012-02-28 Tue>
...   CLOCK: [2012-02-26 Sun 21:10]--[2012-02-26 Sun 21:15] => 0:05
...   Some inactive time range [2012-02-25 Sat]--[2012-02-27 Mon].
...   Some active time range <2012-02-26 Sun>--<2012-02-28 Tue>.
...   Some time interval <2012-02-27 Mon 11:23-12:10>.
... ''')
>>> root.children[0].rangelist     # doctest: +NORMALIZE_WHITESPACE
[OrgDate((2012, 2, 25), (2012, 2, 28)),
 OrgDate((2012, 2, 25), (2012, 2, 27), False),
 OrgDate((2012, 2, 26), (2012, 2, 28)),
 OrgDate((2012, 2, 27, 11, 23, 0), (2012, 2, 27, 12, 10, 0))]
```

has_date()

Return `True` if it has any kind of timestamp

repeated_tasks
Get repeated tasks marked DONE in a entry having repeater.

Return type list of *orgparse.date.OrgDateRepeatedTask*

```
>>> from orgparse import loads
>>> node = loads('''
... * TODO Pay the rent
...   DEADLINE: <2005-10-01 Sat +1m>
...   - State "DONE"  from "TODO"  [2005-09-01 Thu 16:10]
...   - State "DONE"  from "TODO"  [2005-08-01 Mon 19:44]
...   - State "DONE"  from "TODO"  [2005-07-01 Fri 17:27]
... ''').children[0]
>>> node.repeated_tasks            # doctest: +NORMALIZE_WHITESPACE
[OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 8, 1, 19, 44, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 7, 1, 17, 27, 0), 'TODO', 'DONE')]
>>> node.repeated_tasks[0].before
'TODO'
>>> node.repeated_tasks[0].after
'DONE'
```

Repeated tasks in `:LOGBOOK:` can be fetched by the same code.

```
>>> node = loads('''
... * TODO Pay the rent
...   DEADLINE: <2005-10-01 Sat +1m>
...   :LOGBOOK:
...   - State "DONE"  from "TODO"  [2005-09-01 Thu 16:10]
...   - State "DONE"  from "TODO"  [2005-08-01 Mon 19:44]
...   - State "DONE"  from "TODO"  [2005-07-01 Fri 17:27]
...   :END:
... ''').children[0]
>>> node.repeated_tasks            # doctest: +NORMALIZE_WHITESPACE
[OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 8, 1, 19, 44, 0), 'TODO', 'DONE'),
 OrgDateRepeatedTask((2005, 7, 1, 17, 27, 0), 'TODO', 'DONE')]
```

See: (info "(org) Repeated tasks")

5.1.2 orgparse.date

orgparse.date.**total_seconds**(*td*)
Equivalent to *datetime.timedelta.total_seconds*.

orgparse.date.**total_minutes**(*td*)
Alias for `total_seconds(td) / 60`.

orgparse.date.**gene_timestamp_regex**(*brtype*, *prefix=None*, *nocookie=False*)
Generate timetamp regex for active/inactive/nobrace brace type

Parameters

- **brtype** (*{'active', 'inactive', 'nobrace'}*) – It specifies a type of brace. active: <>-type; inactive: []-type; nobrace: no braces.
- **prefix** (*str or None*) – It will be appended to the head of keys of the "groupdict". For example, if prefix is `'active_'` the groupdict has keys such as `'active_year'`, `'active_month'`, and so on. If it is None it will be set to `brtype + '_'`.

- **nocookie** (*bool*) – Cookie part (e.g., `'-3d'` or `'+6m'`) is not included if it is `True`. Default value is `False`.

```
>>> timestamp_re = re.compile(
...     gene_timestamp_regex('active', prefix=''),
...     re.VERBOSE)
>>> timestamp_re.match('no match')  # returns None
>>> m = timestamp_re.match('<2010-06-21 Mon>')
>>> m.group()
'<2010-06-21 Mon>'
>>> '{year}-{month}-{day}'.format(**m.groupdict())
'2010-06-21'
>>> m = timestamp_re.match('<2005-10-01 Sat 12:30 +7m -3d>')
>>> from collections import OrderedDict
>>> sorted(m.groupdict().items())
... # doctest: +NORMALIZE_WHITESPACE
[('day', '01'),
 ('end_hour', None), ('end_min', None),
 ('hour', '12'), ('min', '30'),
 ('month', '10'),
 ('repeatdwmy', 'm'), ('repeatnum', '7'), ('repeatpre', '+'),
 ('warndwmy', 'd'), ('warnnum', '3'), ('warnpre', '-'), ('year', '2005')]
```

When `brtype = 'nobrace'`, cookie part cannot be retrieved.

```
>>> timestamp_re = re.compile(
...     gene_timestamp_regex('nobrace', prefix=''),
...     re.VERBOSE)
>>> timestamp_re.match('no match')  # returns None
>>> m = timestamp_re.match('2010-06-21 Mon')
>>> m.group()
'2010-06-21'
>>> '{year}-{month}-{day}'.format(**m.groupdict())
'2010-06-21'
>>> m = timestamp_re.match('2005-10-01 Sat 12:30 +7m -3d')
>>> sorted(m.groupdict().items())
... # doctest: +NORMALIZE_WHITESPACE
[('day', '01'),
 ('end_hour', None), ('end_min', None),
 ('hour', '12'), ('min', '30'),
 ('month', '10'), ('year', '2005')]
```

class orgparse.date.**OrgDateScheduled**(*start*, *end=None*, *active=None*)
Date object to represent SCHEDULED attribute.

class orgparse.date.**OrgDateDeadline**(*start*, *end=None*, *active=None*)
Date object to represent DEADLINE attribute.

class orgparse.date.**OrgDateClosed**(*start*, *end=None*, *active=None*)
Date object to represent CLOSED attribute.

class orgparse.date.**OrgDateClock**(*start*, *end*, *duration=None*, *active=None*)
Date object to represent CLOCK attributes.

```
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] =>  0:30')
OrgDateClock((2010, 8, 8, 17, 0, 0), (2010, 8, 8, 17, 30, 0))
```

duration

Get duration of CLOCK.

```
>>> duration = OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:30'
... ).duration
>>> duration.seconds
1800
>>> total_minutes(duration)
30.0
```

is_duration_consistent()
Check duration value of CLOCK line.

```
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:30'
... ).is_duration_consistent()
True
>>> OrgDateClock.from_str(
...   'CLOCK: [2010-08-08 Sun 17:00]--[2010-08-08 Sun 17:30] => 0:15'
... ).is_duration_consistent()
False
```

classmethod from_str(*line*)
Get CLOCK from given string.

Return three tuple (start, stop, length) which is datetime object of start time, datetime object of stop time and length in minute.

class orgparse.date.**OrgDateRepeatedTask**(*start*, *before*, *after*, *active=None*)
Date object to represent repeated tasks.

before
The state of task before marked as done.

```
>>> od = OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE')
>>> od.before
'TODO'
```

after
The state of task after marked as done.

```
>>> od = OrgDateRepeatedTask((2005, 9, 1, 16, 10, 0), 'TODO', 'DONE')
>>> od.after
'DONE'
```

class orgparse.date.**OrgDate**(*start*, *end=None*, *active=None*)

_active_default = True
The default active value.

When the *active* argument to __init__ is None, This value will be used.

_active_default = True
The default active value.

When the *active* argument to __init__ is None, This value will be used.

start
Get date or datetime object

```
>>> OrgDate((2012, 2, 10)).start
datetime.date(2012, 2, 10)
>>> OrgDate((2012, 2, 10, 12, 10)).start
datetime.datetime(2012, 2, 10, 12, 10)
```

end
Get date or datetime object

```
>>> OrgDate((2012, 2, 10), (2012, 2, 15)).end
datetime.date(2012, 2, 15)
>>> OrgDate((2012, 2, 10, 12, 10), (2012, 2, 15, 12, 10)).end
datetime.datetime(2012, 2, 15, 12, 10)
```

is_active()
Return true if the date is active

has_end()
Return true if it has the end date

has_time()
Return true if the start date has time field

```
>>> OrgDate((2012, 2, 10)).has_time()
False
>>> OrgDate((2012, 2, 10, 12, 10)).has_time()
True
```

has_overlap(*other*)
Test if it has overlap with other *OrgDate* instance

If the argument is not an instance of *OrgDate*, it is converted to *OrgDate* instance by `OrgDate(other)` first.

```
>>> od = OrgDate((2012, 2, 10), (2012, 2, 15))
>>> od.has_overlap(OrgDate((2012, 2, 11)))
True
>>> od.has_overlap(OrgDate((2012, 2, 20)))
False
>>> od.has_overlap(OrgDate((2012, 2, 11), (2012, 2, 20)))
True
>>> od.has_overlap((2012, 2, 11))
True
```

classmethod list_from_str(*string*)
Parse string and return a list of *OrgDate* objects

```
>>> OrgDate.list_from_str("... <2012-02-10 Fri> and <2012-02-12 Sun>")
[OrgDate((2012, 2, 10)), OrgDate((2012, 2, 12))]
>>> OrgDate.list_from_str("<2012-02-10 Fri>--<2012-02-12 Sun>")
[OrgDate((2012, 2, 10), (2012, 2, 12))]
>>> OrgDate.list_from_str("<2012-02-10 Fri>--[2012-02-12 Sun]")
[OrgDate((2012, 2, 10)), OrgDate((2012, 2, 12), None, False)]
>>> OrgDate.list_from_str("this is not timestamp")
[]
>>> OrgDate.list_from_str("<2012-02-11 Sat 10:11--11:20>")
[OrgDate((2012, 2, 11, 10, 11, 0), (2012, 2, 11, 11, 20, 0))]
```

classmethod from_str(*string*)
Parse string and return an *OrgDate* objects.

```
>>> OrgDate.from_str('2012-02-10 Fri')
OrgDate((2012, 2, 10))
>>> OrgDate.from_str('2012-02-10 Fri 12:05')
OrgDate((2012, 2, 10, 12, 5, 0))
```

- GitHub repository

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

Symbols

A

B

C

D

E

F

G

H

I

L

N

O

P

R

S

T