```
1: """
2: python -m pelican module entry point to run via python -m
3: """
4:
5: from . import main
6:
7:
8: if __name__ == '__main__':
9:     main()
```

```
# -*- coding: utf-8 -*-

import argparse
import logging
import multiprocessing
import os
import pprint
import sys
import time
import traceback
from collections.abc import Iterable
# Combines all paths to `pelican` package accessible from `sys.path`
# Makes it possible to install `pelican` and namespace plugins into different
# locations in the file system (e.g. pip with `-e` or `--user`)
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)

# pelican.log has to be the first pelican module to be loaded
# because logging.setLoggerClass has to be called before logging.getLogger
from pelican.log import init as init_logging
from pelican.generators import (ArticlesGenerator,  # noqa: I100
                                PagesGenerator, SourceFileGenerator,
                                StaticGenerator, TemplatePagesGenerator)
from pelican.plugins import signals
from pelican.plugins._utils import load_plugins
from pelican.readers import Readers
from pelican.server import ComplexHTTPRequestHandler, RootedHTTPServer
from pelican.settings import read_settings
from pelican.utils import (clean_output_dir, file_watcher,
                           folder_watcher, maybe_pluralize)
from pelican.writers import Writer

try:
    __version__ = __import__('pkg_resources') \
        .get_distribution('pelican').version
except Exception:
    __version__ = "unknown"

DEFAULT_CONFIG_NAME = 'pelicanconf.py'
logger = logging.getLogger(__name__)


class Pelican(object):

    def __init__(self, settings):
        """Pelican initialisation

        Performs some checks on the environment before doing anything else.
        """

        # define the default settings
        self.settings = settings

        self.path = settings['PATH']
        self.theme = settings['THEME']
        self.output_path = settings['OUTPUT_PATH']
        self.ignore_files = settings['IGNORE_FILES']
        self.delete_outputdir = settings['DELETE_OUTPUT_DIRECTORY']
        self.output_retention = settings['OUTPUT_RETENTION']

        self.init_path()
```

```
62:         self.init_plugins()
63:         signals.initialized.send(self)
64:
65:     def init_path(self):
66:         if not any(p in sys.path for p in ['', os.curdir]):
67:             logger.debug("Adding current directory to system path")
68:             sys.path.insert(0, '')
69:
70:     def init_plugins(self):
71:         self.plugins = load_plugins(self.settings)
72:         for plugin in self.plugins:
73:             logger.debug('Registering plugin `%s`', plugin.__name__)
74:             try:
75:                 plugin.register()
76:             except Exception as e:
77:                 logger.error('Cannot register plugin `%s`\n%s',
78:                              plugin.__name__, e)
79:
80:     def run(self):
81:         """Run the generators and return"""
82:         start_time = time.time()
83:
84:         context = self.settings.copy()
85:         # Share these among all the generators and content objects
86:         # They map source paths to Content objects or None
87:         context['generated_content'] = {}
88:         context['static_links'] = set()
89:         context['static_content'] = {}
90:         context['localsiteurl'] = self.settings['SITEURL']
91:
92:         generators = [
93:             cls(
94:                 context=context,
95:                 settings=self.settings,
96:                 path=self.path,
97:                 theme=self.theme,
98:                 output_path=self.output_path,
99:             ) for cls in self.get_generator_classes()
100:         ]
101:
102:         # Delete the output directory if (1) the appropriate setting is True
103:         # and (2) that directory is not the parent of the source directory
104:         if (self.delete_outputdir
105:                 and os.path.commonpath([self.output_path]) !=
106:                 os.path.commonpath([self.output_path, self.path])):
107:             clean_output_dir(self.output_path, self.output_retention)
108:
109:         for p in generators:
110:             if hasattr(p, 'generate_context'):
111:                 p.generate_context()
112:
113:         for p in generators:
114:             if hasattr(p, 'refresh_metadata_intersite_links'):
115:                 p.refresh_metadata_intersite_links()
116:
117:         signals.all_generators_finalized.send(generators)
118:
119:         writer = self.get_writer()
120:
121:         for p in generators:
122:             if hasattr(p, 'generate_output'):
```

```
123:                 p.generate_output(writer)
124:
125:         signals.finalized.send(self)
126:
127:         articles_generator = next(g for g in generators
128:                                   if isinstance(g, ArticlesGenerator))
129:         pages_generator = next(g for g in generators
130:                                if isinstance(g, PagesGenerator))
131:
132:         pluralized_articles = maybe_pluralize(
133:             (len(articles_generator.articles) +
134:              len(articles_generator.translations)),
135:             'article',
136:             'articles')
137:         pluralized_drafts = maybe_pluralize(
138:             (len(articles_generator.drafts) +
139:              len(articles_generator.drafts_translations)),
140:             'draft',
141:             'drafts')
142:         pluralized_pages = maybe_pluralize(
143:             (len(pages_generator.pages) +
144:              len(pages_generator.translations)),
145:             'page',
146:             'pages')
147:         pluralized_hidden_pages = maybe_pluralize(
148:             (len(pages_generator.hidden_pages) +
149:              len(pages_generator.hidden_translations)),
150:             'hidden page',
151:             'hidden pages')
152:         pluralized_draft_pages = maybe_pluralize(
153:             (len(pages_generator.draft_pages) +
154:              len(pages_generator.draft_translations)),
155:             'draft page',
156:             'draft pages')
157:
158:         print('Done: Processed {}, {}, {}, {} and {} in {:.2f} seconds.'
159:               .format(
160:                     pluralized_articles,
161:                     pluralized_drafts,
162:                     pluralized_pages,
163:                     pluralized_hidden_pages,
164:                     pluralized_draft_pages,
165:                     time.time() - start_time))
166:
167:     def get_generator_classes(self):
168:         generators = [ArticlesGenerator, PagesGenerator]
169:
170:         if self.settings['TEMPLATE_PAGES']:
171:             generators.append(TemplatePagesGenerator)
172:         if self.settings['OUTPUT_SOURCES']:
173:             generators.append(SourceFileGenerator)
174:
175:         for pair in signals.get_generators.send(self):
176:             (funct, value) = pair
177:
178:             if not isinstance(value, Iterable):
179:                 value = (value, )
180:
181:             for v in value:
182:                 if isinstance(v, type):
183:                     logger.debug('Found generator: %s', v)
```

```
184:                     generators.append(v)
185:
186:         # StaticGenerator must run last, so it can identify files that
187:         # were skipped by the other generators, and so static files can
188:         # have their output paths overridden by the {attach} link syntax.
189:         generators.append(StaticGenerator)
190:         return generators
191:
192:     def get_writer(self):
193:         writers = [w for (_, w) in signals.get_writer.send(self)
194:                    if isinstance(w, type)]
195:         writers_found = len(writers)
196:         if writers_found == 0:
197:             return Writer(self.output_path, settings=self.settings)
198:         else:
199:             writer = writers[0]
200:             if writers_found == 1:
201:                 logger.debug('Found writer: %s', writer)
202:             else:
203:                 logger.warning(
204:                     '%s writers found, using only first one: %s',
205:                     writers_found, writer)
206:             return writer(self.output_path, settings=self.settings)
207:
208:
209: class PrintSettings(argparse.Action):
210:     def __call__(self, parser, namespace, values, option_string):
211:         instance, settings = get_instance(namespace)
212:
213:         if values:
214:             # One or more arguments provided, so only print those settings
215:             for setting in values:
216:                 if setting in settings:
217:                     # Only add newline between setting name and value if dict
218:                     if isinstance(settings[setting], dict):
219:                         setting_format = '\n{}:\n{}'
220:                     else:
221:                         setting_format = '\n{}: {}'
222:                     print(setting_format.format(
223:                         setting,
224:                         pprint.pformat(settings[setting])))
225:                 else:
226:                     print('\n{} is not a recognized setting.'.format(setting))
227:                     break
228:         else:
229:             # No argument was given to --print-settings, so print all settings
230:             pprint.pprint(settings)
231:
232:         parser.exit()
233:
234:
235: def parse_arguments(argv=None):
236:     parser = argparse.ArgumentParser(
237:         description='A tool to generate a static blog, '
238:                     ' with restructured text input files.',
239:         formatter_class=argparse.ArgumentDefaultsHelpFormatter
240:     )
241:
242:     parser.add_argument(dest='path', nargs='?',
243:                         help='Path where to find the content files.',
244:                         default=None)
```

```
245:
246:     parser.add_argument('-t', '--theme-path', dest='theme',
247:                         help='Path where to find the theme templates. If not '
248:                         'specified, it will use the default one included with '
249:                         'pelican.')
250:
251:     parser.add_argument('-o', '--output', dest='output',
252:                         help='Where to output the generated files. If not '
253:                         'specified, a directory will be created, named '
254:                         '"output" in the current path.')
255:
256:     parser.add_argument('-s', '--settings', dest='settings',
257:                         help='The settings of the application, this is '
258:                         'automatically set to {0} if a file exists with this '
259:                         'name.'.format(DEFAULT_CONFIG_NAME))
260:
261:     parser.add_argument('-d', '--delete-output-directory',
262:                         dest='delete_outputdir', action='store_true',
263:                         default=None, help='Delete the output directory.')
264:
265:     parser.add_argument('-v', '--verbose', action='store_const',
266:                         const=logging.INFO, dest='verbosity',
267:                         help='Show all messages.')
268:
269:     parser.add_argument('-q', '--quiet', action='store_const',
270:                         const=logging.CRITICAL, dest='verbosity',
271:                         help='Show only critical errors.')
272:
273:     parser.add_argument('-D', '--debug', action='store_const',
274:                         const=logging.DEBUG, dest='verbosity',
275:                         help='Show all messages, including debug messages.')
276:
277:     parser.add_argument('--version', action='version', version=__version__,
278:                         help='Print the pelican version and exit.')
279:
280:     parser.add_argument('-r', '--autoreload', dest='autoreload',
281:                         action='store_true',
282:                         help='Relaunch pelican each time a modification occurs'
283:                         ' on the content files.')
284:
285:     parser.add_argument('--print-settings', dest='print_settings', nargs='*',
286:                         action=PrintSettings, metavar='SETTING_NAME',
287:                         help='Print current configuration settings and exit. '
288:                         'Append one or more setting name arguments to see the '
289:                         'values for specific settings only.')
290:
291:     parser.add_argument('--relative-urls', dest='relative_paths',
292:                         action='store_true',
293:                         help='Use relative urls in output, '
294:                              'useful for site development')
295:
296:     parser.add_argument('--cache-path', dest='cache_path',
297:                         help=('Directory in which to store cache files. '
298:                               'If not specified, defaults to "cache".'))
299:
300:     parser.add_argument('--ignore-cache', action='store_true',
301:                         dest='ignore_cache', help='Ignore content cache '
302:                         'from previous runs by not loading cache files.')
303:
304:     parser.add_argument('-w', '--write-selected', type=str,
305:                         dest='selected_paths', default=None,
```

```
306:                         help='Comma separated list of selected paths to write')
307:
308:     parser.add_argument('--fatal', metavar='errors|warnings',
309:                         choices=('errors', 'warnings'), default='',
310:                         help=('Exit the program with non-zero status if any '
311:                               'errors/warnings encountered.'))
312:
313:     parser.add_argument('--logs-dedup-min-level', default='WARNING',
314:                         choices=('DEBUG', 'INFO', 'WARNING', 'ERROR'),
315:                         help=('Only enable log de-duplication for levels equal'
316:                               ' to or above the specified value'))
317:
318:     parser.add_argument('-l', '--listen', dest='listen', action='store_true',
319:                         help='Serve content files via HTTP and port 8000.')
320:
321:     parser.add_argument('-p', '--port', dest='port', type=int,
322:                         help='Port to serve HTTP files at. (default: 8000)')
323:
324:     parser.add_argument('-b', '--bind', dest='bind',
325:                         help='IP to bind to when serving files via HTTP '
326:                         '(default: 127.0.0.1)')
327:
328:     args = parser.parse_args(argv)
329:
330:     if args.port is not None and not args.listen:
331:         logger.warning('--port without --listen has no effect')
332:     if args.bind is not None and not args.listen:
333:         logger.warning('--bind without --listen has no effect')
334:
335:     return args
336:
337:
338: def get_config(args):
339:     config = {}
340:     if args.path:
341:         config['PATH'] = os.path.abspath(os.path.expanduser(args.path))
342:     if args.output:
343:         config['OUTPUT_PATH'] = \
344:             os.path.abspath(os.path.expanduser(args.output))
345:     if args.theme:
346:         abstheme = os.path.abspath(os.path.expanduser(args.theme))
347:         config['THEME'] = abstheme if os.path.exists(abstheme) else args.theme
348:     if args.delete_outputdir is not None:
349:         config['DELETE_OUTPUT_DIRECTORY'] = args.delete_outputdir
350:     if args.ignore_cache:
351:         config['LOAD_CONTENT_CACHE'] = False
352:     if args.cache_path:
353:         config['CACHE_PATH'] = args.cache_path
354:     if args.selected_paths:
355:         config['WRITE_SELECTED'] = args.selected_paths.split(',')
356:     if args.relative_paths:
357:         config['RELATIVE_URLS'] = args.relative_paths
358:     if args.port is not None:
359:         config['PORT'] = args.port
360:     if args.bind is not None:
361:         config['BIND'] = args.bind
362:     config['DEBUG'] = args.verbosity == logging.DEBUG
363:
364:     return config
365:
366:
```

```
367: def get_instance(args):
368:
369:     config_file = args.settings
370:     if config_file is None and os.path.isfile(DEFAULT_CONFIG_NAME):
371:         config_file = DEFAULT_CONFIG_NAME
372:         args.settings = DEFAULT_CONFIG_NAME
373:
374:     settings = read_settings(config_file, override=get_config(args))
375:
376:     cls = settings['PELICAN_CLASS']
377:     if isinstance(cls, str):
378:         module, cls_name = cls.rsplit('.', 1)
379:         module = __import__(module)
380:         cls = getattr(module, cls_name)
381:
382:     return cls(settings), settings
383:
384:
385: def autoreload(watchers, args, old_static, reader_descs, excqueue=None):
386:     while True:
387:         try:
388:             # Check source dir for changed files ending with the given
389:             # extension in the settings. In the theme dir is no such
390:             # restriction; all files are recursively checked if they
391:             # have changed, no matter what extension the filenames
392:             # have.
393:             modified = {k: next(v) for k, v in watchers.items()}
394:
395:             if modified['settings']:
396:                 pelican, settings = get_instance(args)
397:
398:                 # Adjust static watchers if there are any changes
399:                 new_static = settings.get("STATIC_PATHS", [])
400:
401:                 # Added static paths
402:                 # Add new watchers and set them as modified
403:                 new_watchers = set(new_static).difference(old_static)
404:                 for static_path in new_watchers:
405:                     static_key = '[static]%s' % static_path
406:                     watchers[static_key] = folder_watcher(
407:                         os.path.join(pelican.path, static_path),
408:                         [''],
409:                         pelican.ignore_files)
410:                     modified[static_key] = next(watchers[static_key])
411:
412:                 # Removed static paths
413:                 # Remove watchers and modified values
414:                 old_watchers = set(old_static).difference(new_static)
415:                 for static_path in old_watchers:
416:                     static_key = '[static]%s' % static_path
417:                     watchers.pop(static_key)
418:                     modified.pop(static_key)
419:
420:                 # Replace old_static with the new one
421:                 old_static = new_static
422:
423:             if any(modified.values()):
424:                 print('\n-> Modified: {}. re-generating...'.format(
425:                     ', '.join(k for k, v in modified.items() if v)))
426:
427:                 if modified['content'] is None:
```

```
428:                     logger.warning(
429:                         'No valid files found in content for '
430:                         + 'the active readers:\n'
431:                         + '\n'.join(reader_descs))
432:
433:                 if modified['theme'] is None:
434:                     logger.warning('Empty theme folder. Using `basic` '
435:                                    'theme.')
436:
437:                 pelican.run()
438:
439:         except KeyboardInterrupt as e:
440:             logger.warning("Keyboard interrupt, quitting.")
441:             if excqueue is not None:
442:                 excqueue.put(traceback.format_exception_only(type(e), e)[-1])
443:             return
444:
445:         except Exception as e:
446:             if (args.verbosity == logging.DEBUG):
447:                 if excqueue is not None:
448:                     excqueue.put(
449:                         traceback.format_exception_only(type(e), e)[-1])
450:                 else:
451:                     raise
452:             logger.warning(
453:                 'Caught exception "%s". Reloading.', e)
454:
455:         finally:
456:             time.sleep(.5)  # sleep to avoid cpu load
457:
458:
459: def listen(server, port, output, excqueue=None):
460:     RootedHTTPServer.allow_reuse_address = True
461:     try:
462:         httpd = RootedHTTPServer(
463:             output, (server, port), ComplexHTTPRequestHandler)
464:     except OSError as e:
465:         logging.error("Could not listen on port %s, server %s.", port, server)
466:         if excqueue is not None:
467:             excqueue.put(traceback.format_exception_only(type(e), e)[-1])
468:         return
469:
470:     try:
471:         print("\nServing site at: {}:{} - Tap CTRL-C to stop".format(
472:             server, port))
473:         httpd.serve_forever()
474:     except Exception as e:
475:         if excqueue is not None:
476:             excqueue.put(traceback.format_exception_only(type(e), e)[-1])
477:         return
478:
479:     except KeyboardInterrupt:
480:         print("\nKeyboard interrupt received. Shutting down server.")
481:         httpd.socket.close()
482:
483:
484: def main(argv=None):
485:     args = parse_arguments(argv)
486:     logs_dedup_min_level = getattr(logging, args.logs_dedup_min_level)
487:     init_logging(args.verbosity, args.fatal,
488:                  logs_dedup_min_level=logs_dedup_min_level)
```

```
489:
490:     logger.debug('Pelican version: %s', __version__)
491:     logger.debug('Python version: %s', sys.version.split()[0])
492:
493:     try:
494:         pelican, settings = get_instance(args)
495:
496:         readers = Readers(settings)
497:         reader_descs = sorted(set(['%s (%s)' %
498:                                    (type(r).__name__,
499:                                     ', '.join(r.file_extensions))
500:                                    for r in readers.readers.values()
501:                                    if r.enabled]))
502:
503:         watchers = {'content': folder_watcher(pelican.path,
504:                                               readers.extensions,
505:                                               pelican.ignore_files),
506:                     'theme': folder_watcher(pelican.theme,
507:                                             [''],
508:                                             pelican.ignore_files),
509:                     'settings': file_watcher(args.settings)}
510:
511:         old_static = settings.get("STATIC_PATHS", [])
512:         for static_path in old_static:
513:             # use a prefix to avoid possible overriding of standard watchers
514:             # above
515:             watchers['[static]%s' % static_path] = folder_watcher(
516:                 os.path.join(pelican.path, static_path),
517:                 [''],
518:                 pelican.ignore_files)
519:
520:         if args.autoreload and args.listen:
521:             excqueue = multiprocessing.Queue()
522:             p1 = multiprocessing.Process(
523:                 target=autoreload,
524:                 args=(watchers, args, old_static, reader_descs, excqueue))
525:             p2 = multiprocessing.Process(
526:                 target=listen,
527:                 args=(settings.get('BIND'), settings.get('PORT'),
528:                       settings.get("OUTPUT_PATH"), excqueue))
529:             p1.start()
530:             p2.start()
531:             exc = excqueue.get()
532:             p1.terminate()
533:             p2.terminate()
534:             logger.critical(exc)
535:         elif args.autoreload:
536:             print('  --- AutoReload Mode: Monitoring `content`, `theme` and'
537:                   ' `settings` for changes. ---')
538:             autoreload(watchers, args, old_static, reader_descs)
539:         elif args.listen:
540:             listen(settings.get('BIND'), settings.get('PORT'),
541:                    settings.get("OUTPUT_PATH"))
542:         else:
543:             if next(watchers['content']) is None:
544:                 logger.warning(
545:                     'No valid files found in content for '
546:                     + 'the active readers:\n'
547:                     + '\n'.join(reader_descs))
548:
549:             if next(watchers['theme']) is None:
```

```
550:                 logger.warning('Empty theme folder. Using `basic` theme.')
551:
552:             pelican.run()
553:
554:     except Exception as e:
555:         logger.critical('%s', e)
556:
557:         if args.verbosity == logging.DEBUG:
558:             raise
559:         else:
560:             sys.exit(getattr(e, 'exitcode', 1))
```

```
# -*- coding: utf-8 -*-

import calendar
import errno
import fnmatch
import logging
import os
from collections import defaultdict
from functools import partial
from itertools import chain, groupby
from operator import attrgetter

from jinja2 import (BaseLoader, ChoiceLoader, Environment, FileSystemLoader,
                    PrefixLoader, TemplateNotFound)

from pelican.cache import FileStampDataCacher
from pelican.contents import Article, Page, Static
from pelican.plugins import signals
from pelican.readers import Readers
from pelican.utils import (DateFormatter, copy, mkdir_p, order_content,
                           posixize_path, process_translations)


logger = logging.getLogger(__name__)


class PelicanTemplateNotFound(Exception):
    pass


class Generator(object):
    """Baseclass generator"""

    def __init__(self, context, settings, path, theme, output_path,
                 readers_cache_name='', **kwargs):
        self.context = context
        self.settings = settings
        self.path = path
        self.theme = theme
        self.output_path = output_path

        for arg, value in kwargs.items():
            setattr(self, arg, value)

        self.readers = Readers(self.settings, readers_cache_name)

        # templates cache
        self._templates = {}
        self._templates_path = list(self.settings['THEME_TEMPLATES_OVERRIDES'])

        theme_templates_path = os.path.expanduser(
            os.path.join(self.theme, 'templates'))
        self._templates_path.append(theme_templates_path)
        theme_loader = FileSystemLoader(theme_templates_path)

        simple_theme_path = os.path.dirname(os.path.abspath(__file__))
        simple_loader = FileSystemLoader(
            os.path.join(simple_theme_path, "themes", "simple", "templates"))

        self.env = Environment(
            loader=ChoiceLoader([
```

```
62:                 FileSystemLoader(self._templates_path),
63:                 simple_loader,  # implicit inheritance
64:                 PrefixLoader({
65:                     '!simple': simple_loader,
66:                     '!theme': theme_loader
67:                 })  # explicit ones
68:             ]),
69:             **self.settings['JINJA_ENVIRONMENT']
70:         )
71:
72:         logger.debug('Template list: %s', self.env.list_templates())
73:
74:         # provide utils.strftime as a jinja filter
75:         self.env.filters.update({'strftime': DateFormatter()})
76:
77:         # get custom Jinja filters from user settings
78:         custom_filters = self.settings['JINJA_FILTERS']
79:         self.env.filters.update(custom_filters)
80:
81:         # get custom Jinja globals from user settings
82:         custom_globals = self.settings['JINJA_GLOBALS']
83:         self.env.globals.update(custom_globals)
84:
85:         # get custom Jinja tests from user settings
86:         custom_tests = self.settings['JINJA_TESTS']
87:         self.env.tests.update(custom_tests)
88:
89:         signals.generator_init.send(self)
90:
91:     def get_template(self, name):
92:         """Return the template by name.
93:         Use self.theme to get the templates to use, and return a list of
94:         templates ready to use with Jinja2.
95:         """
96:         if name not in self._templates:
97:             for ext in self.settings['TEMPLATE_EXTENSIONS']:
98:                 try:
99:                     self._templates[name] = self.env.get_template(name + ext)
100:                    break
101:                except TemplateNotFound:
102:                    continue
103:
104:            if name not in self._templates:
105:                raise PelicanTemplateNotFound(
106:                    '[templates] unable to load {}[{}] from {}'.format(
107:                        name, ', '.join(self.settings['TEMPLATE_EXTENSIONS']),
108:                        self._templates_path))
109:
110:        return self._templates[name]
111:
112:    def _include_path(self, path, extensions=None):
113:        """Inclusion logic for .get_files(), returns True/False
114:
115:        :param path: the path which might be including
116:        :param extensions: the list of allowed extensions, or False if all
117:            extensions are allowed
118:        """
119:        if extensions is None:
120:            extensions = tuple(self.readers.extensions)
121:        basename = os.path.basename(path)
122:
```

```
123:         # check IGNORE_FILES
124:         ignores = self.settings['IGNORE_FILES']
125:         if any(fnmatch.fnmatch(basename, ignore) for ignore in ignores):
126:             return False
127:
128:         ext = os.path.splitext(basename)[1][1:]
129:         if extensions is False or ext in extensions:
130:             return True
131:
132:         return False
133:
134:     def get_files(self, paths, exclude=[], extensions=None):
135:         """Return a list of files to use, based on rules
136:
137:         :param paths: the list pf paths to search (relative to self.path)
138:         :param exclude: the list of path to exclude
139:         :param extensions: the list of allowed extensions (if False, all
140:             extensions are allowed)
141:         """
142:         # backward compatibility for older generators
143:         if isinstance(paths, str):
144:             paths = [paths]
145:
146:         # group the exclude dir names by parent path, for use with os.walk()
147:         exclusions_by_dirpath = {}
148:         for e in exclude:
149:             parent_path, subdir = os.path.split(os.path.join(self.path, e))
150:             exclusions_by_dirpath.setdefault(parent_path, set()).add(subdir)
151:
152:         files = set()
153:         ignores = self.settings['IGNORE_FILES']
154:         for path in paths:
155:             # careful: os.path.join() will add a slash when path == ''.
156:             root = os.path.join(self.path, path) if path else self.path
157:
158:             if os.path.isdir(root):
159:                 for dirpath, dirs, temp_files in os.walk(
160:                         root, topdown=True, followlinks=True):
161:                     excl = exclusions_by_dirpath.get(dirpath, ())
162:                     # We copy the `dirs` list as we will modify it in the loop:
163:                     for d in list(dirs):
164:                         if (d in excl or
165:                             any(fnmatch.fnmatch(d, ignore)
166:                                 for ignore in ignores)):
167:                             if d in dirs:
168:                                 dirs.remove(d)
169:
170:                     reldir = os.path.relpath(dirpath, self.path)
171:                     for f in temp_files:
172:                         fp = os.path.join(reldir, f)
173:                         if self._include_path(fp, extensions):
174:                             files.add(fp)
175:             elif os.path.exists(root) and self._include_path(path, extensions):
176:                 files.add(path)  # can't walk non-directories
177:         return files
178:
179:     def add_source_path(self, content, static=False):
180:         """Record a source file path that a Generator found and processed.
181:         Store a reference to its Content object, for url lookups later.
182:         """
183:         location = content.get_relative_source_path()
```

```
184:         key = 'static_content' if static else 'generated_content'
185:         self.context[key][location] = content
186:
187:     def _add_failed_source_path(self, path, static=False):
188:         """Record a source file path that a Generator failed to process.
189:         (For example, one that was missing mandatory metadata.)
190:         The path argument is expected to be relative to self.path.
191:         """
192:         key = 'static_content' if static else 'generated_content'
193:         self.context[key][posixize_path(os.path.normpath(path))] = None
194:
195:     def _is_potential_source_path(self, path, static=False):
196:         """Return True if path was supposed to be used as a source file.
197:         (This includes all source files that have been found by generators
198:         before this method is called, even if they failed to process.)
199:         The path argument is expected to be relative to self.path.
200:         """
201:         key = 'static_content' if static else 'generated_content'
202:         return (posixize_path(os.path.normpath(path)) in self.context[key])
203:
204:     def add_static_links(self, content):
205:         """Add file links in content to context to be processed as Static
206:         content.
207:         """
208:         self.context['static_links'] |= content.get_static_links()
209:
210:     def _update_context(self, items):
211:         """Update the context with the given items from the currrent
212:         processor.
213:         """
214:         for item in items:
215:             value = getattr(self, item)
216:             if hasattr(value, 'items'):
217:                 value = list(value.items())  # py3k safeguard for iterators
218:             self.context[item] = value
219:
220:     def __str__(self):
221:         # return the name of the class for logging purposes
222:         return self.__class__.__name__
223:
224:
225: class CachingGenerator(Generator, FileStampDataCacher):
226:     '''Subclass of Generator and FileStampDataCacher classes
227:
228:     enables content caching, either at the generator or reader level
229:     '''
230:
231:     def __init__(self, *args, **kwargs):
232:         '''Initialize the generator, then set up caching
233:
234:         note the multiple inheritance structure
235:         '''
236:         cls_name = self.__class__.__name__
237:         Generator.__init__(self, *args,
238:                            readers_cache_name=(cls_name + '-Readers'),
239:                            **kwargs)
240:
241:         cache_this_level = \
242:             self.settings['CONTENT_CACHING_LAYER'] == 'generator'
243:         caching_policy = cache_this_level and self.settings['CACHE_CONTENT']
244:         load_policy = cache_this_level and self.settings['LOAD_CONTENT_CACHE']
```

```
245:         FileStampDataCacher.__init__(self, self.settings, cls_name,
246:                                      caching_policy, load_policy
247:                                      )
248:
249:     def _get_file_stamp(self, filename):
250:         '''Get filestamp for path relative to generator.path'''
251:         filename = os.path.join(self.path, filename)
252:         return super()._get_file_stamp(filename)
253:
254:
255: class _FileLoader(BaseLoader):
256:
257:     def __init__(self, path, basedir):
258:         self.path = path
259:         self.fullpath = os.path.join(basedir, path)
260:
261:     def get_source(self, environment, template):
262:         if template != self.path or not os.path.exists(self.fullpath):
263:             raise TemplateNotFound(template)
264:         mtime = os.path.getmtime(self.fullpath)
265:         with open(self.fullpath, 'r', encoding='utf-8') as f:
266:             source = f.read()
267:         return (source, self.fullpath,
268:                 lambda: mtime == os.path.getmtime(self.fullpath))
269:
270:
271: class TemplatePagesGenerator(Generator):
272:
273:     def generate_output(self, writer):
274:         for source, dest in self.settings['TEMPLATE_PAGES'].items():
275:             self.env.loader.loaders.insert(0, _FileLoader(source, self.path))
276:             try:
277:                 template = self.env.get_template(source)
278:                 rurls = self.settings['RELATIVE_URLS']
279:                 writer.write_file(dest, template, self.context, rurls,
280:                                   override_output=True, url='')
281:             finally:
282:                 del self.env.loader.loaders[0]
283:
284:
285: class ArticlesGenerator(CachingGenerator):
286:     """Generate blog articles"""
287:
288:     def __init__(self, *args, **kwargs):
289:         """initialize properties"""
290:         self.articles = []                  # only articles in default language
291:         self.translations = []
292:         self.dates = {}
293:         self.tags = defaultdict(list)
294:         self.categories = defaultdict(list)
295:         self.related_posts = []
296:         self.authors = defaultdict(list)
297:         self.drafts = []                    # only drafts in default language
298:         self.drafts_translations = []
299:         super().__init__(*args, **kwargs)
300:         signals.article_generator_init.send(self)
301:
302:     def generate_feeds(self, writer):
303:         """Generate the feeds from the current context, and output files."""
304:
305:         if self.settings.get('FEED_ATOM'):
```

```
306:             writer.write_feed(
307:                 self.articles,
308:                 self.context,
309:                 self.settings['FEED_ATOM'],
310:                 self.settings.get('FEED_ATOM_URL', self.settings['FEED_ATOM'])
311:                 )
312:
313:         if self.settings.get('FEED_RSS'):
314:             writer.write_feed(
315:                 self.articles,
316:                 self.context,
317:                 self.settings['FEED_RSS'],
318:                 self.settings.get('FEED_RSS_URL', self.settings['FEED_RSS']),
319:                 feed_type='rss'
320:                 )
321:
322:         if (self.settings.get('FEED_ALL_ATOM') or
323:                 self.settings.get('FEED_ALL_RSS')):
324:             all_articles = list(self.articles)
325:             for article in self.articles:
326:                 all_articles.extend(article.translations)
327:             order_content(all_articles,
328:                           order_by=self.settings['ARTICLE_ORDER_BY'])
329:
330:             if self.settings.get('FEED_ALL_ATOM'):
331:                 writer.write_feed(
332:                     all_articles,
333:                     self.context,
334:                     self.settings['FEED_ALL_ATOM'],
335:                     self.settings.get('FEED_ALL_ATOM_URL',
336:                                       self.settings['FEED_ALL_ATOM'])
337:                     )
338:
339:             if self.settings.get('FEED_ALL_RSS'):
340:                 writer.write_feed(
341:                     all_articles,
342:                     self.context,
343:                     self.settings['FEED_ALL_RSS'],
344:                     self.settings.get('FEED_ALL_RSS_URL',
345:                                       self.settings['FEED_ALL_RSS']),
346:                     feed_type='rss'
347:                     )
348:
349:         for cat, arts in self.categories:
350:             if self.settings.get('CATEGORY_FEED_ATOM'):
351:                 writer.write_feed(
352:                     arts,
353:                     self.context,
354:                     self.settings['CATEGORY_FEED_ATOM'].format(slug=cat.slug),
355:                     self.settings.get(
356:                         'CATEGORY_FEED_ATOM_URL',
357:                         self.settings['CATEGORY_FEED_ATOM']).format(
358:                             slug=cat.slug
359:                         ),
360:                     feed_title=cat.name
361:                     )
362:
363:             if self.settings.get('CATEGORY_FEED_RSS'):
364:                 writer.write_feed(
365:                     arts,
366:                     self.context,
```

```
367:                     self.settings['CATEGORY_FEED_RSS'].format(slug=cat.slug),
368:                     self.settings.get(
369:                         'CATEGORY_FEED_RSS_URL',
370:                         self.settings['CATEGORY_FEED_RSS']).format(
371:                             slug=cat.slug
372:                         ),
373:                     feed_title=cat.name,
374:                     feed_type='rss'
375:                     )
376:
377:         for auth, arts in self.authors:
378:             if self.settings.get('AUTHOR_FEED_ATOM'):
379:                 writer.write_feed(
380:                     arts,
381:                     self.context,
382:                     self.settings['AUTHOR_FEED_ATOM'].format(slug=auth.slug),
383:                     self.settings.get(
384:                         'AUTHOR_FEED_ATOM_URL',
385:                         self.settings['AUTHOR_FEED_ATOM']
386:                         ).format(slug=auth.slug),
387:                     feed_title=auth.name
388:                     )
389:
390:             if self.settings.get('AUTHOR_FEED_RSS'):
391:                 writer.write_feed(
392:                     arts,
393:                     self.context,
394:                     self.settings['AUTHOR_FEED_RSS'].format(slug=auth.slug),
395:                     self.settings.get(
396:                         'AUTHOR_FEED_RSS_URL',
397:                         self.settings['AUTHOR_FEED_RSS']
398:                         ).format(slug=auth.slug),
399:                     feed_title=auth.name,
400:                     feed_type='rss'
401:                     )
402:
403:         if (self.settings.get('TAG_FEED_ATOM') or
404:                 self.settings.get('TAG_FEED_RSS')):
405:             for tag, arts in self.tags.items():
406:                 if self.settings.get('TAG_FEED_ATOM'):
407:                     writer.write_feed(
408:                         arts,
409:                         self.context,
410:                         self.settings['TAG_FEED_ATOM'].format(slug=tag.slug),
411:                         self.settings.get(
412:                             'TAG_FEED_ATOM_URL',
413:                             self.settings['TAG_FEED_ATOM']
414:                             ).format(slug=tag.slug),
415:                         feed_title=tag.name
416:                         )
417:
418:                 if self.settings.get('TAG_FEED_RSS'):
419:                     writer.write_feed(
420:                         arts,
421:                         self.context,
422:                         self.settings['TAG_FEED_RSS'].format(slug=tag.slug),
423:                         self.settings.get(
424:                             'TAG_FEED_RSS_URL',
425:                             self.settings['TAG_FEED_RSS']
426:                             ).format(slug=tag.slug),
427:                         feed_title=tag.name,
```

```
428:                         feed_type='rss'
429:                         )
430:
431:         if (self.settings.get('TRANSLATION_FEED_ATOM') or
432:                 self.settings.get('TRANSLATION_FEED_RSS')):
433:             translations_feeds = defaultdict(list)
434:             for article in chain(self.articles, self.translations):
435:                 translations_feeds[article.lang].append(article)
436:
437:             for lang, items in translations_feeds.items():
438:                 items = order_content(
439:                     items, order_by=self.settings['ARTICLE_ORDER_BY'])
440:                 if self.settings.get('TRANSLATION_FEED_ATOM'):
441:                     writer.write_feed(
442:                         items,
443:                         self.context,
444:                         self.settings['TRANSLATION_FEED_ATOM']
445:                             .format(lang=lang),
446:                         self.settings.get(
447:                             'TRANSLATION_FEED_ATOM_URL',
448:                             self.settings['TRANSLATION_FEED_ATOM']
449:                             ).format(lang=lang),
450:                         )
451:                 if self.settings.get('TRANSLATION_FEED_RSS'):
452:                     writer.write_feed(
453:                         items,
454:                         self.context,
455:                         self.settings['TRANSLATION_FEED_RSS']
456:                             .format(lang=lang),
457:                         self.settings.get(
458:                             'TRANSLATION_FEED_RSS_URL',
459:                             self.settings['TRANSLATION_FEED_RSS']
460:                             ).format(lang=lang),
461:                         feed_type='rss'
462:                         )
463:
464:     def generate_articles(self, write):
465:         """Generate the articles."""
466:         for article in chain(self.translations, self.articles):
467:             signals.article_generator_write_article.send(self, content=article)
468:             write(article.save_as, self.get_template(article.template),
469:                   self.context, article=article, category=article.category,
470:                   override_output=hasattr(article, 'override_save_as'),
471:                   url=article.url, blog=True)
472:
473:     def generate_period_archives(self, write):
474:         """Generate per-year, per-month, and per-day archives."""
475:         try:
476:             template = self.get_template('period_archives')
477:         except PelicanTemplateNotFound:
478:             template = self.get_template('archives')
479:
480:         period_save_as = {
481:             'year': self.settings['YEAR_ARCHIVE_SAVE_AS'],
482:             'month': self.settings['MONTH_ARCHIVE_SAVE_AS'],
483:             'day': self.settings['DAY_ARCHIVE_SAVE_AS'],
484:         }
485:
486:         period_url = {
487:             'year': self.settings['YEAR_ARCHIVE_URL'],
488:             'month': self.settings['MONTH_ARCHIVE_URL'],
```

```
            'day': self.settings['DAY_ARCHIVE_URL'],
        }

        period_date_key = {
            'year': attrgetter('date.year'),
            'month': attrgetter('date.year', 'date.month'),
            'day': attrgetter('date.year', 'date.month', 'date.day')
        }

        def _generate_period_archives(dates, key, save_as_fmt, url_fmt):
            """Generate period archives from `dates`, grouped by
            `key` and written to `save_as`.
            """
            # `dates` is already sorted by date
            for _period, group in groupby(dates, key=key):
                archive = list(group)
                articles = [a for a in self.articles if a in archive]
                # arbitrarily grab the first date so that the usual
                # format string syntax can be used for specifying the
                # period archive dates
                date = archive[0].date
                save_as = save_as_fmt.format(date=date)
                url = url_fmt.format(date=date)
                context = self.context.copy()

                if key == period_date_key['year']:
                    context["period"] = (_period,)
                else:
                    month_name = calendar.month_name[_period[1]]
                    if key == period_date_key['month']:
                        context["period"] = (_period[0],
                                             month_name)
                    else:
                        context["period"] = (_period[0],
                                             month_name,
                                             _period[2])

                write(save_as, template, context, articles=articles,
                      dates=archive, template_name='period_archives',
                      blog=True, url=url, all_articles=self.articles)

        for period in 'year', 'month', 'day':
            save_as = period_save_as[period]
            url = period_url[period]
            if save_as:
                key = period_date_key[period]
                _generate_period_archives(self.dates, key, save_as, url)

    def generate_direct_templates(self, write):
        """Generate direct templates pages"""
        for template in self.settings['DIRECT_TEMPLATES']:
            save_as = self.settings.get("%s_SAVE_AS" % template.upper(),
                                        '%s.html' % template)
            url = self.settings.get("%s_URL" % template.upper(),
                                    '%s.html' % template)
            if not save_as:
                continue

            write(save_as, self.get_template(template), self.context,
                  articles=self.articles, dates=self.dates, blog=True,
                  template_name=template,
```

```
550:                     page_name=os.path.splitext(save_as)[0], url=url)
551:
552:     def generate_tags(self, write):
553:         """Generate Tags pages."""
554:         tag_template = self.get_template('tag')
555:         for tag, articles in self.tags.items():
556:             dates = [article for article in self.dates if article in articles]
557:             write(tag.save_as, tag_template, self.context, tag=tag,
558:                   url=tag.url, articles=articles, dates=dates,
559:                   template_name='tag', blog=True, page_name=tag.page_name,
560:                   all_articles=self.articles)
561:
562:     def generate_categories(self, write):
563:         """Generate category pages."""
564:         category_template = self.get_template('category')
565:         for cat, articles in self.categories:
566:             dates = [article for article in self.dates if article in articles]
567:             write(cat.save_as, category_template, self.context, url=cat.url,
568:                   category=cat, articles=articles, dates=dates,
569:                   template_name='category', blog=True, page_name=cat.page_name,
570:                   all_articles=self.articles)
571:
572:     def generate_authors(self, write):
573:         """Generate Author pages."""
574:         author_template = self.get_template('author')
575:         for aut, articles in self.authors:
576:             dates = [article for article in self.dates if article in articles]
577:             write(aut.save_as, author_template, self.context,
578:                   url=aut.url, author=aut, articles=articles, dates=dates,
579:                   template_name='author', blog=True,
580:                   page_name=aut.page_name, all_articles=self.articles)
581:
582:     def generate_drafts(self, write):
583:         """Generate drafts pages."""
584:         for draft in chain(self.drafts_translations, self.drafts):
585:             write(draft.save_as, self.get_template(draft.template),
586:                   self.context, article=draft, category=draft.category,
587:                   override_output=hasattr(draft, 'override_save_as'),
588:                   blog=True, all_articles=self.articles, url=draft.url)
589:
590:     def generate_pages(self, writer):
591:         """Generate the pages on the disk"""
592:         write = partial(writer.write_file,
593:                         relative_urls=self.settings['RELATIVE_URLS'])
594:
595:         # to minimize the number of relative path stuff modification
596:         # in writer, articles pass first
597:         self.generate_articles(write)
598:         self.generate_period_archives(write)
599:         self.generate_direct_templates(write)
600:
601:         # and subfolders after that
602:         self.generate_tags(write)
603:         self.generate_categories(write)
604:         self.generate_authors(write)
605:         self.generate_drafts(write)
606:
607:     def generate_context(self):
608:         """Add the articles into the shared context"""
609:
610:         all_articles = []
```

```
611:         all_drafts = []
612:         for f in self.get_files(
613:                 self.settings['ARTICLE_PATHS'],
614:                 exclude=self.settings['ARTICLE_EXCLUDES']):
615:             article = self.get_cached_data(f, None)
616:             if article is None:
617:                 try:
618:                     article = self.readers.read_file(
619:                         base_path=self.path, path=f, content_class=Article,
620:                         context=self.context,
621:                         preread_signal=signals.article_generator_preread,
622:                         preread_sender=self,
623:                         context_signal=signals.article_generator_context,
624:                         context_sender=self)
625:                 except Exception as e:
626:                     logger.error(
627:                         'Could not process %s\n%s', f, e,
628:                         exc_info=self.settings.get('DEBUG', False))
629:                     self._add_failed_source_path(f)
630:                     continue
631: 
632:                 if not article.is_valid():
633:                     self._add_failed_source_path(f)
634:                     continue
635: 
636:                 self.cache_data(f, article)
637: 
638:             if article.status == "published":
639:                 all_articles.append(article)
640:             elif article.status == "draft":
641:                 all_drafts.append(article)
642:             self.add_source_path(article)
643:             self.add_static_links(article)
644: 
645:         def _process(arts):
646:             origs, translations = process_translations(
647:                 arts, translation_id=self.settings['ARTICLE_TRANSLATION_ID'])
648:             origs = order_content(origs, self.settings['ARTICLE_ORDER_BY'])
649:             return origs, translations
650: 
651:         self.articles, self.translations = _process(all_articles)
652:         self.drafts, self.drafts_translations = _process(all_drafts)
653: 
654:         signals.article_generator_pretaxonomy.send(self)
655: 
656:         for article in self.articles:
657:             # only main articles are listed in categories and tags
658:             # not translations
659:             self.categories[article.category].append(article)
660:             if hasattr(article, 'tags'):
661:                 for tag in article.tags:
662:                     self.tags[tag].append(article)
663:             for author in getattr(article, 'authors', []):
664:                 self.authors[author].append(article)
665: 
666:         self.dates = list(self.articles)
667:         self.dates.sort(key=attrgetter('date'),
668:                         reverse=self.context['NEWEST_FIRST_ARCHIVES'])
669: 
670:         # and generate the output :)
671: 
```

```
672:         # order the categories per name
673:         self.categories = list(self.categories.items())
674:         self.categories.sort(
675:             reverse=self.settings['REVERSE_CATEGORY_ORDER'])
676:
677:         self.authors = list(self.authors.items())
678:         self.authors.sort()
679:
680:         self._update_context(('articles', 'dates', 'tags', 'categories',
681:                               'authors', 'related_posts', 'drafts'))
682:         self.save_cache()
683:         self.readers.save_cache()
684:         signals.article_generator_finalized.send(self)
685:
686:     def generate_output(self, writer):
687:         self.generate_feeds(writer)
688:         self.generate_pages(writer)
689:         signals.article_writer_finalized.send(self, writer=writer)
690:
691:     def refresh_metadata_intersite_links(self):
692:         for e in chain(self.articles,
693:                        self.translations,
694:                        self.drafts,
695:                        self.drafts_translations):
696:             if hasattr(e, 'refresh_metadata_intersite_links'):
697:                 e.refresh_metadata_intersite_links()
698:
699:
700: class PagesGenerator(CachingGenerator):
701:     """Generate pages"""
702:
703:     def __init__(self, *args, **kwargs):
704:         self.pages = []
705:         self.translations = []
706:         self.hidden_pages = []
707:         self.hidden_translations = []
708:         self.draft_pages = []
709:         self.draft_translations = []
710:         super().__init__(*args, **kwargs)
711:         signals.page_generator_init.send(self)
712:
713:     def generate_context(self):
714:         all_pages = []
715:         hidden_pages = []
716:         draft_pages = []
717:         for f in self.get_files(
718:                 self.settings['PAGE_PATHS'],
719:                 exclude=self.settings['PAGE_EXCLUDES']):
720:             page = self.get_cached_data(f, None)
721:             if page is None:
722:                 try:
723:                     page = self.readers.read_file(
724:                         base_path=self.path, path=f, content_class=Page,
725:                         context=self.context,
726:                         preread_signal=signals.page_generator_preread,
727:                         preread_sender=self,
728:                         context_signal=signals.page_generator_context,
729:                         context_sender=self)
730:                 except Exception as e:
731:                     logger.error(
732:                         'Could not process %s\n%s', f, e,
```

```
733:                         exc_info=self.settings.get('DEBUG', False))
734:                     self._add_failed_source_path(f)
735:                     continue
736:
737:                 if not page.is_valid():
738:                     self._add_failed_source_path(f)
739:                     continue
740:
741:                 self.cache_data(f, page)
742:
743:             if page.status == "published":
744:                 all_pages.append(page)
745:             elif page.status == "hidden":
746:                 hidden_pages.append(page)
747:             elif page.status == "draft":
748:                 draft_pages.append(page)
749:             self.add_source_path(page)
750:             self.add_static_links(page)
751:
752:         def _process(pages):
753:             origs, translations = process_translations(
754:                 pages, translation_id=self.settings['PAGE_TRANSLATION_ID'])
755:             origs = order_content(origs, self.settings['PAGE_ORDER_BY'])
756:             return origs, translations
757:
758:         self.pages, self.translations = _process(all_pages)
759:         self.hidden_pages, self.hidden_translations = _process(hidden_pages)
760:         self.draft_pages, self.draft_translations = _process(draft_pages)
761:
762:         self._update_context(('pages', 'hidden_pages', 'draft_pages'))
763:
764:         self.save_cache()
765:         self.readers.save_cache()
766:         signals.page_generator_finalized.send(self)
767:
768:     def generate_output(self, writer):
769:         for page in chain(self.translations, self.pages,
770:                           self.hidden_translations, self.hidden_pages,
771:                           self.draft_translations, self.draft_pages):
772:             signals.page_generator_write_page.send(self, content=page)
773:             writer.write_file(
774:                 page.save_as, self.get_template(page.template),
775:                 self.context, page=page,
776:                 relative_urls=self.settings['RELATIVE_URLS'],
777:                 override_output=hasattr(page, 'override_save_as'),
778:                 url=page.url)
779:         signals.page_writer_finalized.send(self, writer=writer)
780:
781:     def refresh_metadata_intersite_links(self):
782:         for e in chain(self.pages,
783:                        self.hidden_pages,
784:                        self.hidden_translations,
785:                        self.draft_pages,
786:                        self.draft_translations):
787:             if hasattr(e, 'refresh_metadata_intersite_links'):
788:                 e.refresh_metadata_intersite_links()
789:
790:
791: class StaticGenerator(Generator):
792:     """copy static paths (what you want to copy, like images, medias etc.
793:     to output"""
```

```
794:
795:     def __init__(self, *args, **kwargs):
796:         super().__init__(*args, **kwargs)
797:         self.fallback_to_symlinks = False
798:         signals.static_generator_init.send(self)
799:
800:     def generate_context(self):
801:         self.staticfiles = []
802:         linked_files = set(self.context['static_links'])
803:         found_files = self.get_files(self.settings['STATIC_PATHS'],
804:                                      exclude=self.settings['STATIC_EXCLUDES'],
805:                                      extensions=False)
806:         for f in linked_files | found_files:
807:
808:             # skip content source files unless the user explicitly wants them
809:             if self.settings['STATIC_EXCLUDE_SOURCES']:
810:                 if self._is_potential_source_path(f):
811:                     continue
812:
813:             static = self.readers.read_file(
814:                 base_path=self.path, path=f, content_class=Static,
815:                 fmt='static', context=self.context,
816:                 preread_signal=signals.static_generator_preread,
817:                 preread_sender=self,
818:                 context_signal=signals.static_generator_context,
819:                 context_sender=self)
820:             self.staticfiles.append(static)
821:             self.add_source_path(static, static=True)
822:         self._update_context(('staticfiles',))
823:         signals.static_generator_finalized.send(self)
824:
825:     def generate_output(self, writer):
826:         self._copy_paths(self.settings['THEME_STATIC_PATHS'], self.theme,
827:                          self.settings['THEME_STATIC_DIR'], self.output_path,
828:                          os.curdir)
829:         for sc in self.context['staticfiles']:
830:             if self._file_update_required(sc):
831:                 self._link_or_copy_staticfile(sc)
832:             else:
833:                 logger.debug('%s is up to date, not copying', sc.source_path)
834:
835:     def _copy_paths(self, paths, source, destination, output_path,
836:                     final_path=None):
837:         """Copy all the paths from source to destination"""
838:         for path in paths:
839:             source_path = os.path.join(source, path)
840:
841:             if final_path:
842:                 if os.path.isfile(source_path):
843:                     destination_path = os.path.join(output_path, destination,
844:                                                     final_path,
845:                                                     os.path.basename(path))
846:                 else:
847:                     destination_path = os.path.join(output_path, destination,
848:                                                     final_path)
849:             else:
850:                 destination_path = os.path.join(output_path, destination, path)
851:
852:             copy(source_path, destination_path,
853:                  self.settings['IGNORE_FILES'])
854:
```

```
855:     def _file_update_required(self, staticfile):
856:         source_path = os.path.join(self.path, staticfile.source_path)
857:         save_as = os.path.join(self.output_path, staticfile.save_as)
858:         if not os.path.exists(save_as):
859:             return True
860:         elif (self.settings['STATIC_CREATE_LINKS'] and
861:               os.path.samefile(source_path, save_as)):
862:             return False
863:         elif (self.settings['STATIC_CREATE_LINKS'] and
864:               os.path.realpath(save_as) == source_path):
865:             return False
866:         elif not self.settings['STATIC_CHECK_IF_MODIFIED']:
867:             return True
868:         else:
869:             return self._source_is_newer(staticfile)
870: 
871:     def _source_is_newer(self, staticfile):
872:         source_path = os.path.join(self.path, staticfile.source_path)
873:         save_as = os.path.join(self.output_path, staticfile.save_as)
874:         s_mtime = os.path.getmtime(source_path)
875:         d_mtime = os.path.getmtime(save_as)
876:         return s_mtime - d_mtime > 0.000001
877: 
878:     def _link_or_copy_staticfile(self, sc):
879:         if self.settings['STATIC_CREATE_LINKS']:
880:             self._link_staticfile(sc)
881:         else:
882:             self._copy_staticfile(sc)
883: 
884:     def _copy_staticfile(self, sc):
885:         source_path = os.path.join(self.path, sc.source_path)
886:         save_as = os.path.join(self.output_path, sc.save_as)
887:         self._mkdir(os.path.dirname(save_as))
888:         copy(source_path, save_as)
889:         logger.info('Copying %s to %s', sc.source_path, sc.save_as)
890: 
891:     def _link_staticfile(self, sc):
892:         source_path = os.path.join(self.path, sc.source_path)
893:         save_as = os.path.join(self.output_path, sc.save_as)
894:         self._mkdir(os.path.dirname(save_as))
895:         try:
896:             if os.path.lexists(save_as):
897:                 os.unlink(save_as)
898:             logger.info('Linking %s and %s', sc.source_path, sc.save_as)
899:             if self.fallback_to_symlinks:
900:                 os.symlink(source_path, save_as)
901:             else:
902:                 os.link(source_path, save_as)
903:         except OSError as err:
904:             if err.errno == errno.EXDEV:  # 18: Invalid cross-device link
905:                 logger.debug(
906:                     "Cross-device links not valid. "
907:                     "Creating symbolic links instead."
908:                     )
909:                 self.fallback_to_symlinks = True
910:                 self._link_staticfile(sc)
911:             else:
912:                 raise err
913: 
914:     def _mkdir(self, path):
915:         if os.path.lexists(path) and not os.path.isdir(path):
```

```
916:             os.unlink(path)
917:         mkdir_p(path)
918:
919:
920: class SourceFileGenerator(Generator):
921:
922:     def generate_context(self):
923:         self.output_extension = self.settings['OUTPUT_SOURCES_EXTENSION']
924:
925:     def _create_source(self, obj):
926:         output_path, _ = os.path.splitext(obj.save_as)
927:         dest = os.path.join(self.output_path,
928:                             output_path + self.output_extension)
929:         copy(obj.source_path, dest)
930:
931:     def generate_output(self, writer=None):
932:         logger.info('Generating source files...')
933:         for obj in chain(self.context['articles'], self.context['pages']):
934:             self._create_source(obj)
935:             for obj_trans in obj.translations:
936:                 self._create_source(obj_trans)
```

```
# -*- coding: utf-8 -*-

import datetime
import fnmatch
import locale
import logging
import os
import re
import shutil
import sys
import traceback
import urllib
from collections.abc import Hashable
from contextlib import contextmanager
from functools import partial
from html import entities
from html.parser import HTMLParser
from itertools import groupby
from operator import attrgetter

import dateutil.parser

from jinja2 import Markup

import pytz


logger = logging.getLogger(__name__)


def sanitised_join(base_directory, *parts):
    joined = os.path.abspath(os.path.join(base_directory, *parts))
    if not joined.startswith(os.path.abspath(base_directory)):
        raise RuntimeError(
            "Attempted to break out of output directory to {}".format(
                joined
            )
        )

    return joined


def strftime(date, date_format):
    '''
    Enhanced replacement for built-in strftime with zero stripping

    This works by 'grabbing' possible format strings (those starting with %),
    formatting them with the date, stripping any leading zeros if - prefix is
    used and replacing formatted output back.
    '''
    def strip_zeros(x):
        return x.lstrip('0') or '0'
    c89_directives = 'aAbBcdfHIjmMpSUwWxXyYzZ%'

    # grab candidate format options
    format_options = '%[-]?.'
    candidates = re.findall(format_options, date_format)

    # replace candidates with placeholders for later % formatting
    template = re.sub(format_options, '%s', date_format)

```

```
62:     formatted_candidates = []
63:     for candidate in candidates:
64:         # test for valid C89 directives only
65:         if candidate[-1] in c89_directives:
66:             # check for '-' prefix
67:             if len(candidate) == 3:
68:                 # '-' prefix
69:                 candidate = '%{}'.format(candidate[-1])
70:                 conversion = strip_zeros
71:             else:
72:                 conversion = None
73:
74:             # format date
75:             if isinstance(date, SafeDatetime):
76:                 formatted = date.strftime(candidate, safe=False)
77:             else:
78:                 formatted = date.strftime(candidate)
79:
80:             # strip zeros if '-' prefix is used
81:             if conversion:
82:                 formatted = conversion(formatted)
83:         else:
84:             formatted = candidate
85:         formatted_candidates.append(formatted)
86:
87:     # put formatted candidates back and return
88:     return template % tuple(formatted_candidates)
89:
90:
91: class SafeDatetime(datetime.datetime):
92:     '''Subclass of datetime that works with utf-8 format strings on PY2'''
93:
94:     def strftime(self, fmt, safe=True):
95:         '''Uses our custom strftime if supposed to be *safe*'''
96:         if safe:
97:             return strftime(self, fmt)
98:         else:
99:             return super().strftime(fmt)
100:
101:
102: class DateFormatter(object):
103:     '''A date formatter object used as a jinja filter
104:
105:     Uses the `strftime` implementation and makes sure jinja uses the locale
106:     defined in LOCALE setting
107:     '''
108:
109:     def __init__(self):
110:         self.locale = locale.setlocale(locale.LC_TIME)
111:
112:     def __call__(self, date, date_format):
113:         old_lc_time = locale.setlocale(locale.LC_TIME)
114:         old_lc_ctype = locale.setlocale(locale.LC_CTYPE)
115:
116:         locale.setlocale(locale.LC_TIME, self.locale)
117:         # on OSX, encoding from LC_CTYPE determines the unicode output in PY3
118:         # make sure it's same as LC_TIME
119:         locale.setlocale(locale.LC_CTYPE, self.locale)
120:
121:         formatted = strftime(date, date_format)
122:
```

```
123:         locale.setlocale(locale.LC_TIME, old_lc_time)
124:         locale.setlocale(locale.LC_CTYPE, old_lc_ctype)
125:         return formatted
126:
127:
128: class memoized(object):
129:     """Function decorator to cache return values.
130:
131:     If called later with the same arguments, the cached value is returned
132:     (not reevaluated).
133:
134:     """
135:     def __init__(self, func):
136:         self.func = func
137:         self.cache = {}
138:
139:     def __call__(self, *args):
140:         if not isinstance(args, Hashable):
141:             # uncacheable. a list, for instance.
142:             # better to not cache than blow up.
143:             return self.func(*args)
144:         if args in self.cache:
145:             return self.cache[args]
146:         else:
147:             value = self.func(*args)
148:             self.cache[args] = value
149:             return value
150:
151:     def __repr__(self):
152:         return self.func.__doc__
153:
154:     def __get__(self, obj, objtype):
155:         '''Support instance methods.'''
156:         return partial(self.__call__, obj)
157:
158:
159: def deprecated_attribute(old, new, since=None, remove=None, doc=None):
160:     """Attribute deprecation decorator for gentle upgrades
161:
162:     For example:
163:
164:         class MyClass (object):
165:             @deprecated_attribute(
166:                 old='abc', new='xyz', since=(3, 2, 0), remove=(4, 1, 3))
167:             def abc(): return None
168:
169:             def __init__(self):
170:                 xyz = 5
171:
172:     Note that the decorator needs a dummy method to attach to, but the
173:     content of the dummy method is ignored.
174:     """
175:     def _warn():
176:         version = '.'.join(str(x) for x in since)
177:         message = ['{} has been deprecated since {}'.format(old, version)]
178:         if remove:
179:             version = '.'.join(str(x) for x in remove)
180:             message.append(
181:                 ' and will be removed by version {}'.format(version))
182:         message.append('.  Use {} instead.'.format(new))
183:         logger.warning(''.join(message))
```

```
184:         logger.debug(''.join(str(x) for x
185:                              in traceback.format_stack()))
186:
187:     def fget(self):
188:         _warn()
189:         return getattr(self, new)
190:
191:     def fset(self, value):
192:         _warn()
193:         setattr(self, new, value)
194:
195:     def decorator(dummy):
196:         return property(fget=fget, fset=fset, doc=doc)
197:
198:     return decorator
199:
200:
201: def get_date(string):
202:     """Return a datetime object from a string.
203:
204:     If no format matches the given date, raise a ValueError.
205:     """
206:     string = re.sub(' +', ' ', string)
207:     default = SafeDatetime.now().replace(hour=0, minute=0,
208:                                          second=0, microsecond=0)
209:     try:
210:         return dateutil.parser.parse(string, default=default)
211:     except (TypeError, ValueError):
212:         raise ValueError('{0!r} is not a valid date'.format(string))
213:
214:
215: @contextmanager
216: def pelican_open(filename, mode='r', strip_crs=(sys.platform == 'win32')):
217:     """Open a file and return its content"""
218:
219:     # utf-8-sig will clear any BOM if present
220:     with open(filename, mode, encoding='utf-8-sig') as infile:
221:         content = infile.read()
222:     yield content
223:
224:
225: def slugify(value, regex_subs=(), preserve_case=False):
226:     """
227:     Normalizes string, converts to lowercase, removes non-alpha characters,
228:     and converts spaces to hyphens.
229:
230:     Took from Django sources.
231:     """
232:
233:     # TODO Maybe steal again from current Django 1.5dev
234:     value = Markup(value).striptags()
235:     # value must be unicode per se
236:     import unicodedata
237:     from unidecode import unidecode
238:     value = unidecode(value)
239:     if isinstance(value, bytes):
240:         value = value.decode('ascii')
241:     # still unicode
242:     value = unicodedata.normalize('NFKD', value)
243:
244:     for src, dst in regex_subs:
```

```
245:         value = re.sub(src, dst, value, flags=re.IGNORECASE)
246:
247:     # convert to lowercase
248:     if not preserve_case:
249:         value = value.lower()
250:
251:     # we want only ASCII chars
252:     value = value.encode('ascii', 'ignore').strip()
253:     # but Pelican should generally use only unicode
254:     return value.decode('ascii')
255:
256:
257: def copy(source, destination, ignores=None):
258:     """Recursively copy source into destination.
259:
260:     If source is a file, destination has to be a file as well.
261:     The function is able to copy either files or directories.
262:
263:     :param source: the source file or directory
264:     :param destination: the destination file or directory
265:     :param ignores: either None, or a list of glob patterns;
266:         files matching those patterns will _not_ be copied.
267:     """
268:
269:     def walk_error(err):
270:         logger.warning("While copying %s: %s: %s",
271:                        source_, err.filename, err.strerror)
272:
273:     source_ = os.path.abspath(os.path.expanduser(source))
274:     destination_ = os.path.abspath(os.path.expanduser(destination))
275:
276:     if ignores is None:
277:         ignores = []
278:
279:     if any(fnmatch.fnmatch(os.path.basename(source), ignore)
280:            for ignore in ignores):
281:         logger.info('Not copying %s due to ignores', source_)
282:         return
283:
284:     if os.path.isfile(source_):
285:         dst_dir = os.path.dirname(destination_)
286:         if not os.path.exists(dst_dir):
287:             logger.info('Creating directory %s', dst_dir)
288:             os.makedirs(dst_dir)
289:         logger.info('Copying %s to %s', source_, destination_)
290:         copy_file_metadata(source_, destination_)
291:
292:     elif os.path.isdir(source_):
293:         if not os.path.exists(destination_):
294:             logger.info('Creating directory %s', destination_)
295:             os.makedirs(destination_)
296:         if not os.path.isdir(destination_):
297:             logger.warning('Cannot copy %s (a directory) to %s (a file)',
298:                            source_, destination_)
299:             return
300:
301:         for src_dir, subdirs, others in os.walk(source_, followlinks=True):
302:             dst_dir = os.path.join(destination_,
303:                                    os.path.relpath(src_dir, source_))
304:
305:             subdirs[:] = (s for s in subdirs if not any(fnmatch.fnmatch(s, i)
```

```
306:                                                     for i in ignores))
307:             others[:] = (o for o in others if not any(fnmatch.fnmatch(o, i)
308:                                                       for i in ignores))
309:
310:             if not os.path.isdir(dst_dir):
311:                 logger.info('Creating directory %s', dst_dir)
312:                 # Parent directories are known to exist, so 'mkdir' suffices.
313:                 os.mkdir(dst_dir)
314:
315:             for o in others:
316:                 src_path = os.path.join(src_dir, o)
317:                 dst_path = os.path.join(dst_dir, o)
318:                 if os.path.isfile(src_path):
319:                     logger.info('Copying %s to %s', src_path, dst_path)
320:                     copy_file_metadata(src_path, dst_path)
321:                 else:
322:                     logger.warning('Skipped copy %s (not a file or '
323:                                    'directory) to %s',
324:                                    src_path, dst_path)
325:
326:
327: def copy_file_metadata(source, destination):
328:     '''Copy a file and its metadata (perm bits, access times, ...)'''
329:
330:     # This function is a workaround for Android python copystat
331:     # bug ([issue28141]) https://bugs.python.org/issue28141
332:     try:
333:         shutil.copy2(source, destination)
334:     except OSError as e:
335:         logger.warning("A problem occurred copying file %s to %s; %s",
336:                        source, destination, e)
337:
338:
339: def clean_output_dir(path, retention):
340:     """Remove all files from output directory except those in retention list"""
341:
342:     if not os.path.exists(path):
343:         logger.debug("Directory already removed: %s", path)
344:         return
345:
346:     if not os.path.isdir(path):
347:         try:
348:             os.remove(path)
349:         except Exception as e:
350:             logger.error("Unable to delete file %s; %s", path, e)
351:         return
352:
353:     # remove existing content from output folder unless in retention list
354:     for filename in os.listdir(path):
355:         file = os.path.join(path, filename)
356:         if any(filename == retain for retain in retention):
357:             logger.debug("Skipping deletion; %s is on retention list: %s",
358:                          filename, file)
359:         elif os.path.isdir(file):
360:             try:
361:                 shutil.rmtree(file)
362:                 logger.debug("Deleted directory %s", file)
363:             except Exception as e:
364:                 logger.error("Unable to delete directory %s; %s",
365:                              file, e)
366:         elif os.path.isfile(file) or os.path.islink(file):
```

```
367:             try:
368:                 os.remove(file)
369:                 logger.debug("Deleted file/link %s", file)
370:             except Exception as e:
371:                 logger.error("Unable to delete file %s; %s", file, e)
372:         else:
373:             logger.error("Unable to delete %s, file type unknown", file)
374:
375:
376: def get_relative_path(path):
377:     """Return the relative path from the given path to the root path."""
378:     components = split_all(path)
379:     if len(components) <= 1:
380:         return os.curdir
381:     else:
382:         parents = [os.pardir] * (len(components) - 1)
383:         return os.path.join(*parents)
384:
385:
386: def path_to_url(path):
387:     """Return the URL corresponding to a given path."""
388:     if os.sep == '/':
389:         return path
390:     else:
391:         return '/'.join(split_all(path))
392:
393:
394: def posixize_path(rel_path):
395:     """Use '/' as path separator, so that source references,
396:     like '{static}/foo/bar.jpg' or 'extras/favicon.ico',
397:     will work on Windows as well as on Mac and Linux."""
398:     return rel_path.replace(os.sep, '/')
399:
400:
401: class _HTMLWordTruncator(HTMLParser):
402:
403:     _word_regex = re.compile(r"\w[\w'-]*", re.U)
404:     _word_prefix_regex = re.compile(r'\w', re.U)
405:     _singlets = ('br', 'col', 'link', 'base', 'img', 'param', 'area',
406:                  'hr', 'input')
407:
408:     class TruncationCompleted(Exception):
409:
410:         def __init__(self, truncate_at):
411:             super().__init__(truncate_at)
412:             self.truncate_at = truncate_at
413:
414:     def __init__(self, max_words):
415:         super().__init__(convert_charrefs=False)
416:
417:         self.max_words = max_words
418:         self.words_found = 0
419:         self.open_tags = []
420:         self.last_word_end = None
421:         self.truncate_at = None
422:
423:     def feed(self, *args, **kwargs):
424:         try:
425:             super().feed(*args, **kwargs)
426:         except self.TruncationCompleted as exc:
427:             self.truncate_at = exc.truncate_at
```

```
428:         else:
429:             self.truncate_at = None
430:
431:     def getoffset(self):
432:         line_start = 0
433:         lineno, line_offset = self.getpos()
434:         for i in range(lineno - 1):
435:             line_start = self.rawdata.index('\n', line_start) + 1
436:         return line_start + line_offset
437:
438:     def add_word(self, word_end):
439:         self.words_found += 1
440:         self.last_word_end = None
441:         if self.words_found == self.max_words:
442:             raise self.TruncationCompleted(word_end)
443:
444:     def add_last_word(self):
445:         if self.last_word_end is not None:
446:             self.add_word(self.last_word_end)
447:
448:     def handle_starttag(self, tag, attrs):
449:         self.add_last_word()
450:         if tag not in self._singlets:
451:             self.open_tags.insert(0, tag)
452:
453:     def handle_endtag(self, tag):
454:         self.add_last_word()
455:         try:
456:             i = self.open_tags.index(tag)
457:         except ValueError:
458:             pass
459:         else:
460:             # SGML: An end tag closes, back to the matching start tag,
461:             # all unclosed intervening start tags with omitted end tags
462:             del self.open_tags[:i + 1]
463:
464:     def handle_data(self, data):
465:         word_end = 0
466:         offset = self.getoffset()
467:
468:         while self.words_found < self.max_words:
469:             match = self._word_regex.search(data, word_end)
470:             if not match:
471:                 break
472:
473:             if match.start(0) > 0:
474:                 self.add_last_word()
475:
476:             word_end = match.end(0)
477:             self.last_word_end = offset + word_end
478:
479:         if word_end < len(data):
480:             self.add_last_word()
481:
482:     def _handle_ref(self, name, char):
483:         """
484:         Called by handle_entityref() or handle_charref() when a ref like
485:         `&mdash;`, `&#8212;`, or `&#x2014` is found.
486:
487:         The arguments for this method are:
488:
```

```
489:         - `name`: the HTML entity name (such as `mdash` or `#8212` or `#x2014`)
490:         - `char`: the Unicode representation of the ref (such as `â\200\224`)
491:
492:         This method checks whether the entity is considered to be part of a
493:         word or not and, if not, signals the end of a word.
494:         """
495:         # Compute the index of the character right after the ref.
496:         #
497:         # In a string like 'prefix&mdash;suffix', the end is the sum of:
498:         #
499:         # - `self.getoffset()` (the length of `prefix`)
500:         # - `1` (the length of `&`)
501:         # - `len(name)` (the length of `mdash`)
502:         # - `1` (the length of `;`)
503:         #
504:         # Note that, in case of malformed HTML, the ';' character may
505:         # not be present.
506:
507:         offset = self.getoffset()
508:         ref_end = offset + len(name) + 1
509:
510:         try:
511:             if self.rawdata[ref_end] == ';':
512:                 ref_end += 1
513:         except IndexError:
514:             # We are at the end of the string and there's no ';'
515:             pass
516:
517:         if self.last_word_end is None:
518:             if self._word_prefix_regex.match(char):
519:                 self.last_word_end = ref_end
520:         else:
521:             if self._word_regex.match(char):
522:                 self.last_word_end = ref_end
523:             else:
524:                 self.add_last_word()
525:
526:     def handle_entityref(self, name):
527:         """
528:         Called when an entity ref like '&mdash;' is found
529:
530:         `name` is the entity ref without ampersand and semicolon (e.g. `mdash`)
531:         """
532:         try:
533:             codepoint = entities.name2codepoint[name]
534:             char = chr(codepoint)
535:         except KeyError:
536:             char = ''
537:         self._handle_ref(name, char)
538:
539:     def handle_charref(self, name):
540:         """
541:         Called when a char ref like '&#8212;' or '&#x2014' is found
542:
543:         `name` is the char ref without ampersand and semicolon (e.g. `#8212` or
544:         `#x2014`)
545:         """
546:         try:
547:             if name.startswith('x'):
548:                 codepoint = int(name[1:], 16)
549:             else:
```

```
550:                 codepoint = int(name)
551:             char = chr(codepoint)
552:         except (ValueError, OverflowError):
553:             char = ''
554:         self._handle_ref('#' + name, char)
555:
556:
557: def truncate_html_words(s, num, end_text='â\200¦'):
558:     """Truncates HTML to a certain number of words.
559:
560:     (not counting tags and comments). Closes opened tags if they were correctly
561:     closed in the given html. Takes an optional argument of what should be used
562:     to notify that the string has been truncated, defaulting to ellipsis (â\200¦).
563:
564:     Newlines in the HTML are preserved. (From the django framework).
565:     """
566:     length = int(num)
567:     if length <= 0:
568:         return ''
569:     truncator = _HTMLWordTruncator(length)
570:     truncator.feed(s)
571:     if truncator.truncate_at is None:
572:         return s
573:     out = s[:truncator.truncate_at]
574:     if end_text:
575:         out += ' ' + end_text
576:     # Close any tags still open
577:     for tag in truncator.open_tags:
578:         out += '</%s>' % tag
579:     # Return string
580:     return out
581:
582:
583: def process_translations(content_list, translation_id=None):
584:     """ Finds translations and returns them.
585:
586:     For each content_list item, populates the 'translations' attribute, and
587:     returns a tuple with two lists (index, translations). Index list includes
588:     items in default language or items which have no variant in default
589:     language. Items with the `translation` metadata set to something else than
590:     `False` or `false` will be used as translations, unless all the items in
591:     the same group have that metadata.
592:
593:     Translations and original items are determined relative to one another
594:     amongst items in the same group. Items are in the same group if they
595:     have the same value(s) for the metadata attribute(s) specified by the
596:     'translation_id', which must be a string or a collection of strings.
597:     If 'translation_id' is falsy, the identification of translations is skipped
598:     and all items are returned as originals.
599:     """
600:
601:     if not translation_id:
602:         return content_list, []
603:
604:     if isinstance(translation_id, str):
605:         translation_id = {translation_id}
606:
607:     index = []
608:
609:     try:
610:         content_list.sort(key=attrgetter(*translation_id))
```

```
611:     except TypeError:
612:         raise TypeError('Cannot unpack {}, \'translation_id\' must be falsy, a'
613:                         'string or a collection of strings'
614:                         .format(translation_id))
615:     except AttributeError:
616:         raise AttributeError('Cannot use {} as \'translation_id\', there'
617:                              'appear to be items without these metadata'
618:                              'attributes'.format(translation_id))
619:
620:     for id_vals, items in groupby(content_list, attrgetter(*translation_id)):
621:         # prepare warning string
622:         id_vals = (id_vals,) if len(translation_id) == 1 else id_vals
623:         with_str = 'with' + ', '.join([' {} "{{}}"'] * len(translation_id))\
624:             .format(*translation_id).format(*id_vals)
625:
626:         items = list(items)
627:         original_items = get_original_items(items, with_str)
628:         index.extend(original_items)
629:         for a in items:
630:             a.translations = [x for x in items if x != a]
631:
632:     translations = [x for x in content_list if x not in index]
633:
634:     return index, translations
635:
636:
637: def get_original_items(items, with_str):
638:     def _warn_source_paths(msg, items, *extra):
639:         args = [len(items)]
640:         args.extend(extra)
641:         args.extend((x.source_path for x in items))
642:         logger.warning('{}: {}'.format(msg, '\n%s' * len(items)), *args)
643:
644:     # warn if several items have the same lang
645:     for lang, lang_items in groupby(items, attrgetter('lang')):
646:         lang_items = list(lang_items)
647:         if len(lang_items) > 1:
648:             _warn_source_paths('There are %s items "%s" with lang %s',
649:                                lang_items, with_str, lang)
650:
651:     # items with `translation` metadata will be used as translations...
652:     candidate_items = [
653:         i for i in items
654:         if i.metadata.get('translation', 'false').lower() == 'false']
655:
656:     # ...unless all items with that slug are translations
657:     if not candidate_items:
658:         _warn_source_paths('All items ("%s") "%s" are translations',
659:                            items, with_str)
660:         candidate_items = items
661:
662:     # find items with default language
663:     original_items = [i for i in candidate_items if i.in_default_lang]
664:
665:     # if there is no article with default language, go back one step
666:     if not original_items:
667:         original_items = candidate_items
668:
669:     # warn if there are several original items
670:     if len(original_items) > 1:
671:         _warn_source_paths('There are %s original (not translated) items %s',
```

```
672:                             original_items, with_str)
673:     return original_items
674:
675:
676: def order_content(content_list, order_by='slug'):
677:     """ Sorts content.
678:
679:     order_by can be a string of an attribute or sorting function. If order_by
680:     is defined, content will be ordered by that attribute or sorting function.
681:     By default, content is ordered by slug.
682:
683:     Different content types can have default order_by attributes defined
684:     in settings, e.g. PAGES_ORDER_BY='sort-order', in which case `sort-order`
685:     should be a defined metadata attribute in each page.
686:     """
687:
688:     if order_by:
689:         if callable(order_by):
690:             try:
691:                 content_list.sort(key=order_by)
692:             except Exception:
693:                 logger.error('Error sorting with function %s', order_by)
694:         elif isinstance(order_by, str):
695:             if order_by.startswith('reversed-'):
696:                 order_reversed = True
697:                 order_by = order_by.replace('reversed-', '', 1)
698:             else:
699:                 order_reversed = False
700:
701:             if order_by == 'basename':
702:                 content_list.sort(
703:                     key=lambda x: os.path.basename(x.source_path or ''),
704:                     reverse=order_reversed)
705:             else:
706:                 try:
707:                     content_list.sort(key=attrgetter(order_by),
708:                                       reverse=order_reversed)
709:                 except AttributeError:
710:                     for content in content_list:
711:                         try:
712:                             getattr(content, order_by)
713:                         except AttributeError:
714:                             logger.warning(
715:                                 'There is no "%s" attribute in "%s". '
716:                                 'Defaulting to slug order.',
717:                                 order_by,
718:                                 content.get_relative_source_path(),
719:                                 extra={
720:                                     'limit_msg': ('More files are missing '
721:                                                   'the needed attribute.')
722:                                 })
723:         else:
724:             logger.warning(
725:                 'Invalid *_ORDER_BY setting (%s).'
726:                 'Valid options are strings and functions.', order_by)
727:
728:     return content_list
729:
730:
731: def folder_watcher(path, extensions, ignores=[]):
732:     '''Generator for monitoring a folder for modifications.
```

```
733:
734:     Returns a boolean indicating if files are changed since last check.
735:     Returns None if there are no matching files in the folder'''
736:
737:     def file_times(path):
738:         '''Return `mtime` for each file in path'''
739:
740:         for root, dirs, files in os.walk(path, followlinks=True):
741:             dirs[:] = [x for x in dirs if not x.startswith(os.curdir)]
742:
743:             for f in files:
744:                 valid_extension = f.endswith(tuple(extensions))
745:                 file_ignored = any(
746:                     fnmatch.fnmatch(f, ignore) for ignore in ignores
747:                 )
748:                 if valid_extension and not file_ignored:
749:                     try:
750:                         yield os.stat(os.path.join(root, f)).st_mtime
751:                     except OSError as e:
752:                         logger.warning('Caught Exception: %s', e)
753:
754:     LAST_MTIME = 0
755:     while True:
756:         try:
757:             mtime = max(file_times(path))
758:             if mtime > LAST_MTIME:
759:                 LAST_MTIME = mtime
760:                 yield True
761:         except ValueError:
762:             yield None
763:         else:
764:             yield False
765:
766:
767: def file_watcher(path):
768:     '''Generator for monitoring a file for modifications'''
769:     LAST_MTIME = 0
770:     while True:
771:         if path:
772:             try:
773:                 mtime = os.stat(path).st_mtime
774:             except OSError as e:
775:                 logger.warning('Caught Exception: %s', e)
776:                 continue
777:
778:             if mtime > LAST_MTIME:
779:                 LAST_MTIME = mtime
780:                 yield True
781:             else:
782:                 yield False
783:         else:
784:             yield None
785:
786:
787: def set_date_tzinfo(d, tz_name=None):
788:     """Set the timezone for dates that don't have tzinfo"""
789:     if tz_name and not d.tzinfo:
790:         tz = pytz.timezone(tz_name)
791:         d = tz.localize(d)
792:         return SafeDatetime(d.year, d.month, d.day, d.hour, d.minute, d.second,
793:                             d.microsecond, d.tzinfo)
```

```
794:     return d
795:
796:
797: def mkdir_p(path):
798:     os.makedirs(path, exist_ok=True)
799:
800:
801: def split_all(path):
802:     """Split a path into a list of components
803:
804:     While os.path.split() splits a single component off the back of
805:     `path`, this function splits all components:
806:
807:     >>> split_all(os.path.join('a', 'b', 'c'))
808:     ['a', 'b', 'c']
809:     """
810:     components = []
811:     path = path.lstrip('/')
812:     while path:
813:         head, tail = os.path.split(path)
814:         if tail:
815:             components.insert(0, tail)
816:         elif head == path:
817:             components.insert(0, head)
818:             break
819:         path = head
820:     return components
821:
822:
823: def is_selected_for_writing(settings, path):
824:     '''Check whether path is selected for writing
825:     according to the WRITE_SELECTED list
826:
827:     If WRITE_SELECTED is an empty list (default),
828:     any path is selected for writing.
829:     '''
830:     if settings['WRITE_SELECTED']:
831:         return path in settings['WRITE_SELECTED']
832:     else:
833:         return True
834:
835:
836: def path_to_file_url(path):
837:     '''Convert file-system path to file:// URL'''
838:     return urllib.parse.urljoin("file://", urllib.request.pathname2url(path))
839:
840:
841: def maybe_pluralize(count, singular, plural):
842:     '''
843:     Returns a formatted string containing count and plural if count is not 1
844:     Returns count and singular if count is 1
845:
846:     maybe_pluralize(0, 'Article', 'Articles') -> '0 Articles'
847:     maybe_pluralize(1, 'Article', 'Articles') -> '1 Article'
848:     maybe_pluralize(2, 'Article', 'Articles') -> '2 Articles'
849:
850:     '''
851:     selection = plural
852:     if count == 1:
853:         selection = singular
854:     return '{} {}'.format(count, selection)
```

```
# -*- coding: utf-8 -*-

import datetime
import logging
import os
import re
from collections import OrderedDict
from html import escape
from html.parser import HTMLParser
from io import StringIO

import docutils
import docutils.core
import docutils.io
from docutils.parsers.rst.languages import get_language as get_docutils_lang
from docutils.writers.html4css1 import HTMLTranslator, Writer

from pelican import rstdirectives  # NOQA
from pelican.cache import FileStampDataCacher
from pelican.contents import Author, Category, Page, Tag
from pelican.plugins import signals
from pelican.utils import get_date, pelican_open, posixize_path

try:
    from markdown import Markdown
except ImportError:
    Markdown = False  # NOQA

# Metadata processors have no way to discard an unwanted value, so we have
# them return this value instead to signal that it should be discarded later.
# This means that _filter_discardable_metadata() must be called on processed
# metadata dicts before use, to remove the items with the special value.
_DISCARD = object()

DUPLICATES_DEFINITIONS_ALLOWED = {
    'tags': False,
    'date': False,
    'modified': False,
    'status': False,
    'category': False,
    'author': False,
    'save_as': False,
    'url': False,
    'authors': False,
    'slug': False
}

METADATA_PROCESSORS = {
    'tags': lambda x, y: ([
        Tag(tag, y)
        for tag in ensure_metadata_list(x)
    ] or _DISCARD),
    'date': lambda x, y: get_date(x.replace('_', ' ')),
    'modified': lambda x, y: get_date(x),
    'status': lambda x, y: x.strip() or _DISCARD,
    'category': lambda x, y: _process_if_nonempty(Category, x, y),
    'author': lambda x, y: _process_if_nonempty(Author, x, y),
    'authors': lambda x, y: ([
        Author(author, y)
        for author in ensure_metadata_list(x)
    ] or _DISCARD),
```

```
    'slug': lambda x, y: x.strip() or _DISCARD,
}

logger = logging.getLogger(__name__)


def ensure_metadata_list(text):
    """Canonicalize the format of a list of authors or tags.  This works
       the same way as Docutils' "authors" field: if it's already a list,
       those boundaries are preserved; otherwise, it must be a string;
       if the string contains semicolons, it is split on semicolons;
       otherwise, it is split on commas.  This allows you to write
       author lists in either "Jane Doe, John Doe" or "Doe, Jane; Doe, John"
       format.

       Regardless, all list items undergo .strip() before returning, and
       empty items are discarded.
    """
    if isinstance(text, str):
        if ';' in text:
            text = text.split(';')
        else:
            text = text.split(',')

    return list(OrderedDict.fromkeys(
        [v for v in (w.strip() for w in text) if v]
    ))


def _process_if_nonempty(processor, name, settings):
    """Removes extra whitespace from name and applies a metadata processor.
    If name is empty or all whitespace, returns _DISCARD instead.
    """
    name = name.strip()
    return processor(name, settings) if name else _DISCARD


def _filter_discardable_metadata(metadata):
    """Return a copy of a dict, minus any items marked as discardable."""
    return {name: val for name, val in metadata.items() if val is not _DISCARD}


class BaseReader(object):
    """Base class to read files.

    This class is used to process static files, and it can be inherited for
    other types of file. A Reader class must have the following attributes:

    - enabled: (boolean) tell if the Reader class is enabled. It
      generally depends on the import of some dependency.
    - file_extensions: a list of file extensions that the Reader will process.
    - extensions: a list of extensions to use in the reader (typical use is
      Markdown).

    """
    enabled = True
    file_extensions = ['static']
    extensions = None

    def __init__(self, settings):
        self.settings = settings
```

```
123:
124:     def process_metadata(self, name, value):
125:         if name in METADATA_PROCESSORS:
126:             return METADATA_PROCESSORS[name](value, self.settings)
127:         return value
128:
129:     def read(self, source_path):
130:         "No-op parser"
131:         content = None
132:         metadata = {}
133:         return content, metadata
134:
135:
136: class _FieldBodyTranslator(HTMLTranslator):
137:
138:     def __init__(self, document):
139:         super().__init__(document)
140:         self.compact_p = None
141:
142:     def astext(self):
143:         return ''.join(self.body)
144:
145:     def visit_field_body(self, node):
146:         pass
147:
148:     def depart_field_body(self, node):
149:         pass
150:
151:
152: def render_node_to_html(document, node, field_body_translator_class):
153:     visitor = field_body_translator_class(document)
154:     node.walkabout(visitor)
155:     return visitor.astext()
156:
157:
158: class PelicanHTMLWriter(Writer):
159:
160:     def __init__(self):
161:         super().__init__()
162:         self.translator_class = PelicanHTMLTranslator
163:
164:
165: class PelicanHTMLTranslator(HTMLTranslator):
166:
167:     def visit_abbreviation(self, node):
168:         attrs = {}
169:         if node.hasattr('explanation'):
170:             attrs['title'] = node['explanation']
171:         self.body.append(self.starttag(node, 'abbr', '', **attrs))
172:
173:     def depart_abbreviation(self, node):
174:         self.body.append('</abbr>')
175:
176:     def visit_image(self, node):
177:         # set an empty alt if alt is not specified
178:         # avoids that alt is taken from src
179:         node['alt'] = node.get('alt', '')
180:         return HTMLTranslator.visit_image(self, node)
181:
182:
183: class RstReader(BaseReader):
```

```
184:     """Reader for reStructuredText files
185: 
186:     By default the output HTML is written using
187:     docutils.writers.html4css1.Writer and translated using a subclass of
188:     docutils.writers.html4css1.HTMLTranslator. If you want to override it with
189:     your own writer/translator (e.g. a HTML5-based one), pass your classes to
190:     these two attributes. Look in the source code for details.
191: 
192:         writer_class                    Used for writing contents
193:         field_body_translator_class     Used for translating metadata such
194:             as article summary
195: 
196:     """
197: 
198:     enabled = bool(docutils)
199:     file_extensions = ['rst']
200: 
201:     writer_class = PelicanHTMLWriter
202:     field_body_translator_class = _FieldBodyTranslator
203: 
204:     def __init__(self, *args, **kwargs):
205:         super().__init__(*args, **kwargs)
206: 
207:         lang_code = self.settings.get('DEFAULT_LANG', 'en')
208:         if get_docutils_lang(lang_code):
209:             self._language_code = lang_code
210:         else:
211:             logger.warning("Docutils has no localization for '%s'."
212:                            " Using 'en' instead.", lang_code)
213:             self._language_code = 'en'
214: 
215:     def _parse_metadata(self, document, source_path):
216:         """Return the dict containing document metadata"""
217:         formatted_fields = self.settings['FORMATTED_FIELDS']
218: 
219:         output = {}
220: 
221:         if document.first_child_matching_class(docutils.nodes.title) is None:
222:             logger.warning(
223:                 'Document title missing in file %s: '
224:                 'Ensure exactly one top level section',
225:                 source_path)
226: 
227:         for docinfo in document.traverse(docutils.nodes.docinfo):
228:             for element in docinfo.children:
229:                 if element.tagname == 'field':  # custom fields (e.g. summary)
230:                     name_elem, body_elem = element.children
231:                     name = name_elem.astext()
232:                     if name in formatted_fields:
233:                         value = render_node_to_html(
234:                             document, body_elem,
235:                             self.field_body_translator_class)
236:                     else:
237:                         value = body_elem.astext()
238:                 elif element.tagname == 'authors':  # author list
239:                     name = element.tagname
240:                     value = [element.astext() for element in element.children]
241:                 else:  # standard fields (e.g. address)
242:                     name = element.tagname
243:                     value = element.astext()
244:                 name = name.lower()
```

```
245:
246:                 output[name] = self.process_metadata(name, value)
247:         return output
248:
249:     def _get_publisher(self, source_path):
250:         extra_params = {'initial_header_level': '2',
251:                         'syntax_highlight': 'short',
252:                         'input_encoding': 'utf-8',
253:                         'language_code': self._language_code,
254:                         'halt_level': 2,
255:                         'traceback': True,
256:                         'warning_stream': StringIO(),
257:                         'embed_stylesheet': False}
258:         user_params = self.settings.get('DOCUTILS_SETTINGS')
259:         if user_params:
260:             extra_params.update(user_params)
261:
262:         pub = docutils.core.Publisher(
263:             writer=self.writer_class(),
264:             destination_class=docutils.io.StringOutput)
265:         pub.set_components('standalone', 'restructuredtext', 'html')
266:         pub.process_programmatic_settings(None, extra_params, None)
267:         pub.set_source(source_path=source_path)
268:         pub.publish()
269:         return pub
270:
271:     def read(self, source_path):
272:         """Parses restructured text"""
273:         pub = self._get_publisher(source_path)
274:         parts = pub.writer.parts
275:         content = parts.get('body')
276:
277:         metadata = self._parse_metadata(pub.document, source_path)
278:         metadata.setdefault('title', parts.get('title'))
279:
280:         return content, metadata
281:
282:
283: class MarkdownReader(BaseReader):
284:     """Reader for Markdown files"""
285:
286:     enabled = bool(Markdown)
287:     file_extensions = ['md', 'markdown', 'mkd', 'mdown']
288:
289:     def __init__(self, *args, **kwargs):
290:         super().__init__(*args, **kwargs)
291:         settings = self.settings['MARKDOWN']
292:         settings.setdefault('extension_configs', {})
293:         settings.setdefault('extensions', [])
294:         for extension in settings['extension_configs'].keys():
295:             if extension not in settings['extensions']:
296:                 settings['extensions'].append(extension)
297:         if 'markdown.extensions.meta' not in settings['extensions']:
298:             settings['extensions'].append('markdown.extensions.meta')
299:         self._source_path = None
300:
301:     def _parse_metadata(self, meta):
302:         """Return the dict containing document metadata"""
303:         formatted_fields = self.settings['FORMATTED_FIELDS']
304:
305:         # prevent metadata extraction in fields
```

```
306:         self._md.preprocessors.deregister('meta')
307:
308:         output = {}
309:         for name, value in meta.items():
310:             name = name.lower()
311:             if name in formatted_fields:
312:                 # formatted metadata is special case and join all list values
313:                 formatted_values = "\n".join(value)
314:                 # reset the markdown instance to clear any state
315:                 self._md.reset()
316:                 formatted = self._md.convert(formatted_values)
317:                 output[name] = self.process_metadata(name, formatted)
318:             elif not DUPLICATES_DEFINITIONS_ALLOWED.get(name, True):
319:                 if len(value) > 1:
320:                     logger.warning(
321:                         'Duplicate definition of `%s` '
322:                         'for %s. Using first one.',
323:                         name, self._source_path)
324:                 output[name] = self.process_metadata(name, value[0])
325:             elif len(value) > 1:
326:                 # handle list metadata as list of string
327:                 output[name] = self.process_metadata(name, value)
328:             else:
329:                 # otherwise, handle metadata as single string
330:                 output[name] = self.process_metadata(name, value[0])
331:         return output
332:
333:     def read(self, source_path):
334:         """Parse content and metadata of markdown files"""
335:
336:         self._source_path = source_path
337:         self._md = Markdown(**self.settings['MARKDOWN'])
338:         with pelican_open(source_path) as text:
339:             content = self._md.convert(text)
340:
341:         if hasattr(self._md, 'Meta'):
342:             metadata = self._parse_metadata(self._md.Meta)
343:         else:
344:             metadata = {}
345:         return content, metadata
346:
347:
348: class HTMLReader(BaseReader):
349:     """Parses HTML files as input, looking for meta, title, and body tags"""
350:
351:     file_extensions = ['htm', 'html']
352:     enabled = True
353:
354:     class _HTMLParser(HTMLParser):
355:         def __init__(self, settings, filename):
356:             super().__init__(convert_charrefs=False)
357:             self.body = ''
358:             self.metadata = {}
359:             self.settings = settings
360:
361:             self._data_buffer = ''
362:
363:             self._filename = filename
364:
365:             self._in_top_level = True
366:             self._in_head = False
```

```
367:             self._in_title = False
368:             self._in_body = False
369:             self._in_tags = False
370:
371:         def handle_starttag(self, tag, attrs):
372:             if tag == 'head' and self._in_top_level:
373:                 self._in_top_level = False
374:                 self._in_head = True
375:             elif tag == 'title' and self._in_head:
376:                 self._in_title = True
377:                 self._data_buffer = ''
378:             elif tag == 'body' and self._in_top_level:
379:                 self._in_top_level = False
380:                 self._in_body = True
381:                 self._data_buffer = ''
382:             elif tag == 'meta' and self._in_head:
383:                 self._handle_meta_tag(attrs)
384:
385:             elif self._in_body:
386:                 self._data_buffer += self.build_tag(tag, attrs, False)
387:
388:         def handle_endtag(self, tag):
389:             if tag == 'head':
390:                 if self._in_head:
391:                     self._in_head = False
392:                     self._in_top_level = True
393:             elif self._in_head and tag == 'title':
394:                 self._in_title = False
395:                 self.metadata['title'] = self._data_buffer
396:             elif tag == 'body':
397:                 self.body = self._data_buffer
398:                 self._in_body = False
399:                 self._in_top_level = True
400:             elif self._in_body:
401:                 self._data_buffer += '</{}>'.format(escape(tag))
402:
403:         def handle_startendtag(self, tag, attrs):
404:             if tag == 'meta' and self._in_head:
405:                 self._handle_meta_tag(attrs)
406:             if self._in_body:
407:                 self._data_buffer += self.build_tag(tag, attrs, True)
408:
409:         def handle_comment(self, data):
410:             self._data_buffer += '<!--{}-->'.format(data)
411:
412:         def handle_data(self, data):
413:             self._data_buffer += data
414:
415:         def handle_entityref(self, data):
416:             self._data_buffer += '&{};'.format(data)
417:
418:         def handle_charref(self, data):
419:             self._data_buffer += '&#{};'.format(data)
420:
421:         def build_tag(self, tag, attrs, close_tag):
422:             result = '<{}'.format(escape(tag))
423:             for k, v in attrs:
424:                 result += ' ' + escape(k)
425:                 if v is not None:
426:                     # If the attribute value contains a double quote, surround
427:                     # with single quotes, otherwise use double quotes.
```

```
428:                     if '"' in v:
429:                         result += "='{}'".format(escape(v, quote=False))
430:                     else:
431:                         result += '="{}"'.format(escape(v, quote=False))
432:             if close_tag:
433:                 return result + ' />'
434:             return result + '>'
435:
436:         def _handle_meta_tag(self, attrs):
437:             name = self._attr_value(attrs, 'name')
438:             if name is None:
439:                 attr_list = ['{}="{}"'.format(k, v) for k, v in attrs]
440:                 attr_serialized = ', '.join(attr_list)
441:                 logger.warning("Meta tag in file %s does not have a 'name' "
442:                                "attribute, skipping. Attributes: %s",
443:                                self._filename, attr_serialized)
444:                 return
445:             name = name.lower()
446:             contents = self._attr_value(attrs, 'content', '')
447:             if not contents:
448:                 contents = self._attr_value(attrs, 'contents', '')
449:                 if contents:
450:                     logger.warning(
451:                         "Meta tag attribute 'contents' used in file %s, should"
452:                         " be changed to 'content'",
453:                         self._filename,
454:                         extra={'limit_msg': "Other files have meta tag "
455:                                             "attribute 'contents' that should "
456:                                             "be changed to 'content'"})
457:
458:             if name == 'keywords':
459:                 name = 'tags'
460:
461:             if name in self.metadata:
462:                 # if this metadata already exists (i.e. a previous tag with the
463:                 # same name has already been specified then either convert to
464:                 # list or append to list
465:                 if isinstance(self.metadata[name], list):
466:                     self.metadata[name].append(contents)
467:                 else:
468:                     self.metadata[name] = [self.metadata[name], contents]
469:             else:
470:                 self.metadata[name] = contents
471:
472:         @classmethod
473:         def _attr_value(cls, attrs, name, default=None):
474:             return next((x[1] for x in attrs if x[0] == name), default)
475:
476:     def read(self, filename):
477:         """Parse content and metadata of HTML files"""
478:         with pelican_open(filename) as content:
479:             parser = self._HTMLParser(self.settings, filename)
480:             parser.feed(content)
481:             parser.close()
482:
483:         metadata = {}
484:         for k in parser.metadata:
485:             metadata[k] = self.process_metadata(k, parser.metadata[k])
486:         return parser.body, metadata
487:
488:
```

```
489: class Readers(FileStampDataCacher):
490:     """Interface for all readers.
491:
492:     This class contains a mapping of file extensions / Reader classes, to know
493:     which Reader class must be used to read a file (based on its extension).
494:     This is customizable both with the 'READERS' setting, and with the
495:     'readers_init' signall for plugins.
496:
497:     """
498:
499:     def __init__(self, settings=None, cache_name=''):
500:         self.settings = settings or {}
501:         self.readers = {}
502:         self.reader_classes = {}
503:
504:         for cls in [BaseReader] + BaseReader.__subclasses__():
505:             if not cls.enabled:
506:                 logger.debug('Missing dependencies for %s',
507:                              ', '.join(cls.file_extensions))
508:                 continue
509:
510:             for ext in cls.file_extensions:
511:                 self.reader_classes[ext] = cls
512:
513:         if self.settings['READERS']:
514:             self.reader_classes.update(self.settings['READERS'])
515:
516:         signals.readers_init.send(self)
517:
518:         for fmt, reader_class in self.reader_classes.items():
519:             if not reader_class:
520:                 continue
521:
522:             self.readers[fmt] = reader_class(self.settings)
523:
524:         # set up caching
525:         cache_this_level = (cache_name != '' and
526:                             self.settings['CONTENT_CACHING_LAYER'] == 'reader')
527:         caching_policy = cache_this_level and self.settings['CACHE_CONTENT']
528:         load_policy = cache_this_level and self.settings['LOAD_CONTENT_CACHE']
529:         super().__init__(settings, cache_name, caching_policy, load_policy)
530:
531:     @property
532:     def extensions(self):
533:         return self.readers.keys()
534:
535:     def read_file(self, base_path, path, content_class=Page, fmt=None,
536:                   context=None, preread_signal=None, preread_sender=None,
537:                   context_signal=None, context_sender=None):
538:         """Return a content object parsed with the given format."""
539:
540:         path = os.path.abspath(os.path.join(base_path, path))
541:         source_path = posixize_path(os.path.relpath(path, base_path))
542:         logger.debug(
543:             'Read file %s -> %s',
544:             source_path, content_class.__name__)
545:
546:         if not fmt:
547:             _, ext = os.path.splitext(os.path.basename(path))
548:             fmt = ext[1:]
549:
```

```
550:         if fmt not in self.readers:
551:             raise TypeError(
552:                 'Pelican does not know how to parse %s', path)
553: 
554:         if preread_signal:
555:             logger.debug(
556:                 'Signal %s.send(%s)',
557:                 preread_signal.name, preread_sender)
558:             preread_signal.send(preread_sender)
559: 
560:         reader = self.readers[fmt]
561: 
562:         metadata = _filter_discardable_metadata(default_metadata(
563:             settings=self.settings, process=reader.process_metadata))
564:         metadata.update(path_metadata(
565:             full_path=path, source_path=source_path,
566:             settings=self.settings))
567:         metadata.update(_filter_discardable_metadata(parse_path_metadata(
568:             source_path=source_path, settings=self.settings,
569:             process=reader.process_metadata)))
570:         reader_name = reader.__class__.__name__
571:         metadata['reader'] = reader_name.replace('Reader', '').lower()
572: 
573:         content, reader_metadata = self.get_cached_data(path, (None, None))
574:         if content is None:
575:             content, reader_metadata = reader.read(path)
576:             self.cache_data(path, (content, reader_metadata))
577:         metadata.update(_filter_discardable_metadata(reader_metadata))
578: 
579:         if content:
580:             # find images with empty alt
581:             find_empty_alt(content, path)
582: 
583:         # eventually filter the content with typogrify if asked so
584:         if self.settings['TYPOGRIFY']:
585:             from typogrify.filters import typogrify
586:             import smartypants
587: 
588:             typogrify_dashes = self.settings['TYPOGRIFY_DASHES']
589:             if typogrify_dashes == 'oldschool':
590:                 smartypants.Attr.default = smartypants.Attr.set2
591:             elif typogrify_dashes == 'oldschool_inverted':
592:                 smartypants.Attr.default = smartypants.Attr.set3
593:             else:
594:                 smartypants.Attr.default = smartypants.Attr.set1
595: 
596:             # Tell `smartypants` to also replace &quot; HTML entities with
597:             # smart quotes. This is necessary because Docutils has already
598:             # replaced double quotes with said entities by the time we run
599:             # this filter.
600:             smartypants.Attr.default |= smartypants.Attr.w
601: 
602:             def typogrify_wrapper(text):
603:                 """Ensures ignore_tags feature is backward compatible"""
604:                 try:
605:                     return typogrify(
606:                         text,
607:                         self.settings['TYPOGRIFY_IGNORE_TAGS'])
608:                 except TypeError:
609:                     return typogrify(text)
610: 
```

```
611:             if content:
612:                 content = typogrify_wrapper(content)
613:
614:             if 'title' in metadata:
615:                 metadata['title'] = typogrify_wrapper(metadata['title'])
616:
617:             if 'summary' in metadata:
618:                 metadata['summary'] = typogrify_wrapper(metadata['summary'])
619:
620:         if context_signal:
621:             logger.debug(
622:                 'Signal %s.send(%s, <metadata>)',
623:                 context_signal.name,
624:                 context_sender)
625:             context_signal.send(context_sender, metadata=metadata)
626:
627:         return content_class(content=content, metadata=metadata,
628:                              settings=self.settings, source_path=path,
629:                              context=context)
630:
631:
632: def find_empty_alt(content, path):
633:     """Find images with empty alt
634:
635:     Create warnings for all images with empty alt (up to a certain number),
636:     as they are really likely to be accessibility flaws.
637:
638:     """
639:     imgs = re.compile(r"""
640:         (?:
641:             # src before alt
642:             <img
643:             [^\>]*
644:             src=(['"])(.*?)\1
645:             [^\>]*
646:             alt=(['"])\3
647:         )|(?:
648:             # alt before src
649:             <img
650:             [^\>]*
651:             alt=(['"])\4
652:             [^\>]*
653:             src=(['"])(.*?)\5
654:         )
655:         """, re.X)
656:     for match in re.findall(imgs, content):
657:         logger.warning(
658:             'Empty alt attribute for image %s in %s',
659:             os.path.basename(match[1] + match[5]), path,
660:             extra={'limit_msg': 'Other images have empty alt attributes'})
661:
662:
663: def default_metadata(settings=None, process=None):
664:     metadata = {}
665:     if settings:
666:         for name, value in dict(settings.get('DEFAULT_METADATA', {})).items():
667:             if process:
668:                 value = process(name, value)
669:             metadata[name] = value
670:         if 'DEFAULT_CATEGORY' in settings:
671:             value = settings['DEFAULT_CATEGORY']
```

```
672:             if process:
673:                 value = process('category', value)
674:             metadata['category'] = value
675:         if settings.get('DEFAULT_DATE', None) and \
676:            settings['DEFAULT_DATE'] != 'fs':
677:             if isinstance(settings['DEFAULT_DATE'], str):
678:                 metadata['date'] = get_date(settings['DEFAULT_DATE'])
679:             else:
680:                 metadata['date'] = datetime.datetime(*settings['DEFAULT_DATE'])
681:     return metadata
682:
683:
684: def path_metadata(full_path, source_path, settings=None):
685:     metadata = {}
686:     if settings:
687:         if settings.get('DEFAULT_DATE', None) == 'fs':
688:             metadata['date'] = datetime.datetime.fromtimestamp(
689:                 os.stat(full_path).st_mtime)
690:
691:         # Apply EXTRA_PATH_METADATA for the source path and the paths of any
692:         # parent directories. Sorting EPM first ensures that the most specific
693:         # path wins conflicts.
694:
695:         epm = settings.get('EXTRA_PATH_METADATA', {})
696:         for path, meta in sorted(epm.items()):
697:             # Enforce a trailing slash when checking for parent directories.
698:             # This prevents false positives when one file or directory's name
699:             # is a prefix of another's.
700:             dirpath = os.path.join(path, '')
701:             if source_path == path or source_path.startswith(dirpath):
702:                 metadata.update(meta)
703:
704:     return metadata
705:
706:
707: def parse_path_metadata(source_path, settings=None, process=None):
708:     r"""Extract a metadata dictionary from a file's path
709:
710:     >>> import pprint
711:     >>> settings = {
712:     ...     'FILENAME_METADATA': r'(?P<slug>[^.]*).*',
713:     ...     'PATH_METADATA':
714:     ...         r'(?P<category>[^/]*)/(?P<date>\d{4}-\d{2}-\d{2})/.*',
715:     ...     }
716:     >>> reader = BaseReader(settings=settings)
717:     >>> metadata = parse_path_metadata(
718:     ...     source_path='my-cat/2013-01-01/my-slug.html',
719:     ...     settings=settings,
720:     ...     process=reader.process_metadata)
721:     >>> pprint.pprint(metadata)  # doctest: +ELLIPSIS
722:     {'category': <pelican.urlwrappers.Category object at ...>,
723:      'date': datetime.datetime(2013, 1, 1, 0, 0),
724:      'slug': 'my-slug'}
725:     """
726:     metadata = {}
727:     dirname, basename = os.path.split(source_path)
728:     base, ext = os.path.splitext(basename)
729:     subdir = os.path.basename(dirname)
730:     if settings:
731:         checks = []
732:         for key, data in [('FILENAME_METADATA', base),
```

```
733:                             ('PATH_METADATA', source_path)]:
734:                 checks.append((settings.get(key, None), data))
735:             if settings.get('USE_FOLDER_AS_CATEGORY', None):
736:                 checks.append(('(?P<category>.*)', subdir))
737:             for regexp, data in checks:
738:                 if regexp and data:
739:                     match = re.match(regexp, data)
740:                     if match:
741:                         # .items() for py3k compat.
742:                         for k, v in match.groupdict().items():
743:                             k = k.lower()  # metadata must be lowercase
744:                             if v is not None and k not in metadata:
745:                                 if process:
746:                                     v = process(k, v)
747:                                 metadata[k] = v
748:     return metadata
```

```
# -*- coding: utf-8 -*-

import copy
import importlib.util
import inspect
import locale
import logging
import os
import re
from os.path import isabs
from posixpath import join as posix_join

from pelican.log import LimitFilter


def load_source(name, path):
    spec = importlib.util.spec_from_file_location(name, path)
    mod = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(mod)
    return mod


logger = logging.getLogger(__name__)

DEFAULT_THEME = os.path.join(os.path.dirname(os.path.abspath(__file__)),
                             'themes', 'notmyidea')
DEFAULT_CONFIG = {
    'PATH': os.curdir,
    'ARTICLE_PATHS': [''],
    'ARTICLE_EXCLUDES': [],
    'PAGE_PATHS': ['pages'],
    'PAGE_EXCLUDES': [],
    'THEME': DEFAULT_THEME,
    'OUTPUT_PATH': 'output',
    'READERS': {},
    'STATIC_PATHS': ['images'],
    'STATIC_EXCLUDES': [],
    'STATIC_EXCLUDE_SOURCES': True,
    'THEME_STATIC_DIR': 'theme',
    'THEME_STATIC_PATHS': ['static', ],
    'FEED_ALL_ATOM': posix_join('feeds', 'all.atom.xml'),
    'CATEGORY_FEED_ATOM': posix_join('feeds', '{slug}.atom.xml'),
    'AUTHOR_FEED_ATOM': posix_join('feeds', '{slug}.atom.xml'),
    'AUTHOR_FEED_RSS': posix_join('feeds', '{slug}.rss.xml'),
    'TRANSLATION_FEED_ATOM': posix_join('feeds', 'all-{lang}.atom.xml'),
    'FEED_MAX_ITEMS': '',
    'RSS_FEED_SUMMARY_ONLY': True,
    'SITEURL': '',
    'SITENAME': 'A Pelican Blog',
    'DISPLAY_PAGES_ON_MENU': True,
    'DISPLAY_CATEGORIES_ON_MENU': True,
    'DOCUTILS_SETTINGS': {},
    'OUTPUT_SOURCES': False,
    'OUTPUT_SOURCES_EXTENSION': '.text',
    'USE_FOLDER_AS_CATEGORY': True,
    'DEFAULT_CATEGORY': 'misc',
    'WITH_FUTURE_DATES': True,
    'CSS_FILE': 'main.css',
    'NEWEST_FIRST_ARCHIVES': True,
    'REVERSE_CATEGORY_ORDER': False,
    'DELETE_OUTPUT_DIRECTORY': False,
```

```
    'OUTPUT_RETENTION': [],
    'INDEX_SAVE_AS': 'index.html',
    'ARTICLE_URL': '{slug}.html',
    'ARTICLE_SAVE_AS': '{slug}.html',
    'ARTICLE_ORDER_BY': 'reversed-date',
    'ARTICLE_LANG_URL': '{slug}-{lang}.html',
    'ARTICLE_LANG_SAVE_AS': '{slug}-{lang}.html',
    'DRAFT_URL': 'drafts/{slug}.html',
    'DRAFT_SAVE_AS': posix_join('drafts', '{slug}.html'),
    'DRAFT_LANG_URL': 'drafts/{slug}-{lang}.html',
    'DRAFT_LANG_SAVE_AS': posix_join('drafts', '{slug}-{lang}.html'),
    'PAGE_URL': 'pages/{slug}.html',
    'PAGE_SAVE_AS': posix_join('pages', '{slug}.html'),
    'PAGE_ORDER_BY': 'basename',
    'PAGE_LANG_URL': 'pages/{slug}-{lang}.html',
    'PAGE_LANG_SAVE_AS': posix_join('pages', '{slug}-{lang}.html'),
    'DRAFT_PAGE_URL': 'drafts/pages/{slug}.html',
    'DRAFT_PAGE_SAVE_AS': posix_join('drafts', 'pages', '{slug}.html'),
    'DRAFT_PAGE_LANG_URL': 'drafts/pages/{slug}-{lang}.html',
    'DRAFT_PAGE_LANG_SAVE_AS': posix_join('drafts', 'pages',
                                          '{slug}-{lang}.html'),
    'STATIC_URL': '{path}',
    'STATIC_SAVE_AS': '{path}',
    'STATIC_CREATE_LINKS': False,
    'STATIC_CHECK_IF_MODIFIED': False,
    'CATEGORY_URL': 'category/{slug}.html',
    'CATEGORY_SAVE_AS': posix_join('category', '{slug}.html'),
    'TAG_URL': 'tag/{slug}.html',
    'TAG_SAVE_AS': posix_join('tag', '{slug}.html'),
    'AUTHOR_URL': 'author/{slug}.html',
    'AUTHOR_SAVE_AS': posix_join('author', '{slug}.html'),
    'PAGINATION_PATTERNS': [
        (1, '{name}{extension}', '{name}{extension}'),
        (2, '{name}{number}{extension}', '{name}{number}{extension}'),
    ],
    'YEAR_ARCHIVE_URL': '',
    'YEAR_ARCHIVE_SAVE_AS': '',
    'MONTH_ARCHIVE_URL': '',
    'MONTH_ARCHIVE_SAVE_AS': '',
    'DAY_ARCHIVE_URL': '',
    'DAY_ARCHIVE_SAVE_AS': '',
    'RELATIVE_URLS': False,
    'DEFAULT_LANG': 'en',
    'ARTICLE_TRANSLATION_ID': 'slug',
    'PAGE_TRANSLATION_ID': 'slug',
    'DIRECT_TEMPLATES': ['index', 'tags', 'categories', 'authors', 'archives'],
    'THEME_TEMPLATES_OVERRIDES': [],
    'PAGINATED_TEMPLATES': {'index': None, 'tag': None, 'category': None,
                            'author': None},
    'PELICAN_CLASS': 'pelican.Pelican',
    'DEFAULT_DATE_FORMAT': '%a %d %B %Y',
    'DATE_FORMATS': {},
    'MARKDOWN': {
        'extension_configs': {
            'markdown.extensions.codehilite': {'css_class': 'highlight'},
            'markdown.extensions.extra': {},
            'markdown.extensions.meta': {},
        },
        'output_format': 'html5',
    },
    'JINJA_FILTERS': {},
```

```
    'JINJA_GLOBALS': {},
    'JINJA_TESTS': {},
    'JINJA_ENVIRONMENT': {
        'trim_blocks': True,
        'lstrip_blocks': True,
        'extensions': [],
    },
    'LOG_FILTER': [],
    'LOCALE': [''],  # defaults to user locale
    'DEFAULT_PAGINATION': False,
    'DEFAULT_ORPHANS': 0,
    'DEFAULT_METADATA': {},
    'FILENAME_METADATA': r'(?P<date>\d{4}-\d{2}-\d{2}).*',
    'PATH_METADATA': '',
    'EXTRA_PATH_METADATA': {},
    'ARTICLE_PERMALINK_STRUCTURE': '',
    'TYPOGRIFY': False,
    'TYPOGRIFY_IGNORE_TAGS': [],
    'TYPOGRIFY_DASHES': 'default',
    'SUMMARY_END_MARKER': 'â\200¦',
    'SUMMARY_MAX_LENGTH': 50,
    'PLUGIN_PATHS': [],
    'PLUGINS': None,
    'PYGMENTS_RST_OPTIONS': {},
    'TEMPLATE_PAGES': {},
    'TEMPLATE_EXTENSIONS': ['.html'],
    'IGNORE_FILES': ['.#*'],
    'SLUG_REGEX_SUBSTITUTIONS': [
        (r'[^\w\s-]', ''),  # remove non-alphabetical/whitespace/'-' chars
        (r'(?u)\A\s*', ''),  # strip leading whitespace
        (r'(?u)\s*\Z', ''),  # strip trailing whitespace
        (r'[-\s]+', '-'),  # reduce multiple whitespace or '-' to single '-'
    ],
    'INTRASITE_LINK_REGEX': '[{|](?P<what>.*?)[|}]',
    'SLUGIFY_SOURCE': 'title',
    'CACHE_CONTENT': False,
    'CONTENT_CACHING_LAYER': 'reader',
    'CACHE_PATH': 'cache',
    'GZIP_CACHE': True,
    'CHECK_MODIFIED_METHOD': 'mtime',
    'LOAD_CONTENT_CACHE': False,
    'WRITE_SELECTED': [],
    'FORMATTED_FIELDS': ['summary'],
    'PORT': 8000,
    'BIND': '127.0.0.1',
}

PYGMENTS_RST_OPTIONS = None


def read_settings(path=None, override=None):
    settings = override or {}

    if path:
        settings = dict(get_settings_from_file(path), **settings)

    if settings:
        settings = handle_deprecated_settings(settings)

    if path:
        # Make relative paths absolute
```

```
184:         def getabs(maybe_relative, base_path=path):
185:             if isabs(maybe_relative):
186:                 return maybe_relative
187:             return os.path.abspath(os.path.normpath(os.path.join(
188:                 os.path.dirname(base_path), maybe_relative)))
189:
190:         for p in ['PATH', 'OUTPUT_PATH', 'THEME', 'CACHE_PATH']:
191:             if settings.get(p) is not None:
192:                 absp = getabs(settings[p])
193:                 # THEME may be a name rather than a path
194:                 if p != 'THEME' or os.path.exists(absp):
195:                     settings[p] = absp
196:
197:         if settings.get('PLUGIN_PATHS') is not None:
198:             settings['PLUGIN_PATHS'] = [getabs(pluginpath)
199:                                         for pluginpath
200:                                         in settings['PLUGIN_PATHS']]
201:
202:     settings = dict(copy.deepcopy(DEFAULT_CONFIG), **settings)
203:     settings = configure_settings(settings)
204:
205:     # This is because there doesn't seem to be a way to pass extra
206:     # parameters to docutils directive handlers, so we have to have a
207:     # variable here that we'll import from within Pygments.run (see
208:     # rstdirectives.py) to see what the user defaults were.
209:     global PYGMENTS_RST_OPTIONS
210:     PYGMENTS_RST_OPTIONS = settings.get('PYGMENTS_RST_OPTIONS', None)
211:     return settings
212:
213:
214: def get_settings_from_module(module=None):
215:     """Loads settings from a module, returns a dictionary."""
216:
217:     context = {}
218:     if module is not None:
219:         context.update(
220:             (k, v) for k, v in inspect.getmembers(module) if k.isupper())
221:     return context
222:
223:
224: def get_settings_from_file(path):
225:     """Loads settings from a file path, returning a dict."""
226:
227:     name, ext = os.path.splitext(os.path.basename(path))
228:     module = load_source(name, path)
229:     return get_settings_from_module(module)
230:
231:
232: def get_jinja_environment(settings):
233:     """Sets the environment for Jinja"""
234:
235:     jinja_env = settings.setdefault('JINJA_ENVIRONMENT',
236:                                     DEFAULT_CONFIG['JINJA_ENVIRONMENT'])
237:
238:     # Make sure we include the defaults if the user has set env variables
239:     for key, value in DEFAULT_CONFIG['JINJA_ENVIRONMENT'].items():
240:         if key not in jinja_env:
241:             jinja_env[key] = value
242:
243:     return settings
244:
```

```
245:
246: def _printf_s_to_format_field(printf_string, format_field):
247:     """Tries to replace %s with {format_field} in the provided printf_string.
248:     Raises ValueError in case of failure.
249:     """
250:     TEST_STRING = 'PELICAN_PRINTF_S_DEPRECATION'
251:     expected = printf_string % TEST_STRING
252:
253:     result = printf_string.replace('{', '{{').replace('}', '}}') \
254:         % '{{{}}}'.format(format_field)
255:     if result.format(**{format_field: TEST_STRING}) != expected:
256:         raise ValueError('Failed to safely replace %s with {{{}}}'.format(
257:             format_field))
258:
259:     return result
260:
261:
262: def handle_deprecated_settings(settings):
263:     """Converts deprecated settings and issues warnings. Issues an exception
264:     if both old and new setting is specified.
265:     """
266:
267:     # PLUGIN_PATH -> PLUGIN_PATHS
268:     if 'PLUGIN_PATH' in settings:
269:         logger.warning('PLUGIN_PATH setting has been replaced by '
270:                        'PLUGIN_PATHS, moving it to the new setting name.')
271:         settings['PLUGIN_PATHS'] = settings['PLUGIN_PATH']
272:         del settings['PLUGIN_PATH']
273:
274:     # PLUGIN_PATHS: str -> [str]
275:     if isinstance(settings.get('PLUGIN_PATHS'), str):
276:         logger.warning("Defining PLUGIN_PATHS setting as string "
277:                        "has been deprecated (should be a list)")
278:         settings['PLUGIN_PATHS'] = [settings['PLUGIN_PATHS']]
279:
280:     # JINJA_EXTENSIONS -> JINJA_ENVIRONMENT > extensions
281:     if 'JINJA_EXTENSIONS' in settings:
282:         logger.warning('JINJA_EXTENSIONS setting has been deprecated, '
283:                        'moving it to JINJA_ENVIRONMENT setting.')
284:         settings['JINJA_ENVIRONMENT']['extensions'] = \
285:             settings['JINJA_EXTENSIONS']
286:         del settings['JINJA_EXTENSIONS']
287:
288:     # {ARTICLE,PAGE}_DIR -> {ARTICLE,PAGE}_PATHS
289:     for key in ['ARTICLE', 'PAGE']:
290:         old_key = key + '_DIR'
291:         new_key = key + '_PATHS'
292:         if old_key in settings:
293:             logger.warning(
294:                 'Deprecated setting %s, moving it to %s list',
295:                 old_key, new_key)
296:             settings[new_key] = [settings[old_key]]   # also make a list
297:             del settings[old_key]
298:
299:     # EXTRA_TEMPLATES_PATHS -> THEME_TEMPLATES_OVERRIDES
300:     if 'EXTRA_TEMPLATES_PATHS' in settings:
301:         logger.warning('EXTRA_TEMPLATES_PATHS is deprecated use '
302:                        'THEME_TEMPLATES_OVERRIDES instead.')
303:         if ('THEME_TEMPLATES_OVERRIDES' in settings and
304:                 settings['THEME_TEMPLATES_OVERRIDES']):
305:             raise Exception(
```

```
306:                 'Setting both EXTRA_TEMPLATES_PATHS and '
307:                 'THEME_TEMPLATES_OVERRIDES is not permitted. Please move to '
308:                 'only setting THEME_TEMPLATES_OVERRIDES.')
309:         settings['THEME_TEMPLATES_OVERRIDES'] = \
310:             settings['EXTRA_TEMPLATES_PATHS']
311:         del settings['EXTRA_TEMPLATES_PATHS']
312:
313:     # MD_EXTENSIONS -> MARKDOWN
314:     if 'MD_EXTENSIONS' in settings:
315:         logger.warning('MD_EXTENSIONS is deprecated use MARKDOWN '
316:                        'instead. Falling back to the default.')
317:         settings['MARKDOWN'] = DEFAULT_CONFIG['MARKDOWN']
318:
319:     # LESS_GENERATOR -> Webassets plugin
320:     # FILES_TO_COPY -> STATIC_PATHS, EXTRA_PATH_METADATA
321:     for old, new, doc in [
322:             ('LESS_GENERATOR', 'the Webassets plugin', None),
323:             ('FILES_TO_COPY', 'STATIC_PATHS and EXTRA_PATH_METADATA',
324:                 'https://github.com/getpelican/pelican/'
325:                 'blob/master/docs/settings.rst#path-metadata'),
326:     ]:
327:         if old in settings:
328:             message = 'The {} setting has been removed in favor of {}'.format(
329:                 old, new)
330:             if doc:
331:                 message += ', see {} for details'.format(doc)
332:             logger.warning(message)
333:
334:     # PAGINATED_DIRECT_TEMPLATES -> PAGINATED_TEMPLATES
335:     if 'PAGINATED_DIRECT_TEMPLATES' in settings:
336:         message = 'The {} setting has been removed in favor of {}'.format(
337:             'PAGINATED_DIRECT_TEMPLATES', 'PAGINATED_TEMPLATES')
338:         logger.warning(message)
339:
340:         # set PAGINATED_TEMPLATES
341:         if 'PAGINATED_TEMPLATES' not in settings:
342:             settings['PAGINATED_TEMPLATES'] = {
343:                 'tag': None, 'category': None, 'author': None}
344:
345:         for t in settings['PAGINATED_DIRECT_TEMPLATES']:
346:             if t not in settings['PAGINATED_TEMPLATES']:
347:                 settings['PAGINATED_TEMPLATES'][t] = None
348:         del settings['PAGINATED_DIRECT_TEMPLATES']
349:
350:     # {SLUG,CATEGORY,TAG,AUTHOR}_SUBSTITUTIONS ->
351:     # {SLUG,CATEGORY,TAG,AUTHOR}_REGEX_SUBSTITUTIONS
352:     url_settings_url = \
353:         'http://docs.getpelican.com/en/latest/settings.html#url-settings'
354:     flavours = {'SLUG', 'CATEGORY', 'TAG', 'AUTHOR'}
355:     old_values = {f: settings[f + '_SUBSTITUTIONS']
356:                   for f in flavours if f + '_SUBSTITUTIONS' in settings}
357:     new_values = {f: settings[f + '_REGEX_SUBSTITUTIONS']
358:                   for f in flavours if f + '_REGEX_SUBSTITUTIONS' in settings}
359:     if old_values and new_values:
360:         raise Exception(
361:             'Setting both {new_key} and {old_key} (or variants thereof) is '
362:             'not permitted. Please move to only setting {new_key}.'
363:             .format(old_key='SLUG_SUBSTITUTIONS',
364:                     new_key='SLUG_REGEX_SUBSTITUTIONS'))
365:     if old_values:
366:         message = ('{} and variants thereof are deprecated and will be '
```

```
367:                     'removed in the future. Please use {} and variants thereof '
368:                     'instead. Check {}.'
369:                     .format('SLUG_SUBSTITUTIONS', 'SLUG_REGEX_SUBSTITUTIONS',
370:                             url_settings_url))
371:         logger.warning(message)
372:         if old_values.get('SLUG'):
373:             for f in {'CATEGORY', 'TAG'}:
374:                 if old_values.get(f):
375:                     old_values[f] = old_values['SLUG'] + old_values[f]
376:             old_values['AUTHOR'] = old_values.get('AUTHOR', [])
377:         for f in flavours:
378:             if old_values.get(f) is not None:
379:                 regex_subs = []
380:                 # by default will replace non-alphanum characters
381:                 replace = True
382:                 for tpl in old_values[f]:
383:                     try:
384:                         src, dst, skip = tpl
385:                         if skip:
386:                             replace = False
387:                     except ValueError:
388:                         src, dst = tpl
389:                     regex_subs.append(
390:                         (re.escape(src), dst.replace('\\', r'\\')))
391:
392:                 if replace:
393:                     regex_subs += [
394:                         (r'[^\w\s-]', ''),
395:                         (r'(?u)\A\s*', ''),
396:                         (r'(?u)\s*\Z', ''),
397:                         (r'[-\s]+', '-'),
398:                     ]
399:                 else:
400:                     regex_subs += [
401:                         (r'(?u)\A\s*', ''),
402:                         (r'(?u)\s*\Z', ''),
403:                     ]
404:                 settings[f + '_REGEX_SUBSTITUTIONS'] = regex_subs
405:             settings.pop(f + '_SUBSTITUTIONS', None)
406:
407:     # `%s` -> '{slug}` or `{lang}` in FEED settings
408:     for key in ['TRANSLATION_FEED_ATOM',
409:                 'TRANSLATION_FEED_RSS'
410:                 ]:
411:         if settings.get(key) and '%s' in settings[key]:
412:             logger.warning('%%s usage in %s is deprecated, use {lang} '
413:                            'instead.', key)
414:             try:
415:                 settings[key] = _printf_s_to_format_field(
416:                     settings[key], 'lang')
417:             except ValueError:
418:                 logger.warning('Failed to convert %%s to {lang} for %s. '
419:                                'Falling back to default.', key)
420:                 settings[key] = DEFAULT_CONFIG[key]
421:     for key in ['AUTHOR_FEED_ATOM',
422:                 'AUTHOR_FEED_RSS',
423:                 'CATEGORY_FEED_ATOM',
424:                 'CATEGORY_FEED_RSS',
425:                 'TAG_FEED_ATOM',
426:                 'TAG_FEED_RSS',
427:                 ]:
```

```
428:         if settings.get(key) and '%s' in settings[key]:
429:             logger.warning('%%s usage in %s is deprecated, use {slug} '
430:                            'instead.', key)
431:             try:
432:                 settings[key] = _printf_s_to_format_field(
433:                     settings[key], 'slug')
434:             except ValueError:
435:                 logger.warning('Failed to convert %%s to {slug} for %s. '
436:                                'Falling back to default.', key)
437:                 settings[key] = DEFAULT_CONFIG[key]
438: 
439:     # CLEAN_URLS
440:     if settings.get('CLEAN_URLS', False):
441:         logger.warning('Found deprecated `CLEAN_URLS` in settings.'
442:                        ' Modifying the following settings for the'
443:                        ' same behaviour.')
444: 
445:         settings['ARTICLE_URL'] = '{slug}/'
446:         settings['ARTICLE_LANG_URL'] = '{slug}-{lang}/'
447:         settings['PAGE_URL'] = 'pages/{slug}/'
448:         settings['PAGE_LANG_URL'] = 'pages/{slug}-{lang}/'
449: 
450:         for setting in ('ARTICLE_URL', 'ARTICLE_LANG_URL', 'PAGE_URL',
451:                         'PAGE_LANG_URL'):
452:             logger.warning("%s = '%s'", setting, settings[setting])
453: 
454:     # AUTORELOAD_IGNORE_CACHE -> --ignore-cache
455:     if settings.get('AUTORELOAD_IGNORE_CACHE'):
456:         logger.warning('Found deprecated `AUTORELOAD_IGNORE_CACHE` in '
457:                        'settings. Use --ignore-cache instead.')
458:         settings.pop('AUTORELOAD_IGNORE_CACHE')
459: 
460:     # ARTICLE_PERMALINK_STRUCTURE
461:     if settings.get('ARTICLE_PERMALINK_STRUCTURE', False):
462:         logger.warning('Found deprecated `ARTICLE_PERMALINK_STRUCTURE` in'
463:                        ' settings.  Modifying the following settings for'
464:                        ' the same behaviour.')
465: 
466:         structure = settings['ARTICLE_PERMALINK_STRUCTURE']
467: 
468:         # Convert %(variable) into {variable}.
469:         structure = re.sub(r'%\((\w+)\)s', r'{\g<1>}', structure)
470: 
471:         # Convert %x into {date:%x} for strftime
472:         structure = re.sub(r'(%[A-z])', r'{date:\g<1>}', structure)
473: 
474:         # Strip a / prefix
475:         structure = re.sub('^/', '', structure)
476: 
477:         for setting in ('ARTICLE_URL', 'ARTICLE_LANG_URL', 'PAGE_URL',
478:                         'PAGE_LANG_URL', 'DRAFT_URL', 'DRAFT_LANG_URL',
479:                         'ARTICLE_SAVE_AS', 'ARTICLE_LANG_SAVE_AS',
480:                         'DRAFT_SAVE_AS', 'DRAFT_LANG_SAVE_AS',
481:                         'PAGE_SAVE_AS', 'PAGE_LANG_SAVE_AS'):
482:             settings[setting] = os.path.join(structure,
483:                                              settings[setting])
484:             logger.warning("%s = '%s'", setting, settings[setting])
485: 
486:     # {,TAG,CATEGORY,TRANSLATION}_FEED -> {,TAG,CATEGORY,TRANSLATION}_FEED_ATOM
487:     for new, old in [('FEED', 'FEED_ATOM'), ('TAG_FEED', 'TAG_FEED_ATOM'),
488:                      ('CATEGORY_FEED', 'CATEGORY_FEED_ATOM'),
```

```
489:                         ('TRANSLATION_FEED', 'TRANSLATION_FEED_ATOM')]:
490:         if settings.get(new, False):
491:             logger.warning(
492:                 'Found deprecated `%(new)s` in settings. Modify %(new)s '
493:                 'to %(old)s in your settings and theme for the same '
494:                 'behavior. Temporarily setting %(old)s for backwards '
495:                 'compatibility.',
496:                 {'new': new, 'old': old}
497:             )
498:             settings[old] = settings[new]
499:
500:     return settings
501:
502:
503: def configure_settings(settings):
504:     """Provide optimizations, error checking, and warnings for the given
505:     settings.
506:     Also, specify the log messages to be ignored.
507:     """
508:     if 'PATH' not in settings or not os.path.isdir(settings['PATH']):
509:         raise Exception('You need to specify a path containing the content'
510:                         ' (see pelican --help for more information)')
511:
512:     # specify the log messages to be ignored
513:     log_filter = settings.get('LOG_FILTER', DEFAULT_CONFIG['LOG_FILTER'])
514:     LimitFilter._ignore.update(set(log_filter))
515:
516:     # lookup the theme in "pelican/themes" if the given one doesn't exist
517:     if not os.path.isdir(settings['THEME']):
518:         theme_path = os.path.join(
519:             os.path.dirname(os.path.abspath(__file__)),
520:             'themes',
521:             settings['THEME'])
522:         if os.path.exists(theme_path):
523:             settings['THEME'] = theme_path
524:         else:
525:             raise Exception("Could not find the theme %s"
526:                             % settings['THEME'])
527:
528:     # make paths selected for writing absolute if necessary
529:     settings['WRITE_SELECTED'] = [
530:         os.path.abspath(path) for path in
531:         settings.get('WRITE_SELECTED', DEFAULT_CONFIG['WRITE_SELECTED'])
532:     ]
533:
534:     # standardize strings to lowercase strings
535:     for key in ['DEFAULT_LANG']:
536:         if key in settings:
537:             settings[key] = settings[key].lower()
538:
539:     # set defaults for Jinja environment
540:     settings = get_jinja_environment(settings)
541:
542:     # standardize strings to lists
543:     for key in ['LOCALE']:
544:         if key in settings and isinstance(settings[key], str):
545:             settings[key] = [settings[key]]
546:
547:     # check settings that must be a particular type
548:     for key, types in [
549:             ('OUTPUT_SOURCES_EXTENSION', str),
```

```
            ('FILENAME_METADATA', str),
    ]:
        if key in settings and not isinstance(settings[key], types):
            value = settings.pop(key)
            logger.warn(
                'Detected misconfigured %s (%s), '
                'falling back to the default (%s)',
                key, value, DEFAULT_CONFIG[key])

    # try to set the different locales, fallback on the default.
    locales = settings.get('LOCALE', DEFAULT_CONFIG['LOCALE'])

    for locale_ in locales:
        try:
            locale.setlocale(locale.LC_ALL, str(locale_))
            break  # break if it is successful
        except locale.Error:
            pass
    else:
        logger.warning(
            "Locale could not be set. Check the LOCALE setting, ensuring it "
            "is valid and available on your system.")

    if ('SITEURL' in settings):
        # If SITEURL has a trailing slash, remove it and provide a warning
        siteurl = settings['SITEURL']
        if (siteurl.endswith('/')):
            settings['SITEURL'] = siteurl[:-1]
            logger.warning("Removed extraneous trailing slash from SITEURL.")
        # If SITEURL is defined but FEED_DOMAIN isn't,
        # set FEED_DOMAIN to SITEURL
        if 'FEED_DOMAIN' not in settings:
            settings['FEED_DOMAIN'] = settings['SITEURL']

    # check content caching layer and warn of incompatibilities
    if settings.get('CACHE_CONTENT', False) and \
            settings.get('CONTENT_CACHING_LAYER', '') == 'generator' and \
            settings.get('WITH_FUTURE_DATES', False):
        logger.warning(
            "WITH_FUTURE_DATES conflicts with CONTENT_CACHING_LAYER "
            "set to 'generator', use 'reader' layer instead")

    # Warn if feeds are generated with both SITEURL & FEED_DOMAIN undefined
    feed_keys = [
        'FEED_ATOM', 'FEED_RSS',
        'FEED_ALL_ATOM', 'FEED_ALL_RSS',
        'CATEGORY_FEED_ATOM', 'CATEGORY_FEED_RSS',
        'AUTHOR_FEED_ATOM', 'AUTHOR_FEED_RSS',
        'TAG_FEED_ATOM', 'TAG_FEED_RSS',
        'TRANSLATION_FEED_ATOM', 'TRANSLATION_FEED_RSS',
    ]

    if any(settings.get(k) for k in feed_keys):
        if not settings.get('SITEURL'):
            logger.warning('Feeds generated without SITEURL set properly may'
                           ' not be valid')

    if 'TIMEZONE' not in settings:
        logger.warning(
            'No timezone information specified in the settings. Assuming'
            ' your timezone is UTC for feed generation. Check '
```

```
611:             'http://docs.getpelican.com/en/latest/settings.html#timezone '
612:             'for more information')
613:
614:     # fix up pagination rules
615:     from pelican.paginator import PaginationRule
616:     pagination_rules = [
617:         PaginationRule(*r) for r in settings.get(
618:             'PAGINATION_PATTERNS',
619:             DEFAULT_CONFIG['PAGINATION_PATTERNS'],
620:         )
621:     ]
622:     settings['PAGINATION_PATTERNS'] = sorted(
623:         pagination_rules,
624:         key=lambda r: r[0],
625:     )
626:
627:     # Save people from accidentally setting a string rather than a list
628:     path_keys = (
629:         'ARTICLE_EXCLUDES',
630:         'DEFAULT_METADATA',
631:         'DIRECT_TEMPLATES',
632:         'THEME_TEMPLATES_OVERRIDES',
633:         'FILES_TO_COPY',
634:         'IGNORE_FILES',
635:         'PAGINATED_DIRECT_TEMPLATES',
636:         'PLUGINS',
637:         'STATIC_EXCLUDES',
638:         'STATIC_PATHS',
639:         'THEME_STATIC_PATHS',
640:         'ARTICLE_PATHS',
641:         'PAGE_PATHS',
642:     )
643:     for PATH_KEY in filter(lambda k: k in settings, path_keys):
644:         if isinstance(settings[PATH_KEY], str):
645:             logger.warning("Detected misconfiguration with %s setting "
646:                            "(must be a list), falling back to the default",
647:                            PATH_KEY)
648:             settings[PATH_KEY] = DEFAULT_CONFIG[PATH_KEY]
649:
650:     # Add {PAGE,ARTICLE}_PATHS to {ARTICLE,PAGE}_EXCLUDES
651:     mutually_exclusive = ('ARTICLE', 'PAGE')
652:     for type_1, type_2 in [mutually_exclusive, mutually_exclusive[::-1]]:
653:         try:
654:             includes = settings[type_1 + '_PATHS']
655:             excludes = settings[type_2 + '_EXCLUDES']
656:             for path in includes:
657:                 if path not in excludes:
658:                     excludes.append(path)
659:         except KeyError:
660:             continue            # setting not specified, nothing to do
661:
662:     return settings
```

```
 1: # -*- coding: utf-8 -*-
 2:
 3: import copy
 4: import datetime
 5: import locale
 6: import logging
 7: import os
 8: import re
 9: from urllib.parse import urljoin, urlparse, urlunparse
10:
11: import pytz
12:
13: from pelican.plugins import signals
14: from pelican.settings import DEFAULT_CONFIG
15: from pelican.utils import (deprecated_attribute, memoized, path_to_url,
16:                            posixize_path, sanitised_join, set_date_tzinfo,
17:                            slugify, truncate_html_words)
18:
19: # Import these so that they're avalaible when you import from pelican.contents.
20: from pelican.urlwrappers import (Author, Category, Tag, URLWrapper)  # NOQA
21:
22: logger = logging.getLogger(__name__)
23:
24:
25: class Content(object):
26:     """Represents a content.
27:
28:     :param content: the string to parse, containing the original content.
29:     :param metadata: the metadata associated to this page (optional).
30:     :param settings: the settings dictionary (optional).
31:     :param source_path: The location of the source of this content (if any).
32:     :param context: The shared context between generators.
33:
34:     """
35:     @deprecated_attribute(old='filename', new='source_path', since=(3, 2, 0))
36:     def filename():
37:         return None
38:
39:     def __init__(self, content, metadata=None, settings=None,
40:                  source_path=None, context=None):
41:         if metadata is None:
42:             metadata = {}
43:         if settings is None:
44:             settings = copy.deepcopy(DEFAULT_CONFIG)
45:
46:         self.settings = settings
47:         self._content = content
48:         if context is None:
49:             context = {}
50:         self._context = context
51:         self.translations = []
52:
53:         local_metadata = dict()
54:         local_metadata.update(metadata)
55:
56:         # set metadata as attributes
57:         for key, value in local_metadata.items():
58:             if key in ('save_as', 'url'):
59:                 key = 'override_' + key
60:             setattr(self, key.lower(), value)
61:
```

```
62:         # also keep track of the metadata attributes available
63:         self.metadata = local_metadata
64:
65:         # default template if it's not defined in page
66:         self.template = self._get_template()
67:
68:         # First, read the authors from "authors", if not, fallback to "author"
69:         # and if not use the settings defined one, if any.
70:         if not hasattr(self, 'author'):
71:             if hasattr(self, 'authors'):
72:                 self.author = self.authors[0]
73:             elif 'AUTHOR' in settings:
74:                 self.author = Author(settings['AUTHOR'], settings)
75:
76:         if not hasattr(self, 'authors') and hasattr(self, 'author'):
77:             self.authors = [self.author]
78:
79:         # XXX Split all the following code into pieces, there is too much here.
80:
81:         # manage languages
82:         self.in_default_lang = True
83:         if 'DEFAULT_LANG' in settings:
84:             default_lang = settings['DEFAULT_LANG'].lower()
85:             if not hasattr(self, 'lang'):
86:                 self.lang = default_lang
87:
88:             self.in_default_lang = (self.lang == default_lang)
89:
90:         # create the slug if not existing, generate slug according to
91:         # setting of SLUG_ATTRIBUTE
92:         if not hasattr(self, 'slug'):
93:             if (settings['SLUGIFY_SOURCE'] == 'title' and
94:                     hasattr(self, 'title')):
95:                 self.slug = slugify(
96:                     self.title,
97:                     regex_subs=settings.get('SLUG_REGEX_SUBSTITUTIONS', []))
98:             elif (settings['SLUGIFY_SOURCE'] == 'basename' and
99:                     source_path is not None):
100:                basename = os.path.basename(
101:                    os.path.splitext(source_path)[0])
102:                self.slug = slugify(
103:                    basename,
104:                    regex_subs=settings.get('SLUG_REGEX_SUBSTITUTIONS', []))
105:
106:         self.source_path = source_path
107:         self.relative_source_path = self.get_relative_source_path()
108:
109:         # manage the date format
110:         if not hasattr(self, 'date_format'):
111:             if hasattr(self, 'lang') and self.lang in settings['DATE_FORMATS']:
112:                 self.date_format = settings['DATE_FORMATS'][self.lang]
113:             else:
114:                 self.date_format = settings['DEFAULT_DATE_FORMAT']
115:
116:         if isinstance(self.date_format, tuple):
117:             locale_string = self.date_format[0]
118:             locale.setlocale(locale.LC_ALL, locale_string)
119:             self.date_format = self.date_format[1]
120:
121:         # manage timezone
122:         default_timezone = settings.get('TIMEZONE', 'UTC')
```

```
123:         timezone = getattr(self, 'timezone', default_timezone)
124:         self.timezone = pytz.timezone(timezone)
125:
126:         if hasattr(self, 'date'):
127:             self.date = set_date_tzinfo(self.date, timezone)
128:             self.locale_date = self.date.strftime(self.date_format)
129:
130:         if hasattr(self, 'modified'):
131:             self.modified = set_date_tzinfo(self.modified, timezone)
132:             self.locale_modified = self.modified.strftime(self.date_format)
133:
134:         # manage status
135:         if not hasattr(self, 'status'):
136:             # Previous default of None broke comment plugins and perhaps others
137:             self.status = getattr(self, 'default_status', '')
138:
139:         # store the summary metadata if it is set
140:         if 'summary' in metadata:
141:             self._summary = metadata['summary']
142:
143:         signals.content_object_init.send(self)
144:
145:     def __str__(self):
146:         return self.source_path or repr(self)
147:
148:     def _has_valid_mandatory_properties(self):
149:         """Test mandatory properties are set."""
150:         for prop in self.mandatory_properties:
151:             if not hasattr(self, prop):
152:                 logger.error(
153:                     "Skipping %s: could not find information about '%s'",
154:                     self, prop)
155:                 return False
156:         return True
157:
158:     def _has_valid_save_as(self):
159:         """Return true if save_as doesn't write outside output path, false
160:         otherwise."""
161:         try:
162:             output_path = self.settings["OUTPUT_PATH"]
163:         except KeyError:
164:             # we cannot check
165:             return True
166:
167:         try:
168:             sanitised_join(output_path, self.save_as)
169:         except RuntimeError:  # outside output_dir
170:             logger.error(
171:                 "Skipping %s: file %r would be written outside output path",
172:                 self,
173:                 self.save_as,
174:             )
175:             return False
176:
177:         return True
178:
179:     def _has_valid_status(self):
180:         if hasattr(self, 'allowed_statuses'):
181:             if self.status not in self.allowed_statuses:
182:                 logger.error(
183:                     "Unknown status '%s' for file %s, skipping it.",
```

```
184:                     self.status,
185:                     self
186:                 )
187:                 return False
188:
189:         # if undefined we allow all
190:         return True
191:
192:     def is_valid(self):
193:         """Validate Content"""
194:         # Use all() to not short circuit and get results of all validations
195:         return all([self._has_valid_mandatory_properties(),
196:                     self._has_valid_save_as(),
197:                     self._has_valid_status()])
198:
199:     @property
200:     def url_format(self):
201:         """Returns the URL, formatted with the proper values"""
202:         metadata = copy.copy(self.metadata)
203:         path = self.metadata.get('path', self.get_relative_source_path())
204:         metadata.update({
205:             'path': path_to_url(path),
206:             'slug': getattr(self, 'slug', ''),
207:             'lang': getattr(self, 'lang', 'en'),
208:             'date': getattr(self, 'date', datetime.datetime.now()),
209:             'author': self.author.slug if hasattr(self, 'author') else '',
210:             'category': self.category.slug if hasattr(self, 'category') else ''
211:         })
212:         return metadata
213:
214:     def _expand_settings(self, key, klass=None):
215:         if not klass:
216:             klass = self.__class__.__name__
217:         fq_key = ('%s_%s' % (klass, key)).upper()
218:         return self.settings[fq_key].format(**self.url_format)
219:
220:     def get_url_setting(self, key):
221:         if hasattr(self, 'override_' + key):
222:             return getattr(self, 'override_' + key)
223:         key = key if self.in_default_lang else 'lang_%s' % key
224:         return self._expand_settings(key)
225:
226:     def _link_replacer(self, siteurl, m):
227:         what = m.group('what')
228:         value = urlparse(m.group('value'))
229:         path = value.path
230:         origin = m.group('path')
231:
232:         # urllib.parse.urljoin() produces `a.html` for urljoin("..", "a.html")
233:         # so if RELATIVE_URLS are enabled, we fall back to os.path.join() to
234:         # properly get `../a.html`. However, os.path.join() produces
235:         # `baz/http://foo/bar.html` for join("baz", "http://foo/bar.html")
236:         # instead of correct "http://foo/bar.html", so one has to pick a side
237:         # as there is no silver bullet.
238:         if self.settings['RELATIVE_URLS']:
239:             joiner = os.path.join
240:         else:
241:             joiner = urljoin
242:
243:             # However, it's not *that* simple: urljoin("blog", "index.html")
244:             # produces just `index.html` instead of `blog/index.html` (unlike
```

```
245:             # os.path.join()), so in order to get a correct answer one needs to
246:             # append a trailing slash to siteurl in that case. This also makes
247:             # the new behavior fully compatible with Pelican 3.7.1.
248:             if not siteurl.endswith('/'):
249:                 siteurl += '/'
250:
251:         # XXX Put this in a different location.
252:         if what in {'filename', 'static', 'attach'}:
253:             if path.startswith('/'):
254:                 path = path[1:]
255:             else:
256:                 # relative to the source path of this content
257:                 path = self.get_relative_source_path(
258:                     os.path.join(self.relative_dir, path)
259:                 )
260:
261:             key = 'static_content' if what in ('static', 'attach')\
262:                 else 'generated_content'
263:
264:             def _get_linked_content(key, path):
265:                 try:
266:                     return self._context[key][path]
267:                 except KeyError:
268:                     try:
269:                         # Markdown escapes spaces, try unescaping
270:                         return self._context[key][path.replace('%20', ' ')]
271:                     except KeyError:
272:                         if what == 'filename' and key == 'generated_content':
273:                             key = 'static_content'
274:                             linked_content = _get_linked_content(key, path)
275:                             if linked_content:
276:                                 logger.warning(
277:                                     '{filename} used for linking to static'
278:                                     ' content %s in %s. Use {static} instead',
279:                                     path,
280:                                     self.get_relative_source_path())
281:                                 return linked_content
282:                         return None
283:
284:             linked_content = _get_linked_content(key, path)
285:             if linked_content:
286:                 if what == 'attach':
287:                     linked_content.attach_to(self)
288:                 origin = joiner(siteurl, linked_content.url)
289:                 origin = origin.replace('\\', '/')  # for Windows paths.
290:             else:
291:                 logger.warning(
292:                     "Unable to find '%s', skipping url replacement.",
293:                     value.geturl(), extra={
294:                         'limit_msg': ("Other resources were not found "
295:                                       "and their urls not replaced")})
296:         elif what == 'category':
297:             origin = joiner(siteurl, Category(path, self.settings).url)
298:         elif what == 'tag':
299:             origin = joiner(siteurl, Tag(path, self.settings).url)
300:         elif what == 'index':
301:             origin = joiner(siteurl, self.settings['INDEX_SAVE_AS'])
302:         elif what == 'author':
303:             origin = joiner(siteurl, Author(path, self.settings).url)
304:         else:
305:             logger.warning(
```

```
306:                 "Replacement Indicator '%s' not recognized, "
307:                 "skipping replacement",
308:                 what)
309:
310:         # keep all other parts, such as query, fragment, etc.
311:         parts = list(value)
312:         parts[2] = origin
313:         origin = urlunparse(parts)
314:
315:         return ''.join((m.group('markup'), m.group('quote'), origin,
316:                         m.group('quote')))
317:
318:     def _get_intrasite_link_regex(self):
319:         intrasite_link_regex = self.settings['INTRASITE_LINK_REGEX']
320:         regex = r"""
321:             (?P<markup><[^\>]+  # match tag with all url-value attributes
322:                 (?:href|src|poster|data|cite|formaction|action)\s*=\s*)
323:
324:             (?P<quote>["\'])      # require value to be quoted
325:             (?P<path>{0}(?P<value>.*?))  # the url value
326:             \2""".format(intrasite_link_regex)
327:         return re.compile(regex, re.X)
328:
329:     def _update_content(self, content, siteurl):
330:         """Update the content attribute.
331:
332:         Change all the relative paths of the content to relative paths
333:         suitable for the output content.
334:
335:         :param content: content resource that will be passed to the templates.
336:         :param siteurl: siteurl which is locally generated by the writer in
337:                         case of RELATIVE_URLS.
338:         """
339:         if not content:
340:             return content
341:
342:         hrefs = self._get_intrasite_link_regex()
343:         return hrefs.sub(lambda m: self._link_replacer(siteurl, m), content)
344:
345:     def get_static_links(self):
346:         static_links = set()
347:         hrefs = self._get_intrasite_link_regex()
348:         for m in hrefs.finditer(self._content):
349:             what = m.group('what')
350:             value = urlparse(m.group('value'))
351:             path = value.path
352:             if what not in {'static', 'attach'}:
353:                 continue
354:             if path.startswith('/'):
355:                 path = path[1:]
356:             else:
357:                 # relative to the source path of this content
358:                 path = self.get_relative_source_path(
359:                     os.path.join(self.relative_dir, path)
360:                 )
361:             path = path.replace('%20', ' ')
362:             static_links.add(path)
363:         return static_links
364:
365:     def get_siteurl(self):
366:         return self._context.get('localsiteurl', '')
```

```
367:
368:     @memoized
369:     def get_content(self, siteurl):
370:         if hasattr(self, '_get_content'):
371:             content = self._get_content()
372:         else:
373:             content = self._content
374:         return self._update_content(content, siteurl)
375:
376:     @property
377:     def content(self):
378:         return self.get_content(self.get_siteurl())
379:
380:     @memoized
381:     def get_summary(self, siteurl):
382:         """Returns the summary of an article.
383:
384:         This is based on the summary metadata if set, otherwise truncate the
385:         content.
386:         """
387:         if 'summary' in self.metadata:
388:             return self.metadata['summary']
389:
390:         if self.settings['SUMMARY_MAX_LENGTH'] is None:
391:             return self.content
392:
393:         return truncate_html_words(self.content,
394:                                    self.settings['SUMMARY_MAX_LENGTH'],
395:                                    self.settings['SUMMARY_END_MARKER'])
396:
397:     @property
398:     def summary(self):
399:         return self.get_summary(self.get_siteurl())
400:
401:     def _get_summary(self):
402:         """deprecated function to access summary"""
403:
404:         logger.warning('_get_summary() has been deprecated since 3.6.4. '
405:                        'Use the summary decorator instead')
406:         return self.summary
407:
408:     @summary.setter
409:     def summary(self, value):
410:         """Dummy function"""
411:         pass
412:
413:     @property
414:     def status(self):
415:         return self._status
416:
417:     @status.setter
418:     def status(self, value):
419:         # TODO maybe typecheck
420:         self._status = value.lower()
421:
422:     @property
423:     def url(self):
424:         return self.get_url_setting('url')
425:
426:     @property
427:     def save_as(self):
```

```
428:         return self.get_url_setting('save_as')
429:
430:     def _get_template(self):
431:         if hasattr(self, 'template') and self.template is not None:
432:             return self.template
433:         else:
434:             return self.default_template
435:
436:     def get_relative_source_path(self, source_path=None):
437:         """Return the relative path (from the content path) to the given
438:         source_path.
439:
440:         If no source path is specified, use the source path of this
441:         content object.
442:         """
443:         if not source_path:
444:             source_path = self.source_path
445:         if source_path is None:
446:             return None
447:
448:         return posixize_path(
449:             os.path.relpath(
450:                 os.path.abspath(os.path.join(
451:                     self.settings['PATH'],
452:                     source_path)),
453:                 os.path.abspath(self.settings['PATH'])
454:             ))
455:
456:     @property
457:     def relative_dir(self):
458:         return posixize_path(
459:             os.path.dirname(
460:                 os.path.relpath(
461:                     os.path.abspath(self.source_path),
462:                     os.path.abspath(self.settings['PATH']))))
463:
464:     def refresh_metadata_intersite_links(self):
465:         for key in self.settings['FORMATTED_FIELDS']:
466:             if key in self.metadata and key != 'summary':
467:                 value = self._update_content(
468:                     self.metadata[key],
469:                     self.get_siteurl()
470:                 )
471:                 self.metadata[key] = value
472:                 setattr(self, key.lower(), value)
473:
474:         # _summary is an internal variable that some plugins may be writing to,
475:         # so ensure changes to it are picked up
476:         if ('summary' in self.settings['FORMATTED_FIELDS'] and
477:                 'summary' in self.metadata):
478:             self._summary = self._update_content(
479:                 self._summary,
480:                 self.get_siteurl()
481:             )
482:             self.metadata['summary'] = self._summary
483:
484:
485: class Page(Content):
486:     mandatory_properties = ('title',)
487:     allowed_statuses = ('published', 'hidden', 'draft')
488:     default_status = 'published'
```

```
489:     default_template = 'page'
490:
491:     def _expand_settings(self, key):
492:         klass = 'draft_page' if self.status == 'draft' else None
493:         return super()._expand_settings(key, klass)
494:
495:
496: class Article(Content):
497:     mandatory_properties = ('title', 'date', 'category')
498:     allowed_statuses = ('published', 'draft')
499:     default_status = 'published'
500:     default_template = 'article'
501:
502:     def __init__(self, *args, **kwargs):
503:         super().__init__(*args, **kwargs)
504:
505:         # handle WITH_FUTURE_DATES (designate article to draft based on date)
506:         if not self.settings['WITH_FUTURE_DATES'] and hasattr(self, 'date'):
507:             if self.date.tzinfo is None:
508:                 now = datetime.datetime.now()
509:             else:
510:                 now = datetime.datetime.utcnow().replace(tzinfo=pytz.utc)
511:             if self.date > now:
512:                 self.status = 'draft'
513:
514:         # if we are a draft and there is no date provided, set max datetime
515:         if not hasattr(self, 'date') and self.status == 'draft':
516:             self.date = datetime.datetime.max.replace(tzinfo=self.timezone)
517:
518:     def _expand_settings(self, key):
519:         klass = 'draft' if self.status == 'draft' else 'article'
520:         return super()._expand_settings(key, klass)
521:
522:
523: class Static(Content):
524:     mandatory_properties = ('title',)
525:     default_status = 'published'
526:     default_template = None
527:
528:     def __init__(self, *args, **kwargs):
529:         super().__init__(*args, **kwargs)
530:         self._output_location_referenced = False
531:
532:     @deprecated_attribute(old='filepath', new='source_path', since=(3, 2, 0))
533:     def filepath():
534:         return None
535:
536:     @deprecated_attribute(old='src', new='source_path', since=(3, 2, 0))
537:     def src():
538:         return None
539:
540:     @deprecated_attribute(old='dst', new='save_as', since=(3, 2, 0))
541:     def dst():
542:         return None
543:
544:     @property
545:     def url(self):
546:         # Note when url has been referenced, so we can avoid overriding it.
547:         self._output_location_referenced = True
548:         return super().url
549:
```

```
550:     @property
551:     def save_as(self):
552:         # Note when save_as has been referenced, so we can avoid overriding it.
553:         self._output_location_referenced = True
554:         return super().save_as
555:
556:     def attach_to(self, content):
557:         """Override our output directory with that of the given content object.
558:         """
559:
560:         # Determine our file's new output path relative to the linking
561:         # document. If it currently lives beneath the linking
562:         # document's source directory, preserve that relationship on output.
563:         # Otherwise, make it a sibling.
564:
565:         linking_source_dir = os.path.dirname(content.source_path)
566:         tail_path = os.path.relpath(self.source_path, linking_source_dir)
567:         if tail_path.startswith(os.pardir + os.sep):
568:             tail_path = os.path.basename(tail_path)
569:         new_save_as = os.path.join(
570:             os.path.dirname(content.save_as), tail_path)
571:
572:         # We do not build our new url by joining tail_path with the linking
573:         # document's url, because we cannot know just by looking at the latter
574:         # whether it points to the document itself or to its parent directory.
575:         # (An url like 'some/content' might mean a directory named 'some'
576:         # with a file named 'content', or it might mean a directory named
577:         # 'some/content' with a file named 'index.html'.) Rather than trying
578:         # to figure it out by comparing the linking document's url and save_as
579:         # path, we simply build our new url from our new save_as path.
580:
581:         new_url = path_to_url(new_save_as)
582:
583:         def _log_reason(reason):
584:             logger.warning(
585:                 "The {attach} link in %s cannot relocate "
586:                 "%s because %s. Falling back to "
587:                 "{filename} link behavior instead.",
588:                 content.get_relative_source_path(),
589:                 self.get_relative_source_path(), reason,
590:                 extra={'limit_msg': "More {attach} warnings silenced."})
591:
592:         # We never override an override, because we don't want to interfere
593:         # with user-defined overrides that might be in EXTRA_PATH_METADATA.
594:         if hasattr(self, 'override_save_as') or hasattr(self, 'override_url'):
595:             if new_save_as != self.save_as or new_url != self.url:
596:                 _log_reason("its output location was already overridden")
597:             return
598:
599:         # We never change an output path that has already been referenced,
600:         # because we don't want to break links that depend on that path.
601:         if self._output_location_referenced:
602:             if new_save_as != self.save_as or new_url != self.url:
603:                 _log_reason("another link already referenced its location")
604:             return
605:
606:         self.override_save_as = new_save_as
607:         self.override_url = new_url
```

```
# -*- coding: utf-8 -*-

import logging
import os
from urllib.parse import urljoin

from feedgenerator import Atom1Feed, Rss201rev2Feed, get_tag_uri

from jinja2 import Markup

from pelican.paginator import Paginator
from pelican.plugins import signals
from pelican.utils import (get_relative_path, is_selected_for_writing,
                           path_to_url, sanitised_join, set_date_tzinfo)

logger = logging.getLogger(__name__)


class Writer(object):

    def __init__(self, output_path, settings=None):
        self.output_path = output_path
        self.reminder = dict()
        self.settings = settings or {}
        self._written_files = set()
        self._overridden_files = set()

        # See Content._link_replacer for details
        if self.settings['RELATIVE_URLS']:
            self.urljoiner = os.path.join
        else:
            self.urljoiner = lambda base, url: urljoin(
                base if base.endswith('/') else base + '/', url)

    def _create_new_feed(self, feed_type, feed_title, context):
        feed_class = Rss201rev2Feed if feed_type == 'rss' else Atom1Feed
        if feed_title:
            feed_title = context['SITENAME'] + ' - ' + feed_title
        else:
            feed_title = context['SITENAME']
        feed = feed_class(
            title=Markup(feed_title).striptags(),
            link=(self.site_url + '/'),
            feed_url=self.feed_url,
            description=context.get('SITESUBTITLE', ''),
            subtitle=context.get('SITESUBTITLE', None))
        return feed

    def _add_item_to_the_feed(self, feed, item):
        title = Markup(item.title).striptags()
        link = self.urljoiner(self.site_url, item.url)

        if isinstance(feed, Rss201rev2Feed):
            # RSS feeds use a single tag called 'description' for both the full
            # content and the summary
            content = None
            if self.settings.get('RSS_FEED_SUMMARY_ONLY'):
                description = item.summary
            else:
                description = item.get_content(self.site_url)

```

```
62:         else:
63:             # Atom feeds have two different tags for full content (called
64:             # 'content' by feedgenerator) and summary (called 'description' by
65:             # feedgenerator).
66:             #
67:             # It does not make sense to have the summary be the
68:             # exact same thing as the full content. If we detect that
69:             # they are we just remove the summary.
70:             content = item.get_content(self.site_url)
71:             description = item.summary
72:             if description == content:
73:                 description = None
74:
75:         categories = list()
76:         if hasattr(item, 'category'):
77:             categories.append(item.category)
78:         if hasattr(item, 'tags'):
79:             categories.extend(item.tags)
80:
81:         feed.add_item(
82:             title=title,
83:             link=link,
84:             unique_id=get_tag_uri(link, item.date),
85:             description=description,
86:             content=content,
87:             categories=categories if categories else None,
88:             author_name=getattr(item, 'author', ''),
89:             pubdate=set_date_tzinfo(
90:                 item.date, self.settings.get('TIMEZONE', None)),
91:             updateddate=set_date_tzinfo(
92:                 item.modified, self.settings.get('TIMEZONE', None)
93:                 ) if hasattr(item, 'modified') else None)
94:
95:     def _open_w(self, filename, encoding, override=False):
96:         """Open a file to write some content to it.
97:
98:         Exit if we have already written to that file, unless one (and no more
99:         than one) of the writes has the override parameter set to True.
100:         """
101:         if filename in self._overridden_files:
102:             if override:
103:                 raise RuntimeError('File %s is set to be overridden twice'
104:                                    % filename)
105:             else:
106:                 logger.info('Skipping %s', filename)
107:                 filename = os.devnull
108:         elif filename in self._written_files:
109:             if override:
110:                 logger.info('Overwriting %s', filename)
111:             else:
112:                 raise RuntimeError('File %s is to be overwritten' % filename)
113:         if override:
114:             self._overridden_files.add(filename)
115:         self._written_files.add(filename)
116:         return open(filename, 'w', encoding=encoding)
117:
118:     def write_feed(self, elements, context, path=None, url=None,
119:                    feed_type='atom', override_output=False, feed_title=None):
120:         """Generate a feed with the list of articles provided
121:
122:         Return the feed. If no path or output_path is specified, just
```

```
123:         return the feed object.
124:
125:         :param elements: the articles to put on the feed.
126:         :param context: the context to get the feed metadata.
127:         :param path: the path to output.
128:         :param url: the publicly visible feed URL; if None, path is used
129:             instead
130:         :param feed_type: the feed type to use (atom or rss)
131:         :param override_output: boolean telling if we can override previous
132:             output with the same name (and if next files written with the same
133:             name should be skipped to keep that one)
134:         :param feed_title: the title of the feed.o
135:         """
136:         if not is_selected_for_writing(self.settings, path):
137:             return
138:
139:         self.site_url = context.get(
140:             'SITEURL', path_to_url(get_relative_path(path)))
141:
142:         self.feed_domain = context.get('FEED_DOMAIN')
143:         self.feed_url = self.urljoiner(self.feed_domain, url if url else path)
144:
145:         feed = self._create_new_feed(feed_type, feed_title, context)
146:
147:         max_items = len(elements)
148:         if self.settings['FEED_MAX_ITEMS']:
149:             max_items = min(self.settings['FEED_MAX_ITEMS'], max_items)
150:         for i in range(max_items):
151:             self._add_item_to_the_feed(feed, elements[i])
152:
153:         signals.feed_generated.send(context, feed=feed)
154:         if path:
155:             complete_path = sanitised_join(self.output_path, path)
156:
157:             try:
158:                 os.makedirs(os.path.dirname(complete_path))
159:             except Exception:
160:                 pass
161:
162:             with self._open_w(complete_path, 'utf-8', override_output) as fp:
163:                 feed.write(fp, 'utf-8')
164:                 logger.info('Writing %s', complete_path)
165:
166:             signals.feed_written.send(
167:                 complete_path, context=context, feed=feed)
168:         return feed
169:
170:     def write_file(self, name, template, context, relative_urls=False,
171:                    paginated=None, template_name=None, override_output=False,
172:                    url=None, **kwargs):
173:         """Render the template and write the file.
174:
175:         :param name: name of the file to output
176:         :param template: template to use to generate the content
177:         :param context: dict to pass to the templates.
178:         :param relative_urls: use relative urls or absolutes ones
179:         :param paginated: dict of article list to paginate - must have the
180:             same length (same list in different orders)
181:         :param template_name: the template name, for pagination
182:         :param override_output: boolean telling if we can override previous
183:             output with the same name (and if next files written with the same
```

```
184:             name should be skipped to keep that one)
185:         :param url: url of the file (needed by the paginator)
186:         :param **kwargs: additional variables to pass to the templates
187:         """
188:
189:         if name is False or \
190:            name == "" or \
191:            not is_selected_for_writing(self.settings,
192:                                        os.path.join(self.output_path, name)):
193:             return
194:         elif not name:
195:             # other stuff, just return for now
196:             return
197:
198:         def _write_file(template, localcontext, output_path, name, override):
199:             """Render the template write the file."""
200:             # set localsiteurl for context so that Contents can adjust links
201:             if localcontext['localsiteurl']:
202:                 context['localsiteurl'] = localcontext['localsiteurl']
203:             output = template.render(localcontext)
204:             path = sanitised_join(output_path, name)
205:
206:             try:
207:                 os.makedirs(os.path.dirname(path))
208:             except Exception:
209:                 pass
210:
211:             with self._open_w(path, 'utf-8', override=override) as f:
212:                 f.write(output)
213:             logger.info('Writing %s', path)
214:
215:             # Send a signal to say we're writing a file with some specific
216:             # local context.
217:             signals.content_written.send(path, context=localcontext)
218:
219:         def _get_localcontext(context, name, kwargs, relative_urls):
220:             localcontext = context.copy()
221:             localcontext['localsiteurl'] = localcontext.get(
222:                 'localsiteurl', None)
223:             if relative_urls:
224:                 relative_url = path_to_url(get_relative_path(name))
225:                 localcontext['SITEURL'] = relative_url
226:                 localcontext['localsiteurl'] = relative_url
227:             localcontext['output_file'] = name
228:             localcontext.update(kwargs)
229:             return localcontext
230:
231:         if paginated is None:
232:             paginated = {key: val for key, val in kwargs.items()
233:                          if key in {'articles', 'dates'}}
234:
235:         # pagination
236:         if paginated and template_name in self.settings['PAGINATED_TEMPLATES']:
237:             # pagination needed
238:             per_page = self.settings['PAGINATED_TEMPLATES'][template_name] \
239:                 or self.settings['DEFAULT_PAGINATION']
240:
241:             # init paginators
242:             paginators = {key: Paginator(name, url, val, self.settings,
243:                                          per_page)
244:                           for key, val in paginated.items()}
```

```
245:
246:             # generated pages, and write
247:             for page_num in range(list(paginators.values())[0].num_pages):
248:                 paginated_kwargs = kwargs.copy()
249:                 for key in paginators.keys():
250:                     paginator = paginators[key]
251:                     previous_page = paginator.page(page_num) \
252:                         if page_num > 0 else None
253:                     page = paginator.page(page_num + 1)
254:                     next_page = paginator.page(page_num + 2) \
255:                         if page_num + 1 < paginator.num_pages else None
256:                     paginated_kwargs.update(
257:                         {'%s_paginator' % key: paginator,
258:                          '%s_page' % key: page,
259:                          '%s_previous_page' % key: previous_page,
260:                          '%s_next_page' % key: next_page})
261:
262:                 localcontext = _get_localcontext(
263:                     context, page.save_as, paginated_kwargs, relative_urls)
264:                 _write_file(template, localcontext, self.output_path,
265:                             page.save_as, override_output)
266:         else:
267:             # no pagination
268:             localcontext = _get_localcontext(
269:                 context, name, kwargs, relative_urls)
270:             _write_file(template, localcontext, self.output_path, name,
271:                         override_output)
```

```
# -*- coding: utf-8 -*-

import logging
import os
import sys
from collections import defaultdict

__all__ = [
    'init'
]


class BaseFormatter(logging.Formatter):
    def __init__(self, fmt=None, datefmt=None):
        FORMAT = '%(customlevelname)s %(message)s'
        super().__init__(fmt=FORMAT, datefmt=datefmt)

    def format(self, record):
        customlevel = self._get_levelname(record.levelname)
        record.__dict__['customlevelname'] = customlevel
        # format multiline messages 'nicely' to make it clear they are together
        record.msg = record.msg.replace('\n', '\n  | ')
        record.args = tuple(arg.replace('\n', '\n  | ') if
                            isinstance(arg, str) else
                            arg for arg in record.args)
        return super().format(record)

    def formatException(self, ei):
        ''' prefix traceback info for better representation '''
        s = super().formatException(ei)
        # fancy format traceback
        s = '\n'.join('  | ' + line for line in s.splitlines())
        # separate the traceback from the preceding lines
        s = '  |___\n{}'.format(s)
        return s

    def _get_levelname(self, name):
        ''' NOOP: overridden by subclasses '''
        return name


class ANSIFormatter(BaseFormatter):
    ANSI_CODES = {
        'red': '\033[1;31m',
        'yellow': '\033[1;33m',
        'cyan': '\033[1;36m',
        'white': '\033[1;37m',
        'bgred': '\033[1;41m',
        'bggrey': '\033[1;100m',
        'reset': '\033[0;m'}

    LEVEL_COLORS = {
        'INFO': 'cyan',
        'WARNING': 'yellow',
        'ERROR': 'red',
        'CRITICAL': 'bgred',
        'DEBUG': 'bggrey'}

    def _get_levelname(self, name):
        color = self.ANSI_CODES[self.LEVEL_COLORS.get(name, 'white')]
        if name == 'INFO':
```

```
62:             fmt = '{0}->{2}'
63:         else:
64:             fmt = '{0}{1}{2}:'
65:         return fmt.format(color, name, self.ANSI_CODES['reset'])
66:
67:
68: class TextFormatter(BaseFormatter):
69:     """
70:     Convert a `logging.LogRecord' object into text.
71:     """
72:
73:     def _get_levelname(self, name):
74:         if name == 'INFO':
75:             return '->'
76:         else:
77:             return name + ':'
78:
79:
80: class LimitFilter(logging.Filter):
81:     """
82:     Remove duplicates records, and limit the number of records in the same
83:     group.
84:
85:     Groups are specified by the message to use when the number of records in
86:     the same group hit the limit.
87:     E.g.: log.warning(('43 is not the answer', 'More erroneous answers'))
88:     """
89:
90:     LOGS_DEDUP_MIN_LEVEL = logging.WARNING
91:
92:     _ignore = set()
93:     _raised_messages = set()
94:     _threshold = 5
95:     _group_count = defaultdict(int)
96:
97:     def filter(self, record):
98:         # don't limit log messages for anything above "warning"
99:         if record.levelno > self.LOGS_DEDUP_MIN_LEVEL:
100:             return True
101:
102:         # extract group
103:         group = record.__dict__.get('limit_msg', None)
104:         group_args = record.__dict__.get('limit_args', ())
105:
106:         # ignore record if it was already raised
107:         message_key = (record.levelno, record.getMessage())
108:         if message_key in self._raised_messages:
109:             return False
110:         else:
111:             self._raised_messages.add(message_key)
112:
113:         # ignore LOG_FILTER records by templates or messages
114:         # when "debug" isn't enabled
115:         logger_level = logging.getLogger().getEffectiveLevel()
116:         if logger_level > logging.DEBUG:
117:             template_key = (record.levelno, record.msg)
118:             message_key = (record.levelno, record.getMessage())
119:             if (template_key in self._ignore or message_key in self._ignore):
120:                 return False
121:
122:         # check if we went over threshold
```

```
123:         if group:
124:             key = (record.levelno, group)
125:             self._group_count[key] += 1
126:             if self._group_count[key] == self._threshold:
127:                 record.msg = group
128:                 record.args = group_args
129:             elif self._group_count[key] > self._threshold:
130:                 return False
131:         return True
132:
133:
134: class LimitLogger(logging.Logger):
135:     """
136:     A logger which adds LimitFilter automatically
137:     """
138:
139:     limit_filter = LimitFilter()
140:
141:     def __init__(self, *args, **kwargs):
142:         super().__init__(*args, **kwargs)
143:         self.enable_filter()
144:
145:     def disable_filter(self):
146:         self.removeFilter(LimitLogger.limit_filter)
147:
148:     def enable_filter(self):
149:         self.addFilter(LimitLogger.limit_filter)
150:
151:
152: class FatalLogger(LimitLogger):
153:     warnings_fatal = False
154:     errors_fatal = False
155:
156:     def warning(self, *args, **kwargs):
157:         super().warning(*args, **kwargs)
158:         if FatalLogger.warnings_fatal:
159:             raise RuntimeError('Warning encountered')
160:
161:     def error(self, *args, **kwargs):
162:         super().error(*args, **kwargs)
163:         if FatalLogger.errors_fatal:
164:             raise RuntimeError('Error encountered')
165:
166:
167: logging.setLoggerClass(FatalLogger)
168:
169:
170: def supports_color():
171:     """
172:     Returns True if the running system's terminal supports color,
173:     and False otherwise.
174:
175:     from django.core.management.color
176:     """
177:     plat = sys.platform
178:     supported_platform = plat != 'Pocket PC' and \
179:         (plat != 'win32' or 'ANSICON' in os.environ)
180:
181:     # isatty is not always implemented, #6223.
182:     is_a_tty = hasattr(sys.stdout, 'isatty') and sys.stdout.isatty()
183:     if not supported_platform or not is_a_tty:
```

```
184:         return False
185:     return True
186:
187:
188: def get_formatter():
189:     if supports_color():
190:         return ANSIFormatter()
191:     else:
192:         return TextFormatter()
193:
194:
195: def init(level=None, fatal='', handler=logging.StreamHandler(), name=None,
196:          logs_dedup_min_level=None):
197:     FatalLogger.warnings_fatal = fatal.startswith('warning')
198:     FatalLogger.errors_fatal = bool(fatal)
199:
200:     logger = logging.getLogger(name)
201:
202:     handler.setFormatter(get_formatter())
203:     logger.addHandler(handler)
204:
205:     if level:
206:         logger.setLevel(level)
207:     if logs_dedup_min_level:
208:         LimitFilter.LOGS_DEDUP_MIN_LEVEL = logs_dedup_min_level
209:
210:
211: def log_warnings():
212:     import warnings
213:     logging.captureWarnings(True)
214:     warnings.simplefilter("default", DeprecationWarning)
215:     init(logging.DEBUG, name='py.warnings')
216:
217:
218: if __name__ == '__main__':
219:     init(level=logging.DEBUG)
220:
221:     root_logger = logging.getLogger()
222:     root_logger.debug('debug')
223:     root_logger.info('info')
224:     root_logger.warning('warning')
225:     root_logger.error('error')
226:     root_logger.critical('critical')
```

```
# -*- coding: utf-8 -*-

import functools
import logging
import os
from collections import namedtuple
from math import ceil

logger = logging.getLogger(__name__)
PaginationRule = namedtuple(
    'PaginationRule',
    'min_page URL SAVE_AS',
)


class Paginator(object):
    def __init__(self, name, url, object_list, settings, per_page=None):
        self.name = name
        self.url = url
        self.object_list = object_list
        self.settings = settings
        if per_page:
            self.per_page = per_page
            self.orphans = settings['DEFAULT_ORPHANS']
        else:
            self.per_page = len(object_list)
            self.orphans = 0

        self._num_pages = self._count = None

    def page(self, number):
        "Returns a Page object for the given 1-based page number."
        bottom = (number - 1) * self.per_page
        top = bottom + self.per_page
        if top + self.orphans >= self.count:
            top = self.count
        return Page(self.name, self.url, self.object_list[bottom:top], number,
                    self, self.settings)

    def _get_count(self):
        "Returns the total number of objects, across all pages."
        if self._count is None:
            self._count = len(self.object_list)
        return self._count
    count = property(_get_count)

    def _get_num_pages(self):
        "Returns the total number of pages."
        if self._num_pages is None:
            hits = max(1, self.count - self.orphans)
            self._num_pages = int(ceil(hits / (float(self.per_page) or 1)))
        return self._num_pages
    num_pages = property(_get_num_pages)

    def _get_page_range(self):
        """
        Returns a 1-based range of pages for iterating through within
        a template for loop.
        """
        return list(range(1, self.num_pages + 1))
    page_range = property(_get_page_range)
```

```
 62:
 63:
 64: class Page(object):
 65:     def __init__(self, name, url, object_list, number, paginator, settings):
 66:         self.full_name = name
 67:         self.name, self.extension = os.path.splitext(name)
 68:         dn, fn = os.path.split(name)
 69:         self.base_name = dn if fn in ('index.htm', 'index.html') else self.name
 70:         self.base_url = url
 71:         self.object_list = object_list
 72:         self.number = number
 73:         self.paginator = paginator
 74:         self.settings = settings
 75:
 76:     def __repr__(self):
 77:         return '<Page %s of %s>' % (self.number, self.paginator.num_pages)
 78:
 79:     def has_next(self):
 80:         return self.number < self.paginator.num_pages
 81:
 82:     def has_previous(self):
 83:         return self.number > 1
 84:
 85:     def has_other_pages(self):
 86:         return self.has_previous() or self.has_next()
 87:
 88:     def next_page_number(self):
 89:         return self.number + 1
 90:
 91:     def previous_page_number(self):
 92:         return self.number - 1
 93:
 94:     def start_index(self):
 95:         """
 96:         Returns the 1-based index of the first object on this page,
 97:         relative to total objects in the paginator.
 98:         """
 99:         # Special case, return zero if no items.
100:         if self.paginator.count == 0:
101:             return 0
102:         return (self.paginator.per_page * (self.number - 1)) + 1
103:
104:     def end_index(self):
105:         """
106:         Returns the 1-based index of the last object on this page,
107:         relative to total objects found (hits).
108:         """
109:         # Special case for the last page because there can be orphans.
110:         if self.number == self.paginator.num_pages:
111:             return self.paginator.count
112:         return self.number * self.paginator.per_page
113:
114:     def _from_settings(self, key):
115:         """Returns URL information as defined in settings. Similar to
116:         URLWrapper._from_settings, but specialized to deal with pagination
117:         logic."""
118:
119:         rule = None
120:
121:         # find the last matching pagination rule
122:         for p in self.settings['PAGINATION_PATTERNS']:
```

```
123:             if p.min_page <= self.number:
124:                 rule = p
125:
126:         if not rule:
127:             return ''
128:
129:         prop_value = getattr(rule, key)
130:
131:         if not isinstance(prop_value, str):
132:             logger.warning('%s is set to %s', key, prop_value)
133:             return prop_value
134:
135:         # URL or SAVE_AS is a string, format it with a controlled context
136:         context = {
137:             'save_as': self.full_name,
138:             'url': self.base_url,
139:             'name': self.name,
140:             'base_name': self.base_name,
141:             'extension': self.extension,
142:             'number': self.number,
143:         }
144:
145:         ret = prop_value.format(**context)
146:         # Remove a single leading slash, if any. This is done for backwards
147:         # compatibility reasons. If a leading slash is needed (for URLs
148:         # relative to server root or absolute URLs without the scheme such as
149:         # //blog.my.site/), it can be worked around by prefixing the pagination
150:         # pattern by an additional slash (which then gets removed, preserving
151:         # the other slashes). This also means the following code *can't* be
152:         # changed to lstrip() because that would remove all leading slashes and
153:         # thus make the workaround impossible. See
154:         # test_custom_pagination_pattern() for a verification of this.
155:         if ret[0] == '/':
156:             ret = ret[1:]
157:         return ret
158:
159:     url = property(functools.partial(_from_settings, key='URL'))
160:     save_as = property(functools.partial(_from_settings, key='SAVE_AS'))
```

```
# -*- coding: utf-8 -*-

import argparse
import logging
import os
import posixpath
import ssl
import sys
import urllib
from http import server

try:
    from magic import from_file as magic_from_file
except ImportError:
    magic_from_file = None

from pelican.log import init as init_logging
logger = logging.getLogger(__name__)


def parse_arguments():
    parser = argparse.ArgumentParser(
        description='Pelican Development Server',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter
    )
    parser.add_argument("port", default=8000, type=int, nargs="?",
                        help="Port to Listen On")
    parser.add_argument("server", default="", nargs="?",
                        help="Interface to Listen On")
    parser.add_argument('--ssl', action="store_true",
                        help='Activate SSL listener')
    parser.add_argument('--cert', default="./cert.pem", nargs="?",
                        help='Path to certificate file. ' +
                        'Relative to current directory')
    parser.add_argument('--key', default="./key.pem", nargs="?",
                        help='Path to certificate key file. ' +
                        'Relative to current directory')
    parser.add_argument('--path', default=".",
                        help='Path to pelican source directory to serve. ' +
                        'Relative to current directory')
    return parser.parse_args()


class ComplexHTTPRequestHandler(server.SimpleHTTPRequestHandler):
    SUFFIXES = ['.html', '/index.html', '/', '']

    def translate_path(self, path):
        # abandon query parameters
        path = path.split('?', 1)[0]
        path = path.split('#', 1)[0]
        # Don't forget explicit trailing slash when normalizing. Issue17324
        trailing_slash = path.rstrip().endswith('/')
        path = urllib.parse.unquote(path)
        path = posixpath.normpath(path)
        words = path.split('/')
        words = filter(None, words)
        path = self.base_path
        for word in words:
            if os.path.dirname(word) or word in (os.curdir, os.pardir):
                # Ignore components that are not a simple file/directory name
                continue
```

```
62:             path = os.path.join(path, word)
63:         if trailing_slash:
64:             path += '/'
65:         return path
66:
67:     def do_GET(self):
68:         # cut off a query string
69:         original_path = self.path.split('?', 1)[0]
70:         # try to find file
71:         self.path = self.get_path_that_exists(original_path)
72:
73:         if not self.path:
74:             return
75:
76:         server.SimpleHTTPRequestHandler.do_GET(self)
77:
78:     def get_path_that_exists(self, original_path):
79:         # Try to strip trailing slash
80:         original_path = original_path.rstrip('/')
81:         # Try to detect file by applying various suffixes
82:         tries = []
83:         for suffix in self.SUFFIXES:
84:             path = original_path + suffix
85:             if os.path.exists(self.translate_path(path)):
86:                 return path
87:             tries.append(path)
88:         logger.warning("Unable to find `%s` or variations:\n%s",
89:                        original_path,
90:                        '\n'.join(tries))
91:         return None
92:
93:     def guess_type(self, path):
94:         """Guess at the mime type for the specified file.
95:         """
96:         mimetype = server.SimpleHTTPRequestHandler.guess_type(self, path)
97:
98:         # If the default guess is too generic, try the python-magic library
99:         if mimetype == 'application/octet-stream' and magic_from_file:
100:             mimetype = magic_from_file(path, mime=True)
101:
102:         return mimetype
103:
104:
105: class RootedHTTPServer(server.HTTPServer):
106:     def __init__(self, base_path, *args, **kwargs):
107:         server.HTTPServer.__init__(self, *args, **kwargs)
108:         self.RequestHandlerClass.base_path = base_path
109:
110:
111: if __name__ == '__main__':
112:     init_logging(level=logging.INFO)
113:     logger.warning("'python -m pelican.server' is deprecated.\nThe "
114:                    "Pelican development server should be run via "
115:                    "'pelican --listen' or 'pelican -l'.\nThis can be combined "
116:                    "with regeneration as 'pelican -lr'.\nRerun 'pelican-"
117:                    "quickstart' to get new Makefile and tasks.py files.")
118:     args = parse_arguments()
119:     RootedHTTPServer.allow_reuse_address = True
120:     try:
121:         httpd = RootedHTTPServer(
122:             args.path, (args.server, args.port), ComplexHTTPRequestHandler)
```

```
123:         if args.ssl:
124:             httpd.socket = ssl.wrap_socket(
125:                 httpd.socket, keyfile=args.key,
126:                 certfile=args.cert, server_side=True)
127:     except ssl.SSLError as e:
128:         logger.error("Couldn't open certificate file %s or key file %s",
129:                      args.cert, args.key)
130:         logger.error("Could not listen on port %s, server %s.",
131:                      args.port, args.server)
132:         sys.exit(getattr(e, 'exitcode', 1))
133: 
134:     logger.info("Serving at port %s, server %s.",
135:                 args.port, args.server)
136:     try:
137:         httpd.serve_forever()
138:     except KeyboardInterrupt:
139:         logger.info("Shutting down server.")
140:         httpd.socket.close()
```

```
# -*- coding: utf-8 -*-

import hashlib
import logging
import os
import pickle

from pelican.utils import mkdir_p

logger = logging.getLogger(__name__)


class FileDataCacher(object):
    """Class that can cache data contained in files"""

    def __init__(self, settings, cache_name, caching_policy, load_policy):
        """Load the specified cache within CACHE_PATH in settings

        only if *load_policy* is True,
        May use gzip if GZIP_CACHE ins settings is True.
        Sets caching policy according to *caching_policy*.
        """
        self.settings = settings
        self._cache_path = os.path.join(self.settings['CACHE_PATH'],
                                        cache_name)
        self._cache_data_policy = caching_policy
        if self.settings['GZIP_CACHE']:
            import gzip
            self._cache_open = gzip.open
        else:
            self._cache_open = open
        if load_policy:
            try:
                with self._cache_open(self._cache_path, 'rb') as fhandle:
                    self._cache = pickle.load(fhandle)
            except (IOError, OSError) as err:
                logger.debug('Cannot load cache %s (this is normal on first '
                             'run). Proceeding with empty cache.\n%s',
                             self._cache_path, err)
                self._cache = {}
            except pickle.PickleError as err:
                logger.warning('Cannot unpickle cache %s, cache may be using '
                               'an incompatible protocol (see pelican '
                               'caching docs). '
                               'Proceeding with empty cache.\n%s',
                               self._cache_path, err)
                self._cache = {}
        else:
            self._cache = {}

    def cache_data(self, filename, data):
        """Cache data for given file"""
        if self._cache_data_policy:
            self._cache[filename] = data

    def get_cached_data(self, filename, default=None):
        """Get cached data for the given file

        if no data is cached, return the default object
        """
        return self._cache.get(filename, default)
```

```
 62:
 63:     def save_cache(self):
 64:         """Save the updated cache"""
 65:         if self._cache_data_policy:
 66:             try:
 67:                 mkdir_p(self.settings['CACHE_PATH'])
 68:                 with self._cache_open(self._cache_path, 'wb') as fhandle:
 69:                     pickle.dump(self._cache, fhandle)
 70:             except (IOError, OSError, pickle.PicklingError) as err:
 71:                 logger.warning('Could not save cache %s\n ... %s',
 72:                                self._cache_path, err)
 73:
 74:
 75: class FileStampDataCacher(FileDataCacher):
 76:     """Subclass that also caches the stamp of the file"""
 77:
 78:     def __init__(self, settings, cache_name, caching_policy, load_policy):
 79:         """This sublcass additionally sets filestamp function
 80:         and base path for filestamping operations
 81:         """
 82:
 83:         super().__init__(settings, cache_name, caching_policy, load_policy)
 84:
 85:         method = self.settings['CHECK_MODIFIED_METHOD']
 86:         if method == 'mtime':
 87:             self._filestamp_func = os.path.getmtime
 88:         else:
 89:             try:
 90:                 hash_func = getattr(hashlib, method)
 91:
 92:                 def filestamp_func(filename):
 93:                     """return hash of file contents"""
 94:                     with open(filename, 'rb') as fhandle:
 95:                         return hash_func(fhandle.read()).digest()
 96:
 97:                 self._filestamp_func = filestamp_func
 98:             except AttributeError as err:
 99:                 logger.warning('Could not get hashing function\n\t%s', err)
100:                 self._filestamp_func = None
101:
102:     def cache_data(self, filename, data):
103:         """Cache stamp and data for the given file"""
104:         stamp = self._get_file_stamp(filename)
105:         super().cache_data(filename, (stamp, data))
106:
107:     def _get_file_stamp(self, filename):
108:         """Check if the given file has been modified
109:         since the previous build.
110:
111:         depending on CHECK_MODIFIED_METHOD
112:         a float may be returned for 'mtime',
113:         a hash for a function name in the hashlib module
114:         or an empty bytes string otherwise
115:         """
116:
117:         try:
118:             return self._filestamp_func(filename)
119:         except (IOError, OSError, TypeError) as err:
120:             logger.warning('Cannot get modification stamp for %s\n\t%s',
121:                            filename, err)
122:             return ''
```

```
123:
124:     def get_cached_data(self, filename, default=None):
125:         """Get the cached data for the given filename
126:         if the file has not been modified.
127:
128:         If no record exists or file has been modified, return default.
129:         Modification is checked by comparing the cached
130:         and current file stamp.
131:         """
132:
133:         stamp, data = super().get_cached_data(filename, (None, default))
134:         if stamp != self._get_file_stamp(filename):
135:             return default
136:         return data
```

```
# -*- coding: utf-8 -*-

import functools
import logging
import os

from pelican.utils import slugify

logger = logging.getLogger(__name__)


@functools.total_ordering
class URLWrapper(object):
    def __init__(self, name, settings):
        self.settings = settings
        self._name = name
        self._slug = None
        self._slug_from_name = True

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name
        # if slug wasn't explicitly set, it needs to be regenerated from name
        # so, changing name should reset slug for slugification
        if self._slug_from_name:
            self._slug = None

    @property
    def slug(self):
        if self._slug is None:
            class_key = '{}_REGEX_SUBSTITUTIONS'.format(
                self.__class__.__name__.upper())
            if class_key in self.settings:
                self._slug = slugify(
                    self.name,
                    regex_subs=self.settings[class_key])
            else:
                self._slug = slugify(
                    self.name,
                    regex_subs=self.settings.get(
                        'SLUG_REGEX_SUBSTITUTIONS', []))
        return self._slug

    @slug.setter
    def slug(self, slug):
        # if slug is expliticly set, changing name won't alter slug
        self._slug_from_name = False
        self._slug = slug

    def as_dict(self):
        d = self.__dict__
        d['name'] = self.name
        d['slug'] = self.slug
        return d

    def __hash__(self):
        return hash(self.slug)
```

```
62:
63:     def _normalize_key(self, key):
64:         subs = self.settings.get('SLUG_REGEX_SUBSTITUTIONS', [])
65:         return slugify(key, regex_subs=subs)
66:
67:     def __eq__(self, other):
68:         if isinstance(other, self.__class__):
69:             return self.slug == other.slug
70:         if isinstance(other, str):
71:             return self.slug == self._normalize_key(other)
72:         return False
73:
74:     def __ne__(self, other):
75:         if isinstance(other, self.__class__):
76:             return self.slug != other.slug
77:         if isinstance(other, str):
78:             return self.slug != self._normalize_key(other)
79:         return True
80:
81:     def __lt__(self, other):
82:         if isinstance(other, self.__class__):
83:             return self.slug < other.slug
84:         if isinstance(other, str):
85:             return self.slug < self._normalize_key(other)
86:         return False
87:
88:     def __str__(self):
89:         return self.name
90:
91:     def __repr__(self):
92:         return '<{} {}>'.format(type(self).__name__, repr(self._name))
93:
94:     def _from_settings(self, key, get_page_name=False):
95:         """Returns URL information as defined in settings.
96:
97:         When get_page_name=True returns URL without anything after {slug} e.g.
98:         if in settings: CATEGORY_URL="cat/{slug}.html" this returns
99:         "cat/{slug}" Useful for pagination.
100:
101:         """
102:         setting = "%s_%s" % (self.__class__.__name__.upper(), key)
103:         value = self.settings[setting]
104:         if not isinstance(value, str):
105:             logger.warning('%s is set to %s', setting, value)
106:             return value
107:         else:
108:             if get_page_name:
109:                 return os.path.splitext(value)[0].format(**self.as_dict())
110:             else:
111:                 return value.format(**self.as_dict())
112:
113:     page_name = property(functools.partial(_from_settings, key='URL',
114:                                            get_page_name=True))
115:     url = property(functools.partial(_from_settings, key='URL'))
116:     save_as = property(functools.partial(_from_settings, key='SAVE_AS'))
117:
118:
119: class Category(URLWrapper):
120:     pass
121:
122:
```

```
123: class Tag(URLWrapper):
124:     def __init__(self, name, *args, **kwargs):
125:         super().__init__(name.strip(), *args, **kwargs)
126:
127:
128: class Author(URLWrapper):
129:     pass
```

```
# -*- coding: utf-8 -*-

import re

from docutils import nodes, utils
from docutils.parsers.rst import Directive, directives, roles

from pygments import highlight
from pygments.formatters import HtmlFormatter
from pygments.lexers import TextLexer, get_lexer_by_name

import pelican.settings as pys


class Pygments(Directive):
    """ Source code syntax highlighting.
    """
    required_arguments = 1
    optional_arguments = 0
    final_argument_whitespace = True
    option_spec = {
        'anchorlinenos': directives.flag,
        'classprefix': directives.unchanged,
        'hl_lines': directives.unchanged,
        'lineanchors': directives.unchanged,
        'linenos': directives.unchanged,
        'linenospecial': directives.nonnegative_int,
        'linenostart': directives.nonnegative_int,
        'linenostep': directives.nonnegative_int,
        'lineseparator': directives.unchanged,
        'linespans': directives.unchanged,
        'nobackground': directives.flag,
        'nowrap': directives.flag,
        'tagsfile': directives.unchanged,
        'tagurlformat': directives.unchanged,
    }
    has_content = True

    def run(self):
        self.assert_has_content()
        try:
            lexer = get_lexer_by_name(self.arguments[0])
        except ValueError:
            # no lexer found - use the text one instead of an exception
            lexer = TextLexer()

        # Fetch the defaults
        if pys.PYGMENTS_RST_OPTIONS is not None:
            for k, v in pys.PYGMENTS_RST_OPTIONS.items():
                # Locally set options overrides the defaults
                if k not in self.options:
                    self.options[k] = v

        if ('linenos' in self.options and
                self.options['linenos'] not in ('table', 'inline')):
            if self.options['linenos'] == 'none':
                self.options.pop('linenos')
            else:
                self.options['linenos'] = 'table'

        for flag in ('nowrap', 'nobackground', 'anchorlinenos'):
```

```
62:             if flag in self.options:
63:                 self.options[flag] = True
64:
65:         # noclasses should already default to False, but just in case...
66:         formatter = HtmlFormatter(noclasses=False, **self.options)
67:         parsed = highlight('\n'.join(self.content), lexer, formatter)
68:         return [nodes.raw('', parsed, format='html')]
69:
70:
71: directives.register_directive('code-block', Pygments)
72: directives.register_directive('sourcecode', Pygments)
73:
74:
75: _abbr_re = re.compile(r'\((.*)\)$', re.DOTALL)
76:
77:
78: class abbreviation(nodes.Inline, nodes.TextElement):
79:     pass
80:
81:
82: def abbr_role(typ, rawtext, text, lineno, inliner, options={}, content=[]):
83:     text = utils.unescape(text)
84:     m = _abbr_re.search(text)
85:     if m is None:
86:         return [abbreviation(text, text)], []
87:     abbr = text[:m.start()].strip()
88:     expl = m.group(1)
89:     return [abbreviation(abbr, abbr, explanation=expl)], []
90:
91:
92: roles.register_local_role('abbr', abbr_role)
```