





moz://a

erik@mozilla.com · IRC: ErikRose · @ErikRose

```
11 + render_goto_tabs()
12 + sneeze_loudly(True)
```

 erikrose added a note 12 minutes ago Owner   

Your code is bad, and you are bad. Have a bad day.


[Add a line note](#)

```
13 + do_other_stuff()
```

Constructive Code Review

erik@mozilla.com · IRC: ErikRose · @ErikRose

```
11 + render_goto_tabs()  
12 + sneeze_loudly(True)
```

 erikrose added a note 12 minutes ago Owner +@ ✎ ✕

Your code is bad, and you are bad. Have a bad day.

```
13 + do_other_stuff()
```


Build an excellent product

Build an excellent product

Build people

Build an excellent product

Build people

Build yourself*

* Assumes you are not a person

Build an excellent product

Build people

Build yourself*

* Assumes you are not a person

Build an excellent product

Build people

Build yourself*

*Assumes you are not a person

Creative work
is powered by
enthusiasm.

Creative work
is powered by
enthusiasm.

Creative work
is powered by
enthusiasm.

Creative work
is powered by
enthusiasm.

Truth


Nature cannot
be fooled.

Kindness

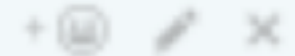
We are made
of meat.

Clarity of Explanation

```
13 + do_other_stuff()  
14 +  
15 + if thing == 5:  
16 +     stir('B')
```

 erikrose added a note 18 seconds ago

Owner




This isn't great.

Add a line note

```
17 +     print "Flibbety jibbet!"  
18 +     render_golfclubs()  
19 +     sneeze_loudly(True)  
20 +     do_other_stuff()  
21 +  
22 + if thing == 7:
```

Clarity of Explanation

```
13 + do_other_stuff()  
14 +  
15 + if thing == 5:  
16 +     stir('B')
```


 erikrose added a note 18 seconds ago

Owner

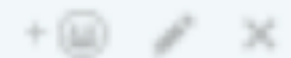


This isn't great.

```
13 + do_other_stuff()  
14 +  
15 + if thing == 5:  
16 +     stir('B')
```

 erikrose added a note 6 minutes ago

Owner



If we pass "B" to `stir` here, it will cause a mem leak as we allocate the whatzit, since the two loops of the B will get caught on adjacent gear teeth.

Add a line note

```
17 + print "Flibbety jibbet!"  
18 + render_golfclubs()  
19 +
```


Clarity of Explanation

Clarity of Explanation

Code

Clarity of Explanation

Code

Links

Clarity of Explanation

Code

Links

Higher-bandwidth communications

Clarity of Explanation

Code


Links

Higher-bandwidth communications

Write down the result!

Clarity of Expectation

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note just now

Owner

+ 😊 ✎ ✕

Internationalization would be better.

Add a line note

```
4 +     render_golfclubs()  
5 +     sneeze_loudly(True)  
6 +     do_other_stuff()
```

Clarity of Expectation

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note just now


Owner



Internationalization would be better.

Add

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note 2 minutes ago

Owner




It would be great to internationalize this message, but it doesn't need to block the merge.

Add a line note

```
4 +     render_golfclubs()  
5 +     sneeze_loudly(True)  
6 +     do_other_stuff()  
7 +
```

Clarity of Expectation

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note just now


Owner



Internationalization would be better.

Add

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note 2 minutes ago

It would be great to internationalize this message, but it doesn't

Add a line note

```
4 +     render_golfclubs()  
5 +     sneeze_loudly(True)  
6 +     do_other_stuff()
```

Unified

Split

Review changes

Submit your review

Review summary

Leave a comment

Comment

Submit general feedback without explicit approval.

Approve

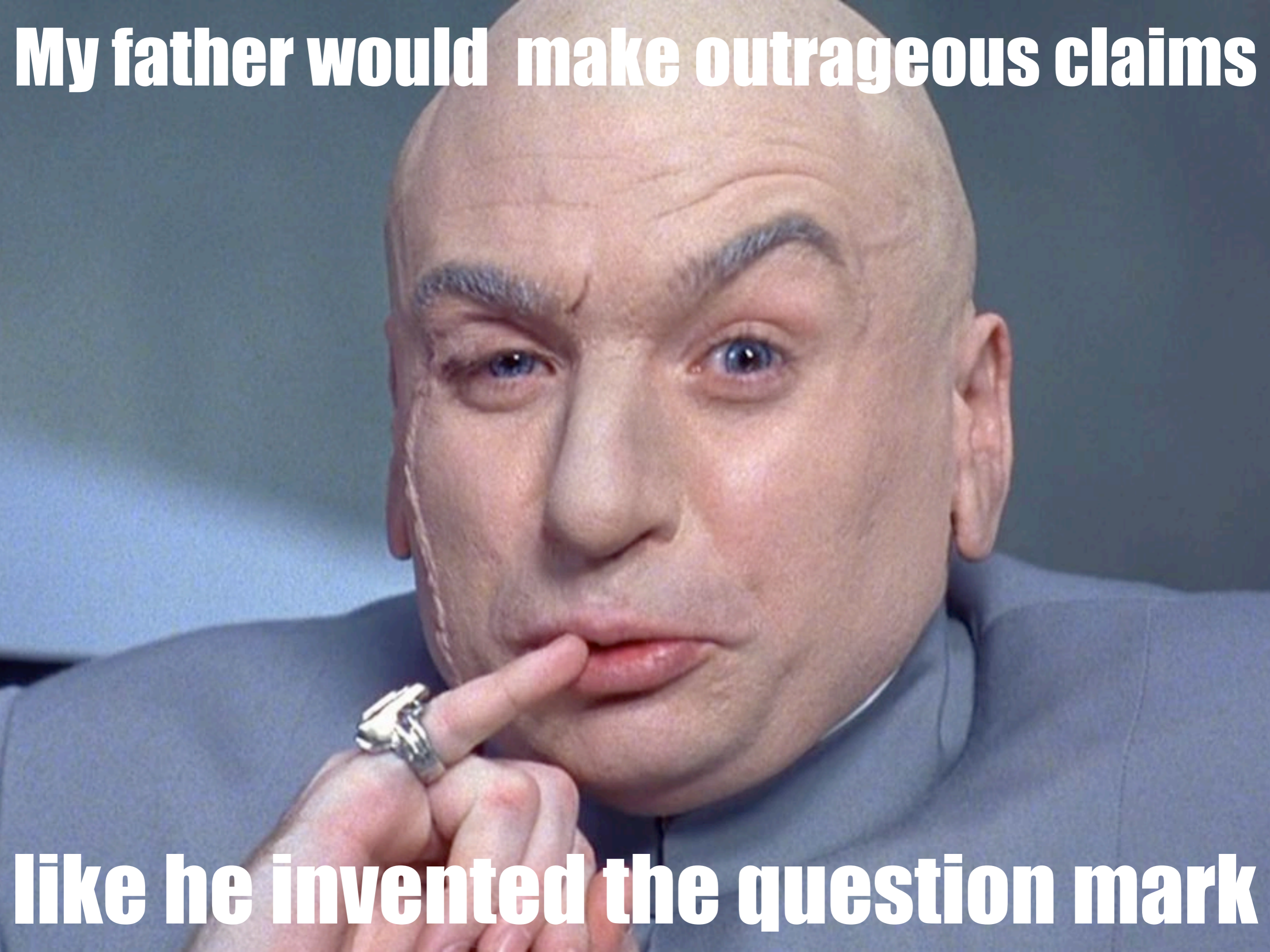
Submit feedback and approve merging these changes.

Request changes

Submit feedback that must be addressed before merging.

Submit review

Tact Hacks



My father would make outrageous claims

like he invented the question mark

The Question Mark

55

-

return term

87

+

```
text_terms = [term for term in self._terms if term['name'] == 't
```



erikrose added a note 19 days ago

Mozilla member



There's no point returning path results when there is more than one term.

88

+

```
if len(text_terms) == 1:
```

89


+

```
return text_terms[0]
```

The Question Mark

```
return term
```


```
87 + text_terms = [term for term in self._terms if term['name'] == 't
```

 erikrose added a note 19 days ago

Mozilla member




There's no point returning path results when there is more than one term.

 erikrose added a note 19 days ago

Mozilla member




Can you remind me of some use cases for returning path results when there is more than one term (but only one text term, of course)?

 pelmers added a note 19 days ago

Mozilla member



For example it's common to exclude the object directory in a search, and it can be helpful to still have the promoted results.

 erikrose added a note 16 days ago

Mozilla member



Of course. Thanks. Any FILE-domain filter could be useful.

Add a line note

```
88 + if len(text_terms) == 1:
```

You, We, & This

55

return term

87

+

```
text_terms = [term for term in self._terms if term['name'] == 't
```



erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries

88

+

```
if len(text_terms) == 1:
```

89

+

```
return text_terms[0]
```

You, We, & This

55

return term

87

+

```
text_terms = [term for term in self._terms if term['name'] == 't
```



erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries [you idiot]

88

+

```
if len(text_terms) == 1:
```


89

+

```
return text_terms[0]
```

You, We, & This

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries [you idiot]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```

 erikrose added a note 19 days ago

Mozilla member




If we do it this way, it'll break Unicode queries

```
88 + if len(text_terms) == 1:
89 +     return text_terms[0]
```

You, We, & This

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries [you idiot]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```

 erikrose added a note 19 days ago

Mozilla member




If we do it this way, it'll break Unicode queries [my fellow code steward]

```
88 + if len(text_terms) == 1:
89 +     return text_terms[0]
```


You, We, & This

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries [you idiot]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If we do it this way, it'll break Unicode queries [my fellow code steward]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```

 erikrose added a note 19 days ago

Mozilla member




This casting will break Unicode queries

```
88 + if len(text_terms) == 1:
89 +     return text_terms[0]
```

You, We, & This

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If you do it this way, you'll break Unicode queries [you idiot]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```


 erikrose added a note 19 days ago

Mozilla member



If we do it this way, it'll break Unicode queries [my fellow code steward]

```
53 - return term
87 + text_terms = [term for term in self._terms if term['name'] == 't
```

 erikrose added a note 19 days ago

Mozilla member



This casting will break Unicode queries [as a matter of fact]

```
88 + if len(text_terms) == 1:
89 +     return text_terms[0]
```

Compliments

Code

Issues 0

Pull requests 1

Pulse

Graphs

Settings

Add dispatch handler for flibbety jibbeting. #3

Open erikrose wants to merge 1 commit into `master` from `screenshots`

Conversation 0

Commits 1

Files changed 1



gerbiltickler commented 21 hours ago

Owner



This oughtta do it.

Add dispatch handler for flibbety jibbeting.

729d31b



erikrose commented 2 minutes ago

Owner



Really looking forward to having this; I know a lot of our users need to jibbet flibbetily!

Add more commits by pushing to the **screenshots** branch on **erikrose/presentations**.

Compliments

Compliments

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note just now

Owner



Thank you for refactoring this scary mess!

Add a line note

```
4 +     render_golfclubs()  
5 +     sneeze_loudly(True)  
6 +     do_other_stuff()
```

Compliments

```
1 +def do_stuff(thing):  
2 +     if thing == 2:  
3 +         print "Flibbety jibbet!"
```

 erikrose added a note just now

Owner

+   

Thank you for refactoring this scary mess!

Add a line note

```
4 +  
5 +  
6 +
```

 pelmers added a note 19 days ago

Mozilla member

+   

I think this is an off-by-one on the end of the list.

 erikrose added a note 16 days ago

Mozilla member

+   

Yikes, nice catch!

Add a line note

```
88 +     if len(text_terms) == 1:  
89 +         return text_terms[0]
```

Humor



Humor

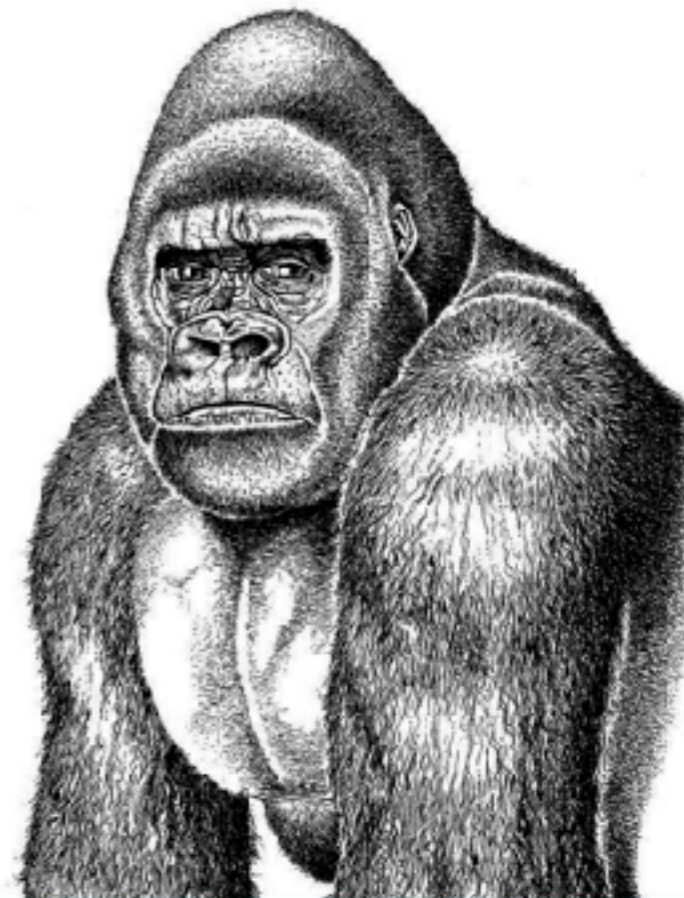


HIDE AND SEEK

I WANT A PONY



Who are you kidding?



“Temporary”
Workarounds

O RLY?

@ThePracticalDev

Antipatterns

TL;DR; LGTM

```
from __future__ import print_function

from collections import Counter, defaultdict, deque
from functools import partial, wraps
from heapq import merge
from itertools import chain, count, groupby, islice, repeat
from operator import itemgetter
from sys import version_info

from six import binary_type, string_types, text_type
from six.moves import filter, map, zip, zip_longest

from .recipes import flatten, take

__all__ = [
    'adjacent',
    'always_iterable',
    'bucket',
    'chunked',
    'collapse',
    'collate',
    'consumer',
    'distinct_permutations',
    'distribute',
    'divide',
    'first',
    'groupby_transform',
    'ilen',
    'interleave_longest',
    'interleave',
    'intersperse',
    'iterate',
    'one',
    'padded',
    'peekable',
    'side_effect',
    'sliced',
    'sort_together',
    'split_after',
    'split_before',
    'spy',
    'stagger',
    'unique_to_each',
    'windowed',
    'with_iter',
    'zip_offset',
]

_marker = object()

def chunked(iterable, n):
    """Break an iterable into lists of a given length:

    >>> list(chunked([1, 2, 3, 4, 5, 6, 7], 3))
    [[1, 2, 3], [4, 5, 6], [7]]

    If the length of ``iterable`` is not evenly divisible, the
    returned list will be shorter.

    This is useful for splitting up a computation on a large
    into batches, to be pickled and sent off to worker
    is operations on rows in MySQL, which does not imp
    cursors properly and would otherwise load the entire
    the client.

    """
    return iter(partial(take, n, iter(iterable)), [])

def first(iterable, default=_marker):
    """Return the first item of an iterable, ``default``

    >>> first([0, 1, 2, 3])
    0
    >>> first([], 'some default')
    'some default'

    If ``default`` is not provided and there are no items,
    raise ``ValueError``.

    ``first()`` is useful when you have a generator of
    values and want any arbitrary one. It is marginally
    ``next(iter(...), default)``.

    """
    try:
        return next(iter(iterable))
    except StopIteration:
        # I'm on the edge about raising ValueError ins
        # the moment, ValueError wins, because the cal
        # want to do something different with flow con
        # exception, and it's weird to explicitly catc
        if default is _marker:
            raise ValueError('first() was called on an
            'default value was provided')
        return default

class peekable(object):
    """Wrap an iterator to allow lookahead and prepending elements.

    Call ``peek()`` on the result to get the value that will next pop out of
    ``next()`` , without advancing the iterator:

    >>> p = peekable(['a', 'b'])
    >>> p.peek()
    'a'
    >>> next(p)
    'a'

    Pass ``peek()`` a default value to
    ``StopIteration`` when the iterator

    >>> p = peekable([])
    >>> p.peek('hi')
    'hi'

    peekables also offer a ``prepend()``
    the remaining part of the underlying

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> p.peek()
    11
    >>> list(p)
    [11, 12, 1, 2, 3]

    Prepend items are treated by oth
    had come from the source iterator.

    You may index the peekable to look
    The values up to the index you spe
    Index 0 is the item that will be r
    item after that, and so on:

    >>> p = peekable(['a', 'b', 'c'])
    >>> p[0]
    'a'
    >>> p[1]
    'b'
    >>> next(p)
    'a'
    >>> p.prepend('x')
    >>> p[1]
    'b'
    >>> next(p)
    'x'
    >>> next(p)
    'b'

    Negative indexes are supported, bu
    remaining items in the source iter
    storage.

    To test whether there are more ite
    peekable's truth value. If it is t
    have been prepended or obtained fr

    >>> assert peekable([])
    >>> p = peekable([])
    >>> assert not p
    >>> p.prepend(1)
    >>> assert p

    """
    def __init__(self, iterable):
        self._it = iter(iterable)
        self._cache = deque()

    def __iter__(self):
        return self

    def __bool__(self):
        try:
            self._peek()
        except StopIteration:
            return False
        return True

    def __nonzero__(self):
        # For Python 2 compatibility
        return self.__bool__()

    def peek(self, default=_marker):
        """Return the item that will be next returned from ``next()``
        Return ``default`` if there are no items left. If ``default``
        provided, raise ``StopIteration``.

        """
        if not self._cache:
            try:
                self._cache.append(next(self._it))
            except StopIteration:
                if default is _marker:
                    raise
                return default
        return self._cache[0]

    def prepend(self, *items):
        """Stack up items to be the next ones returned from ``next()``
        ``self._peek()``. The items will be returned in
        first in, first out order:

        >>> p = peekable([1, 2, 3])
        >>> p.prepend(10, 11, 12)
        >>> next(p)
        10
        >>> list(p)
        [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
class peekable(object):
    """Wrap an iterator to allow lookahead and prepending elements.

    Call ``peek()`` on the result to get the value that will next pop out of
    ``next()`` , without advancing the iterator:

    >>> p = peekable(['a', 'b'])
    >>> p.peek()
    'a'
    >>> next(p)
    'a'

    Pass ``peek()`` a default value to
    ``StopIteration`` when the iterator

    >>> p = peekable([])
    >>> p.peek('hi')
    'hi'

    peekables also offer a ``prepend()``
    the remaining part of the underlying

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> p.peek()
    11
    >>> list(p)
    [11, 12, 1, 2, 3]

    Prepend items are treated by oth
    had come from the source iterator.

    You may index the peekable to look
    The values up to the index you spe
    Index 0 is the item that will be r
    item after that, and so on:

    >>> p = peekable(['a', 'b', 'c'])
    >>> p[0]
    'a'
    >>> p[1]
    'b'
    >>> next(p)
    'a'
    >>> p.prepend('x')
    >>> p[1]
    'b'
    >>> next(p)
    'x'
    >>> next(p)
    'b'

    Negative indexes are supported, bu
    remaining items in the source iter
    storage.

    To test whether there are more ite
    peekable's truth value. If it is t
    have been prepended or obtained fr

    >>> assert peekable([])
    >>> p = peekable([])
    >>> assert not p
    >>> p.prepend(1)
    >>> assert p

    """
    def __init__(self, iterable):
        self._it = iter(iterable)
        self._cache = deque()

    def __iter__(self):
        return self

    def __bool__(self):
        try:
            self._peek()
        except StopIteration:
            return False
        return True

    def __nonzero__(self):
        # For Python 2 compatibility
        return self.__bool__()

    def peek(self, default=_marker):
        """Return the item that will be next returned from ``next()``
        Return ``default`` if there are no items left. If ``default``
        provided, raise ``StopIteration``.

        """
        if not self._cache:
            try:
                self._cache.append(next(self._it))
            except StopIteration:
                if default is _marker:
                    raise
                return default
        return self._cache[0]

    def prepend(self, *items):
        """Stack up items to be the next ones returned from ``next()``
        ``self._peek()``. The items will be returned in
        first in, first out order:

        >>> p = peekable([1, 2, 3])
        >>> p.prepend(10, 11, 12)
        >>> next(p)
        10
        >>> list(p)
        [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
def __init__(self, iterable):
    self._it = iter(iterable)
    self._cache = deque()

def __iter__(self):
    return self

def __bool__(self):
    try:
        self._peek()
    except StopIteration:
        return False
    return True

def __nonzero__(self):
    # For Python 2 compatibility
    return self.__bool__()

def peek(self, default=_marker):
    """Return the item that will be next returned from ``next()``
    Return ``default`` if there are no items left. If ``default``
    provided, raise ``StopIteration``.

    """
    if not self._cache:
        try:
            self._cache.append(next(self._it))
        except StopIteration:
            if default is _marker:
                raise
            return default
    return self._cache[0]

def prepend(self, *items):
    """Stack up items to be the next ones returned from ``next()``
    ``self._peek()``. The items will be returned in
    first in, first out order:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> list(p)
    [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
def __init__(self, iterable):
    self._it = iter(iterable)
    self._cache = deque()

def __iter__(self):
    return self

def __bool__(self):
    try:
        self._peek()
    except StopIteration:
        return False
    return True

def __nonzero__(self):
    # For Python 2 compatibility
    return self.__bool__()

def peek(self, default=_marker):
    """Return the item that will be next returned from ``next()``
    Return ``default`` if there are no items left. If ``default``
    provided, raise ``StopIteration``.

    """
    if not self._cache:
        try:
            self._cache.append(next(self._it))
        except StopIteration:
            if default is _marker:
                raise
            return default
    return self._cache[0]

def prepend(self, *items):
    """Stack up items to be the next ones returned from ``next()``
    ``self._peek()``. The items will be returned in
    first in, first out order:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> list(p)
    [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
def __init__(self, iterable):
    self._it = iter(iterable)
    self._cache = deque()

def __iter__(self):
    return self

def __bool__(self):
    try:
        self._peek()
    except StopIteration:
        return False
    return True

def __nonzero__(self):
    # For Python 2 compatibility
    return self.__bool__()

def peek(self, default=_marker):
    """Return the item that will be next returned from ``next()``
    Return ``default`` if there are no items left. If ``default``
    provided, raise ``StopIteration``.

    """
    if not self._cache:
        try:
            self._cache.append(next(self._it))
        except StopIteration:
            if default is _marker:
                raise
            return default
    return self._cache[0]

def prepend(self, *items):
    """Stack up items to be the next ones returned from ``next()``
    ``self._peek()``. The items will be returned in
    first in, first out order:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> list(p)
    [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
def __init__(self, iterable):
    self._it = iter(iterable)
    self._cache = deque()

def __iter__(self):
    return self

def __bool__(self):
    try:
        self._peek()
    except StopIteration:
        return False
    return True

def __nonzero__(self):
    # For Python 2 compatibility
    return self.__bool__()

def peek(self, default=_marker):
    """Return the item that will be next returned from ``next()``
    Return ``default`` if there are no items left. If ``default``
    provided, raise ``StopIteration``.

    """
    if not self._cache:
        try:
            self._cache.append(next(self._it))
        except StopIteration:
            if default is _marker:
                raise
            return default
    return self._cache[0]

def prepend(self, *items):
    """Stack up items to be the next ones returned from ``next()``
    ``self._peek()``. The items will be returned in
    first in, first out order:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> list(p)
    [11, 12, 1, 2, 3]

    It is possible, by prepending items, to "resurrect" a peekable
    previously raised ``StopIteration``.

    >>> p = peekable([])
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    >>> p.prepend(1)
    >>> next(p)
    1
    >>> next(p)
    Traceback (most recent call last):
    ...
    StopIteration
    """
    self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))

        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)

        cache_len = len(self._cache)
        if index < 0:
            # ...

```

```
def _collate(*iterables, **kwargs):
    """Helper for ``collate()`` , called when the user is using the ``reverse``
    or ``key`` keyword arguments on Python versions below 3.5.

    """
    key = kwargs.pop('key', lambda a: a)
    reverse = kwargs.pop('reverse', False)

    min_or_max = partial(max if reverse else min, key=lambda a_b: a_b[0])
    peekables = [peekable(it) for it in iterables]
    peekables = [p for p in peekables if p] # Kill empties.
    while peekables:
        p = min_or_max((key(p.peek()), p) for p in peekables)
        yield next(p)
        peekables = [x for x in peekables if x]

def collate(*iterables, **kwargs):
    """Return a sorted merge of the items from each of several already-sorted
    ``iterables``.

    >>> list(collate('ACDZ', 'AZ', 'JKL'))
    ['A', 'A', 'C', 'D', 'J', 'K', 'L', 'Z', 'Z']

    Works lazily, keeping only the next value from each iterable in memory. Use
    ``collate()`` to, for example, perform a n-way mergesort of items that
    don't fit in memory.

    :arg key: A function that returns a comparison value for an item. Defaults
    to the identity function.
    :arg reverse: If ``reverse=True`` , yield results in descending order
    rather than ascending. ``iterables`` must also yield their elements in
    descending order.

    If the elements of the passed-in iterables are out of order, you might get
    unexpected results.

    If neither of the keyword arguments are specified, this function delegates
    to ``heapq.merge()`` .

    """
    if not kwargs:
        return merge(*iterables)

    return _collate(*iterables, **kwargs)

# If using Python version 3.5 or greater, heapq.merge() will be faster than
# collate - use that instead.
if version_info >= (3, 5, 0):
    collate = merge

def consumer(func):
    """Decorator that automatically advances a PEP-342-style "reverse iterator"
    to its first yield point so you don't have to call ``next()`` on it
    manually.

    >>> @consumer
    ... def tally():
    ...     i = 0
    ...     while True:
    ...         print('Thing number %s is %s.' % (i, (yield)))
    ...         i += 1
    ...
    >>> t = tally()
    >>> t.send('red')
    Thing number 0 is red.
    >>> t.send('fish')
    Thing number 1 is fish.

    Without the decorator, you would have to call ``next(t)`` before
    ``t.send()`` could be used.

    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return wrapper

def ilen(iterable):
    """Return the number of items in ``iterable``.

    >>> ilen(x for x in range(1000000) if x % 3 == 0)
    333334

    This consumes the iterable, so handle with care.

    """
    d = deque(enumerate(iterable, 1), maxlen=1)
    return d[0][0] if d else 0

def iterate(func, start):
    """Return ``start`` , ``func(start)`` , ``func(func(start))`` , ...

    >>> from itertools import islice
    >>> list(islice(iterate(lambda x: 2*x, 1), 10))
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

    """
    while True:
        yield start
        start = func(start)

def with_iter(context_manager):

```

TL;DR: LGTM

```
from __future__ import print_function

from collections import Counter, defaultdict, deque
from functools import partial, wraps
from heapq import merge
from itertools import chain, count, groupby, islice, repeat
from operator import itemgetter
from sys import version_info

from six import binary_type, string_types, text_type
from six.moves import filter, map, zip, zip_longest

from .recipes import flatten, take

__all__ = [
    'adjacent',
    'always_iterable',
    'bucket',
    'chunked',
    'collapsed',
    'collate',
    'collate',
    'consumer',
    'distinct_permutations',
    'distribute',
    'divide',
    'first',
    'groupby_transform',
    'ilen',
    'interleave_longest',
    'interleave',
    'intersperse',
    'iterate',
    'one',
    'padded',
    'peekable',
    'side_effect',
    'sliced',
    'sort_together',
    'split_after',
    'split_before',
    'spy',
    'stagger',
    'unique_to_each',
    'windowed',
    'with_iter',
    'zip_offset',
]

_marker = object()

def chunked(iterable, n):
    """Break an iterable into lists of a given length:

    >>> list(chunked([1, 2, 3, 4, 5, 6, 7], 3))
    [[1, 2, 3], [4, 5, 6], [7]]

    If the length of ``iterable`` is not evenly divisible, the
    returned list will be shorter.

    This is useful for splitting up a computation on a large
    into batches, to be pickled and sent off to worker
    is operations on rows in MySQL, which does not imp
    cursors properly and would otherwise load the entire
    the client.

    """
    return iter(partial(take, n, iter(iterable)), [])

def first(iterable, default=_marker):
    """Return the first item of an iterable, ``default`` if
    the iterable is empty.

    >>> first([0, 1, 2, 3])
    0
    >>> first([], 'some default')
    'some default'

    If ``default`` is not provided and there are no items,
    raise ``ValueError``.

    ``first()`` is useful when you have a generator of
    values and want any arbitrary one. It is marginally
    ``next(iter(...), default)``.

    """
    try:
        return next(iter(iterable))
    except StopIteration:
        # I'm on the edge about raising ValueError ins
        # the moment, ValueError wins, because the cal
        # want to do something different with flow con
        # exception, and it's weird to explicitly catc
        if default is _marker:
            raise ValueError('first() was called on an
            empty iterable; default value was provided')
        return default

class peekable(object):
    """Wrap an iterator to allow lookahead and prepending elements.

    Call ``peek()`` on the result to get the value that will next pop out of
    ``next()`` , without advancing the iterator:

    >>> p = peekable(['a', 'b'])
    >>> p.peek()
    'a'
    >>> next(p)
    'a'

    Pass ``peek()`` a default value to
    ``StopIteration`` when the iterator
    has been exhausted:

    >>> p = peekable([])
    >>> p.peek('hi')
    'hi'

    peekables also offer a ``prepend()``
    the remaining part of the underlying
    iterator:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> p.peek()
    11
    >>> list(p)
    [11, 12, 1, 2, 3]

    Prepend items are treated by oth
    had come from the source iterator.

    You may index the peekable to look
    The values up to the index you spe
    Index 0 is the item that will be r
    item after that, and so on:

    >>> p = peekable(['a', 'b', 'c'])
    >>> p[0]
    'a'
    >>> p[1]
    'b'
    >>> next(p)
    'a'
    >>> p.prepend('x')
    >>> p[1]
    'b'
    >>> next(p)
    'x'
    >>> next(p)
    'b'

    Negative indexes are supported, bu
    remaining items in the source iter
    storage.

    To test whether there are more ite
    peekable's truth value. If it is t
    have been prepended or obtained fr

    >>> assert peekable([1])
    >>> p = peekable([])
    >>> assert not p
    >>> p.prepend(1)
    >>> assert p

    """
    def __init__(self, iterable):
        self._it = iter(iterable)
        self._cache = deque()

    def __iter__(self):
        return self

    def __bool__(self):
        try:
            self.peek()
        except StopIteration:
            return False
        return True

    def __nonzero__(self):
        # For Python 2 compatibility
        return self.__bool__()

    def peek(self, default=_marker):
        """Return the item that will be next returned from ``next()``
        if there are no items left. If ``default``
        is provided, raise ``StopIteration``.

        """
        if not self._cache:
            try:
                self._cache.append(next(self._it))
            except StopIteration:
                if default is _marker:
                    raise
                return default
        return self._cache[0]

    def prepend(self, *items):
        """Stack up items to be the next ones returned from ``next()``
        ``self.peek()``. The items will
        first in, first out order:

        >>> p = peekable([1, 2, 3])
        >>> p.prepend(10, 11, 12)
        >>> next(p)
        10
        >>> list(p)
        [11, 12, 1, 2, 3]

        It is possible, by prepending items, to "resurrect" a peekable
        previously raised ``StopIteration``.

        >>> p = peekable([])
        >>> next(p)
        Traceback (most recent call last):
        ...
        StopIteration
        >>> p.prepend(1)
        >>> next(p)
        1
        >>> next(p)
        Traceback (most recent call last):
        ...
        StopIteration
        """
        self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()
        else:
            return next(self._it)

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))
        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)
        cache_len = len(self._cache)
        if index < 0:
            index = cache_len + index
            if index < 0:
                raise IndexError('peekable index out of range')
```

```
class peekable(object):
    """Wrap an iterator to allow lookahead and prepending elements.

    Call ``peek()`` on the result to get the value that will next pop out of
    ``next()`` , without advancing the iterator:

    >>> p = peekable(['a', 'b'])
    >>> p.peek()
    'a'
    >>> next(p)
    'a'

    Pass ``peek()`` a default value to
    ``StopIteration`` when the iterator
    has been exhausted:

    >>> p = peekable([])
    >>> p.peek('hi')
    'hi'

    peekables also offer a ``prepend()``
    the remaining part of the underlying
    iterator:

    >>> p = peekable([1, 2, 3])
    >>> p.prepend(10, 11, 12)
    >>> next(p)
    10
    >>> p.peek()
    11
    >>> list(p)
    [11, 12, 1, 2, 3]

    Prepend items are treated by oth
    had come from the source iterator.

    You may index the peekable to look
    The values up to the index you spe
    Index 0 is the item that will be r
    item after that, and so on:

    >>> p = peekable(['a', 'b', 'c'])
    >>> p[0]
    'a'
    >>> p[1]
    'b'
    >>> next(p)
    'a'
    >>> p.prepend('x')
    >>> p[1]
    'b'
    >>> next(p)
    'x'
    >>> next(p)
    'b'

    Negative indexes are supported, bu
    remaining items in the source iter
    storage.

    To test whether there are more ite
    peekable's truth value. If it is t
    have been prepended or obtained fr

    >>> assert peekable([1])
    >>> p = peekable([])
    >>> assert not p
    >>> p.prepend(1)
    >>> assert p

    """
    def __init__(self, iterable):
        self._it = iter(iterable)
        self._cache = deque()

    def __iter__(self):
        return self

    def __bool__(self):
        try:
            self.peek()
        except StopIteration:
            return False
        return True

    def __nonzero__(self):
        # For Python 2 compatibility
        return self.__bool__()

    def peek(self, default=_marker):
        """Return the item that will be next returned from ``next()``
        if there are no items left. If ``default``
        is provided, raise ``StopIteration``.

        """
        if not self._cache:
            try:
                self._cache.append(next(self._it))
            except StopIteration:
                if default is _marker:
                    raise
                return default
        return self._cache[0]

    def prepend(self, *items):
        """Stack up items to be the ne
        ``self.peek()``. The items will
        first in, first out order:

        >>> p = peekable([1, 2, 3])
        >>> p.prepend(10, 11, 12)
        >>> next(p)
        10
        >>> list(p)
        [11, 12, 1, 2, 3]

        It is possible, by prepending items, to "resurrect" a peekable
        previously raised ``StopIteration``.

        >>> p = peekable([])
        >>> next(p)
        Traceback (most recent call last):
        ...
        StopIteration
        >>> p.prepend(1)
        >>> next(p)
        1
        >>> next(p)
        Traceback (most recent call last):
        ...
        StopIteration
        """
        self._cache.extendleft(reversed(items))

    def __next__(self):
        if self._cache:
            return self._cache.popleft()
        else:
            return next(self._it)

    def next(self):
        # For Python 2 compatibility
        return self.__next__()

    def __get_slice(self, index):
        start = index.start
        stop = index.stop

        if (
            ((start is not None) and (start < 0)) or
            ((stop is not None) and (stop < 0))
        ):
            stop = None
        elif (
            (start is not None) and (stop is not None) and (start > stop)
        ):
            stop = start + 1

        cache_len = len(self._cache)
        if stop is None:
            self._cache.extend(self._it)
        elif stop >= cache_len:
            self._cache.extend(islice(self._it, stop - cache_len))
        return list(self._cache)[index]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return self.__get_slice(index)
        cache_len = len(self._cache)
        if index < 0:
            index = cache_len + index
            if index < 0:
                raise IndexError('peekable index out of range')
```

```
def __collate(*iterables, **kwargs):
    """Helper for ``collate()`` , called when the user is using the ``reverse``
    or ``key`` keyword arguments on Python versions below 3.5.

    """
    key = kwargs.pop('key', lambda a: a)
    reverse = kwargs.pop('reverse', False)

    min_or_max = partial(max if reverse else min, key=key)
    peekables = [peekable(it) for it in iterables]
    peekables = [p for p in peekables if p] # Kill empties.
    while peekables:
        p = min_or_max((key(p.peek()), p) for p in peekables)
        yield next(p)
        peekables = [x for x in peekables if x]

def collate(*iterables, **kwargs):
    """Return a sorted merge of the items from each of several already-sorted
    ``iterables``.

    >>> list(collate('ACDZ', 'AZ', 'JKL'))
    ['A', 'A', 'C', 'D', 'J', 'K', 'L', 'Z', 'Z']

    Works lazily, keeping only the next value from each iterable in memory. Use
    ``collate()`` to, for example, perform a n-way mergesort of items that
    don't fit in memory.

    :arg key: A function that returns a comparison value for an item. Defaults
    to the identity function.
    :arg reverse: If ``reverse=True`` , yield results in descending order
    rather than ascending. ``iterables`` must also yield their elements in
    descending order.

    If the elements of the passed-in iterables are out of order, you might get
    unexpected results.

    If neither of the keyword arguments are specified, this function delegates
    to ``heapq.merge()``.

    """
    if not kwargs:
        return merge(*iterables)

    return __collate(*iterables, **kwargs)

# If using Python version 3.5 or greater, heapq.merge() will be faster than
# collate - use that instead.
if version_info >= (3, 5, 0):
    collate = merge

def consumer(func):
    """Decorator that automatically advances a PEP-342-style "reverse iterator"
    to its first yield point so you don't have to call ``next()`` on it
    manually.

    >>> @consumer
    ... def tally():
    ...     i = 0
    ...     while True:
    ...         print('Thing number %s is %s.' % (i, (yield)))
    ...         i += 1
    ...
    >>> t = tally()
    >>> t.send('red')
    Thing number 0 is red.
    >>> t.send('fish')
    Thing number 1 is fish.

    Without the decorator, you would have to call ``next(t)`` before
    ``t.send()`` could be used.

    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return wrapper

def ilen(iterable):
    """Return the number of items in ``iterable``.

    >>> ilen(x for x in range(1000000) if x % 3 == 0)
    333334

    This consumes the iterable, so handle with care.

    """
    d = deque(enumerate(iterable, 1), maxlen=1)
    return d[0][0] if d else 0

def iterate(func, start):
    """Return ``start`` , ``func(start)`` , ``func(func(start))`` , ...

    >>> from itertools import islice
    >>> list(islice(iterate(lambda x: 2*x, 1), 10))
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

    """
    while True:
        yield start
        start = func(start)

def with_iter(context_manager):
```

```
def __collate(*iterables, **kwargs):
    """Helper for ``collate()`` , called when the user is using the ``reverse``
    or ``key`` keyword arguments on Python versions below 3.5.

    """
    key = kwargs.pop('key', lambda a: a)
    reverse = kwargs.pop('reverse', False)

    min_or_max = partial(max if reverse else min, key=key)
    peekables = [peekable(it) for it in iterables]
    peekables = [p for p in peekables if p] # Kill empties.
    while peekables:
        p = min_or_max((key(p.peek()), p) for p in peekables)
        yield next(p)
        peekables = [x for x in peekables if x]

def collate(*iterables, **kwargs):
    """Return a sorted merge of the items from each of several already-sorted
    ``iterables``.

    >>> list(collate('ACDZ', 'AZ', 'JKL'))
    ['A', 'A', 'C', 'D', 'J', 'K', 'L', 'Z', 'Z']

    Works lazily, keeping only the next value from each iterable in memory. Use
    ``collate()`` to, for example, perform a n-way mergesort of items that
    don't fit in memory.

    :arg key: A function that returns a comparison value for an item. Defaults
    to the identity function.
    :arg reverse: If ``reverse=True`` , yield results in descending order
    rather than ascending. ``iterables`` must also yield their elements in
    descending order.

    If the elements of the passed-in iterables are out of order, you might get
    unexpected results.

    If neither of the keyword arguments are specified, this function delegates
    to ``heapq.merge()``.

    """
    if not kwargs:
        return merge(*iterables)

    return __collate(*iterables, **kwargs)

# If using Python version 3.5 or greater, heapq.merge() will be faster than
# collate - use that instead.
if version_info >= (3, 5, 0):
    collate = merge

def consumer(func):
    """Decorator that automatically advances a PEP-342-style "reverse iterator"
    to its first yield point so you don't have to call ``next()`` on it
    manually.

    >>> @consumer
    ... def tally():
    ...     i = 0
    ...     while True:
    ...         print('Thing number %s is %s.' % (i, (yield)))
    ...         i += 1
    ...
    >>> t = tally()
    >>> t.send('red')
    Thing number 0 is red.
    >>> t.send('fish')
    Thing number 1 is fish.

    Without the decorator, you would have to call ``next(t)`` before
    ``t.send()`` could be used.

    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return wrapper

def ilen(iterable):
    """Return the number of items in ``iterable``.

    >>> ilen(x for x in range(1000000) if x % 3 == 0)
    333334

    This consumes the iterable, so handle with care.

    """
    d = deque(enumerate(iterable, 1), maxlen=1)
    return d[0][0] if d else 0

def iterate(func, start):
    """Return ``start`` , ``func(start)`` , ``func(func(start))`` , ...

    >>> from itertools import islice
    >>> list(islice(iterate(lambda x: 2*x, 1), 10))
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

    """
    while True:
        yield start
        start = func(start)

def with_iter(context_manager):
```



LGTM!
:-D

TL;DR;LGTM

TL;DR;LGTM

prose overview of patch

TL;DR;LGTM

prose overview of patch

long commit messages

TL;DR;LGTM

prose overview of patch

long commit messages

small commits

TL;DR;LGTM

prose overview of patch

long commit messages

small commits

comments, docstrings, naming

TL;DR;LGTM

GitX

The screenshot shows the GitX application window for a repository named 'sphinx-js' on the 'temp' branch. The main area displays a diff for the file 'tox.ini'. The diff shows changes to the '[tox]' and '[testenv]' sections. The '[tox]' section has two lines: '-envlist = py27, py33' (removed, red) and '+envlist = py27, py36' (added, green). The '[testenv]' section has three lines: '-commands = nosetests' (removed, red), '-deps = nose' (removed, red), and '+commands = python setup.py test' (added, green). Buttons for 'Stage', 'Discard', and 'Stage lines' are visible next to the diff lines.

The bottom section of the window is divided into three panels: 'Unstaged Changes', 'Commit Message', and 'Staged Changes'. The 'Unstaged Changes' panel lists files: '.gitignore', 'setup.py', and 'tox.ini'. The 'Commit Message' panel contains the text: 'Bump Python 3 ~~tox~~ env to 3.6.'. The 'Staged Changes' panel is currently empty.

At the bottom of the window, there are three buttons: 'Amend' (with an unchecked checkbox), 'Sign-Off', and 'Commit'.

TL;DR; LGTM

GitX

The screenshot shows the GitX application window for a repository named 'sphinx-js' on the 'temp' branch. The main area displays a diff for the file 'tox.ini'. The diff shows the following changes:

```
@@ -1,6 +1,5 @@
1 1 [tox]
2  -envlist = py27, py33
2  +envlist = py27, py36
3 3
4 4 [testenv]
5  -commands = nosetests
6  -deps = nose
5  +commands = python setup.py test
```

The diff is color-coded: the new line '+envlist = py27, py36' is highlighted in blue, the removed lines '-envlist = py27, py33' and '-commands = nosetests' are highlighted in red, and the new line '+commands = python setup.py test' is highlighted in green. A red arrow points to the 'Stage lines' button next to the blue-highlighted line. Other buttons visible are 'Stage' and 'Discard'.

The bottom of the window is divided into three panels:

- Unstaged Changes:** Lists files that have been modified but not yet staged: .gitignore, setup.py, and tox.ini.
- Commit Message:** Contains the text 'Bump Python 3 tox env to 3.6.' with 'tox' underlined in red.
- Staged Changes:** Currently empty.

At the bottom of the window, there are three buttons: 'Amend' (with an unchecked checkbox), 'Sign-Off', and 'Commit'.

TL;DR; LGTM

FileMerge

8r8k09_renderers.py - /var/folders/39/cttgwlvj0rs_75t1z11pltrm0000gn/T

```
class JsRenderer
from collections import OrderedDict
from os.path import dirname, join
from re import sub

from docutils.parsers.rst import Parser as RstParser
from docutils.utils import new_document
from Jinja2 import Environment, PackageLoader
from six import iteritems
from sphinx.ext.autodoc import ALL

class JsRenderer(object):
    """Abstract superclass for renderers of various sphinx-js directives

    Provides an inversion-of-control framework for rendering and bridges us
    from the hidden, closed-over JsDirective subclasses to top-level classes
    that can see and use each other.

    """

    def __init__(self, directive, app):
        """
        :arg directive: The associated Sphinx directive
        :arg app: The Sphinx global app object. Some methods need this.
        """

        # content, arguments, options, app: all need to be accessible to
        # template_vars, so we bring them in on construction and stow them away
        # on the instance so calls to template_vars don't need to concern
        # themselves with what it needs.
        self._arguments = directive.arguments
        self._content = directive.content
        self._options = directive.options
        self._directive = directive
        self._app = app

    def rst_nodes(self):
        """Render into RST nodes a thing shaped like a function, having a name
        and arguments.

        Fill in args, docstrings, and info fields from stored JSdoc output.

        """
        # Get the relevant documentation together:
        name = self._name()
        doclet = self._app._sphinxjs_doclets_by_longname.get(name)
        if doclet is None:
            app.warn('No JSdoc documentation for the longname "%s" was found.' &
                    name)
            return []
        rst = self.rst(name, doclet, use_short_name='short-name' in self._options)

        # Parse the RST into docutils nodes with a fresh doc, and return
        # them.
        #
        # Not sure if passing the settings from the "real" doc is the right
        # thing to do here:
        meta = doclet['meta']
```

2edMZ9_renderers.py - /var/folders/39/cttgwlvj0rs_75t1z11pltrm0000gn/T

```
class JsRenderer
from os.path import dirname, join
from re import sub

from docutils.parsers.rst import Parser as RstParser
from docutils.statemachine import StringList
from docutils.utils import new_document
from Jinja2 import Environment, PackageLoader
from six import iteritems
from sphinx.ext.autodoc import ALL

class JsRenderer(object):
    """Abstract superclass for renderers of various sphinx-js directives

    Provides an inversion-of-control framework for rendering and bridges us
    from the hidden, closed-over JsDirective subclasses to top-level classes
    that can see and use each other.

    """

    def __init__(self, directive, app, arguments=None, content=None, options=None):
        self._directive = directive

        # content, arguments, options, app: all need to be accessible to
        # template_vars, so we bring them in on construction and stow them away
        # on the instance so calls to template_vars don't need to concern
        # themselves with what it needs.
        self._app = app
        self._arguments = arguments or []
        self._content = content or StringList()
        self._options = options or {}

    @classmethod
    def from_directive(cls, directive, app):
        """Return one of these whose state is all derived from a directive.

        This is suitable for top-level calls but not for when a renderer is
        being called from a different renderer, lest content and such from the
        outer directive be duplicated in the inner directive.

        :arg directive: The associated Sphinx directive
        :arg app: The Sphinx global app object. Some methods need this.

        """
        return cls(directive,
                  app,
                  arguments=directive.arguments,
                  content=directive.content,
                  options=directive.options)

    def rst_nodes(self):
        """Render into RST nodes a thing shaped like a function, having a name
        and arguments.

        Fill in args, docstrings, and info fields from stored JSdoc output.

        """
```

status: 6 differences

Actions

Nitpicks

```
print 'Hello'
```

Nitpicks

Lowercase please.

Should we be using the Python-3-style parentheses via `import future`?

```
print 'Hello'
```

If we use a logging framework, we have the advantage of levels.

`l8n?`

Too intimate a greeting, I think

Nitpicks

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path'][0]): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
            path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                             content_highlighters)))
            for line in lines])
print 'Hello'
```

Nitpicks

Should be aligned
with "h" above

Line too long

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path']): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
            path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))]
           for line in lines])
print 'Hello'
```

Some rogue
camelCase escaped.

Too intimate a
greeting, I think

If we use a logging
framework, we have the
advantage of levels.

Nitpicks

Should be aligned
with "h" above

Line too long

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path']): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
                               path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))]
           for line in lines])
print 'Hello'
```

Some rogue
camelCase escaped.

Nitpicks

Should be aligned
with “h” above

Line too long

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path']): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
            path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))]
           for line in lines])
print 'Hello'
```

Some rogue
camelCase escaped.

PEP 8, PEP 257, Pycodestyle style guide, Sphinx

Nitpicks

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path'][0]): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
                               path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))
            for line in lines])
print 'Hello'
```

PEP 8, PEP 257, Pycocoo style guide, Sphinx

Nitpicks

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path'][0]): # noqa: E234
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123
                               path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))
           for line in lines])
print 'Hello'
```

PEP 8, PEP 257, Pycodestyle style guide, Sphinx
flake8

Nitpicks

```
# Group lines into files:
for path, lines in groupby(results, lambda r: r['path'][0]): # noqa: E234 🍌
    lines = list(lines)
    highlit_path = highlight( # noqa: E234
        path,
        chain.from_iterable((h(lines[0]) for h in # noqa: E123 🍌
                               path_highlighters)))
    here_is_some_new(code, that.is_ridiculously_longer_than(the_surrounding_code)).and_thus(really).distracting("isn't it?")
    icon_for_path = icon(path)
    yield (icon_for_path,
           highlit_path,
           [(line['number'][0],
             highlight(line['content'][0].rstrip('\n\r'),
                       chain.from_iterable(h(line) for h in
                                           content_highlighters)))
            for line in lines])
print 'Hello'
```

PEP 8, PEP 257, Pycodestyle style guide, Sphinx
flake8

While you're at it...

While you're at it...

HaHaOnlySerious

While you're at it...

HaHa~~Only~~Serious

While you're at it...

HaHa~~Only~~Serious

GettingBetter

While you're at it...

HaHa~~Only~~Serious

GettingBetter

~~BeingPerfect~~

Slow Turnarounds

Slow Turnarounds

Energizing

Slow Turnarounds

Energizing

Comprehensiveness not required.

Slow Turnarounds

Energizing

Comprehensiveness not required.

Respect working memory.

Slow Turnarounds

Energizing

Comprehensiveness not required.

Respect working memory.

Quick “no”s

Those

Pesky

Human

Emotions

Insecurity

Insecurity

Insecurity == fear.

Insecurity

Insecurity == fear.

Everybody is wrapped up in themselves.

Insecurity

Insecurity == fear.

Everybody is wrapped up in themselves.

When someone corrects you,
that means you just got smarter.

Insecurity

Insecurity == fear.

Everybody is wrapped up in themselves.

When someone corrects you,
that means you just got smarter.

What are you so afraid of?
What's the worst that can happen?

Feeling Short on Time

Feeling Short on Time

Lower standards.

Feeling Short on Time

Lower standards.

Never sleep.

Feeling Short on Time

Lower standards.

Never sleep.

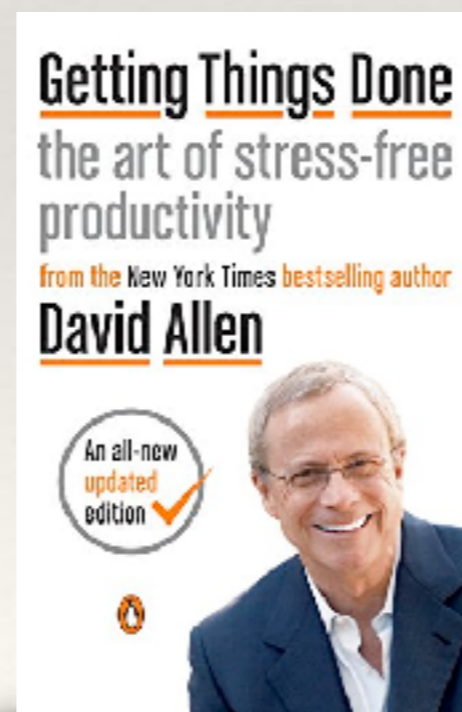
Or pace, prioritize, and peace.

Feeling Short on Time

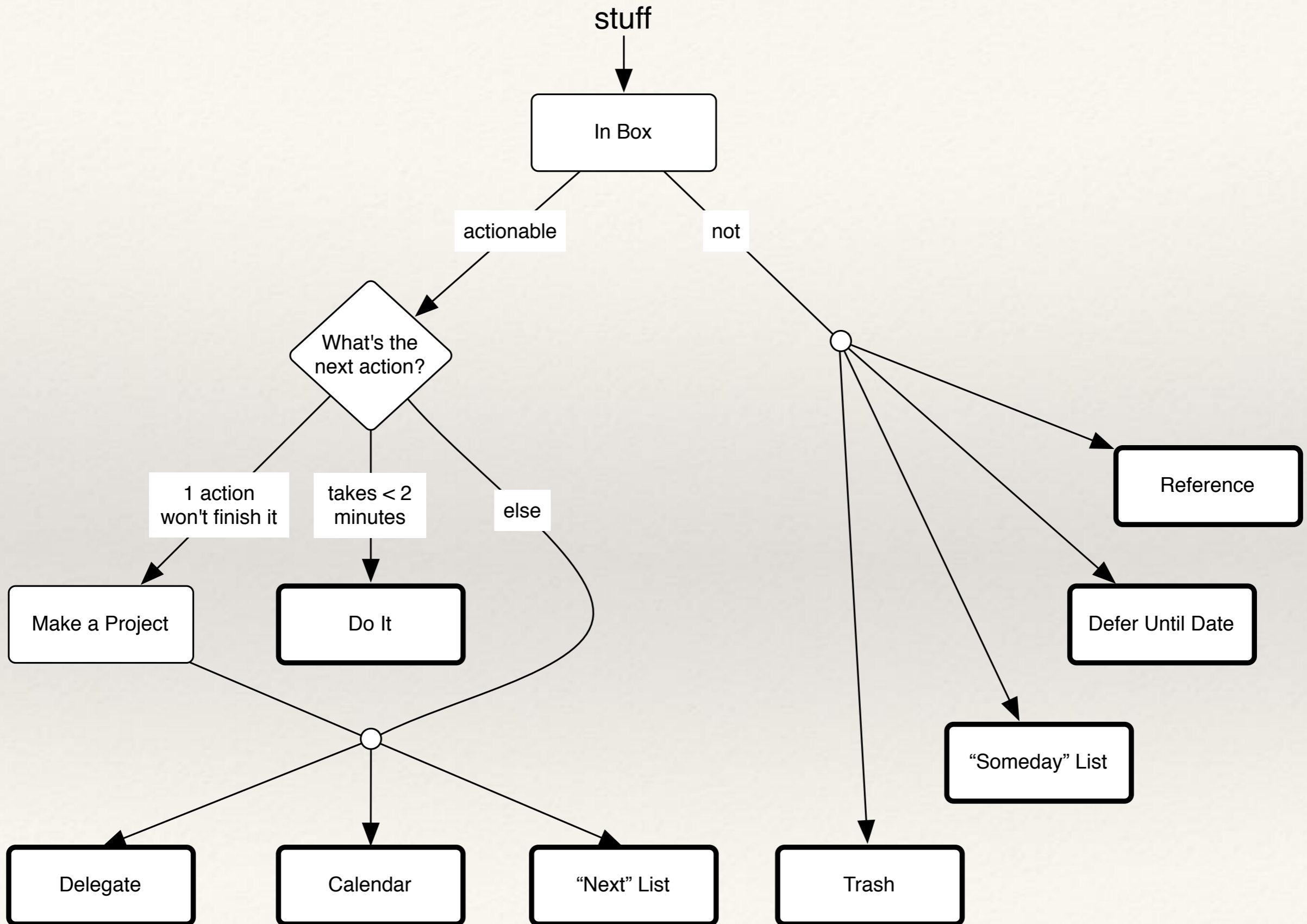
Lower standards.

Never sleep.

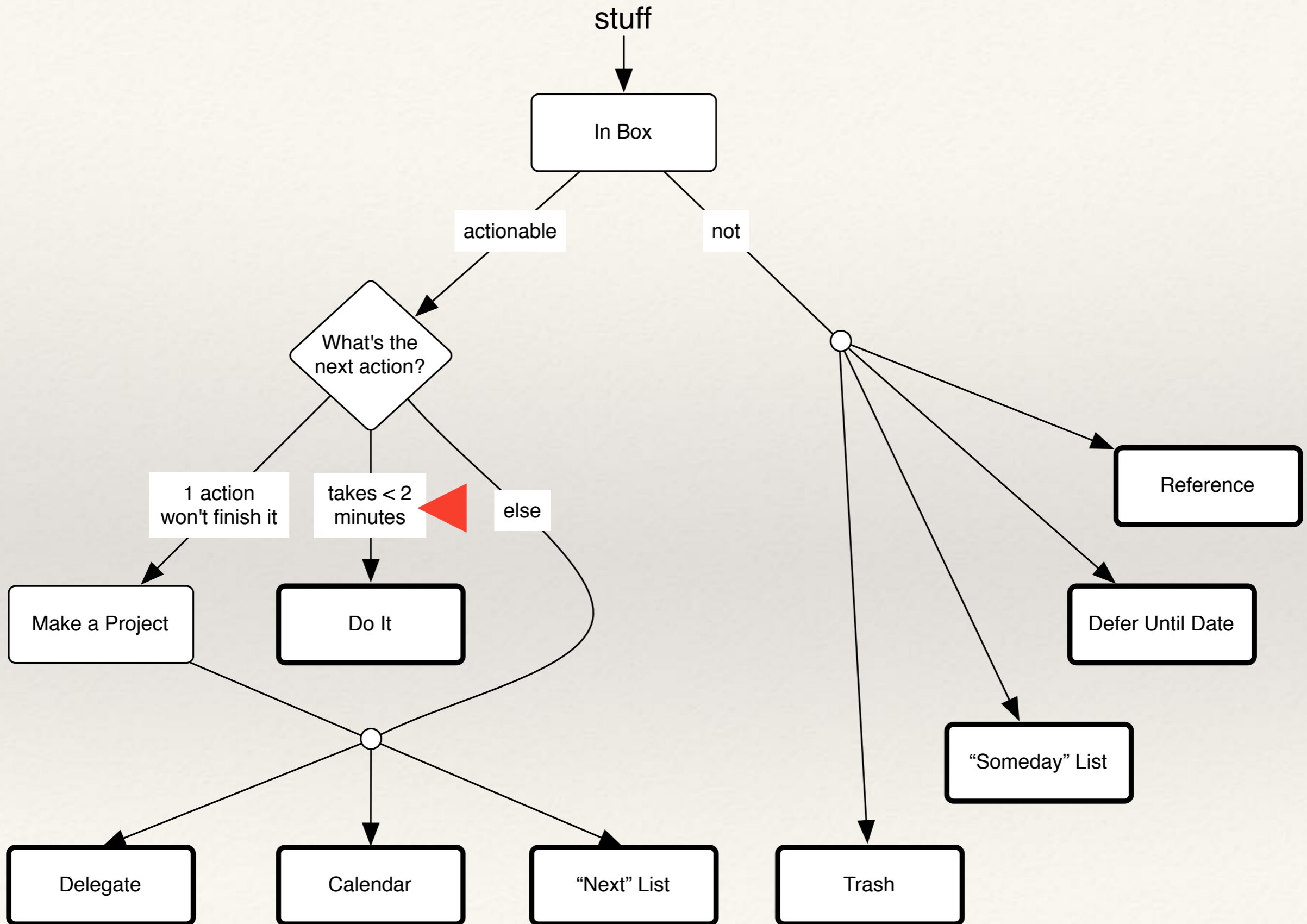
Or pace, prioritize, and peace.



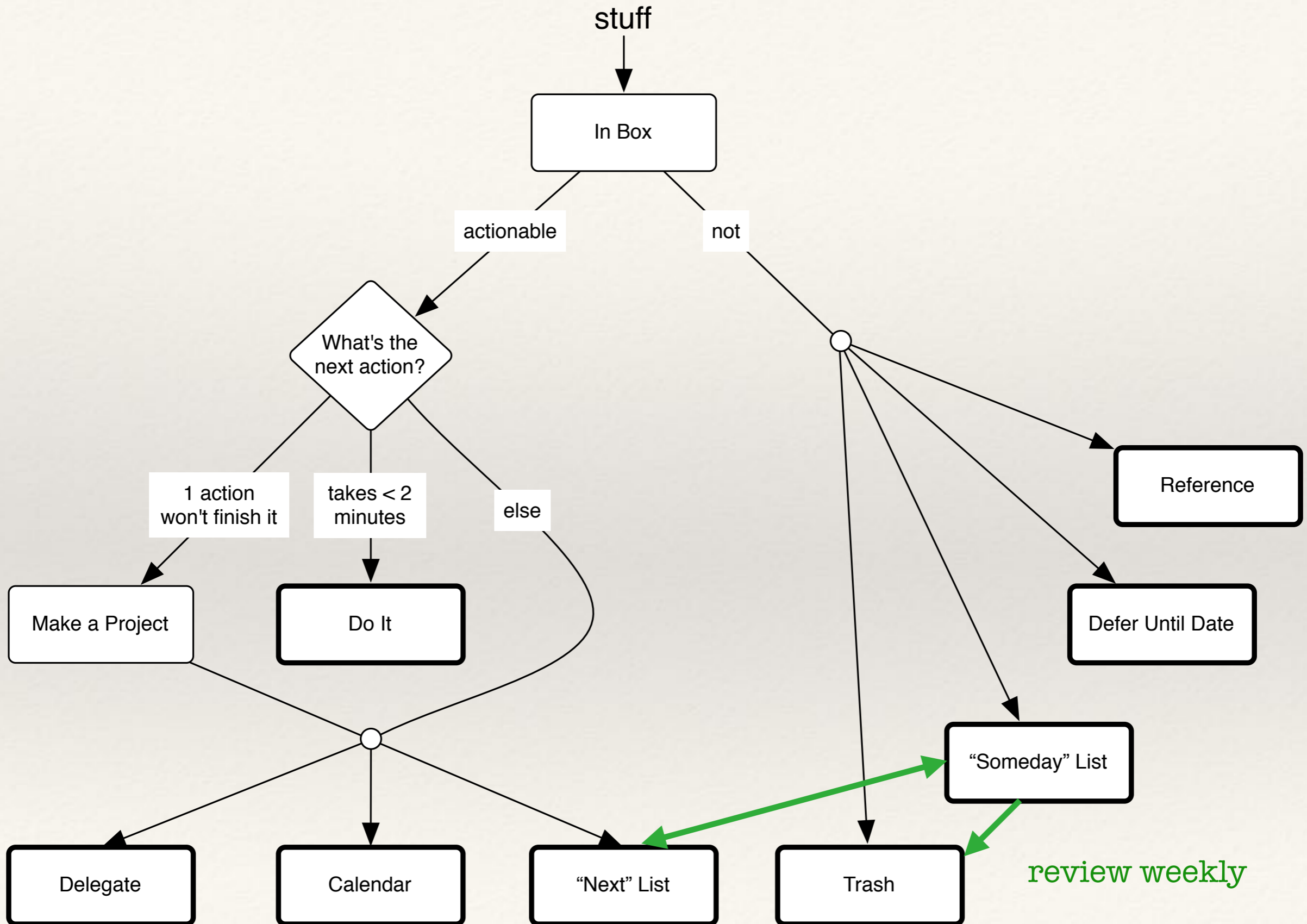
Feeling Short on Time



Feeling Short on Time



Feeling Short on Time



Feeling Short on Time

Feeling Short on Time

Patch-batching

Feeling Short on Time

Patch-batching

Leveling up newcomers

Feeling Short on Time

Patch-batching

Leveling up newcomers **1**

Feeling Short on Time

Patch-batching

Leveling up newcomers **1 2**

Feeling Short on Time

Patch-batching

Leveling up newcomers **1 2 3**

The Trust Bank

Never eat lunch alone.

When all else fails...

When all else fails...

Say what you feel.

When all else fails...

Say what you feel.

Invite people into the decision.

Review Checklist

- Tact hacks
 - Question mark
 - You → we/this
 - Compliments
 - Humor
- Antipatterns
 - TL;DR;LGTM
 - Nitpicks
 - While you're at it...
 - Slow Turnarounds
- Clarity of explanation
- Clarity of expectation
- Pesky Emotions
 - Insecurity
 - Feeling short on time
 - Pace & peace
 - Getting Things Done
 - Patch-batching
 - Leveling up newcomers
 - The trust bank
 - Articulate emotions

erik@mozilla.com · IRC: ErikRose · @ErikRose