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

```
 1: # -*- coding: utf-8 -*-
 2: from __future__ import print_function, unicode_literals
 3:
 4: import argparse
 5: try:
 6:     import collections.abc as collections
 7: except ImportError:
 8:     import collections
 9: import locale
10: import logging
11: import multiprocessing
12: import os
13: import pprint
14: import sys
15: import time
16: import traceback
17:
18: import six
19:
20: # pelican.log has to be the first pelican module to be loaded
21: # because logging.setLoggerClass has to be called before logging.getLogger
22: from pelican.log import init as init_logging
23: from pelican import signals  # noqa
24: from pelican.generators import (ArticlesGenerator, PagesGenerator,
25:                                 SourceFileGenerator, StaticGenerator,
26:                                 TemplatePagesGenerator)
27: from pelican.readers import Readers
28: from pelican.server import ComplexHTTPRequestHandler, RootedHTTPServer
29: from pelican.settings import read_settings
30: from pelican.utils import (clean_output_dir, file_watcher,
31:                             folder_watcher, maybe_pluralize)
32: from pelican.writers import Writer
33:
34: try:
35:     __version__ = __import__('pkg_resources') \
36:         .get_distribution('pelican').version
37: except Exception:
38:     __version__ = "unknown"
39:
40: DEFAULT_CONFIG_NAME = 'pelicanconf.py'
41: logger = logging.getLogger(__name__)
42:
43:
44: class Pelican(object):
45:
46:     def __init__(self, settings):
47:         """Pelican initialisation
48:
49:         Performs some checks on the environment before doing anything else.
50:         """
51:
52:         # define the default settings
53:         self.settings = settings
54:
55:         self.path = settings['PATH']
56:         self.theme = settings['THEME']
57:         self.output_path = settings['OUTPUT_PATH']
58:         self.ignore_files = settings['IGNORE_FILES']
59:         self.delete_outputdir = settings['DELETE_OUTPUT_DIRECTORY']
60:         self.output_retention = settings['OUTPUT_RETENTION']
61:
```

```
62:         self.init_path()
63:         self.init_plugins()
64:         signals.initialized.send(self)
65:
66:     def init_path(self):
67:         if not any(p in sys.path for p in ['', os.curdir]):
68:             logger.debug("Adding current directory to system path")
69:             sys.path.insert(0, '')
70:
71:     def init_plugins(self):
72:         self.plugins = []
73:         logger.debug('Temporarily adding PLUGIN_PATHS to system path')
74:         _sys_path = sys.path[:]
75:         for pluginpath in self.settings['PLUGIN_PATHS']:
76:             sys.path.insert(0, pluginpath)
77:         for plugin in self.settings['PLUGINS']:
78:             # if it's a string, then import it
79:             if isinstance(plugin, six.string_types):
80:                 logger.debug("Loading plugin `%s`", plugin)
81:                 try:
82:                     plugin = __import__(plugin, globals(), locals(),
83:                                         str('module'))
84:                 except ImportError as e:
85:                     logger.error(
86:                         "Cannot load plugin `%s`\n%s", plugin, e)
87:                     continue
88:
89:             logger.debug("Registering plugin `%s`", plugin.__name__)
90:             plugin.register()
91:             self.plugins.append(plugin)
92:         logger.debug('Restoring system path')
93:         sys.path = _sys_path
94:
95:     def run(self):
96:         """Run the generators and return"""
97:         start_time = time.time()
98:
99:         context = self.settings.copy()
100:         # Share these among all the generators and content objects
101:         # They map source paths to Content objects or None
102:         context['generated_content'] = {}
103:         context['static_links'] = set()
104:         context['static_content'] = {}
105:         context['localsiteurl'] = self.settings['SITEURL']
106:
107:         generators = [
108:             cls(
109:                 context=context,
110:                 settings=self.settings,
111:                 path=self.path,
112:                 theme=self.theme,
113:                 output_path=self.output_path,
114:             ) for cls in self.get_generator_classes()
115:         ]
116:
117:         # erase the directory if it is not the source and if that's
118:         # explicitly asked
119:         if (self.delete_outputdir and not
120:                 os.path.realpath(self.path).startswith(self.output_path)):
121:             clean_output_dir(self.output_path, self.output_retention)
122:
```

```
123:         for p in generators:
124:             if hasattr(p, 'generate_context'):
125:                 p.generate_context()
126: 
127:         for p in generators:
128:             if hasattr(p, 'refresh_metadata_intersite_links'):
129:                 p.refresh_metadata_intersite_links()
130: 
131:         signals.all_generators_finalized.send(generators)
132: 
133:         writer = self.get_writer()
134: 
135:         for p in generators:
136:             if hasattr(p, 'generate_output'):
137:                 p.generate_output(writer)
138: 
139:         signals.finalized.send(self)
140: 
141:         articles_generator = next(g for g in generators
142:                                   if isinstance(g, ArticlesGenerator))
143:         pages_generator = next(g for g in generators
144:                                if isinstance(g, PagesGenerator))
145: 
146:         pluralized_articles = maybe_pluralize(
147:             (len(articles_generator.articles) +
148:              len(articles_generator.translations)),
149:             'article',
150:             'articles')
151:         pluralized_drafts = maybe_pluralize(
152:             (len(articles_generator.drafts) +
153:              len(articles_generator.drafts_translations)),
154:             'draft',
155:             'drafts')
156:         pluralized_pages = maybe_pluralize(
157:             (len(pages_generator.pages) +
158:              len(pages_generator.translations)),
159:             'page',
160:             'pages')
161:         pluralized_hidden_pages = maybe_pluralize(
162:             (len(pages_generator.hidden_pages) +
163:              len(pages_generator.hidden_translations)),
164:             'hidden page',
165:             'hidden pages')
166:         pluralized_draft_pages = maybe_pluralize(
167:             (len(pages_generator.draft_pages) +
168:              len(pages_generator.draft_translations)),
169:             'draft page',
170:             'draft pages')
171: 
172:         print('Done: Processed {}, {}, {}, {} and {} in {:.2f} seconds.'
173:               .format(
174:                     pluralized_articles,
175:                     pluralized_drafts,
176:                     pluralized_pages,
177:                     pluralized_hidden_pages,
178:                     pluralized_draft_pages,
179:                     time.time() - start_time))
180: 
181:     def get_generator_classes(self):
182:         generators = [ArticlesGenerator, PagesGenerator]
183: 
```

```
184:         if self.settings['TEMPLATE_PAGES']:
185:             generators.append(TemplatePagesGenerator)
186:         if self.settings['OUTPUT_SOURCES']:
187:             generators.append(SourceFileGenerator)
188:
189:         for pair in signals.get_generators.send(self):
190:             (funct, value) = pair
191:
192:             if not isinstance(value, collections.Iterable):
193:                 value = (value, )
194:
195:             for v in value:
196:                 if isinstance(v, type):
197:                     logger.debug('Found generator: %s', v)
198:                     generators.append(v)
199:
200:         # StaticGenerator must run last, so it can identify files that
201:         # were skipped by the other generators, and so static files can
202:         # have their output paths overridden by the {attach} link syntax.
203:         generators.append(StaticGenerator)
204:         return generators
205:
206:     def get_writer(self):
207:         writers = [w for (_, w) in signals.get_writer.send(self)
208:                    if isinstance(w, type)]
209:         writers_found = len(writers)
210:         if writers_found == 0:
211:             return Writer(self.output_path, settings=self.settings)
212:         else:
213:             writer = writers[0]
214:             if writers_found == 1:
215:                 logger.debug('Found writer: %s', writer)
216:             else:
217:                 logger.warning(
218:                     '%s writers found, using only first one: %s',
219:                     writers_found, writer)
220:             return writer(self.output_path, settings=self.settings)
221:
222:
223: class PrintSettings(argparse.Action):
224:     def __call__(self, parser, namespace, values, option_string):
225:         instance, settings = get_instance(namespace)
226:
227:         if values:
228:             # One or more arguments provided, so only print those settings
229:             for setting in values:
230:                 if setting in settings:
231:                     # Only add newline between setting name and value if dict
232:                     if isinstance(settings[setting], dict):
233:                         setting_format = '\n{}:\n{}'
234:                     else:
235:                         setting_format = '\n{}: {}'
236:                     print(setting_format.format(
237:                         setting,
238:                         pprint.pformat(settings[setting])))
239:                 else:
240:                     print('\n{} is not a recognized setting.'.format(setting))
241:                     break
242:         else:
243:             # No argument was given to --print-settings, so print all settings
244:             pprint.pprint(settings)
```

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

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

```
367:         config['CACHE_PATH'] = args.cache_path
368:     if args.selected_paths:
369:         config['WRITE_SELECTED'] = args.selected_paths.split(',')
370:     if args.relative_paths:
371:         config['RELATIVE_URLS'] = args.relative_paths
372:     if args.port is not None:
373:         config['PORT'] = args.port
374:     if args.bind is not None:
375:         config['BIND'] = args.bind
376:     config['DEBUG'] = args.verbosity == logging.DEBUG
377:
378:     # argparse returns bytes in Py2. There is no definite answer as to which
379:     # encoding argparse (or sys.argv) uses.
380:     # "Best" option seems to be locale.getpreferredencoding()
381:     # http://mail.python.org/pipermail/python-list/2006-October/405766.html
382:     if not six.PY3:
383:         enc = locale.getpreferredencoding()
384:         for key in config:
385:             if key in ('PATH', 'OUTPUT_PATH', 'THEME'):
386:                 config[key] = config[key].decode(enc)
387:     return config
388:
389:
390: def get_instance(args):
391:
392:     config_file = args.settings
393:     if config_file is None and os.path.isfile(DEFAULT_CONFIG_NAME):
394:         config_file = DEFAULT_CONFIG_NAME
395:         args.settings = DEFAULT_CONFIG_NAME
396:
397:     settings = read_settings(config_file, override=get_config(args))
398:
399:     cls = settings['PELICAN_CLASS']
400:     if isinstance(cls, six.string_types):
401:         module, cls_name = cls.rsplit('.', 1)
402:         module = __import__(module)
403:         cls = getattr(module, cls_name)
404:
405:     return cls(settings), settings
406:
407:
408: def autoreload(watchers, args, old_static, reader_descs, excqueue=None):
409:     while True:
410:         try:
411:             # Check source dir for changed files ending with the given
412:             # extension in the settings. In the theme dir is no such
413:             # restriction; all files are recursively checked if they
414:             # have changed, no matter what extension the filenames
415:             # have.
416:             modified = {k: next(v) for k, v in watchers.items()}
417:
418:             if modified['settings']:
419:                 pelican, settings = get_instance(args)
420:
421:                 # Adjust static watchers if there are any changes
422:                 new_static = settings.get("STATIC_PATHS", [])
423:
424:                 # Added static paths
425:                 # Add new watchers and set them as modified
426:                 new_watchers = set(new_static).difference(old_static)
427:                 for static_path in new_watchers:
```

```
428:                     static_key = '[static]%s' % static_path
429:                     watchers[static_key] = folder_watcher(
430:                         os.path.join(pelican.path, static_path),
431:                         [''],
432:                         pelican.ignore_files)
433:                     modified[static_key] = next(watchers[static_key])
434:
435:                 # Removed static paths
436:                 # Remove watchers and modified values
437:                 old_watchers = set(old_static).difference(new_static)
438:                 for static_path in old_watchers:
439:                     static_key = '[static]%s' % static_path
440:                     watchers.pop(static_key)
441:                     modified.pop(static_key)
442:
443:                 # Replace old_static with the new one
444:                 old_static = new_static
445:
446:             if any(modified.values()):
447:                 print('\n-> Modified: {}. re-generating...'.format(
448:                     ', '.join(k for k, v in modified.items() if v)))
449:
450:                 if modified['content'] is None:
451:                     logger.warning(
452:                         'No valid files found in content for '
453:                         + 'the active readers:\n'
454:                         + '\n'.join(reader_descs))
455:
456:                 if modified['theme'] is None:
457:                     logger.warning('Empty theme folder. Using `basic` '
458:                                    'theme.')
459:
460:                 pelican.run()
461:
462:         except KeyboardInterrupt as e:
463:             logger.warning("Keyboard interrupt, quitting.")
464:             if excqueue is not None:
465:                 excqueue.put(traceback.format_exception_only(type(e), e)[-1])
466:             return
467:
468:         except Exception as e:
469:             if (args.verbosity == logging.DEBUG):
470:                 if excqueue is not None:
471:                     excqueue.put(
472:                         traceback.format_exception_only(type(e), e)[-1])
473:                 else:
474:                     raise
475:             logger.warning(
476:                 'Caught exception "%s". Reloading.', e)
477:
478:         finally:
479:             time.sleep(.5)  # sleep to avoid cpu load
480:
481:
482: def listen(server, port, output, excqueue=None):
483:     RootedHTTPServer.allow_reuse_address = True
484:     try:
485:         httpd = RootedHTTPServer(
486:             output, (server, port), ComplexHTTPRequestHandler)
487:     except OSError as e:
488:         logging.error("Could not listen on port %s, server %s.", port, server)
```

```
489:         if excqueue is not None:
490:             excqueue.put(traceback.format_exception_only(type(e), e)[-1])
491:         return
492:
493:     logging.info("Serving at port %s, server %s.", port, server)
494:     try:
495:         httpd.serve_forever()
496:     except Exception as e:
497:         if excqueue is not None:
498:             excqueue.put(traceback.format_exception_only(type(e), e)[-1])
499:         return
500:
501:
502: def main(argv=None):
503:     args = parse_arguments(argv)
504:     logs_dedup_min_level = getattr(logging, args.logs_dedup_min_level)
505:     init_logging(args.verbosity, args.fatal,
506:                  logs_dedup_min_level=logs_dedup_min_level)
507:
508:     logger.debug('Pelican version: %s', __version__)
509:     logger.debug('Python version: %s', sys.version.split()[0])
510:
511:     try:
512:         pelican, settings = get_instance(args)
513:
514:         readers = Readers(settings)
515:         reader_descs = sorted(set(['%s (%s)' %
516:                                    (type(r).__name__,
517:                                     ', '.join(r.file_extensions))
518:                                    for r in readers.readers.values()
519:                                    if r.enabled]))
520:
521:         watchers = {'content': folder_watcher(pelican.path,
522:                                               readers.extensions,
523:                                               pelican.ignore_files),
524:                     'theme': folder_watcher(pelican.theme,
525:                                             [''],
526:                                             pelican.ignore_files),
527:                     'settings': file_watcher(args.settings)}
528:
529:         old_static = settings.get("STATIC_PATHS", [])
530:         for static_path in old_static:
531:             # use a prefix to avoid possible overriding of standard watchers
532:             # above
533:             watchers['[static]%s' % static_path] = folder_watcher(
534:                 os.path.join(pelican.path, static_path),
535:                 [''],
536:                 pelican.ignore_files)
537:
538:         if args.autoreload and args.listen:
539:             excqueue = multiprocessing.Queue()
540:             p1 = multiprocessing.Process(
541:                 target=autoreload,
542:                 args=(watchers, args, old_static, reader_descs, excqueue))
543:             p2 = multiprocessing.Process(
544:                 target=listen,
545:                 args=(settings.get('BIND'), settings.get('PORT'),
546:                       settings.get("OUTPUT_PATH"), excqueue))
547:             p1.start()
548:             p2.start()
549:             exc = excqueue.get()
```

```
550:             p1.terminate()
551:             p2.terminate()
552:             logger.critical(exc)
553:         elif args.autoreload:
554:             print('  --- AutoReload Mode: Monitoring `content`, `theme` and'
555:                   ' `settings` for changes. ---')
556:             autoreload(watchers, args, old_static, reader_descs)
557:         elif args.listen:
558:             listen(settings.get('BIND'), settings.get('PORT'),
559:                    settings.get("OUTPUT_PATH"))
560:         else:
561:             if next(watchers['content']) is None:
562:                 logger.warning(
563:                     'No valid files found in content for '
564:                     + 'the active readers:\n'
565:                     + '\n'.join(reader_descs))
566: 
567:             if next(watchers['theme']) is None:
568:                 logger.warning('Empty theme folder. Using `basic` theme.')
569: 
570:             pelican.run()
571: 
572:     except Exception as e:
573:         logger.critical('%s', e)
574: 
575:         if args.verbosity == logging.DEBUG:
576:             raise
577:         else:
578:             sys.exit(getattr(e, 'exitcode', 1))
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import calendar
import errno
import fnmatch
import logging
import os
from codecs import open
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)

import six

from pelican import signals
from pelican.cache import FileStampDataCacher
from pelican.contents import Article, Page, Static
from pelican.readers import Readers
from pelican.utils import (DateFormatter, copy, mkdir_p, order_content,
                           posixize_path, process_translations,
                           python_2_unicode_compatible)


logger = logging.getLogger(__name__)


class PelicanTemplateNotFound(Exception):
    pass


@python_2_unicode_compatible
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)

```

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

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

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

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

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

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

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

```
489:
490:         period_date_key = {
491:             'year': attrgetter('date.year'),
492:             'month': attrgetter('date.year', 'date.month'),
493:             'day': attrgetter('date.year', 'date.month', 'date.day')
494:         }
495:
496:         def _generate_period_archives(dates, key, save_as_fmt, url_fmt):
497:             """Generate period archives from `dates`, grouped by
498:             `key` and written to `save_as`.
499:             """
500:             # `dates` is already sorted by date
501:             for _period, group in groupby(dates, key=key):
502:                 archive = list(group)
503:                 articles = [a for a in self.articles if a in archive]
504:                 # arbitrarily grab the first date so that the usual
505:                 # format string syntax can be used for specifying the
506:                 # period archive dates
507:                 date = archive[0].date
508:                 save_as = save_as_fmt.format(date=date)
509:                 url = url_fmt.format(date=date)
510:                 context = self.context.copy()
511:
512:                 if key == period_date_key['year']:
513:                     context["period"] = (_period,)
514:                 else:
515:                     month_name = calendar.month_name[_period[1]]
516:                     if not six.PY3:
517:                         month_name = month_name.decode('utf-8')
518:                     if key == period_date_key['month']:
519:                         context["period"] = (_period[0],
520:                                              month_name)
521:                     else:
522:                         context["period"] = (_period[0],
523:                                              month_name,
524:                                              _period[2])
525:
526:                 write(save_as, template, context, articles=articles,
527:                       dates=archive, template_name='period_archives',
528:                       blog=True, url=url, all_articles=self.articles)
529:
530:         for period in 'year', 'month', 'day':
531:             save_as = period_save_as[period]
532:             url = period_url[period]
533:             if save_as:
534:                 key = period_date_key[period]
535:                 _generate_period_archives(self.dates, key, save_as, url)
536:
537:     def generate_direct_templates(self, write):
538:         """Generate direct templates pages"""
539:         for template in self.settings['DIRECT_TEMPLATES']:
540:             save_as = self.settings.get("%s_SAVE_AS" % template.upper(),
541:                                         '%s.html' % template)
542:             url = self.settings.get("%s_URL" % template.upper(),
543:                                     '%s.html' % template)
544:             if not save_as:
545:                 continue
546:
547:             write(save_as, self.get_template(template), self.context,
548:                   articles=self.articles, dates=self.dates, blog=True,
549:                   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(PagesGenerator, self).__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(StaticGenerator, self).__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 = {os.path.join(self.path, path)
803:                         for path in self.context['static_links']}
804:         found_files = self.get_files(self.settings['STATIC_PATHS'],
805:                                      exclude=self.settings['STATIC_EXCLUDES'],
806:                                      extensions=False)
807:         for f in linked_files | found_files:
808:
809:             # skip content source files unless the user explicitly wants them
810:             if self.settings['STATIC_EXCLUDE_SOURCES']:
811:                 if self._is_potential_source_path(f):
812:                     continue
813:
814:             static = self.readers.read_file(
815:                 base_path=self.path, path=f, content_class=Static,
816:                 fmt='static', context=self.context,
817:                 preread_signal=signals.static_generator_preread,
818:                 preread_sender=self,
819:                 context_signal=signals.static_generator_context,
820:                 context_sender=self)
821:             self.staticfiles.append(static)
822:             self.add_source_path(static, static=True)
823:         self._update_context(('staticfiles',))
824:         signals.static_generator_finalized.send(self)
825:
826:     def generate_output(self, writer):
827:         self._copy_paths(self.settings['THEME_STATIC_PATHS'], self.theme,
828:                          self.settings['THEME_STATIC_DIR'], self.output_path,
829:                          os.curdir)
830:         for sc in self.context['staticfiles']:
831:             if self._file_update_required(sc):
832:                 self._link_or_copy_staticfile(sc)
833:             else:
834:                 logger.debug('%s is up to date, not copying', sc.source_path)
835:
836:     def _copy_paths(self, paths, source, destination, output_path,
837:                     final_path=None):
838:         """Copy all the paths from source to destination"""
839:         for path in paths:
840:             source_path = os.path.join(source, path)
841:
842:             if final_path:
843:                 if os.path.isfile(source_path):
844:                     destination_path = os.path.join(output_path, destination,
845:                                                     final_path,
846:                                                     os.path.basename(path))
847:                 else:
848:                     destination_path = os.path.join(output_path, destination,
849:                                                     final_path)
850:             else:
851:                 destination_path = os.path.join(output_path, destination, path)
852:
853:             copy(source_path, destination_path,
854:                  self.settings['IGNORE_FILES'])
```

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

```
916:         if os.path.lexists(path) and not os.path.isdir(path):
917:             os.unlink(path)
918:         mkdir_p(path)
919:
920:
921: class SourceFileGenerator(Generator):
922:
923:     def generate_context(self):
924:         self.output_extension = self.settings['OUTPUT_SOURCES_EXTENSION']
925:
926:     def _create_source(self, obj):
927:         output_path, _ = os.path.splitext(obj.save_as)
928:         dest = os.path.join(self.output_path,
929:                             output_path + self.output_extension)
930:         copy(obj.source_path, dest)
931:
932:     def generate_output(self, writer=None):
933:         logger.info('Generating source files...')
934:         for obj in chain(self.context['articles'], self.context['pages']):
935:             self._create_source(obj)
936:             for obj_trans in obj.translations:
937:                 self._create_source(obj_trans)
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import codecs
import datetime
import errno
import fnmatch
import locale
import logging
import os
import re
import shutil
import sys
import traceback
try:
    from collections.abc import Hashable
except ImportError:
    from collections import Hashable
from contextlib import contextmanager
from functools import partial
from itertools import groupby
from operator import attrgetter

import dateutil.parser

from jinja2 import Markup

import pytz

import six
from six.moves import html_entities
from six.moves.html_parser import HTMLParser

try:
    from html import escape
except ImportError:
    from cgi import escape

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):
    '''
    Replacement for built-in strftime

    This is necessary because of the way Py2 handles date format strings.
    Specifically, Py2 strftime takes a bytestring. In the case of text output
    (e.g. %b, %a, etc), the output is encoded with an encoding defined by
    locale.LC_TIME. Things get messy if the formatting string has chars that
```

```
    are not valid in LC_TIME defined encoding.

    This works by 'grabbing' possible format strings (those starting with %),
    formatting them with the date, (if necessary) decoding the output 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)

    # we need to convert formatted dates back to unicode in Py2
    # LC_TIME determines the encoding for built-in strftime outputs
    lang_code, enc = locale.getlocale(locale.LC_TIME)

    formatted_candidates = []
    for candidate in candidates:
        # test for valid C89 directives only
        if candidate[-1] in c89_directives:
            # check for '-' prefix
            if len(candidate) == 3:
                # '-' prefix
                candidate = '%{}'.format(candidate[-1])
                conversion = strip_zeros
            else:
                conversion = None

            # format date
            if isinstance(date, SafeDatetime):
                formatted = date.strftime(candidate, safe=False)
            else:
                formatted = date.strftime(candidate)

            # convert Py2 result to unicode
            if not six.PY3 and enc is not None:
                formatted = formatted.decode(enc)

            # strip zeros if '-' prefix is used
            if conversion:
                formatted = conversion(formatted)
        else:
            formatted = candidate
        formatted_candidates.append(formatted)

    # put formatted candidates back and return
    return template % tuple(formatted_candidates)


class SafeDatetime(datetime.datetime):
    '''Subclass of datetime that works with utf-8 format strings on PY2'''

    def strftime(self, fmt, safe=True):
        '''Uses our custom strftime if supposed to be *safe*'''
        if safe:
            return strftime(self, fmt)
```

```
123:         else:
124:             return super(SafeDatetime, self).strftime(fmt)
125:
126:
127: class DateFormatter(object):
128:     '''A date formatter object used as a jinja filter
129:
130:     Uses the `strftime` implementation and makes sure jinja uses the locale
131:     defined in LOCALE setting
132:     '''
133:
134:     def __init__(self):
135:         self.locale = locale.setlocale(locale.LC_TIME)
136:
137:     def __call__(self, date, date_format):
138:         old_lc_time = locale.setlocale(locale.LC_TIME)
139:         old_lc_ctype = locale.setlocale(locale.LC_CTYPE)
140:
141:         locale.setlocale(locale.LC_TIME, self.locale)
142:         # on OSX, encoding from LC_CTYPE determines the unicode output in PY3
143:         # make sure it's same as LC_TIME
144:         locale.setlocale(locale.LC_CTYPE, self.locale)
145:
146:         formatted = strftime(date, date_format)
147:
148:         locale.setlocale(locale.LC_TIME, old_lc_time)
149:         locale.setlocale(locale.LC_CTYPE, old_lc_ctype)
150:         return formatted
151:
152:
153: def python_2_unicode_compatible(klass):
154:     """
155:     A decorator that defines __unicode__ and __str__ methods under Python 2.
156:     Under Python 3 it does nothing.
157:
158:     To support Python 2 and 3 with a single code base, define a __str__ method
159:     returning text and apply this decorator to the class.
160:
161:     From django.utils.encoding.
162:     """
163:     if not six.PY3:
164:         klass.__unicode__ = klass.__str__
165:         klass.__str__ = lambda self: self.__unicode__().encode('utf-8')
166:     return klass
167:
168:
169: class memoized(object):
170:     """Function decorator to cache return values.
171:
172:     If called later with the same arguments, the cached value is returned
173:     (not reevaluated).
174:
175:     """
176:     def __init__(self, func):
177:         self.func = func
178:         self.cache = {}
179:
180:     def __call__(self, *args):
181:         if not isinstance(args, Hashable):
182:             # uncacheable. a list, for instance.
183:             # better to not cache than blow up.
```

```
184:             return self.func(*args)
185:         if args in self.cache:
186:             return self.cache[args]
187:         else:
188:             value = self.func(*args)
189:             self.cache[args] = value
190:             return value
191:
192:     def __repr__(self):
193:         return self.func.__doc__
194:
195:     def __get__(self, obj, objtype):
196:         '''Support instance methods.'''
197:         return partial(self.__call__, obj)
198:
199:
200: def deprecated_attribute(old, new, since=None, remove=None, doc=None):
201:     """Attribute deprecation decorator for gentle upgrades
202:
203:     For example:
204:
205:         class MyClass (object):
206:             @deprecated_attribute(
207:                 old='abc', new='xyz', since=(3, 2, 0), remove=(4, 1, 3))
208:             def abc(): return None
209:
210:             def __init__(self):
211:                 xyz = 5
212:
213:     Note that the decorator needs a dummy method to attach to, but the
214:     content of the dummy method is ignored.
215:     """
216:     def _warn():
217:         version = '.'.join(six.text_type(x) for x in since)
218:         message = ['{} has been deprecated since {}'.format(old, version)]
219:         if remove:
220:             version = '.'.join(six.text_type(x) for x in remove)
221:             message.append(
222:                 ' and will be removed by version {}'.format(version))
223:         message.append('.  Use {} instead.'.format(new))
224:         logger.warning(''.join(message))
225:         logger.debug(''.join(six.text_type(x) for x
226:                              in traceback.format_stack()))
227:
228:     def fget(self):
229:         _warn()
230:         return getattr(self, new)
231:
232:     def fset(self, value):
233:         _warn()
234:         setattr(self, new, value)
235:
236:     def decorator(dummy):
237:         return property(fget=fget, fset=fset, doc=doc)
238:
239:     return decorator
240:
241:
242: def get_date(string):
243:     """Return a datetime object from a string.
244:
```

```
245:     If no format matches the given date, raise a ValueError.
246:     """
247:     string = re.sub(' +', ' ', string)
248:     default = SafeDatetime.now().replace(hour=0, minute=0,
249:                                          second=0, microsecond=0)
250:     try:
251:         return dateutil.parser.parse(string, default=default)
252:     except (TypeError, ValueError):
253:         raise ValueError('{0!r} is not a valid date'.format(string))
254:
255:
256: @contextmanager
257: def pelican_open(filename, mode='rb', strip_crs=(sys.platform == 'win32')):
258:     """Open a file and return its content"""
259:
260:     with codecs.open(filename, mode, encoding='utf-8') as infile:
261:         content = infile.read()
262:     if content[:1] == codecs.BOM_UTF8.decode('utf8'):
263:         content = content[1:]
264:     if strip_crs:
265:         content = content.replace('\r\n', '\n')
266:     yield content
267:
268:
269: def slugify(value, regex_subs=()):
270:     """
271:     Normalizes string, converts to lowercase, removes non-alpha characters,
272:     and converts spaces to hyphens.
273:
274:     Took from Django sources.
275:     """
276:
277:     # TODO Maybe steal again from current Django 1.5dev
278:     value = Markup(value).striptags()
279:     # value must be unicode per se
280:     import unicodedata
281:     from unidecode import unidecode
282:     # unidecode returns str in Py2 and 3, so in Py2 we have to make
283:     # it unicode again
284:     value = unidecode(value)
285:     if isinstance(value, six.binary_type):
286:         value = value.decode('ascii')
287:     # still unicode
288:     value = unicodedata.normalize('NFKD', value)
289:
290:     for src, dst in regex_subs:
291:         value = re.sub(src, dst, value, flags=re.IGNORECASE)
292:
293:     # convert to lowercase
294:     value = value.lower()
295:
296:     # we want only ASCII chars
297:     value = value.encode('ascii', 'ignore').strip()
298:     # but Pelican should generally use only unicode
299:     return value.decode('ascii')
300:
301:
302: def copy(source, destination, ignores=None):
303:     """Recursively copy source into destination.
304:
305:     If source is a file, destination has to be a file as well.
```

```
306:     The function is able to copy either files or directories.
307:
308:     :param source: the source file or directory
309:     :param destination: the destination file or directory
310:     :param ignores: either None, or a list of glob patterns;
311:         files matching those patterns will _not_ be copied.
312:     """
313:
314:     def walk_error(err):
315:         logger.warning("While copying %s: %s: %s",
316:                        source_, err.filename, err.strerror)
317:
318:     source_ = os.path.abspath(os.path.expanduser(source))
319:     destination_ = os.path.abspath(os.path.expanduser(destination))
320:
321:     if ignores is None:
322:         ignores = []
323:
324:     if any(fnmatch.fnmatch(os.path.basename(source), ignore)
325:            for ignore in ignores):
326:         logger.info('Not copying %s due to ignores', source_)
327:         return
328:
329:     if os.path.isfile(source_):
330:         dst_dir = os.path.dirname(destination_)
331:         if not os.path.exists(dst_dir):
332:             logger.info('Creating directory %s', dst_dir)
333:             os.makedirs(dst_dir)
334:         logger.info('Copying %s to %s', source_, destination_)
335:         copy_file_metadata(source_, destination_)
336:
337:     elif os.path.isdir(source_):
338:         if not os.path.exists(destination_):
339:             logger.info('Creating directory %s', destination_)
340:             os.makedirs(destination_)
341:         if not os.path.isdir(destination_):
342:             logger.warning('Cannot copy %s (a directory) to %s (a file)',
343:                            source_, destination_)
344:             return
345:
346:         for src_dir, subdirs, others in os.walk(source_, followlinks=True):
347:             dst_dir = os.path.join(destination_,
348:                                    os.path.relpath(src_dir, source_))
349:
350:             subdirs[:] = (s for s in subdirs if not any(fnmatch.fnmatch(s, i)
351:                                                         for i in ignores))
352:             others[:] = (o for o in others if not any(fnmatch.fnmatch(o, i)
353:                                                       for i in ignores))
354:
355:             if not os.path.isdir(dst_dir):
356:                 logger.info('Creating directory %s', dst_dir)
357:                 # Parent directories are known to exist, so 'mkdir' suffices.
358:                 os.mkdir(dst_dir)
359:
360:             for o in others:
361:                 src_path = os.path.join(src_dir, o)
362:                 dst_path = os.path.join(dst_dir, o)
363:                 if os.path.isfile(src_path):
364:                     logger.info('Copying %s to %s', src_path, dst_path)
365:                     copy_file_metadata(src_path, dst_path)
366:                 else:
```

```
367:                     logger.warning('Skipped copy %s (not a file or '
368:                                    'directory) to %s',
369:                                    src_path, dst_path)
370:
371:
372: def copy_file_metadata(source, destination):
373:     '''Copy a file and its metadata (perm bits, access times, ...)'''
374:
375:     # This function is a workaround for Android python copystat
376:     # bug ([issue28141]) https://bugs.python.org/issue28141
377:     try:
378:         shutil.copy2(source, destination)
379:     except OSError as e:
380:         logger.warning("A problem occurred copying file %s to %s; %s",
381:                        source, destination, e)
382:
383:
384: def clean_output_dir(path, retention):
385:     """Remove all files from output directory except those in retention list"""
386:
387:     if not os.path.exists(path):
388:         logger.debug("Directory already removed: %s", path)
389:         return
390:
391:     if not os.path.isdir(path):
392:         try:
393:             os.remove(path)
394:         except Exception as e:
395:             logger.error("Unable to delete file %s; %s", path, e)
396:         return
397:
398:     # remove existing content from output folder unless in retention list
399:     for filename in os.listdir(path):
400:         file = os.path.join(path, filename)
401:         if any(filename == retain for retain in retention):
402:             logger.debug("Skipping deletion; %s is on retention list: %s",
403:                          filename, file)
404:         elif os.path.isdir(file):
405:             try:
406:                 shutil.rmtree(file)
407:                 logger.debug("Deleted directory %s", file)
408:             except Exception as e:
409:                 logger.error("Unable to delete directory %s; %s",
410:                              file, e)
411:         elif os.path.isfile(file) or os.path.islink(file):
412:             try:
413:                 os.remove(file)
414:                 logger.debug("Deleted file/link %s", file)
415:             except Exception as e:
416:                 logger.error("Unable to delete file %s; %s", file, e)
417:         else:
418:             logger.error("Unable to delete %s, file type unknown", file)
419:
420:
421: def get_relative_path(path):
422:     """Return the relative path from the given path to the root path."""
423:     components = split_all(path)
424:     if len(components) <= 1:
425:         return os.curdir
426:     else:
427:         parents = [os.pardir] * (len(components) - 1)
```

```
428:         return os.path.join(*parents)
429:
430:
431: def path_to_url(path):
432:     """Return the URL corresponding to a given path."""
433:     if os.sep == '/':
434:         return path
435:     else:
436:         return '/'.join(split_all(path))
437:
438:
439: def posixize_path(rel_path):
440:     """Use '/' as path separator, so that source references,
441:     like '{static}/foo/bar.jpg' or 'extras/favicon.ico',
442:     will work on Windows as well as on Mac and Linux."""
443:     return rel_path.replace(os.sep, '/')
444:
445:
446: class _HTMLWordTruncator(HTMLParser):
447:
448:     _word_regex = re.compile(r"\w[\w'-]*", re.U)
449:     _word_prefix_regex = re.compile(r'\w', re.U)
450:     _singlets = ('br', 'col', 'link', 'base', 'img', 'param', 'area',
451:                  'hr', 'input')
452:
453:     class TruncationCompleted(Exception):
454:
455:         def __init__(self, truncate_at):
456:             super(_HTMLWordTruncator.TruncationCompleted, self).__init__(
457:                 truncate_at)
458:             self.truncate_at = truncate_at
459:
460:     def __init__(self, max_words):
461:         # In Python 2, HTMLParser is not a new-style class,
462:         # hence super() cannot be used.
463:         try:
464:             HTMLParser.__init__(self, convert_charrefs=False)
465:         except TypeError:
466:             # pre Python 3.3
467:             HTMLParser.__init__(self)
468:
469:         self.max_words = max_words
470:         self.words_found = 0
471:         self.open_tags = []
472:         self.last_word_end = None
473:         self.truncate_at = None
474:
475:     def feed(self, *args, **kwargs):
476:         try:
477:             # With Python 2, super() cannot be used.
478:             # See the comment for __init__().
479:             HTMLParser.feed(self, *args, **kwargs)
480:         except self.TruncationCompleted as exc:
481:             self.truncate_at = exc.truncate_at
482:         else:
483:             self.truncate_at = None
484:
485:     def getoffset(self):
486:         line_start = 0
487:         lineno, line_offset = self.getpos()
488:         for i in range(lineno - 1):
```

```
489:             line_start = self.rawdata.index('\n', line_start) + 1
490:         return line_start + line_offset
491:
492:     def add_word(self, word_end):
493:         self.words_found += 1
494:         self.last_word_end = None
495:         if self.words_found == self.max_words:
496:             raise self.TruncationCompleted(word_end)
497:
498:     def add_last_word(self):
499:         if self.last_word_end is not None:
500:             self.add_word(self.last_word_end)
501:
502:     def handle_starttag(self, tag, attrs):
503:         self.add_last_word()
504:         if tag not in self._singlets:
505:             self.open_tags.insert(0, tag)
506:
507:     def handle_endtag(self, tag):
508:         self.add_last_word()
509:         try:
510:             i = self.open_tags.index(tag)
511:         except ValueError:
512:             pass
513:         else:
514:             # SGML: An end tag closes, back to the matching start tag,
515:             # all unclosed intervening start tags with omitted end tags
516:             del self.open_tags[:i + 1]
517:
518:     def handle_data(self, data):
519:         word_end = 0
520:         offset = self.getoffset()
521:
522:         while self.words_found < self.max_words:
523:             match = self._word_regex.search(data, word_end)
524:             if not match:
525:                 break
526:
527:             if match.start(0) > 0:
528:                 self.add_last_word()
529:
530:             word_end = match.end(0)
531:             self.last_word_end = offset + word_end
532:
533:         if word_end < len(data):
534:             self.add_last_word()
535:
536:     def _handle_ref(self, name, char):
537:         """
538:         Called by handle_entityref() or handle_charref() when a ref like
539:         `&mdash;`, `&#8212;`, or `&#x2014` is found.
540:
541:         The arguments for this method are:
542:
543:         - `name`: the HTML entity name (such as `mdash` or `#8212` or `#x2014`)
544:         - `char`: the Unicode representation of the ref (such as `â\200\224`)
545:
546:         This method checks whether the entity is considered to be part of a
547:         word or not and, if not, signals the end of a word.
548:         """
549:         # Compute the index of the character right after the ref.
```

```
550:         #
551:         # In a string like 'prefix&mdash;suffix', the end is the sum of:
552:         #
553:         # - `self.getoffset()` (the length of `prefix`)
554:         # - `1` (the length of `&`)
555:         # - `len(name)` (the length of `mdash`)
556:         # - `1` (the length of `;`)
557:         #
558:         # Note that, in case of malformed HTML, the ';' character may
559:         # not be present.
560:
561:         offset = self.getoffset()
562:         ref_end = offset + len(name) + 1
563:
564:         try:
565:             if self.rawdata[ref_end] == ';':
566:                 ref_end += 1
567:         except IndexError:
568:             # We are at the end of the string and there's no ';'
569:             pass
570:
571:         if self.last_word_end is None:
572:             if self._word_prefix_regex.match(char):
573:                 self.last_word_end = ref_end
574:         else:
575:             if self._word_regex.match(char):
576:                 self.last_word_end = ref_end
577:             else:
578:                 self.add_last_word()
579:
580:     def handle_entityref(self, name):
581:         """
582:         Called when an entity ref like '&mdash;' is found
583:
584:         `name` is the entity ref without ampersand and semicolon (e.g. `mdash`)
585:         """
586:         try:
587:             codepoint = html_entities.name2codepoint[name]
588:             char = six.unichr(codepoint)
589:         except KeyError:
590:             char = ''
591:         self._handle_ref(name, char)
592:
593:     def handle_charref(self, name):
594:         """
595:         Called when a char ref like '&#8212;' or '&#x2014' is found
596:
597:         `name` is the char ref without ampersand and semicolon (e.g. `#8212` or
598:         `#x2014`)
599:         """
600:         try:
601:             if name.startswith('x'):
602:                 codepoint = int(name[1:], 16)
603:             else:
604:                 codepoint = int(name)
605:             char = six.unichr(codepoint)
606:         except (ValueError, OverflowError):
607:             char = ''
608:         self._handle_ref('#' + name, char)
609:
610:
```

```
611: def truncate_html_words(s, num, end_text='â\200¦'):
612:     """Truncates HTML to a certain number of words.
613:
614:     (not counting tags and comments). Closes opened tags if they were correctly
615:     closed in the given html. Takes an optional argument of what should be used
616:     to notify that the string has been truncated, defaulting to ellipsis (â\200¦).
617:
618:     Newlines in the HTML are preserved. (From the django framework).
619:     """
620:     length = int(num)
621:     if length <= 0:
622:         return ''
623:     truncator = _HTMLWordTruncator(length)
624:     truncator.feed(s)
625:     if truncator.truncate_at is None:
626:         return s
627:     out = s[:truncator.truncate_at]
628:     if end_text:
629:         out += ' ' + end_text
630:     # Close any tags still open
631:     for tag in truncator.open_tags:
632:         out += '</%s>' % tag
633:     # Return string
634:     return out
635:
636:
637: def escape_html(text, quote=True):
638:     """Escape '&', '<' and '>' to HTML-safe sequences.
639:
640:     In Python 2 this uses cgi.escape and in Python 3 this uses html.escape. We
641:     wrap here to ensure the quote argument has an identical default."""
642:     return escape(text, quote=quote)
643:
644:
645: def process_translations(content_list, translation_id=None):
646:     """ Finds translations and returns them.
647:
648:     For each content_list item, populates the 'translations' attribute, and
649:     returns a tuple with two lists (index, translations). Index list includes
650:     items in default language or items which have no variant in default
651:     language. Items with the `translation` metadata set to something else than
652:     `False` or `false` will be used as translations, unless all the items in
653:     the same group have that metadata.
654:
655:     Translations and original items are determined relative to one another
656:     amongst items in the same group. Items are in the same group if they
657:     have the same value(s) for the metadata attribute(s) specified by the
658:     'translation_id', which must be a string or a collection of strings.
659:     If 'translation_id' is falsy, the identification of translations is skipped
660:     and all items are returned as originals.
661:     """
662:
663:     if not translation_id:
664:         return content_list, []
665:
666:     if isinstance(translation_id, six.string_types):
667:         translation_id = {translation_id}
668:
669:     index = []
670:
671:     try:
```

```
672:         content_list.sort(key=attrgetter(*translation_id))
673:     except TypeError:
674:         raise TypeError('Cannot unpack {}, \'translation_id\' must be falsy, a'
675:                         'string or a collection of strings'
676:                         .format(translation_id))
677:     except AttributeError:
678:         raise AttributeError('Cannot use {} as \'translation_id\', there'
679:                              'appear to be items without these metadata'
680:                              'attributes'.format(translation_id))
681:
682:     for id_vals, items in groupby(content_list, attrgetter(*translation_id)):
683:         # prepare warning string
684:         id_vals = (id_vals,) if len(translation_id) == 1 else id_vals
685:         with_str = 'with' + ', '.join([' {} "{{}}"'] * len(translation_id))\
686:             .format(*translation_id).format(*id_vals)
687:
688:         items = list(items)
689:         original_items = get_original_items(items, with_str)
690:         index.extend(original_items)
691:         for a in items:
692:             a.translations = [x for x in items if x != a]
693:
694:     translations = [x for x in content_list if x not in index]
695:
696:     return index, translations
697:
698:
699: def get_original_items(items, with_str):
700:     def _warn_source_paths(msg, items, *extra):
701:         args = [len(items)]
702:         args.extend(extra)
703:         args.extend((x.source_path for x in items))
704:         logger.warning('{}: {}'.format(msg, '\n%s' * len(items)), *args)
705:
706:     # warn if several items have the same lang
707:     for lang, lang_items in groupby(items, attrgetter('lang')):
708:         lang_items = list(lang_items)
709:         if len(lang_items) > 1:
710:             _warn_source_paths('There are %s items "%s" with lang %s',
711:                                lang_items, with_str, lang)
712:
713:     # items with `translation` metadata will be used as translations...
714:     candidate_items = [
715:         i for i in items
716:         if i.metadata.get('translation', 'false').lower() == 'false']
717:
718:     # ...unless all items with that slug are translations
719:     if not candidate_items:
720:         _warn_source_paths('All items ("%s") "%s" are translations',
721:                            items, with_str)
722:         candidate_items = items
723:
724:     # find items with default language
725:     original_items = [i for i in candidate_items if i.in_default_lang]
726:
727:     # if there is no article with default language, go back one step
728:     if not original_items:
729:         original_items = candidate_items
730:
731:     # warn if there are several original items
732:     if len(original_items) > 1:
```

```
733:         _warn_source_paths('There are %s original (not translated) items %s',
734:                            original_items, with_str)
735:     return original_items
736:
737:
738: def order_content(content_list, order_by='slug'):
739:     """ Sorts content.
740:
741:     order_by can be a string of an attribute or sorting function. If order_by
742:     is defined, content will be ordered by that attribute or sorting function.
743:     By default, content is ordered by slug.
744:
745:     Different content types can have default order_by attributes defined
746:     in settings, e.g. PAGES_ORDER_BY='sort-order', in which case `sort-order`
747:     should be a defined metadata attribute in each page.
748:     """
749:
750:     if order_by:
751:         if callable(order_by):
752:             try:
753:                 content_list.sort(key=order_by)
754:             except Exception:
755:                 logger.error('Error sorting with function %s', order_by)
756:         elif isinstance(order_by, six.string_types):
757:             if order_by.startswith('reversed-'):
758:                 order_reversed = True
759:                 order_by = order_by.replace('reversed-', '', 1)
760:             else:
761:                 order_reversed = False
762:
763:             if order_by == 'basename':
764:                 content_list.sort(
765:                     key=lambda x: os.path.basename(x.source_path or ''),
766:                     reverse=order_reversed)
767:             else:
768:                 try:
769:                     content_list.sort(key=attrgetter(order_by),
770:                                       reverse=order_reversed)
771:                 except AttributeError:
772:                     for content in content_list:
773:                         try:
774:                             getattr(content, order_by)
775:                         except AttributeError:
776:                             logger.warning(
777:                                 'There is no "%s" attribute in "%s". '
778:                                 'Defaulting to slug order.',
779:                                 order_by,
780:                                 content.get_relative_source_path(),
781:                                 extra={
782:                                     'limit_msg': ('More files are missing '
783:                                                   'the needed attribute.')
784:                                 })
785:         else:
786:             logger.warning(
787:                 'Invalid *_ORDER_BY setting (%s).'
788:                 'Valid options are strings and functions.', order_by)
789:
790:     return content_list
791:
792:
793: def folder_watcher(path, extensions, ignores=[]):
```

```
794:     '''Generator for monitoring a folder for modifications.
795:
796:     Returns a boolean indicating if files are changed since last check.
797:     Returns None if there are no matching files in the folder'''
798:
799:     def file_times(path):
800:         '''Return `mtime` for each file in path'''
801:
802:         for root, dirs, files in os.walk(path, followlinks=True):
803:             dirs[:] = [x for x in dirs if not x.startswith(os.curdir)]
804:
805:             for f in files:
806:                 valid_extension = f.endswith(tuple(extensions))
807:                 file_ignored = any(
808:                     fnmatch.fnmatch(f, ignore) for ignore in ignores
809:                 )
810:                 if valid_extension and not file_ignored:
811:                     try:
812:                         yield os.stat(os.path.join(root, f)).st_mtime
813:                     except OSError as e:
814:                         logger.warning('Caught Exception: %s', e)
815:
816:     LAST_MTIME = 0
817:     while True:
818:         try:
819:             mtime = max(file_times(path))
820:             if mtime > LAST_MTIME:
821:                 LAST_MTIME = mtime
822:                 yield True
823:         except ValueError:
824:             yield None
825:         else:
826:             yield False
827:
828:
829: def file_watcher(path):
830:     '''Generator for monitoring a file for modifications'''
831:     LAST_MTIME = 0
832:     while True:
833:         if path:
834:             try:
835:                 mtime = os.stat(path).st_mtime
836:             except OSError as e:
837:                 logger.warning('Caught Exception: %s', e)
838:                 continue
839:
840:             if mtime > LAST_MTIME:
841:                 LAST_MTIME = mtime
842:                 yield True
843:             else:
844:                 yield False
845:         else:
846:             yield None
847:
848:
849: def set_date_tzinfo(d, tz_name=None):
850:     """Set the timezone for dates that don't have tzinfo"""
851:     if tz_name and not d.tzinfo:
852:         tz = pytz.timezone(tz_name)
853:         d = tz.localize(d)
854:         return SafeDatetime(d.year, d.month, d.day, d.hour, d.minute, d.second,
```

```
                                d.microsecond, d.tzinfo)
    return d


def mkdir_p(path):
    try:
        os.makedirs(path)
    except OSError as e:
        if e.errno != errno.EEXIST or not os.path.isdir(path):
            raise


def split_all(path):
    """Split a path into a list of components

    While os.path.split() splits a single component off the back of
    `path`, this function splits all components:

    >>> split_all(os.path.join('a', 'b', 'c'))
    ['a', 'b', 'c']
    """
    components = []
    path = path.lstrip('/')
    while path:
        head, tail = os.path.split(path)
        if tail:
            components.insert(0, tail)
        elif head == path:
            components.insert(0, head)
            break
        path = head
    return components


def is_selected_for_writing(settings, path):
    '''Check whether path is selected for writing
    according to the WRITE_SELECTED list

    If WRITE_SELECTED is an empty list (default),
    any path is selected for writing.
    '''
    if settings['WRITE_SELECTED']:
        return path in settings['WRITE_SELECTED']
    else:
        return True


def path_to_file_url(path):
    '''Convert file-system path to file:// URL'''
    return six.moves.urllib_parse.urljoin(
        "file://", six.moves.urllib.request.pathname2url(path))


def maybe_pluralize(count, singular, plural):
    '''
    Returns a formatted string containing count and plural if count is not 1
    Returns count and singular if count is 1

    maybe_pluralize(0, 'Article', 'Articles') -> '0 Articles'
    maybe_pluralize(1, 'Article', 'Articles') -> '1 Article'
    maybe_pluralize(2, 'Article', 'Articles') -> '2 Articles'
```

```
916:
917:     '''
918:     selection = plural
919:     if count == 1:
920:         selection = singular
921:     return '{} {}'.format(count, selection)
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import logging
import os
import re
from collections import OrderedDict

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

import six
from six import StringIO
from six.moves.html_parser import HTMLParser

from pelican import rstdirectives  # NOQA
from pelican import signals
from pelican.cache import FileStampDataCacher
from pelican.contents import Author, Category, Page, Tag
from pelican.utils import SafeDatetime, escape_html, 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)
```

```
62:         for author in ensure_metadata_list(x)
63:     ] or _DISCARD),
64:     'slug': lambda x, y: x.strip() or _DISCARD,
65: }
66:
67: logger = logging.getLogger(__name__)
68:
69:
70: def ensure_metadata_list(text):
71:     """Canonicalize the format of a list of authors or tags.  This works
72:        the same way as Docutils' "authors" field: if it's already a list,
73:        those boundaries are preserved; otherwise, it must be a string;
74:        if the string contains semicolons, it is split on semicolons;
75:        otherwise, it is split on commas.  This allows you to write
76:        author lists in either "Jane Doe, John Doe" or "Doe, Jane; Doe, John"
77:        format.
78:
79:        Regardless, all list items undergo .strip() before returning, and
80:        empty items are discarded.
81:     """
82:     if isinstance(text, six.text_type):
83:         if ';' in text:
84:             text = text.split(';')
85:         else:
86:             text = text.split(',')
87:
88:     return list(OrderedDict.fromkeys(
89:         [v for v in (w.strip() for w in text) if v]
90:     ))
91:
92:
93: def _process_if_nonempty(processor, name, settings):
94:     """Removes extra whitespace from name and applies a metadata processor.
95:     If name is empty or all whitespace, returns _DISCARD instead.
96:     """
97:     name = name.strip()
98:     return processor(name, settings) if name else _DISCARD
99:
100:
101: def _filter_discardable_metadata(metadata):
102:     """Return a copy of a dict, minus any items marked as discardable."""
103:     return {name: val for name, val in metadata.items() if val is not _DISCARD}
104:
105:
106: class BaseReader(object):
107:     """Base class to read files.
108:
109:     This class is used to process static files, and it can be inherited for
110:     other types of file. A Reader class must have the following attributes:
111:
112:     - enabled: (boolean) tell if the Reader class is enabled. It
113:       generally depends on the import of some dependency.
114:     - file_extensions: a list of file extensions that the Reader will process.
115:     - extensions: a list of extensions to use in the reader (typical use is
116:       Markdown).
117:
118:     """
119:     enabled = True
120:     file_extensions = ['static']
121:     extensions = None
122:
```

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

```
184:
185: class RstReader(BaseReader):
186:     """Reader for reStructuredText files
187:
188:     By default the output HTML is written using
189:     docutils.writers.html4css1.Writer and translated using a subclass of
190:     docutils.writers.html4css1.HTMLTranslator. If you want to override it with
191:     your own writer/translator (e.g. a HTML5-based one), pass your classes to
192:     these two attributes. Look in the source code for details.
193:
194:         writer_class                    Used for writing contents
195:         field_body_translator_class     Used for translating metadata such
196:             as article summary
197:
198:     """
199:
200:     enabled = bool(docutils)
201:     file_extensions = ['rst']
202:
203:     writer_class = PelicanHTMLWriter
204:     field_body_translator_class = _FieldBodyTranslator
205:
206:     class FileInput(docutils.io.FileInput):
207:         """Patch docutils.io.FileInput to remove "U" mode in py3.
208:
209:         Universal newlines is enabled by default and "U" mode is deprecated
210:         in py3.
211:
212:         """
213:
214:         def __init__(self, *args, **kwargs):
215:             if six.PY3:
216:                 kwargs['mode'] = kwargs.get('mode', 'r').replace('U', '')
217:             docutils.io.FileInput.__init__(self, *args, **kwargs)
218:
219:     def __init__(self, *args, **kwargs):
220:         super(RstReader, self).__init__(*args, **kwargs)
221:
222:         lang_code = self.settings.get('DEFAULT_LANG', 'en')
223:         if get_docutils_lang(lang_code):
224:             self._language_code = lang_code
225:         else:
226:             logger.warning("Docutils has no localization for '%s'."
227:                            " Using 'en' instead.", lang_code)
228:             self._language_code = 'en'
229:
230:     def _parse_metadata(self, document, source_path):
231:         """Return the dict containing document metadata"""
232:         formatted_fields = self.settings['FORMATTED_FIELDS']
233:
234:         output = {}
235:
236:         if document.first_child_matching_class(docutils.nodes.title) is None:
237:             logger.warning(
238:                 'Document title missing in file %s: '
239:                 'Ensure exactly one top level section',
240:                 source_path)
241:
242:         for docinfo in document.traverse(docutils.nodes.docinfo):
243:             for element in docinfo.children:
244:                 if element.tagname == 'field':  # custom fields (e.g. summary)
```

```
245:                     name_elem, body_elem = element.children
246:                     name = name_elem.astext()
247:                     if name in formatted_fields:
248:                         value = render_node_to_html(
249:                             document, body_elem,
250:                             self.field_body_translator_class)
251:                     else:
252:                         value = body_elem.astext()
253:                 elif element.tagname == 'authors':  # author list
254:                     name = element.tagname
255:                     value = [element.astext() for element in element.children]
256:                 else:  # standard fields (e.g. address)
257:                     name = element.tagname
258:                     value = element.astext()
259:                 name = name.lower()
260:
261:                 output[name] = self.process_metadata(name, value)
262:         return output
263:
264:     def _get_publisher(self, source_path):
265:         extra_params = {'initial_header_level': '2',
266:                         'syntax_highlight': 'short',
267:                         'input_encoding': 'utf-8',
268:                         'language_code': self._language_code,
269:                         'halt_level': 2,
270:                         'traceback': True,
271:                         'warning_stream': StringIO(),
272:                         'embed_stylesheet': False}
273:         user_params = self.settings.get('DOCUTILS_SETTINGS')
274:         if user_params:
275:             extra_params.update(user_params)
276:
277:         pub = docutils.core.Publisher(
278:             writer=self.writer_class(),
279:             source_class=self.FileInput,
280:             destination_class=docutils.io.StringOutput)
281:         pub.set_components('standalone', 'restructuredtext', 'html')
282:         pub.process_programmatic_settings(None, extra_params, None)
283:         pub.set_source(source_path=source_path)
284:         pub.publish()
285:         return pub
286:
287:     def read(self, source_path):
288:         """Parses restructured text"""
289:         pub = self._get_publisher(source_path)
290:         parts = pub.writer.parts
291:         content = parts.get('body')
292:
293:         metadata = self._parse_metadata(pub.document, source_path)
294:         metadata.setdefault('title', parts.get('title'))
295:
296:         return content, metadata
297:
298:
299: class MarkdownReader(BaseReader):
300:     """Reader for Markdown files"""
301:
302:     enabled = bool(Markdown)
303:     file_extensions = ['md', 'markdown', 'mkd', 'mdown']
304:
305:     def __init__(self, *args, **kwargs):
```

```
306:         super(MarkdownReader, self).__init__(*args, **kwargs)
307:         settings = self.settings['MARKDOWN']
308:         settings.setdefault('extension_configs', {})
309:         settings.setdefault('extensions', [])
310:         for extension in settings['extension_configs'].keys():
311:             if extension not in settings['extensions']:
312:                 settings['extensions'].append(extension)
313:         if 'markdown.extensions.meta' not in settings['extensions']:
314:             settings['extensions'].append('markdown.extensions.meta')
315:         self._source_path = None
316:
317:     def _parse_metadata(self, meta):
318:         """Return the dict containing document metadata"""
319:         formatted_fields = self.settings['FORMATTED_FIELDS']
320:
321:         output = {}
322:         for name, value in meta.items():
323:             name = name.lower()
324:             if name in formatted_fields:
325:                 # formatted metadata is special case and join all list values
326:                 formatted_values = "\n".join(value)
327:                 # reset the markdown instance to clear any state
328:                 self._md.reset()
329:                 formatted = self._md.convert(formatted_values)
330:                 output[name] = self.process_metadata(name, formatted)
331:             elif not DUPLICATES_DEFINITIONS_ALLOWED.get(name, True):
332:                 if len(value) > 1:
333:                     logger.warning(
334:                         'Duplicate definition of `%s` '
335:                         'for %s. Using first one.',
336:                         name, self._source_path)
337:                 output[name] = self.process_metadata(name, value[0])
338:             elif len(value) > 1:
339:                 # handle list metadata as list of string
340:                 output[name] = self.process_metadata(name, value)
341:             else:
342:                 # otherwise, handle metadata as single string
343:                 output[name] = self.process_metadata(name, value[0])
344:         return output
345:
346:     def read(self, source_path):
347:         """Parse content and metadata of markdown files"""
348:
349:         self._source_path = source_path
350:         self._md = Markdown(**self.settings['MARKDOWN'])
351:         with pelican_open(source_path) as text:
352:             content = self._md.convert(text)
353:
354:         if hasattr(self._md, 'Meta'):
355:             metadata = self._parse_metadata(self._md.Meta)
356:         else:
357:             metadata = {}
358:         return content, metadata
359:
360:
361: class HTMLReader(BaseReader):
362:     """Parses HTML files as input, looking for meta, title, and body tags"""
363:
364:     file_extensions = ['htm', 'html']
365:     enabled = True
366:
```

```
367:     class _HTMLParser(HTMLParser):
368:         def __init__(self, settings, filename):
369:             try:
370:                 # Python 3.5+
371:                 HTMLParser.__init__(self, convert_charrefs=False)
372:             except TypeError:
373:                 HTMLParser.__init__(self)
374:             self.body = ''
375:             self.metadata = {}
376:             self.settings = settings
377:
378:             self._data_buffer = ''
379:
380:             self._filename = filename
381:
382:             self._in_top_level = True
383:             self._in_head = False
384:             self._in_title = False
385:             self._in_body = False
386:             self._in_tags = False
387:
388:         def handle_starttag(self, tag, attrs):
389:             if tag == 'head' and self._in_top_level:
390:                 self._in_top_level = False
391:                 self._in_head = True
392:             elif tag == 'title' and self._in_head:
393:                 self._in_title = True
394:                 self._data_buffer = ''
395:             elif tag == 'body' and self._in_top_level:
396:                 self._in_top_level = False
397:                 self._in_body = True
398:                 self._data_buffer = ''
399:             elif tag == 'meta' and self._in_head:
400:                 self._handle_meta_tag(attrs)
401:
402:             elif self._in_body:
403:                 self._data_buffer += self.build_tag(tag, attrs, False)
404:
405:         def handle_endtag(self, tag):
406:             if tag == 'head':
407:                 if self._in_head:
408:                     self._in_head = False
409:                     self._in_top_level = True
410:             elif self._in_head and tag == 'title':
411:                 self._in_title = False
412:                 self.metadata['title'] = self._data_buffer
413:             elif tag == 'body':
414:                 self.body = self._data_buffer
415:                 self._in_body = False
416:                 self._in_top_level = True
417:             elif self._in_body:
418:                 self._data_buffer += '</{}>'.format(escape_html(tag))
419:
420:         def handle_startendtag(self, tag, attrs):
421:             if tag == 'meta' and self._in_head:
422:                 self._handle_meta_tag(attrs)
423:             if self._in_body:
424:                 self._data_buffer += self.build_tag(tag, attrs, True)
425:
426:         def handle_comment(self, data):
427:             self._data_buffer += '<!--{}-->'.format(data)
```

```
428:
429:         def handle_data(self, data):
430:             self._data_buffer += data
431:
432:         def handle_entityref(self, data):
433:             self._data_buffer += '&{};'.format(data)
434:
435:         def handle_charref(self, data):
436:             self._data_buffer += '&#{};'.format(data)
437:
438:         def build_tag(self, tag, attrs, close_tag):
439:             result = '<{}'.format(escape_html(tag))
440:             for k, v in attrs:
441:                 result += ' ' + escape_html(k)
442:                 if v is not None:
443:                     # If the attribute value contains a double quote, surround
444:                     # with single quotes, otherwise use double quotes.
445:                     if '"' in v:
446:                         result += "='{}'".format(escape_html(v, quote=False))
447:                     else:
448:                         result += '="{}"'.format(escape_html(v, quote=False))
449:             if close_tag:
450:                 return result + ' />'
451:             return result + '>'
452:
453:         def _handle_meta_tag(self, attrs):
454:             name = self._attr_value(attrs, 'name')
455:             if name is None:
456:                 attr_list = ['{}="{}"'.format(k, v) for k, v in attrs]
457:                 attr_serialized = ', '.join(attr_list)
458:                 logger.warning("Meta tag in file %s does not have a 'name' "
459:                                "attribute, skipping. Attributes: %s",
460:                                self._filename, attr_serialized)
461:                 return
462:             name = name.lower()
463:             contents = self._attr_value(attrs, 'content', '')
464:             if not contents:
465:                 contents = self._attr_value(attrs, 'contents', '')
466:                 if contents:
467:                     logger.warning(
468:                         "Meta tag attribute 'contents' used in file %s, should"
469:                         " be changed to 'content'",
470:                         self._filename,
471:                         extra={'limit_msg': "Other files have meta tag "
472:                                             "attribute 'contents' that should "
473:                                             "be changed to 'content'"})
474:
475:             if name == 'keywords':
476:                 name = 'tags'
477:
478:             if name in self.metadata:
479:                 # if this metadata already exists (i.e. a previous tag with the
480:                 # same name has already been specified then either convert to
481:                 # list or append to list
482:                 if isinstance(self.metadata[name], list):
483:                     self.metadata[name].append(contents)
484:                 else:
485:                     self.metadata[name] = [self.metadata[name], contents]
486:             else:
487:                 self.metadata[name] = contents
488:
```

```
489:         @classmethod
490:         def _attr_value(cls, attrs, name, default=None):
491:             return next((x[1] for x in attrs if x[0] == name), default)
492:
493:     def read(self, filename):
494:         """Parse content and metadata of HTML files"""
495:         with pelican_open(filename) as content:
496:             parser = self._HTMLParser(self.settings, filename)
497:             parser.feed(content)
498:             parser.close()
499:
500:         metadata = {}
501:         for k in parser.metadata:
502:             metadata[k] = self.process_metadata(k, parser.metadata[k])
503:         return parser.body, metadata
504:
505:
506: class Readers(FileStampDataCacher):
507:     """Interface for all readers.
508:
509:     This class contains a mapping of file extensions / Reader classes, to know
510:     which Reader class must be used to read a file (based on its extension).
511:     This is customizable both with the 'READERS' setting, and with the
512:     'readers_init' signall for plugins.
513:
514:     """
515:
516:     def __init__(self, settings=None, cache_name=''):
517:         self.settings = settings or {}
518:         self.readers = {}
519:         self.reader_classes = {}
520:
521:         for cls in [BaseReader] + BaseReader.__subclasses__():
522:             if not cls.enabled:
523:                 logger.debug('Missing dependencies for %s',
524:                              ', '.join(cls.file_extensions))
525:                 continue
526:
527:             for ext in cls.file_extensions:
528:                 self.reader_classes[ext] = cls
529:
530:         if self.settings['READERS']:
531:             self.reader_classes.update(self.settings['READERS'])
532:
533:         signals.readers_init.send(self)
534:
535:         for fmt, reader_class in self.reader_classes.items():
536:             if not reader_class:
537:                 continue
538:
539:             self.readers[fmt] = reader_class(self.settings)
540:
541:         # set up caching
542:         cache_this_level = (cache_name != '' and
543:                             self.settings['CONTENT_CACHING_LAYER'] == 'reader')
544:         caching_policy = cache_this_level and self.settings['CACHE_CONTENT']
545:         load_policy = cache_this_level and self.settings['LOAD_CONTENT_CACHE']
546:         super(Readers, self).__init__(settings, cache_name,
547:                                       caching_policy, load_policy,
548:                                       )
549:
```

```
550:     @property
551:     def extensions(self):
552:         return self.readers.keys()
553:
554:     def read_file(self, base_path, path, content_class=Page, fmt=None,
555:                   context=None, preread_signal=None, preread_sender=None,
556:                   context_signal=None, context_sender=None):
557:         """Return a content object parsed with the given format."""
558:
559:         path = os.path.abspath(os.path.join(base_path, path))
560:         source_path = posixize_path(os.path.relpath(path, base_path))
561:         logger.debug(
562:             'Read file %s -> %s',
563:             source_path, content_class.__name__)
564:
565:         if not fmt:
566:             _, ext = os.path.splitext(os.path.basename(path))
567:             fmt = ext[1:]
568:
569:         if fmt not in self.readers:
570:             raise TypeError(
571:                 'Pelican does not know how to parse %s', path)
572:
573:         if preread_signal:
574:             logger.debug(
575:                 'Signal %s.send(%s)',
576:                 preread_signal.name, preread_sender)
577:             preread_signal.send(preread_sender)
578:
579:         reader = self.readers[fmt]
580:
581:         metadata = _filter_discardable_metadata(default_metadata(
582:             settings=self.settings, process=reader.process_metadata))
583:         metadata.update(path_metadata(
584:             full_path=path, source_path=source_path,
585:             settings=self.settings))
586:         metadata.update(_filter_discardable_metadata(parse_path_metadata(
587:             source_path=source_path, settings=self.settings,
588:             process=reader.process_metadata)))
589:         reader_name = reader.__class__.__name__
590:         metadata['reader'] = reader_name.replace('Reader', '').lower()
591:
592:         content, reader_metadata = self.get_cached_data(path, (None, None))
593:         if content is None:
594:             content, reader_metadata = reader.read(path)
595:             self.cache_data(path, (content, reader_metadata))
596:         metadata.update(_filter_discardable_metadata(reader_metadata))
597:
598:         if content:
599:             # find images with empty alt
600:             find_empty_alt(content, path)
601:
602:         # eventually filter the content with typogrify if asked so
603:         if self.settings['TYPOGRIFY']:
604:             from typogrify.filters import typogrify
605:             import smartypants
606:
607:             # Tell `smartypants` to also replace &quot; HTML entities with
608:             # smart quotes. This is necessary because Docutils has already
609:             # replaced double quotes with said entities by the time we run
610:             # this filter.
```

```
611:             smartypants.Attr.default |= smartypants.Attr.w
612:
613:             def typogrify_wrapper(text):
614:                 """Ensures ignore_tags feature is backward compatible"""
615:                 try:
616:                     return typogrify(
617:                         text,
618:                         self.settings['TYPOGRIFY_IGNORE_TAGS'])
619:                 except TypeError:
620:                     return typogrify(text)
621:
622:             if content:
623:                 content = typogrify_wrapper(content)
624:
625:             if 'title' in metadata:
626:                 metadata['title'] = typogrify_wrapper(metadata['title'])
627:
628:             if 'summary' in metadata:
629:                 metadata['summary'] = typogrify_wrapper(metadata['summary'])
630:
631:         if context_signal:
632:             logger.debug(
633:                 'Signal %s.send(%s, <metadata>)',
634:                 context_signal.name,
635:                 context_sender)
636:             context_signal.send(context_sender, metadata=metadata)
637:
638:         return content_class(content=content, metadata=metadata,
639:                              settings=self.settings, source_path=path,
640:                              context=context)
641:
642:
643: def find_empty_alt(content, path):
644:     """Find images with empty alt
645:
646:     Create warnings for all images with empty alt (up to a certain number),
647:     as they are really likely to be accessibility flaws.
648:
649:     """
650:     imgs = re.compile(r"""
651:         (?:
652:             # src before alt
653:             <img
654:             [^\>]*
655:             src=(['"])(.*?)\1
656:             [^\>]*
657:             alt=(['"])\3
658:         )|(?:
659:             # alt before src
660:             <img
661:             [^\>]*
662:             alt=(['"])\4
663:             [^\>]*
664:             src=(['"])(.*?)\5
665:         )
666:         """, re.X)
667:     for match in re.findall(imgs, content):
668:         logger.warning(
669:             'Empty alt attribute for image %s in %s',
670:             os.path.basename(match[1] + match[5]), path,
671:             extra={'limit_msg': 'Other images have empty alt attributes'})
```

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

```
733:     {'category': <pelican.urlwrappers.Category object at ...>,
734:      'date': SafeDatetime(2013, 1, 1, 0, 0),
735:      'slug': 'my-slug'}
736:     """
737:     metadata = {}
738:     dirname, basename = os.path.split(source_path)
739:     base, ext = os.path.splitext(basename)
740:     subdir = os.path.basename(dirname)
741:     if settings:
742:         checks = []
743:         for key, data in [('FILENAME_METADATA', base),
744:                           ('PATH_METADATA', source_path)]:
745:             checks.append((settings.get(key, None), data))
746:         if settings.get('USE_FOLDER_AS_CATEGORY', None):
747:             checks.append(('(?P<category>.*)', subdir))
748:         for regexp, data in checks:
749:             if regexp and data:
750:                 match = re.match(regexp, data)
751:                 if match:
752:                     # .items() for py3k compat.
753:                     for k, v in match.groupdict().items():
754:                         k = k.lower()  # metadata must be lowercase
755:                         if v is not None and k not in metadata:
756:                             if process:
757:                                 v = process(k, v)
758:                             metadata[k] = v
759:     return metadata
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

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

import six

from pelican.log import LimitFilter


try:
    # spec_from_file_location is the recommended way in Python 3.5+
    import importlib.util

    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
except ImportError:
    # but it does not exist in Python 2.7, so fall back to imp
    import imp
    load_source = imp.load_source


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_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': [],
    'SUMMARY_MAX_LENGTH': 50,
    'PLUGIN_PATHS': [],
    'PLUGINS': [],
    '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)
```

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

```
245:     for key, value in DEFAULT_CONFIG['JINJA_ENVIRONMENT'].items():
246:         if key not in jinja_env:
247:             jinja_env[key] = value
248:
249:     return settings
250:
251:
252: def _printf_s_to_format_field(printf_string, format_field):
253:     """Tries to replace %s with {format_field} in the provided printf_string.
254:     Raises ValueError in case of failure.
255:     """
256:     TEST_STRING = 'PELICAN_PRINTF_S_DEPRECATION'
257:     expected = printf_string % TEST_STRING
258:
259:     result = printf_string.replace('{', '{{').replace('}', '}}') \
260:         % '{{{}}}'.format(format_field)
261:     if result.format(**{format_field: TEST_STRING}) != expected:
262:         raise ValueError('Failed to safely replace %s with {{{}}}'.format(
263:             format_field))
264:
265:     return result
266:
267:
268: def handle_deprecated_settings(settings):
269:     """Converts deprecated settings and issues warnings. Issues an exception
270:     if both old and new setting is specified.
271:     """
272:
273:     # PLUGIN_PATH -> PLUGIN_PATHS
274:     if 'PLUGIN_PATH' in settings:
275:         logger.warning('PLUGIN_PATH setting has been replaced by '
276:                        'PLUGIN_PATHS, moving it to the new setting name.')
277:         settings['PLUGIN_PATHS'] = settings['PLUGIN_PATH']
278:         del settings['PLUGIN_PATH']
279:
280:     # PLUGIN_PATHS: str -> [str]
281:     if isinstance(settings.get('PLUGIN_PATHS'), six.string_types):
282:         logger.warning("Defining PLUGIN_PATHS setting as string "
283:                        "has been deprecated (should be a list)")
284:         settings['PLUGIN_PATHS'] = [settings['PLUGIN_PATHS']]
285:
286:     # JINJA_EXTENSIONS -> JINJA_ENVIRONMENT > extensions
287:     if 'JINJA_EXTENSIONS' in settings:
288:         logger.warning('JINJA_EXTENSIONS setting has been deprecated, '
289:                        'moving it to JINJA_ENVIRONMENT setting.')
290:         settings['JINJA_ENVIRONMENT']['extensions'] = \
291:             settings['JINJA_EXTENSIONS']
292:         del settings['JINJA_EXTENSIONS']
293:
294:     # {ARTICLE,PAGE}_DIR -> {ARTICLE,PAGE}_PATHS
295:     for key in ['ARTICLE', 'PAGE']:
296:         old_key = key + '_DIR'
297:         new_key = key + '_PATHS'
298:         if old_key in settings:
299:             logger.warning(
300:                 'Deprecated setting %s, moving it to %s list',
301:                 old_key, new_key)
302:             settings[new_key] = [settings[old_key]]   # also make a list
303:             del settings[old_key]
304:
305:     # EXTRA_TEMPLATES_PATHS -> THEME_TEMPLATES_OVERRIDES
```

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

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

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

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

```
550:         if key in settings and isinstance(settings[key], six.string_types):
551:             settings[key] = [settings[key]]
552:
553:     # check settings that must be a particular type
554:     for key, types in [
555:             ('OUTPUT_SOURCES_EXTENSION', six.string_types),
556:             ('FILENAME_METADATA', six.string_types),
557:     ]:
558:         if key in settings and not isinstance(settings[key], types):
559:             value = settings.pop(key)
560:             logger.warn(
561:                 'Detected misconfigured %s (%s), '
562:                 'falling back to the default (%s)',
563:                 key, value, DEFAULT_CONFIG[key])
564:
565:     # try to set the different locales, fallback on the default.
566:     locales = settings.get('LOCALE', DEFAULT_CONFIG['LOCALE'])
567:
568:     for locale_ in locales:
569:         try:
570:             locale.setlocale(locale.LC_ALL, str(locale_))
571:             break  # break if it is successful
572:         except locale.Error:
573:             pass
574:     else:
575:         logger.warning(
576:             "Locale could not be set. Check the LOCALE setting, ensuring it "
577:             "is valid and available on your system.")
578:
579:     if ('SITEURL' in settings):
580:         # If SITEURL has a trailing slash, remove it and provide a warning
581:         siteurl = settings['SITEURL']
582:         if (siteurl.endswith('/')):
583:             settings['SITEURL'] = siteurl[:-1]
584:             logger.warning("Removed extraneous trailing slash from SITEURL.")
585:         # If SITEURL is defined but FEED_DOMAIN isn't,
586:         # set FEED_DOMAIN to SITEURL
587:         if 'FEED_DOMAIN' not in settings:
588:             settings['FEED_DOMAIN'] = settings['SITEURL']
589:
590:     # check content caching layer and warn of incompatibilities
591:     if settings.get('CACHE_CONTENT', False) and \
592:             settings.get('CONTENT_CACHING_LAYER', '') == 'generator' and \
593:             settings.get('WITH_FUTURE_DATES', False):
594:         logger.warning(
595:             "WITH_FUTURE_DATES conflicts with CONTENT_CACHING_LAYER "
596:             "set to 'generator', use 'reader' layer instead")
597:
598:     # Warn if feeds are generated with both SITEURL & FEED_DOMAIN undefined
599:     feed_keys = [
600:         'FEED_ATOM', 'FEED_RSS',
601:         'FEED_ALL_ATOM', 'FEED_ALL_RSS',
602:         'CATEGORY_FEED_ATOM', 'CATEGORY_FEED_RSS',
603:         'AUTHOR_FEED_ATOM', 'AUTHOR_FEED_RSS',
604:         'TAG_FEED_ATOM', 'TAG_FEED_RSS',
605:         'TRANSLATION_FEED_ATOM', 'TRANSLATION_FEED_RSS',
606:     ]
607:
608:     if any(settings.get(k) for k in feed_keys):
609:         if not settings.get('SITEURL'):
610:             logger.warning('Feeds generated without SITEURL set properly may'
```

```
611:                             ' not be valid')
612:
613:     if 'TIMEZONE' not in settings:
614:         logger.warning(
615:             'No timezone information specified in the settings. Assuming'
616:             ' your timezone is UTC for feed generation. Check '
617:             'http://docs.getpelican.com/en/latest/settings.html#timezone '
618:             'for more information')
619:
620:     # fix up pagination rules
621:     from pelican.paginator import PaginationRule
622:     pagination_rules = [
623:         PaginationRule(*r) for r in settings.get(
624:             'PAGINATION_PATTERNS',
625:             DEFAULT_CONFIG['PAGINATION_PATTERNS'],
626:         )
627:     ]
628:     settings['PAGINATION_PATTERNS'] = sorted(
629:         pagination_rules,
630:         key=lambda r: r[0],
631:     )
632:
633:     # Save people from accidentally setting a string rather than a list
634:     path_keys = (
635:         'ARTICLE_EXCLUDES',
636:         'DEFAULT_METADATA',
637:         'DIRECT_TEMPLATES',
638:         'THEME_TEMPLATES_OVERRIDES',
639:         'FILES_TO_COPY',
640:         'IGNORE_FILES',
641:         'PAGINATED_DIRECT_TEMPLATES',
642:         'PLUGINS',
643:         'STATIC_EXCLUDES',
644:         'STATIC_PATHS',
645:         'THEME_STATIC_PATHS',
646:         'ARTICLE_PATHS',
647:         'PAGE_PATHS',
648:     )
649:     for PATH_KEY in filter(lambda k: k in settings, path_keys):
650:         if isinstance(settings[PATH_KEY], six.string_types):
651:             logger.warning("Detected misconfiguration with %s setting "
652:                            "(must be a list), falling back to the default",
653:                            PATH_KEY)
654:             settings[PATH_KEY] = DEFAULT_CONFIG[PATH_KEY]
655:
656:     # Add {PAGE,ARTICLE}_PATHS to {ARTICLE,PAGE}_EXCLUDES
657:     mutually_exclusive = ('ARTICLE', 'PAGE')
658:     for type_1, type_2 in [mutually_exclusive, mutually_exclusive[::-1]]:
659:         try:
660:             includes = settings[type_1 + '_PATHS']
661:             excludes = settings[type_2 + '_EXCLUDES']
662:             for path in includes:
663:                 if path not in excludes:
664:                     excludes.append(path)
665:         except KeyError:
666:             continue            # setting not specified, nothing to do
667:
668:     return settings
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import copy
import locale
import logging
import os
import re
import sys

import pytz

import six
from six.moves.urllib.parse import urljoin, urlparse, urlunparse

from pelican import signals
from pelican.settings import DEFAULT_CONFIG
from pelican.utils import (SafeDatetime, deprecated_attribute, memoized,
                           path_to_url, posixize_path,
                           python_2_unicode_compatible, sanitised_join,
                           set_date_tzinfo, slugify, strftime,
                           truncate_html_words)

# Import these so that they're avalaible when you import from pelican.contents.
from pelican.urlwrappers import (Author, Category, Tag, URLWrapper)  # NOQA

logger = logging.getLogger(__name__)


@python_2_unicode_compatible
class Content(object):
    """Represents a content.

    :param content: the string to parse, containing the original content.
    :param metadata: the metadata associated to this page (optional).
    :param settings: the settings dictionary (optional).
    :param source_path: The location of the source of this content (if any).
    :param context: The shared context between generators.

    """
    @deprecated_attribute(old='filename', new='source_path', since=(3, 2, 0))
    def filename():
        return None

    def __init__(self, content, metadata=None, settings=None,
                 source_path=None, context=None):
        if metadata is None:
            metadata = {}
        if settings is None:
            settings = copy.deepcopy(DEFAULT_CONFIG)

        self.settings = settings
        self._content = content
        if context is None:
            context = {}
        self._context = context
        self.translations = []

        local_metadata = dict()
        local_metadata.update(metadata)

```

```
        # set metadata as attributes
        for key, value in local_metadata.items():
            if key in ('save_as', 'url'):
                key = 'override_' + key
            setattr(self, key.lower(), value)

        # also keep track of the metadata attributes available
        self.metadata = local_metadata

        # default template if it's not defined in page
        self.template = self._get_template()

        # First, read the authors from "authors", if not, fallback to "author"
        # and if not use the settings defined one, if any.
        if not hasattr(self, 'author'):
            if hasattr(self, 'authors'):
                self.author = self.authors[0]
            elif 'AUTHOR' in settings:
                self.author = Author(settings['AUTHOR'], settings)

        if not hasattr(self, 'authors') and hasattr(self, 'author'):
            self.authors = [self.author]

        # XXX Split all the following code into pieces, there is too much here.

        # manage languages
        self.in_default_lang = True
        if 'DEFAULT_LANG' in settings:
            default_lang = settings['DEFAULT_LANG'].lower()
            if not hasattr(self, 'lang'):
                self.lang = default_lang

            self.in_default_lang = (self.lang == default_lang)

        # create the slug if not existing, generate slug according to
        # setting of SLUG_ATTRIBUTE
        if not hasattr(self, 'slug'):
            if (settings['SLUGIFY_SOURCE'] == 'title' and
                    hasattr(self, 'title')):
                self.slug = slugify(
                    self.title,
                    regex_subs=settings.get('SLUG_REGEX_SUBSTITUTIONS', []))
            elif (settings['SLUGIFY_SOURCE'] == 'basename' and
                    source_path is not None):
                basename = os.path.basename(
                    os.path.splitext(source_path)[0])
                self.slug = slugify(
                    basename,
                    regex_subs=settings.get('SLUG_REGEX_SUBSTITUTIONS', []))

        self.source_path = source_path
        self.relative_source_path = self.get_relative_source_path()

        # manage the date format
        if not hasattr(self, 'date_format'):
            if hasattr(self, 'lang') and self.lang in settings['DATE_FORMATS']:
                self.date_format = settings['DATE_FORMATS'][self.lang]
            else:
                self.date_format = settings['DEFAULT_DATE_FORMAT']

        if isinstance(self.date_format, tuple):
```

```
123:             locale_string = self.date_format[0]
124:             if sys.version_info < (3, ) and isinstance(locale_string,
125:                                                        six.text_type):
126:                 locale_string = locale_string.encode('ascii')
127:             locale.setlocale(locale.LC_ALL, locale_string)
128:             self.date_format = self.date_format[1]
129: 
130:         # manage timezone
131:         default_timezone = settings.get('TIMEZONE', 'UTC')
132:         timezone = getattr(self, 'timezone', default_timezone)
133: 
134:         if hasattr(self, 'date'):
135:             self.date = set_date_tzinfo(self.date, timezone)
136:             self.locale_date = strftime(self.date, self.date_format)
137: 
138:         if hasattr(self, 'modified'):
139:             self.modified = set_date_tzinfo(self.modified, timezone)
140:             self.locale_modified = strftime(self.modified, self.date_format)
141: 
142:         # manage status
143:         if not hasattr(self, 'status'):
144:             # Previous default of None broke comment plugins and perhaps others
145:             self.status = getattr(self, 'default_status', '')
146: 
147:         # store the summary metadata if it is set
148:         if 'summary' in metadata:
149:             self._summary = metadata['summary']
150: 
151:         signals.content_object_init.send(self)
152: 
153:     def __str__(self):
154:         return self.source_path or repr(self)
155: 
156:     def _has_valid_mandatory_properties(self):
157:         """Test mandatory properties are set."""
158:         for prop in self.mandatory_properties:
159:             if not hasattr(self, prop):
160:                 logger.error(
161:                     "Skipping %s: could not find information about '%s'",
162:                     self, prop)
163:                 return False
164:         return True
165: 
166:     def _has_valid_save_as(self):
167:         """Return true if save_as doesn't write outside output path, false
168:         otherwise."""
169:         try:
170:             output_path = self.settings["OUTPUT_PATH"]
171:         except KeyError:
172:             # we cannot check
173:             return True
174: 
175:         try:
176:             sanitised_join(output_path, self.save_as)
177:         except RuntimeError:  # outside output_dir
178:             logger.error(
179:                 "Skipping %s: file %r would be written outside output path",
180:                 self,
181:                 self.save_as,
182:             )
183:             return False
```

```
184:
185:         return True
186:
187:     def _has_valid_status(self):
188:         if hasattr(self, 'allowed_statuses'):
189:             if self.status not in self.allowed_statuses:
190:                 logger.error(
191:                     "Unknown status '%s' for file %s, skipping it.",
192:                     self.status,
193:                     self
194:                 )
195:                 return False
196:
197:         # if undefined we allow all
198:         return True
199:
200:     def is_valid(self):
201:         """Validate Content"""
202:         # Use all() to not short circuit and get results of all validations
203:         return all([self._has_valid_mandatory_properties(),
204:                     self._has_valid_save_as(),
205:                     self._has_valid_status()])
206:
207:     @property
208:     def url_format(self):
209:         """Returns the URL, formatted with the proper values"""
210:         metadata = copy.copy(self.metadata)
211:         path = self.metadata.get('path', self.get_relative_source_path())
212:         metadata.update({
213:             'path': path_to_url(path),
214:             'slug': getattr(self, 'slug', ''),
215:             'lang': getattr(self, 'lang', 'en'),
216:             'date': getattr(self, 'date', SafeDatetime.now()),
217:             'author': self.author.slug if hasattr(self, 'author') else '',
218:             'category': self.category.slug if hasattr(self, 'category') else ''
219:         })
220:         return metadata
221:
222:     def _expand_settings(self, key, klass=None):
223:         if not klass:
224:             klass = self.__class__.__name__
225:         fq_key = ('%s_%s' % (klass, key)).upper()
226:         return self.settings[fq_key].format(**self.url_format)
227:
228:     def get_url_setting(self, key):
229:         if hasattr(self, 'override_' + key):
230:             return getattr(self, 'override_' + key)
231:         key = key if self.in_default_lang else 'lang_%s' % key
232:         return self._expand_settings(key)
233:
234:     def _link_replacer(self, siteurl, m):
235:         what = m.group('what')
236:         value = urlparse(m.group('value'))
237:         path = value.path
238:         origin = m.group('path')
239:
240:         # urllib.parse.urljoin() produces `a.html` for urljoin("..", "a.html")
241:         # so if RELATIVE_URLS are enabled, we fall back to os.path.join() to
242:         # properly get `../a.html`. However, os.path.join() produces
243:         # `baz/http://foo/bar.html` for join("baz", "http://foo/bar.html")
244:         # instead of correct "http://foo/bar.html", so one has to pick a side
```

```
245:         # as there is no silver bullet.
246:         if self.settings['RELATIVE_URLS']:
247:             joiner = os.path.join
248:         else:
249:             joiner = urljoin
250:
251:             # However, it's not *that* simple: urljoin("blog", "index.html")
252:             # produces just `index.html` instead of `blog/index.html` (unlike
253:             # os.path.join()), so in order to get a correct answer one needs to
254:             # append a trailing slash to siteurl in that case. This also makes
255:             # the new behavior fully compatible with Pelican 3.7.1.
256:             if not siteurl.endswith('/'):
257:                 siteurl += '/'
258:
259:         # XXX Put this in a different location.
260:         if what in {'filename', 'static', 'attach'}:
261:             if path.startswith('/'):
262:                 path = path[1:]
263:             else:
264:                 # relative to the source path of this content
265:                 path = self.get_relative_source_path(
266:                     os.path.join(self.relative_dir, path)
267:                 )
268:
269:             key = 'static_content' if what in ('static', 'attach')\
270:                 else 'generated_content'
271:
272:             def _get_linked_content(key, path):
273:                 try:
274:                     return self._context[key][path]
275:                 except KeyError:
276:                     try:
277:                         # Markdown escapes spaces, try unescaping
278:                         return self._context[key][path.replace('%20', ' ')]
279:                     except KeyError:
280:                         if what == 'filename' and key == 'generated_content':
281:                             key = 'static_content'
282:                             linked_content = _get_linked_content(key, path)
283:                             if linked_content:
284:                                 logger.warning(
285:                                     '{filename} used for linking to static'
286:                                     ' content %s in %s. Use {static} instead',
287:                                     path,
288:                                     self.get_relative_source_path())
289:                                 return linked_content
290:                         return None
291:
292:             linked_content = _get_linked_content(key, path)
293:             if linked_content:
294:                 if what == 'attach':
295:                     linked_content.attach_to(self)
296:                 origin = joiner(siteurl, linked_content.url)
297:                 origin = origin.replace('\\', '/')  # for Windows paths.
298:             else:
299:                 logger.warning(
300:                     "Unable to find '%s', skipping url replacement.",
301:                     value.geturl(), extra={
302:                         'limit_msg': ("Other resources were not found "
303:                                       "and their urls not replaced")})
304:         elif what == 'category':
305:             origin = joiner(siteurl, Category(path, self.settings).url)
```

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

```
367:                     os.path.join(self.relative_dir, path)
368:                 )
369:             path = path.replace('%20', ' ')
370:             static_links.add(path)
371:         return static_links
372:
373:     def get_siteurl(self):
374:         return self._context.get('localsiteurl', '')
375:
376:     @memoized
377:     def get_content(self, siteurl):
378:         if hasattr(self, '_get_content'):
379:             content = self._get_content()
380:         else:
381:             content = self._content
382:         return self._update_content(content, siteurl)
383:
384:     @property
385:     def content(self):
386:         return self.get_content(self.get_siteurl())
387:
388:     @memoized
389:     def get_summary(self, siteurl):
390:         """Returns the summary of an article.
391:
392:         This is based on the summary metadata if set, otherwise truncate the
393:         content.
394:         """
395:         if 'summary' in self.metadata:
396:             return self.metadata['summary']
397:
398:         if self.settings['SUMMARY_MAX_LENGTH'] is None:
399:             return self.content
400:
401:         return truncate_html_words(self.content,
402:                                    self.settings['SUMMARY_MAX_LENGTH'])
403:
404:     @property
405:     def summary(self):
406:         return self.get_summary(self.get_siteurl())
407:
408:     def _get_summary(self):
409:         """deprecated function to access summary"""
410:
411:         logger.warning('_get_summary() has been deprecated since 3.6.4. '
412:                        'Use the summary decorator instead')
413:         return self.summary
414:
415:     @summary.setter
416:     def summary(self, value):
417:         """Dummy function"""
418:         pass
419:
420:     @property
421:     def status(self):
422:         return self._status
423:
424:     @status.setter
425:     def status(self, value):
426:         # TODO maybe typecheck
427:         self._status = value.lower()
```

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

```
489:             self.metadata['summary'] = self._summary
490:
491:
492: class Page(Content):
493:     mandatory_properties = ('title',)
494:     allowed_statuses = ('published', 'hidden', 'draft')
495:     default_status = 'published'
496:     default_template = 'page'
497:
498:     def _expand_settings(self, key):
499:         klass = 'draft_page' if self.status == 'draft' else None
500:         return super(Page, self)._expand_settings(key, klass)
501:
502:
503: class Article(Content):
504:     mandatory_properties = ('title', 'date', 'category')
505:     allowed_statuses = ('published', 'draft')
506:     default_status = 'published'
507:     default_template = 'article'
508:
509:     def __init__(self, *args, **kwargs):
510:         super(Article, self).__init__(*args, **kwargs)
511:
512:         # handle WITH_FUTURE_DATES (designate article to draft based on date)
513:         if not self.settings['WITH_FUTURE_DATES'] and hasattr(self, 'date'):
514:             if self.date.tzinfo is None:
515:                 now = SafeDatetime.now()
516:             else:
517:                 now = SafeDatetime.utcnow().replace(tzinfo=pytz.utc)
518:             if self.date > now:
519:                 self.status = 'draft'
520:
521:         # if we are a draft and there is no date provided, set max datetime
522:         if not hasattr(self, 'date') and self.status == 'draft':
523:             self.date = SafeDatetime.max
524:
525:     def _expand_settings(self, key):
526:         klass = 'draft' if self.status == 'draft' else 'article'
527:         return super(Article, self)._expand_settings(key, klass)
528:
529:
530: @python_2_unicode_compatible
531: class Static(Content):
532:     mandatory_properties = ('title',)
533:     default_status = 'published'
534:     default_template = None
535:
536:     def __init__(self, *args, **kwargs):
537:         super(Static, self).__init__(*args, **kwargs)
538:         self._output_location_referenced = False
539:
540:     @deprecated_attribute(old='filepath', new='source_path', since=(3, 2, 0))
541:     def filepath():
542:         return None
543:
544:     @deprecated_attribute(old='src', new='source_path', since=(3, 2, 0))
545:     def src():
546:         return None
547:
548:     @deprecated_attribute(old='dst', new='save_as', since=(3, 2, 0))
549:     def dst():
```

```
550:         return None
551:
552:     @property
553:     def url(self):
554:         # Note when url has been referenced, so we can avoid overriding it.
555:         self._output_location_referenced = True
556:         return super(Static, self).url
557:
558:     @property
559:     def save_as(self):
560:         # Note when save_as has been referenced, so we can avoid overriding it.
561:         self._output_location_referenced = True
562:         return super(Static, self).save_as
563:
564:     def attach_to(self, content):
565:         """Override our output directory with that of the given content object.
566:         """
567:
568:         # Determine our file's new output path relative to the linking
569:         # document. If it currently lives beneath the linking
570:         # document's source directory, preserve that relationship on output.
571:         # Otherwise, make it a sibling.
572:
573:         linking_source_dir = os.path.dirname(content.source_path)
574:         tail_path = os.path.relpath(self.source_path, linking_source_dir)
575:         if tail_path.startswith(os.pardir + os.sep):
576:             tail_path = os.path.basename(tail_path)
577:         new_save_as = os.path.join(
578:             os.path.dirname(content.save_as), tail_path)
579:
580:         # We do not build our new url by joining tail_path with the linking
581:         # document's url, because we cannot know just by looking at the latter
582:         # whether it points to the document itself or to its parent directory.
583:         # (An url like 'some/content' might mean a directory named 'some'
584:         # with a file named 'content', or it might mean a directory named
585:         # 'some/content' with a file named 'index.html'.) Rather than trying
586:         # to figure it out by comparing the linking document's url and save_as
587:         # path, we simply build our new url from our new save_as path.
588:
589:         new_url = path_to_url(new_save_as)
590:
591:         def _log_reason(reason):
592:             logger.warning(
593:                 "The {attach} link in %s cannot relocate "
594:                 "%s because %s. Falling back to "
595:                 "{filename} link behavior instead.",
596:                 content.get_relative_source_path(),
597:                 self.get_relative_source_path(), reason,
598:                 extra={'limit_msg': "More {attach} warnings silenced."})
599:
600:         # We never override an override, because we don't want to interfere
601:         # with user-defined overrides that might be in EXTRA_PATH_METADATA.
602:         if hasattr(self, 'override_save_as') or hasattr(self, 'override_url'):
603:             if new_save_as != self.save_as or new_url != self.url:
604:                 _log_reason("its output location was already overridden")
605:             return
606:
607:         # We never change an output path that has already been referenced,
608:         # because we don't want to break links that depend on that path.
609:         if self._output_location_referenced:
610:             if new_save_as != self.save_as or new_url != self.url:
```

```
611:                 _log_reason("another link already referenced its location")
612:             return
613:
614:         self.override_save_as = new_save_as
615:         self.override_url = new_url
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals, with_statement

import logging
import os

from feedgenerator import Atom1Feed, Rss201rev2Feed, get_tag_uri

from jinja2 import Markup

import six
from six.moves.urllib.parse import urljoin

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

if not six.PY3:
    from codecs import open

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)

        else:
            # Atom feeds have two different tags for full content (called
            # 'content' by feedgenerator) and summary (called 'description' by
            # feedgenerator).
            #
            # It does not make sense to have the summary be the
            # exact same thing as the full content. If we detect that
            # they are we just remove the summary.
            content = item.get_content(self.site_url)
            description = item.summary
            if description == content:
                description = None

        categories = list()
        if hasattr(item, 'category'):
            categories.append(item.category)
        if hasattr(item, 'tags'):
            categories.extend(item.tags)

        feed.add_item(
            title=title,
            link=link,
            unique_id=get_tag_uri(link, item.date),
            description=description,
            content=content,
            categories=categories if categories else None,
            author_name=getattr(item, 'author', ''),
            pubdate=set_date_tzinfo(
                item.date, self.settings.get('TIMEZONE', None)),
            updateddate=set_date_tzinfo(
                item.modified, self.settings.get('TIMEZONE', None)
                ) if hasattr(item, 'modified') else None)

    def _open_w(self, filename, encoding, override=False):
        """Open a file to write some content to it.

        Exit if we have already written to that file, unless one (and no more
        than one) of the writes has the override parameter set to True.
        """
        if filename in self._overridden_files:
            if override:
                raise RuntimeError('File %s is set to be overridden twice'
                                   % filename)
            else:
                logger.info('Skipping %s', filename)
                filename = os.devnull
        elif filename in self._written_files:
            if override:
                logger.info('Overwriting %s', filename)
            else:
                raise RuntimeError('File %s is to be overwritten' % filename)
        if override:
            self._overridden_files.add(filename)
        self._written_files.add(filename)
        return open(filename, 'w', encoding=encoding)
```

```
123:
124:     def write_feed(self, elements, context, path=None, url=None,
125:                    feed_type='atom', override_output=False, feed_title=None):
126:         """Generate a feed with the list of articles provided
127:
128:         Return the feed. If no path or output_path is specified, just
129:         return the feed object.
130:
131:         :param elements: the articles to put on the feed.
132:         :param context: the context to get the feed metadata.
133:         :param path: the path to output.
134:         :param url: the publicly visible feed URL; if None, path is used
135:             instead
136:         :param feed_type: the feed type to use (atom or rss)
137:         :param override_output: boolean telling if we can override previous
138:             output with the same name (and if next files written with the same
139:             name should be skipped to keep that one)
140:         :param feed_title: the title of the feed.o
141:         """
142:         if not is_selected_for_writing(self.settings, path):
143:             return
144:
145:         self.site_url = context.get(
146:             'SITEURL', path_to_url(get_relative_path(path)))
147:
148:         self.feed_domain = context.get('FEED_DOMAIN')
149:         self.feed_url = self.urljoiner(self.feed_domain, url if url else path)
150:
151:         feed = self._create_new_feed(feed_type, feed_title, context)
152:
153:         max_items = len(elements)
154:         if self.settings['FEED_MAX_ITEMS']:
155:             max_items = min(self.settings['FEED_MAX_ITEMS'], max_items)
156:         for i in range(max_items):
157:             self._add_item_to_the_feed(feed, elements[i])
158:
159:         signals.feed_generated.send(context, feed=feed)
160:         if path:
161:             complete_path = sanitised_join(self.output_path, path)
162:
163:             try:
164:                 os.makedirs(os.path.dirname(complete_path))
165:             except Exception:
166:                 pass
167:
168:             encoding = 'utf-8' if six.PY3 else None
169:             with self._open_w(complete_path, encoding, override_output) as fp:
170:                 feed.write(fp, 'utf-8')
171:                 logger.info('Writing %s', complete_path)
172:
173:             signals.feed_written.send(
174:                 complete_path, context=context, feed=feed)
175:         return feed
176:
177:     def write_file(self, name, template, context, relative_urls=False,
178:                    paginated=None, template_name=None, override_output=False,
179:                    url=None, **kwargs):
180:         """Render the template and write the file.
181:
182:         :param name: name of the file to output
183:         :param template: template to use to generate the content
```

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

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

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import locale
import logging
import os
import sys
from collections import defaultdict
try:
    from collections.abc import Mapping
except ImportError:
    from collections import Mapping

import six

__all__ = [
    'init'
]


class BaseFormatter(logging.Formatter):
    def __init__(self, fmt=None, datefmt=None):
        FORMAT = '%(customlevelname)s %(message)s'
        super(BaseFormatter, self).__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, six.string_types) else
                            arg for arg in record.args)
        return super(BaseFormatter, self).format(record)

    def formatException(self, ei):
        ''' prefix traceback info for better representation '''
        # .formatException returns a bytestring in py2 and unicode in py3
        # since .format will handle unicode conversion,
        # str() calls are used to normalize formatting string
        s = super(BaseFormatter, self).formatException(ei)
        # fancy format traceback
        s = str('\n').join(str('  | ') + line for line in s.splitlines())
        # separate the traceback from the preceding lines
        s = str('  |___\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'}
```

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

```
123:
124:         # ignore LOG_FILTER records by templates when "debug" isn't enabled
125:         logger_level = logging.getLogger().getEffectiveLevel()
126:         if logger_level > logging.DEBUG:
127:             ignore_key = (record.levelno, record.msg)
128:             if ignore_key in self._ignore:
129:                 return False
130:
131:         # check if we went over threshold
132:         if group:
133:             key = (record.levelno, group)
134:             self._group_count[key] += 1
135:             if self._group_count[key] == self._threshold:
136:                 record.msg = group
137:                 record.args = group_args
138:             elif self._group_count[key] > self._threshold:
139:                 return False
140:         return True
141:
142:
143: class SafeLogger(logging.Logger):
144:     """
145:     Base Logger which properly encodes Exceptions in Py2
146:     """
147:     _exc_encoding = locale.getpreferredencoding()
148:
149:     def _log(self, level, msg, args, exc_info=None, extra=None):
150:         # if the only argument is a Mapping, Logger uses that for formatting
151:         # format values for that case
152:         if args and len(args) == 1 and isinstance(args[0], Mapping):
153:             args = ({k: self._decode_arg(v) for k, v in args[0].items()},)
154:         # otherwise, format each arg
155:         else:
156:             args = tuple(self._decode_arg(arg) for arg in args)
157:         super(SafeLogger, self)._log(
158:             level, msg, args, exc_info=exc_info, extra=extra)
159:
160:     def _decode_arg(self, arg):
161:         '''
162:         properly decode an arg for Py2 if it's Exception
163:
164:
165:         localized systems have errors in native language if locale is set
166:         so convert the message to unicode with the correct encoding
167:         '''
168:         if isinstance(arg, Exception):
169:             text = str('%s: %s') % (arg.__class__.__name__, arg)
170:             if six.PY2:
171:                 text = text.decode(self._exc_encoding)
172:             return text
173:         else:
174:             return arg
175:
176:
177: class LimitLogger(SafeLogger):
178:     """
179:     A logger which adds LimitFilter automatically
180:     """
181:
182:     limit_filter = LimitFilter()
183:
```

```
184:     def __init__(self, *args, **kwargs):
185:         super(LimitLogger, self).__init__(*args, **kwargs)
186:         self.enable_filter()
187:
188:     def disable_filter(self):
189:         self.removeFilter(LimitLogger.limit_filter)
190:
191:     def enable_filter(self):
192:         self.addFilter(LimitLogger.limit_filter)
193:
194:
195: class FatalLogger(LimitLogger):
196:     warnings_fatal = False
197:     errors_fatal = False
198:
199:     def warning(self, *args, **kwargs):
200:         super(FatalLogger, self).warning(*args, **kwargs)
201:         if FatalLogger.warnings_fatal:
202:             raise RuntimeError('Warning encountered')
203:
204:     def error(self, *args, **kwargs):
205:         super(FatalLogger, self).error(*args, **kwargs)
206:         if FatalLogger.errors_fatal:
207:             raise RuntimeError('Error encountered')
208:
209:
210: logging.setLoggerClass(FatalLogger)
211:
212:
213: def supports_color():
214:     """
215:     Returns True if the running system's terminal supports color,
216:     and False otherwise.
217:
218:     from django.core.management.color
219:     """
220:     plat = sys.platform
221:     supported_platform = plat != 'Pocket PC' and \
222:         (plat != 'win32' or 'ANSICON' in os.environ)
223:
224:     # isatty is not always implemented, #6223.
225:     is_a_tty = hasattr(sys.stdout, 'isatty') and sys.stdout.isatty()
226:     if not supported_platform or not is_a_tty:
227:         return False
228:     return True
229:
230:
231: def get_formatter():
232:     if supports_color():
233:         return ANSIFormatter()
234:     else:
235:         return TextFormatter()
236:
237:
238: def init(level=None, fatal='', handler=logging.StreamHandler(), name=None,
239:          logs_dedup_min_level=None):
240:     FatalLogger.warnings_fatal = fatal.startswith('warning')
241:     FatalLogger.errors_fatal = bool(fatal)
242:
243:     logger = logging.getLogger(name)
244:
```

```
245:     handler.setFormatter(get_formatter())
246:     logger.addHandler(handler)
247:
248:     if level:
249:         logger.setLevel(level)
250:     if logs_dedup_min_level:
251:         LimitFilter.LOGS_DEDUP_MIN_LEVEL = logs_dedup_min_level
252:
253:
254: def log_warnings():
255:     import warnings
256:     logging.captureWarnings(True)
257:     warnings.simplefilter("default", DeprecationWarning)
258:     init(logging.DEBUG, name='py.warnings')
259:
260:
261: if __name__ == '__main__':
262:     init(level=logging.DEBUG)
263:
264:     root_logger = logging.getLogger()
265:     root_logger.debug('debug')
266:     root_logger.info('info')
267:     root_logger.warning('warning')
268:     root_logger.error('error')
269:     root_logger.critical('critical')
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

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

import six

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.
```

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

```
123:
124:         # find the last matching pagination rule
125:         for p in self.settings['PAGINATION_PATTERNS']:
126:             if p.min_page <= self.number:
127:                 rule = p
128:
129:         if not rule:
130:             return ''
131:
132:         prop_value = getattr(rule, key)
133:
134:         if not isinstance(prop_value, six.string_types):
135:             logger.warning('%s is set to %s', key, prop_value)
136:             return prop_value
137:
138:         # URL or SAVE_AS is a string, format it with a controlled context
139:         context = {
140:             'save_as': self.full_name,
141:             'url': self.base_url,
142:             'name': self.name,
143:             'base_name': self.base_name,
144:             'extension': self.extension,
145:             'number': self.number,
146:         }
147:
148:         ret = prop_value.format(**context)
149:         # Remove a single leading slash, if any. This is done for backwards
150:         # compatibility reasons. If a leading slash is needed (for URLs
151:         # relative to server root or absolute URLs without the scheme such as
152:         # //blog.my.site/), it can be worked around by prefixing the pagination
153:         # pattern by an additional slash (which then gets removed, preserving
154:         # the other slashes). This also means the following code *can't* be
155:         # changed to lstrip() because that would remove all leading slashes and
156:         # thus make the workaround impossible. See
157:         # test_custom_pagination_pattern() for a verification of this.
158:         if ret[0] == '/':
159:             ret = ret[1:]
160:         return ret
161:
162:     url = property(functools.partial(_from_settings, key='URL'))
163:     save_as = property(functools.partial(_from_settings, key='SAVE_AS'))
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

import argparse
import logging
import os
import posixpath
import ssl
import sys

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

from six.moves import BaseHTTPServer
from six.moves import SimpleHTTPServer as srvmod
from six.moves import urllib

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(srvmod.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:
```

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

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

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

import hashlib
import logging
import os

from six.moves import cPickle as 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)

    def save_cache(self):
        """Save the updated cache"""
        if self._cache_data_policy:
            try:
                mkdir_p(self.settings['CACHE_PATH'])
                with self._cache_open(self._cache_path, 'wb') as fhandle:
                    pickle.dump(self._cache, fhandle)
            except (IOError, OSError, pickle.PicklingError) as err:
                logger.warning('Could not save cache %s\n ... %s',
                               self._cache_path, err)


class FileStampDataCacher(FileDataCacher):
    """Subclass that also caches the stamp of the file"""

    def __init__(self, settings, cache_name, caching_policy, load_policy):
        """This sublcass additionally sets filestamp function
        and base path for filestamping operations
        """

        super(FileStampDataCacher, self).__init__(settings, cache_name,
                                                  caching_policy,
                                                  load_policy)

        method = self.settings['CHECK_MODIFIED_METHOD']
        if method == 'mtime':
            self._filestamp_func = os.path.getmtime
        else:
            try:
                hash_func = getattr(hashlib, method)

                def filestamp_func(filename):
                    """return hash of file contents"""
                    with open(filename, 'rb') as fhandle:
                        return hash_func(fhandle.read()).digest()

                self._filestamp_func = filestamp_func
            except AttributeError as err:
                logger.warning('Could not get hashing function\n\t%s', err)
                self._filestamp_func = None

    def cache_data(self, filename, data):
        """Cache stamp and data for the given file"""
        stamp = self._get_file_stamp(filename)
        super(FileStampDataCacher, self).cache_data(filename, (stamp, data))

    def _get_file_stamp(self, filename):
        """Check if the given file has been modified
        since the previous build.

        depending on CHECK_MODIFIED_METHOD
        a float may be returned for 'mtime',
        a hash for a function name in the hashlib module
        or an empty bytes string otherwise
        """

        try:
            return self._filestamp_func(filename)
```

```
123:             except (IOError, OSError, TypeError) as err:
124:                 logger.warning('Cannot get modification stamp for %s\n\t%s',
125:                                filename, err)
126:                 return ''
127:
128:         def get_cached_data(self, filename, default=None):
129:             """Get the cached data for the given filename
130:             if the file has not been modified.
131:
132:             If no record exists or file has been modified, return default.
133:             Modification is checked by comparing the cached
134:             and current file stamp.
135:             """
136:
137:             stamp, data = super(FileStampDataCacher, self).get_cached_data(
138:                 filename, (None, default))
139:             if stamp != self._get_file_stamp(filename):
140:                 return default
141:             return data
```

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

import functools
import logging
import os

import six

from pelican.utils import python_2_unicode_compatible, slugify

logger = logging.getLogger(__name__)


@python_2_unicode_compatible
@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
```

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

```
123: class Category(URLWrapper):
124:     pass
125:
126:
127: class Tag(URLWrapper):
128:     def __init__(self, name, *args, **kwargs):
129:         super(Tag, self).__init__(name.strip(), *args, **kwargs)
130:
131:
132: class Author(URLWrapper):
133:     pass
```

```
# -*- coding: utf-8 -*-
from __future__ import print_function, unicode_literals

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 six

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 six.iteritems(pys.PYGMENTS_RST_OPTIONS):
                # 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:
```

```
62:                 self.options['linenos'] = 'table'
63:
64:         for flag in ('nowrap', 'nobackground', 'anchorlinenos'):
65:             if flag in self.options:
66:                 self.options[flag] = True
67:
68:         # noclasses should already default to False, but just in case...
69:         formatter = HtmlFormatter(noclasses=False, **self.options)
70:         parsed = highlight('\n'.join(self.content), lexer, formatter)
71:         return [nodes.raw('', parsed, format='html')]
72:
73:
74: directives.register_directive('code-block', Pygments)
75: directives.register_directive('sourcecode', Pygments)
76:
77:
78: _abbr_re = re.compile(r'\((.*)\)$', re.DOTALL)
79:
80:
81: class abbreviation(nodes.Inline, nodes.TextElement):
82:     pass
83:
84:
85: def abbr_role(typ, rawtext, text, lineno, inliner, options={}, content=[]):
86:     text = utils.unescape(text)
87:     m = _abbr_re.search(text)
88:     if m is None:
89:         return [abbreviation(text, text)], []
90:     abbr = text[:m.start()].strip()
91:     expl = m.group(1)
92:     return [abbreviation(abbr, abbr, explanation=expl)], []
93:
94:
95: roles.register_local_role('abbr', abbr_role)
```

```
 1: # -*- coding: utf-8 -*-
 2: from __future__ import print_function, unicode_literals
 3:
 4: from blinker import signal
 5:
 6: # Run-level signals:
 7:
 8: initialized = signal('pelican_initialized')
 9: get_generators = signal('get_generators')
10: all_generators_finalized = signal('all_generators_finalized')
11: get_writer = signal('get_writer')
12: finalized = signal('pelican_finalized')
13:
14: # Reader-level signals
15:
16: readers_init = signal('readers_init')
17:
18: # Generator-level signals
19:
20: generator_init = signal('generator_init')
21:
22: article_generator_init = signal('article_generator_init')
23: article_generator_pretaxonomy = signal('article_generator_pretaxonomy')
24: article_generator_finalized = signal('article_generator_finalized')
25: article_generator_write_article = signal('article_generator_write_article')
26: article_writer_finalized = signal('article_writer_finalized')
27:
28: page_generator_init = signal('page_generator_init')
29: page_generator_finalized = signal('page_generator_finalized')
30: page_generator_write_page = signal('page_generator_write_page')
31: page_writer_finalized = signal('page_writer_finalized')
32:
33: static_generator_init = signal('static_generator_init')
34: static_generator_finalized = signal('static_generator_finalized')
35:
36: # Page-level signals
37:
38: article_generator_preread = signal('article_generator_preread')
39: article_generator_context = signal('article_generator_context')
40:
41: page_generator_preread = signal('page_generator_preread')
42: page_generator_context = signal('page_generator_context')
43:
44: static_generator_preread = signal('static_generator_preread')
45: static_generator_context = signal('static_generator_context')
46:
47: content_object_init = signal('content_object_init')
48:
49: # Writers signals
50: content_written = signal('content_written')
51: feed_generated = signal('feed_generated')
52: feed_written = signal('feed_written')
```