Whoosh Documentation

Release 2.7.4

Matt Chaput

Jun 30, 2017

Contents

Whoosh was created by Matt Chaput. You can view outstanding issues on the Whoosh Bitbucket page and get help on the Whoosh mailing list.

CHAPTER 1

Contents

Release notes

Whoosh 2.x release notes

Whoosh 2.7

- Removed on-disk word graph implementation of spell checking in favor of much simpler and faster FSA implementation over the term file.
- Many bug fixes.
- Removed backwards compatibility with indexes created by versions prior to 2.5. You may need to re-index if you are using an old index that hasn't been updated.
- This is the last 2.x release before a major overhaul that will break backwards compatibility.

Whoosh 2.5

- Whoosh 2.5 will read existing indexes, but segments created by 2.5 will not be readable by older versions of Whoosh.
- As a replacement for field caches to speed up sorting, Whoosh now supports adding a `sortable=True` keyword argument to fields. This makes Whoosh store a sortable representation of the field's values in a "column" format (which associates a "key" value with each document). This is more robust, efficient, and customizable than the old behavior. You should now specify `sortable=True` on fields that you plan on using to sort or group search results.

  (You can still sort/group on fields that don't have `sortable=True`, however it will use more RAM and be slower as Whoosh caches the field values in memory.)

  Fields that use `sortable=True` can avoid specifying `stored=True`. The field's value will still be available on `Hit` objects (the value will be retrieved from the column instead of from the stored fields). This may actually be faster for certain types of values.

- Whoosh will now detect common types of OR queries and use optimized read-ahead matchers to speed them up by several times.
- Whoosh now includes pure-Python implementations of the Snowball stemmers and stop word lists for various languages adapted from NLTK. These are available through the *whoosh.analysis.LanguageAnalyzer* analyzer or through the `lang=` keyword argument to the *TEXT* field.
- You can now use the *whoosh.filedb.filestore.Storage.create()* and `whoosh.filedb.filestore.Storage.destory()` methods as a consistent API to set up and tear down different types of storage.
- Many bug fixes and speed improvements.
- Switched unit tests to use `py.test` instead of `nose`.
- Removed obsolete `SpellChecker` class.

Whoosh 2.4

- By default, Whoosh now assembles the individual files of a segment into a single file when committing. This has a small performance penalty but solves a problem where Whoosh can keep too many files open. Whoosh is also now smarter about using mmap.
- Added functionality to index and search hierarchical documents. See *Indexing and searching document hierarchies*.
- Rewrote the Directed Acyclic Word Graph implementation (used in spell checking) to be faster and more space-efficient. Word graph files created by previous versions will be ignored, meaning that spell checking may become slower unless/until you replace the old segments (for example, by optimizing).
- Rewrote multiprocessing indexing to be faster and simpler. You can now do `myindex.writer(procs=n)` to get a multiprocessing writer, or `myindex.writer(procs=n, multisegment=True)` to get a multiprocessing writer that leaves behind multiple segments, like the old MultiSegmentWriter. (`MultiSegmentWriter` is still available as a function that returns the new class.)
- When creating `Term` query objects for special fields (e.g. NUMERIC or BOOLEAN), you can now use the field's literal type instead of a string as the second argument, for example `Term("num", 20)` or `Term("bool", True)`. (This change may cause problems interacting with functions that expect query objects to be pure textual, such as spell checking.)
- All writing to and reading from on-disk indexes is now done through "codec" objects. This architecture should make it easier to add optional or experimental features, and maintain backwards compatibility.
- Fixes issues #75, #137, #206, #213, #215, #219, #223, #226, #230, #233, #238, #239, #240, #241, #243, #244, #245, #252, #253, and other bugs. Thanks to Thomas Waldmann and Alexei Gousev for the help!

Whoosh 2.3.2

- Fixes bug in BM25F scoring function, leading to increased precision in search results.
- Fixes issues #203, #205, #206, #208, #209, #212.

Whoosh 2.3.1

- Fixes issue #200.

Whoosh 2.3

- Added a *whoosh.query.Regex* term query type, similar to *whoosh.query.Wildcard*. The parser does not allow regex term queries by default. You need to add the *whoosh.qparser.RegexPlugin* plugin. After you add the plugin, you can use `r"expression"` query syntax for regular expression term queries. For example, `r"foo.*bar"`.
- Added the `whoosh.qparser.PseudoFieldPlugin` parser plugin. This plugin lets you create "pseudo-fields" that run a transform function on whatever query syntax the user applies the field to. This is fairly advanced functionality right now; I'm trying to think of ways to make its power easier to access.
- The documents in the lists in the dictionary returned by `Results.groups()` by default are now in the same relative order as in the results. This makes it much easier to display the "top N" results in each category, for example.
- The `groupids` keyword argument to `Searcher.search` has been removed. Instead you can now pass a *whoosh.sorting.FacetMap* object to the `Searcher.search` method's `maptype` argument to control how faceted documents are grouped, and/or set the `maptype` argument on individual `whoosh.sorting.FacetType`` objects to set custom grouping per facet. See *Sorting and faceting* for more information.
- Calling `Searcher.documents()` or `Searcher.document_numbers()` with no arguments now yields all documents/numbers.
- Calling `Writer.update_document()` with no unique fields is now equivalent to calling `Writer.add_document()` with the same arguments.
- Fixed a problem with keyword expansion where the code was building a cache that was fast on small indexes, but unacceptably slow on large indexes.
- Added the hyphen (-) to the list of characters that match a "wildcard" token, to make parsing slightly more predictable. A true fix will have to wait for another parser rewrite.
- Fixed an unused `__future__` import and use of `float("nan")` which were breaking under Python 2.5.
- Fixed a bug where vectored fields with only one term stored an empty term vector.
- Various other bug fixes.

Whoosh 2.2

- Fixes several bugs, including a bad bug in BM25F scoring.
- Added `allow_overlap` option to *whoosh.sorting.StoredFieldFacet*.
- In *add_document()*, You can now pass query-like strings for BOOLEAN and DATETIME fields (e.g `boolfield="true"` and `dtfield="20101131-16:01"`) as an alternative to actual `bool` or `datetime` objects. The implementation of this is incomplete: it only works in the default `filedb` backend, and if the field is stored, the stored value will be the string, not the parsed object.
- Added *whoosh.analysis.CompoundWordFilter* and *whoosh.analysis.TeeFilter*.

Whoosh 2.1

This release fixes several bugs, and contains speed improvments to highlighting. See *How to create highlighted search result excerpts* for more information.

Whoosh 2.0

Improvements

- Whoosh is now compatible with Python 3 (tested with Python 3.2). Special thanks to Vinay Sajip who did the work, and also Jordan Sherer who helped fix later issues.
- Sorting and grouping (faceting) now use a new system of "facet" objects which are much more flexible than the previous field-based system.

  For example, to sort by first name and then score:

```
from whoosh import sorting

mf = sorting.MultiFacet([sorting.FieldFacet("firstname"),
                         sorting.ScoreFacet()])
results = searcher.search(myquery, sortedby=mf)
```

  In addition to the previously supported sorting/grouping by field contents and/or query results, you can now use numeric ranges, date ranges, score, and more. The new faceting system also supports overlapping groups.

  (The old "Sorter" API still works but is deprecated and may be removed in a future version.)

  See *Sorting and faceting* for more information.

- Completely revamped spell-checking to make it much faster, easier, and more flexible. You can enable generation of the graph files use by spell checking using the `spelling=True` argument to a field type:

```
schema = fields.Schema(text=fields.TEXT(spelling=True))
```

  (Spelling suggestion methods will work on fields without `spelling=True` but will slower.) The spelling graph will be updated automatically as new documents are added – it is no longer necessary to maintain a separate "spelling index".

  You can get suggestions for individual words using *whoosh.searching.Searcher.suggest()*:

```
suglist = searcher.suggest("content", "werd", limit=3)
```

  Whoosh now includes convenience methods to spell-check and correct user queries, with optional highlighting of corrections using the `whoosh.highlight` module:

```
from whoosh import highlight, qparser

# User query string
qstring = request.get("q")

# Parse into query object
parser = qparser.QueryParser("content", myindex.schema)
qobject = parser.parse(qstring)

results = searcher.search(qobject)

if not results:
  correction = searcher.correct_query(gobject, gstring)
  # correction.query = corrected query object
  # correction.string = corrected query string

  # Format the corrected query string with HTML highlighting
  cstring = correction.format_string(highlight.HtmlFormatter())
```

Spelling suggestions can come from field contents and/or lists of words. For stemmed fields the spelling suggestions automatically use the unstemmed forms of the words.

There are APIs for spelling suggestions and query correction, so highly motivated users could conceivably replace the defaults with more sophisticated behaviors (for example, to take context into account).

See *"Did you mean... ?" Correcting errors in user queries* for more information.

- *whoosh.query.FuzzyTerm* now uses the new word graph feature as well and so is much faster.
- You can now set a boost factor for individual documents as you index them, to increase the score of terms in those documents in searches. See the documentation for the *add_document()* for more information.
- Added built-in recording of which terms matched in which documents. Use the `terms=True` argument to *whoosh.searching.Searcher.search()* and use *whoosh.searching.Hit.matched_terms()* and `whoosh.searching.Hit.contains_term()` to check matched terms.
- Whoosh now supports whole-term quality optimizations, so for example if the system knows that a UnionMatcher cannot possibly contribute to the "top N" results unless both sub-matchers match, it will replace the UnionMatcher with an IntersectionMatcher which is faster to compute. The performance improvement is not as dramatic as from block quality optimizations, but it can be noticeable.
- Fixed a bug that prevented block quality optimizations in queries with words not in the index, which could severely degrade performance.
- Block quality optimizations now use the actual scoring algorithm to calculate block quality instead of an approximation, which fixes issues where ordering of results could be different for searches with and without the optimizations.
- the BOOLEAN field type now supports field boosts.
- Re-architected the query parser to make the code easier to understand. Custom parser plugins from previous versions will probably break in Whoosh 2.0.
- Various bug-fixes and performance improvements.
- Removed the "read lock", which caused more problems than it solved. Now when opening a reader, if segments are deleted out from under the reader as it is opened, the code simply retries.

Compatibility

- The term quality optimizations required changes to the on-disk formats. Whoosh 2.0 if backwards-compatible with the old format. As you rewrite an index using Whoosh 2.0, by default it will use the new formats for new segments, making the index incompatible with older versions.

  To upgrade an existing index to use the new formats immediately, use `Index.optimize()`.
- Removed the experimental `TermTrackingCollector` since it is replaced by the new built-in term recording functionality.
- Removed the experimental `Searcher.define_facets` feature until a future release when it will be replaced by a more robust and useful feature.
- Reader iteration methods (`__iter__`, `iter_from`, `iter_field`, etc.) now yield *whoosh.reading.TermInfo* objects.
- The arguments to *whoosh.query.FuzzyTerm* changed.

Whoosh 1.x release notes

Whoosh 1.8.3

Whoosh 1.8.3 contains important bugfixes and new functionality. Thanks to all the mailing list and BitBucket users who helped with the fixes!

Fixed a bad `Collector` bug where the docset of a Results object did not match the actual results.

You can now pass a sequence of objects to a keyword argument in `add_document` and `update_document` (currently this will not work for unique fields in `update_document`). This is useful for non-text fields such as `DATETIME` and `NUMERIC`, allowing you to index multiple dates/numbers for a document:

```
writer.add_document(shoe=u"Saucony Kinvara", sizes=[10.0, 9.5, 12])
```

This version reverts to using the CDB hash function for hash files instead of Python's `hash()` because the latter is not meant to be stored externally. This change maintains backwards compatibility with old files.

The `Searcher.search` method now takes a `mask` keyword argument. This is the opposite of the `filter` argument. Where the `filter` specifies the set of documents that can appear in the results, the `mask` specifies a set of documents that must not appear in the results.

Fixed performance problems in `Searcher.more_like`. This method now also takes a `filter` keyword argument like `Searcher.search`.

Improved documentation.

Whoosh 1.8.2

Whoosh 1.8.2 fixes some bugs, including a mistyped signature in Searcher.more_like and a bad bug in Collector that could screw up the ordering of results given certain parameters.

Whoosh 1.8.1

Whoosh 1.8.1 includes a few recent bugfixes/improvements:

- ListMatcher.skip_to_quality() wasn't returning an integer, resulting in a "None + int" error.
- Fixed locking and memcache sync bugs in the Google App Engine storage object.
- MultifieldPlugin wasn't working correctly with groups.
  - The binary matcher trees of Or and And are now generated using a Huffman-like algorithm instead perfectly balanced. This gives a noticeable speed improvement because less information has to be passed up/down the tree.

Whoosh 1.8

This release relicensed the Whoosh source code under the Simplified BSD (A.K.A. "two-clause" or "FreeBSD") license. See LICENSE.txt for more information.

Whoosh 1.7.7

Setting a TEXT field to store term vectors is now much easier. Instead of having to pass an instantiated whoosh.formats.Format object to the vector= keyword argument, you can pass True to automatically use the same

format and analyzer as the inverted index. Alternatively, you can pass a Format subclass and Whoosh will instantiate it for you.

For example, to store term vectors using the same settings as the inverted index (Positions format and StandardAnalyzer):

```
from whoosh.fields import Schema, TEXT

schema = Schema(content=TEXT(vector=True))
```

To store term vectors that use the same analyzer as the inverted index (StandardAnalyzer by default) but only store term frequency:

```
from whoosh.formats import Frequency

schema = Schema(content=TEXT(vector=Frequency))
```

Note that currently the only place term vectors are used in Whoosh is keyword extraction/more like this, but they can be useful for expert users with custom code.

Added *whoosh.searching.Searcher.more_like()* and *whoosh.searching.Hit.more_like_this()* methods, as shortcuts for doing keyword extraction yourself. Return a Results object.

"python setup.py test" works again, as long as you have nose installed.

The `whoosh.searching.Searcher.sort_query_using()` method lets you sort documents matching a given query using an arbitrary function. Note that like "complex" searching with the Sorter object, this can be slow on large multi-segment indexes.

Whoosh 1.7

You can once again perform complex sorting of search results (that is, a sort with some fields ascending and some fields descending).

You can still use the `sortedby` keyword argument to *whoosh.searching.Searcher.search()* to do a simple sort (where all fields are sorted in the same direction), or you can use the new `Sorter` class to do a simple or complex sort:

```
searcher = myindex.searcher()
sorter = searcher.sorter()
# Sort first by the group field, ascending
sorter.add_field("group")
# Then by the price field, descending
sorter.add_field("price", reverse=True)
# Get the Results
results = sorter.sort_query(myquery)
```

See the documentation for the `Sorter` class for more information. Bear in mind that complex sorts will be much slower on large indexes because they can't use the per-segment field caches.

You can now get highlighted snippets for a hit automatically using *whoosh.searching.Hit.highlights()*:

```
results = searcher.search(myquery, limit=20)
for hit in results:
    print hit["title"]
    print hit.highlights("content")
```

See *whoosh.searching.Hit.highlights()* for more information.

Added the ability to filter search results so that only hits in a Results set, a set of docnums, or matching a query are returned. The filter is cached on the searcher.

> # Search within previous results newresults = searcher.search(newquery, filter=oldresults)
>
> # Search within the "basics" chapter results = searcher.search(userquery, filter=query.Term("chapter", "basics"))

You can now specify a time limit for a search. If the search does not finish in the given time, a *whoosh.searching.TimeLimit* exception is raised, but you can still retrieve the partial results from the collector. See the `timelimit` and `greedy` arguments in the `whoosh.searching.Collector` documentation.

Added back the ability to set *whoosh.analysis.StemFilter* to use an unlimited cache. This is useful for one-shot batch indexing (see *Tips for speeding up batch indexing*).

The `normalize()` method of the `And` and `Or` queries now merges overlapping range queries for more efficient queries.

Query objects now have `__hash__` methods allowing them to be used as dictionary keys.

The API of the highlight module has changed slightly. Most of the functions in the module have been converted to classes. However, most old code should still work. The `NullFragmeter` is now called `WholeFragmenter`, but the old name is still available as an alias.

Fixed MultiPool so it won't fill up the temp directory with job files.

Fixed a bug where Phrase query objects did not use their boost factor.

Fixed a bug where a fieldname after an open parenthesis wasn't parsed correctly. The change alters the semantics of certain parsing "corner cases" (such as `a:b:c:d`).

Whoosh 1.6

The `whoosh.writing.BatchWriter` class is now called *whoosh.writing.BufferedWriter*. It is similar to the old `BatchWriter` class but allows you to search and update the buffered documents as well as the documents that have been flushed to disk:

```
writer = writing.BufferedWriter(myindex)

# You can update (replace) documents in RAM without having to commit them
# to disk
writer.add_document(path="/a", text="Hi there")
writer.update_document(path="/a", text="Hello there")

# Search committed and uncommited documents by getting a searcher from the
# writer instead of the index
searcher = writer.searcher()
```

(BatchWriter is still available as an alias for backwards compatibility.)

The *whoosh.qparser.QueryParser* initialization method now requires a schema as the second argument. Previously the default was to create a `QueryParser` without a schema, which was confusing:

```
qp = qparser.QueryParser("content", myindex.schema)
```

The *whoosh.searching.Searcher.search()* method now takes a `scored` keyword. If you search with `scored=False`, the results will be in "natural" order (the order the documents were added to the index). This is useful when you don't need scored results but want the convenience of the Results object.

Added the *whoosh.qparser.GtLtPlugin* parser plugin to allow greater than/less as an alternative syntax for ranges:

```
count:>100 tag:<=zebra date:>='29 march 2001'
```

Added the ability to define schemas declaratively, similar to Django models:

```
from whoosh import index
from whoosh.fields import SchemaClass, ID, KEYWORD, STORED, TEXT

class MySchema(SchemaClass):
    uuid = ID(stored=True, unique=True)
    path = STORED
    tags = KEYWORD(stored=True)
    content = TEXT

index.create_in("indexdir", MySchema)
```

Whoosh 1.6.2: Added `whoosh.searching.TermTrackingCollector` which tracks which part of the query matched which documents in the final results.

Replaced the unbounded cache in *`whoosh.analysis.StemFilter`* with a bounded LRU (least recently used) cache. This will make stemming analysis slightly slower but prevent it from eating up too much memory over time.

Added a simple `whoosh.analysis.PyStemmerFilter` that works when the py-stemmer library is installed:

```
ana = RegexTokenizer() | PyStemmerFilter("spanish")
```

The estimation of memory usage for the `limitmb` keyword argument to `FileIndex.writer()` is more accurate, which should help keep memory usage memory usage by the sorting pool closer to the limit.

The `whoosh.ramdb` package was removed and replaced with a single `whoosh.ramindex` module.

Miscellaneous bug fixes.

Whoosh 1.5

Note: Whoosh 1.5 is incompatible with previous indexes. You must recreate existing indexes with Whoosh 1.5.

Fixed a bug where postings were not portable across different endian platforms.

New generalized field cache system, using per-reader caches, for much faster sorting and faceting of search results, as well as much faster multi-term (e.g. prefix and wildcard) and range queries, especially for large indexes and/or indexes with multiple segments.

Changed the faceting API. See *Sorting and faceting*.

Faster storage and retrieval of posting values.

Added per-field `multitoken_query` attribute to control how the query parser deals with a "term" that when analyzed generates multiple tokens. The default value is *"first"* which throws away all but the first token (the previous behavior). Other possible values are *"and"*, *"or"*, or *"phrase"*.

Added *`whoosh.analysis.DoubleMetaphoneFilter`*, *`whoosh.analysis.SubstitutionFilter`*, and *`whoosh.analysis.ShingleFilter`*.

Added *`whoosh.qparser.CopyFieldPlugin`*.

Added *`whoosh.query.Otherwise`*.

Generalized parsing of operators (such as OR, AND, NOT, etc.) in the query parser to make it easier to add new operators. In intend to add a better API for this in a future release.

Switched NUMERIC and DATETIME fields to use more compact on-disk representations of numbers.

Fixed a bug in the porter2 stemmer when stemming the string *"y"*.

Added methods to *whoosh.searching.Hit* to make it more like a *dict*.

Short posting lists (by default, single postings) are inline in the term file instead of written to the posting file for faster retrieval and a small saving in disk space.

Whoosh 1.3

Whoosh 1.3 adds a more efficient DATETIME field based on the new tiered NUMERIC field, and the DateParserPlugin. See *Indexing and parsing dates/times*.

Whoosh 1.2

Whoosh 1.2 adds tiered indexing for NUMERIC fields, resulting in much faster range queries on numeric fields.

Whoosh 1.0

Whoosh 1.0 is a major milestone release with vastly improved performance and several useful new features.

The index format of this version is not compatibile with indexes created by previous versions of Whoosh. You will need to reindex your data to use this version.

Orders of magnitude faster searches for common terms. Whoosh now uses optimizations similar to those in Xapian to skip reading low-scoring postings.

Faster indexing and ability to use multiple processors (via `multiprocessing` module) to speed up indexing.

Flexible Schema: you can now add and remove fields in an index with the *whoosh.writing.IndexWriter.add_field()* and *whoosh.writing.IndexWriter.remove_field()* methods.

New hand-written query parser based on plug-ins. Less brittle, more robust, more flexible, and easier to fix/improve than the old pyparsing-based parser.

On-disk formats now use 64-bit disk pointers allowing files larger than 4 GB.

New `whoosh.searching.Facets` class efficiently sorts results into facets based on any criteria that can be expressed as queries, for example tags or price ranges.

New `whoosh.writing.BatchWriter` class automatically batches up individual `add_document` and/or `delete_document` calls until a certain number of calls or a certain amount of time passes, then commits them all at once.

New *whoosh.analysis.BiWordFilter* lets you create bi-word indexed fields a possible alternative to phrase searching.

Fixed bug where files could be deleted before a reader could open them in threaded situations.

New *whoosh.analysis.NgramFilter* filter, *whoosh.analysis.NgramWordAnalyzer* analyzer, and *whoosh.fields.NGRAMWORDS* field type allow producing n-grams from tokenized text.

Errors in query parsing now raise a specific `whoosh.qparse.QueryParserError` exception instead of a generic exception.

Previously, the query string `*` was optimized to a *whoosh.query.Every* query which matched every document. Now the `Every` query only matches documents that actually have an indexed term from the given field, to better match the intuitive sense of what a query string like `tag:*` should do.

New *whoosh.searching.Searcher.key_terms_from_text()* method lets you extract key words from arbitrary text instead of documents in the index.

Previously the *whoosh.searching.Searcher.key_terms()* and *whoosh.searching.Results.key_terms()* methods required that the given field store term vectors. They now also work if the given field is stored instead. They will analyze the stored string into a term vector on-the-fly. The field must still be indexed.

User API changes

The default for the `limit` keyword argument to *whoosh.searching.Searcher.search()* is now `10`. To return all results in a single `Results` object, use `limit=None`.

The `Index` object no longer represents a snapshot of the index at the time the object was instantiated. Instead it always represents the index in the abstract. `Searcher` and `IndexReader` objects obtained from the `Index` object still represent the index as it was at the time they were created.

Because the `Index` object no longer represents the index at a specific version, several methods such as `up_to_date` and `refresh` were removed from its interface. The Searcher object now has `last_modified()`, *up_to_date()*, and *refresh()* methods similar to those that used to be on `Index`.

The document deletion and field add/remove methods on the `Index` object now create a writer behind the scenes to accomplish each call. This means they write to the index immediately, so you don't need to call `commit` on the `Index`. Also, it will be much faster if you need to call them multiple times to create your own writer instead:

```
# Don't do this
for id in my_list_of_ids_to_delete:
    myindex.delete_by_term("id", id)
myindex.commit()

# Instead do this
writer = myindex.writer()
for id in my_list_of_ids_to_delete:
    writer.delete_by_term("id", id)
writer.commit()
```

The `postlimit` argument to `Index.writer()` has been changed to `postlimitmb` and is now expressed in megabytes instead of bytes:

```
writer = myindex.writer(postlimitmb=128)
```

Instead of having to import `whoosh.filedb.filewriting.NO_MERGE` or `whoosh.filedb.filewriting.OPTIMIZE` to use as arguments to `commit()`, you can now simply do the following:

```
# Do not merge segments
writer.commit(merge=False)

# or

# Merge all segments
writer.commit(optimize=True)
```

The `whoosh.postings` module is gone. The `whoosh.matching` module contains classes for posting list readers.

Whoosh no longer maps field names to numbers for internal use or writing to disk. Any low-level method that accepted field numbers now accept field names instead.

Custom Weighting implementations that use the `final()` method must now set the `use_final` attribute to `True`:

```
from whoosh.scoring import BM25F

class MyWeighting(BM25F):
    use_final = True

    def final(searcher, docnum, score):
        return score + docnum * 10
```

This disables the new optimizations, forcing Whoosh to score every matching document.

whoosh.writing.AsyncWriter now takes an *whoosh.index.Index* object as its first argument, not a callable. Also, the keyword arguments to pass to the index's `writer()` method should now be passed as a dictionary using the `writerargs` keyword argument.

Whoosh now stores per-document field length using an approximation rather than exactly. For low numbers the approximation is perfectly accurate, while high numbers will be approximated less accurately.

The `doc_field_length` method on searchers and readers now takes a second argument representing the default to return if the given document and field do not have a length (i.e. the field is not scored or the field was not provided for the given document).

The *whoosh.analysis.StopFilter* now has a `maxsize` argument as well as a `minsize` argument to its initializer. Analyzers that use the `StopFilter` have the `maxsize` argument in their initializers now also.

The interface of *whoosh.writing.AsyncWriter* has changed.

Misc

- Because the file backend now writes 64-bit disk pointers and field names instead of numbers, the size of an index on disk will grow compared to previous versions.
- Unit tests should no longer leave directories and files behind.

Whoosh 0.3 release notes

- Major improvements to reading/writing of postings and query performance.
- Changed default post limit (run size) from 4 MB to 32 MB.
- Finished migrating backend-specific code into `whoosh.filedb` package.
- Moved formats from whoosh.fields module into new whoosh.formats module.
- DocReader and TermReader classes combined into new IndexReader interface. You can get an IndexReader implementation by calling Index.reader(). Searcher is now a wrapper around an IndexReader.
- Range query object changed, with new signature and new syntax in the default query parser. Now you can use `[start TO end]` in the query parser for an inclusive range, and `{start TO end}` for an exclusive range. You can also mix the delimiters, for example `[start TO end}` for a range with an inclusive start but exclusive end term.
- Added experimental DATETIME field type lets you pass a `datetime.datetime` object as a field value to `add_document`:

```
from whoosh.fields import Schema, ID, DATETIME
from whoosh.filedb.filestore import RamStorage
from datetime import datetime

schema = Schema(id=ID, date=DATETIME)
```

```
storage = RamStorage()
ix = storage.create_index(schema)
w = ix.writer()
w.add_document(id=u"A", date=datetime.now())
w.close()
```

Internally, the DATETIME field indexes the datetime object as text using the format (4 digit year + 2 digit month + 2 digit day + 'T' + 2 digit hour + 2 digit minute + 2 digit second + 6 digit microsecond), for example `20090817T160203109000`.

- The default query parser now lets you use quoted strings in prefix and range queries, e.g. `["2009-05" TO "2009-12"]`, `"alfa/bravo"*`, making it easier to work with terms containing special characters.
- `DocReader.vector_as(docnum, fieldid, astype)` is now `IndexReader.vector_as(astype, docnum, fieldid)` (i.e. the astype argument has moved from the last to the first argument), e.g. `v = ixreader.vector_as("frequency", 102, "content")`.
- Added whoosh.support.charset for translating Sphinx charset table files.
- Added whoosh.analysis.CharsetTokenizer and CharsetFilter to enable case and accent folding.
- Added experimental `whoosh.ramdb` in-memory backend.
- Added experimental `whoosh.query.FuzzyTerm` query type.
- Added `whoosh.lang.wordnet` module containing `Thesaurus` object for using WordNet synonym database.

Quick start

Whoosh is a library of classes and functions for indexing text and then searching the index. It allows you to develop custom search engines for your content. For example, if you were creating blogging software, you could use Whoosh to add a search function to allow users to search blog entries.

A quick introduction

```
>>> from whoosh.index import create_in
>>> from whoosh.fields import *
>>> schema = Schema(title=TEXT(stored=True), path=ID(stored=True), content=TEXT)
>>> ix = create_in("indexdir", schema)
>>> writer = ix.writer()
>>> writer.add_document(title=u"First document", path=u"/a",
...                     content=u"This is the first document we've added!")
>>> writer.add_document(title=u"Second document", path=u"/b",
...                     content=u"The second one is even more interesting!")
>>> writer.commit()
>>> from whoosh.qparser import QueryParser
>>> with ix.searcher() as searcher:
...     query = QueryParser("content", ix.schema).parse("first")
...     results = searcher.search(query)
...     results[0]
...
{"title": u"First document", "path": u"/a"}
```

The `Index` and `Schema` objects

To begin using Whoosh, you need an *index object*. The first time you create an index, you must define the index's *schema*. The schema lists the *fields* in the index. A field is a piece of information for each document in the index, such as its title or text content. A field can be *indexed* (meaning it can be searched) and/or *stored* (meaning the value that gets indexed is returned with the results; this is useful for fields such as the title).

This schema has two fields, "title" and "content":

```
from whoosh.fields import Schema, TEXT

schema = Schema(title=TEXT, content=TEXT)
```

You only need to do create the schema once, when you create the index. The schema is pickled and stored with the index.

When you create the `Schema` object, you use keyword arguments to map field names to field types. The list of fields and their types defines what you are indexing and what's searchable. Whoosh comes with some very useful predefined field types, and you can easily create your own.

whoosh.fields.ID This type simply indexes (and optionally stores) the entire value of the field as a single unit (that is, it doesn't break it up into individual words). This is useful for fields such as a file path, URL, date, category, etc.

whoosh.fields.STORED This field is stored with the document, but not indexed. This field type is not indexed and not searchable. This is useful for document information you want to display to the user in the search results.

whoosh.fields.KEYWORD This type is designed for space- or comma-separated keywords. This type is indexed and searchable (and optionally stored). To save space, it does not support phrase searching.

whoosh.fields.TEXT This type is for body text. It indexes (and optionally stores) the text and stores term positions to allow phrase searching.

whoosh.fields.NUMERIC This type is for numbers. You can store integers or floating point numbers.

whoosh.fields.BOOLEAN This type is for boolean (true/false) values.

whoosh.fields.DATETIME This type is for `datetime` objects. See *Indexing and parsing dates/times* for more information.

whoosh.fields.NGRAM* and *whoosh.fields.NGRAMWORDS These types break the field text or individual terms into N-grams. See *Indexing and searching N-grams* for more information.

(As a shortcut, if you don't need to pass any arguments to the field type, you can just give the class name and Whoosh will instantiate the object for you.)

```
from whoosh.fields import Schema, STORED, ID, KEYWORD, TEXT

schema = Schema(title=TEXT(stored=True), content=TEXT,
                path=ID(stored=True), tags=KEYWORD, icon=STORED)
```

See *Designing a schema* for more information.

Once you have the schema, you can create an index using the `create_in` function:

```
import os.path
from whoosh.index import create_in

if not os.path.exists("index"):
    os.mkdir("index")
ix = create_in("index", schema)
```

(At a low level, this creates a *Storage* object to contain the index. A `Storage` object represents that medium in which the index will be stored. Usually this will be `FileStorage`, which stores the index as a set of files in a directory.)

After you've created an index, you can open it using the `open_dir` convenience function:

```
from whoosh.index import open_dir

ix = open_dir("index")
```

The `IndexWriter` object

OK, so we've got an `Index` object, now we can start adding documents. The `writer()` method of the `Index` object returns an `IndexWriter` object that lets you add documents to the index. The IndexWriter's `add_document(**kwargs)` method accepts keyword arguments where the field name is mapped to a value:

```
writer = ix.writer()
writer.add_document(title=u"My document", content=u"This is my document!",
                    path=u"/a", tags=u"first short", icon=u"/icons/star.png")
writer.add_document(title=u"Second try", content=u"This is the second example.",
                    path=u"/b", tags=u"second short", icon=u"/icons/sheep.png")
writer.add_document(title=u"Third time's the charm", content=u"Examples are many.",
                    path=u"/c", tags=u"short", icon=u"/icons/book.png")
writer.commit()
```

Two important notes:

- You don't have to fill in a value for every field. Whoosh doesn't care if you leave out a field from a document.
- Indexed text fields must be passed a unicode value. Fields that are stored but not indexed (`STORED` field type) can be passed any pickle-able object.

If you have a text field that is both indexed and stored, you can index a unicode value but store a different object if necessary (it's usually not, but sometimes this is really useful) using this trick:

```
writer.add_document(title=u"Title to be indexed", _stored_title=u"Stored title")
```

Calling commit() on the `IndexWriter` saves the added documents to the index:

```
writer.commit()
```

See *How to index documents* for more information.

Once your documents are committed to the index, you can search for them.

The `Searcher` object

To begin searching the index, we'll need a `Searcher` object:

```
searcher = ix.searcher()
```

You'll usually want to open the searcher using a `with` statement so the searcher is automatically closed when you're done with it (searcher objects represent a number of open files, so if you don't explicitly close them and the system is slow to collect them, you can run out of file handles):

```
with ix.searcher() as searcher:
    ...
```

This is of course equivalent to:

```
try:
    searcher = ix.searcher()
    ...
finally:
    searcher.close()
```

The Searcher's `search()` method takes a *Query object*. You can construct query objects directly or use a query parser to parse a query string.

For example, this query would match documents that contain both "apple" and "bear" in the "content" field:

```
# Construct query objects directly

from whoosh.query import *
myquery = And([Term("content", u"apple"), Term("content", "bear")])
```

To parse a query string, you can use the default query parser in the `qparser` module. The first argument to the `QueryParser` constructor is the default field to search. This is usually the "body text" field. The second optional argument is a schema to use to understand how to parse the fields:

```
# Parse a query string

from whoosh.qparser import QueryParser
parser = QueryParser("content", ix.schema)
myquery = parser.parse(querystring)
```

Once you have a `Searcher` and a query object, you can use the `Searcher`'s `search()` method to run the query and get a `Results` object:

```
>>> results = searcher.search(myquery)
>>> print(len(results))
1
>>> print(results[0])
{"title": "Second try", "path": "/b", "icon": "/icons/sheep.png"}
```

The default `QueryParser` implements a query language very similar to Lucene's. It lets you connect terms with `AND` or `OR`, eleminate terms with `NOT`, group terms together into clauses with parentheses, do range, prefix, and wilcard queries, and specify different fields to search. By default it joins clauses together with `AND` (so by default, all terms you specify must be in the document for the document to match):

```
>>> print(parser.parse(u"render shade animate"))
And([Term("content", "render"), Term("content", "shade"), Term("content", "animate")])

>>> print(parser.parse(u"render OR (title:shade keyword:animate)"))
Or([Term("content", "render"), And([Term("title", "shade"), Term("keyword", "animate
↪")])])

>>> print(parser.parse(u"rend*"))
Prefix("content", "rend")
```

Whoosh includes extra features for dealing with search results, such as

- Sorting results by the value of an indexed field, instead of by relelvance.
- Highlighting the search terms in excerpts from the original documents.
- Expanding the query terms based on the top few documents found.

- Paginating the results (e.g. "Showing results 1-20, page 1 of 4").

See *How to search* for more information.

Introduction to Whoosh

About Whoosh

Whoosh was created by Matt Chaput. It started as a quick and dirty search server for the online documentation of the Houdini 3D animation software package. Side Effects Software generously allowed Matt to open source the code in case it might be useful to anyone else who needs a very flexible or pure-Python search engine (or both!).

- Whoosh is fast, but uses only pure Python, so it will run anywhere Python runs, without requiring a compiler.
- By default, Whoosh uses the Okapi BM25F ranking function, but like most things the ranking function can be easily customized.
- Whoosh creates fairly small indexes compared to many other search libraries.
- All indexed text in Whoosh must be *unicode*.
- Whoosh lets you store arbitrary Python objects with indexed documents.

What is Whoosh?

Whoosh is a fast, pure Python search engine library.

The primary design impetus of Whoosh is that it is pure Python. You should be able to use Whoosh anywhere you can use Python, no compiler or Java required.

Like one of its ancestors, Lucene, Whoosh is not really a search engine, it's a programmer library for creating a search engine[1].

Practically no important behavior of Whoosh is hard-coded. Indexing of text, the level of information stored for each term in each field, parsing of search queries, the types of queries allowed, scoring algorithms, etc. are all customizable, replaceable, and extensible.

What can Whoosh do for you?

Whoosh lets you index free-form or structured text and then quickly find matching documents based on simple or complex search criteria.

Getting help with Whoosh

You can view outstanding issues on the Whoosh Bitbucket page and get help on the Whoosh mailing list.

Glossary

Analysis The process of breaking the text of a field into individual *terms* to be indexed. This consists of tokenizing the text into terms, and then optionally filtering the tokenized terms (for example, lowercasing and removing *stop words*). Whoosh includes several different analyzers.

[1] It would of course be possible to build a turnkey search engine on top of Whoosh, like Nutch and Solr use Lucene.

Corpus The set of documents you are indexing.

Documents The individual pieces of content you want to make searchable. The word "documents" might imply files, but the data source could really be anything – articles in a content management system, blog posts in a blogging system, chunks of a very large file, rows returned from an SQL query, individual email messages from a mailbox file, or whatever. When you get search results from Whoosh, the results are a list of documents, whatever "documents" means in your search engine.

Fields Each document contains a set of fields. Typical fields might be "title", "content", "url", "keywords", "status", "date", etc. Fields can be indexed (so they're searchable) and/or stored with the document. Storing the field makes it available in search results. For example, you typically want to store the "title" field so your search results can display it.

Forward index A table listing every document and the words that appear in the document. Whoosh lets you store *term vectors* that are a kind of forward index.

Indexing The process of examining documents in the corpus and adding them to the *reverse index*.

Postings The *reverse index* lists every word in the corpus, and for each word, a list of documents in which that word appears, along with some optional information (such as the number of times the word appears in that document). These items in the list, containing a document number and any extra information, are called *postings*. In Whoosh the information stored in postings is customizable for each *field*.

Reverse index Basically a table listing every word in the corpus, and for each word, the list of documents in which it appears. It can be more complicated (the index can also list how many times the word appears in each document, the positions at which it appears, etc.) but that's how it basically works.

Schema Whoosh requires that you specify the *fields* of the index before you begin indexing. The Schema associates field names with metadata about the field, such as the format of the *postings* and whether the contents of the field are stored in the index.

Term vector A *forward index* for a certain field in a certain document. You can specify in the Schema that a given field should store term vectors.

Designing a schema

About schemas and fields

The schema specifies the fields of documents in an index.

Each document can have multiple fields, such as title, content, url, date, etc.

Some fields can be indexed, and some fields can be stored with the document so the field value is available in search results. Some fields will be both indexed and stored.

The schema is the set of all possible fields in a document. Each individual document might only use a subset of the available fields in the schema.

For example, a simple schema for indexing emails might have fields like `from_addr`, `to_addr`, `subject`, `body`, and `attachments`, where the `attachments` field lists the names of attachments to the email. For emails without attachments, you would omit the attachments field.

Built-in field types

Whoosh provides some useful predefined field types:

whoosh.fields.TEXT This type is for body text. It indexes (and optionally stores) the text and stores term positions to allow phrase searching.

`TEXT` fields use *`StandardAnalyzer`* by default. To specify a different analyzer, use the `analyzer` keyword argument to the constructor, e.g. `TEXT(analyzer=analysis.StemmingAnalyzer())`. See *About analyzers*.

By default, `TEXT` fields store position information for each indexed term, to allow you to search for phrases. If you don't need to be able to search for phrases in a text field, you can turn off storing term positions to save space. Use `TEXT(phrase=False)`.

By default, `TEXT` fields are not stored. Usually you will not want to store the body text in the search index. Usually you have the indexed documents themselves available to read or link to based on the search results, so you don't need to store their text in the search index. However, in some circumstances it can be useful (see *How to create highlighted search result excerpts*). Use `TEXT(stored=True)` to specify that the text should be stored in the index.

whoosh.fields.KEYWORD This field type is designed for space- or comma-separated keywords. This type is indexed and searchable (and optionally stored). To save space, it does not support phrase searching.

To store the value of the field in the index, use `stored=True` in the constructor. To automatically lowercase the keywords before indexing them, use `lowercase=True`.

By default, the keywords are space separated. To separate the keywords by commas instead (to allow keywords containing spaces), use `commas=True`.

If your users will use the keyword field for searching, use `scorable=True`.

whoosh.fields.ID The `ID` field type simply indexes (and optionally stores) the entire value of the field as a single unit (that is, it doesn't break it up into individual terms). This type of field does not store frequency information, so it's quite compact, but not very useful for scoring.

Use `ID` for fields like url or path (the URL or file path of a document), date, category – fields where the value must be treated as a whole, and each document only has one value for the field.

By default, `ID` fields are not stored. Use `ID(stored=True)` to specify that the value of the field should be stored with the document for use in the search results. For example, you would want to store the value of a url field so you could provide links to the original in your search results.

whoosh.fields.STORED This field is stored with the document, but not indexed and not searchable. This is useful for document information you want to display to the user in the search results, but don't need to be able to search for.

whoosh.fields.NUMERIC This field stores int, long, or floating point numbers in a compact, sortable format.

whoosh.fields.DATETIME This field stores datetime objects in a compact, sortable format.

whoosh.fields.BOOLEAN This simple filed indexes boolean values and allows users to search for `yes`, `no`, `true`, `false`, `1`, `0`, `t` or `f`.

whoosh.fields.NGRAM TBD.

Expert users can create their own field types.

Creating a Schema

To create a schema:

```
from whoosh.fields import Schema, TEXT, KEYWORD, ID, STORED
from whoosh.analysis import StemmingAnalyzer
```

```
schema = Schema(from_addr=ID(stored=True),
                to_addr=ID(stored=True),
                subject=TEXT(stored=True),
                body=TEXT(analyzer=StemmingAnalyzer()),
                tags=KEYWORD)
```

If you aren't specifying any constructor keyword arguments to one of the predefined fields, you can leave off the brackets (e.g. `fieldname=TEXT` instead of `fieldname=TEXT()`). Whoosh will instantiate the class for you.

Alternatively you can create a schema declaratively using the `SchemaClass` base class:

```
from whoosh.fields import SchemaClass, TEXT, KEYWORD, ID, STORED

class MySchema(SchemaClass):
    path = ID(stored=True)
    title = TEXT(stored=True)
    content = TEXT
    tags = KEYWORD
```

You can pass a declarative class to *`create_in()`* or `create_index()` instead of a *`Schema`* instance.

Modifying the schema after indexing

After you have created an index, you can add or remove fields to the schema using the `add_field()` and `remove_field()` methods. These methods are on the `Writer` object:

```
writer = ix.writer()
writer.add_field("fieldname", fields.TEXT(stored=True))
writer.remove_field("content")
writer.commit()
```

(If you're going to modify the schema *and* add documents using the same writer, you must call `add_field()` and/or `remove_field` *before* you add any documents.)

These methods are also on the `Index` object as a convenience, but when you call them on an `Index`, the Index object simply creates the writer, calls the corresponding method on it, and commits, so if you want to add or remove more than one field, it's much more efficient to create the writer yourself:

```
ix.add_field("fieldname", fields.KEYWORD)
```

In the `filedb` backend, removing a field simply removes that field from the *schema* – the index will not get smaller, data about that field will remain in the index until you optimize. Optimizing will compact the index, removing references to the deleted field as it goes:

```
writer = ix.writer()
writer.add_field("uuid", fields.ID(stored=True))
writer.remove_field("path")
writer.commit(optimize=True)
```

Because data is stored on disk with the field name, *do not* add a new field with the same name as a deleted field without optimizing the index in between:

```
writer = ix.writer()
writer.delete_field("path")
# Don't do this!!!
writer.add_field("path", fields.KEYWORD)
```

(A future version of Whoosh may automatically prevent this error.)

Dynamic fields

Dynamic fields let you associate a field type with any field name that matches a given "glob" (a name pattern containing `*`, `?`, and/or `[abc]` wildcards).

You can add dynamic fields to a new schema using the `add()` method with the `glob` keyword set to True:

```
schema = fields.Schema(...)
# Any name ending in "_d" will be treated as a stored
# DATETIME field
schema.add("*_d", fields.DATETIME(stored=True), glob=True)
```

To set up a dynamic field on an existing index, use the same `IndexWriter.add_field` method as if you were adding a regular field, but with the `glob` keyword argument set to `True`:

```
writer = ix.writer()
writer.add_field("*_d", fields.DATETIME(stored=True), glob=True)
writer.commit()
```

To remove a dynamic field, use the `IndexWriter.remove_field()` method with the glob as the name:

```
writer = ix.writer()
writer.remove_field("*_d")
writer.commit()
```

For example, to allow documents to contain any field name that ends in `_id` and associate it with the `ID` field type:

```
schema = fields.Schema(path=fields.ID)
schema.add("*_id", fields.ID, glob=True)

ix = index.create_in("myindex", schema)

w = ix.writer()
w.add_document(path=u"/a", test_id=u"alfa")
w.add_document(path=u"/b", class_id=u"MyClass")
# ...
w.commit()

qp = qparser.QueryParser("path", schema=schema)
q = qp.parse(u"test_id:alfa")
with ix.searcher() as s:
    results = s.search(q)
```

Advanced schema setup

Field boosts

You can specify a field boost for a field. This is a multiplier applied to the score of any term found in the field. For example, to make terms found in the title field score twice as high as terms in the body field:

```
schema = Schema(title=TEXT(field_boost=2.0), body=TEXT)
```

Field types

The predefined field types listed above are subclasses of `fields.FieldType`. `FieldType` is a pretty simple class. Its attributes contain information that define the behavior of a field.

Attribute	Type	Description
format	fields.Format	Defines what kind of information a field records about each term, and how the information is stored on disk.
vector	fields.Format	Optional: if defined, the format in which to store per-document forward-index information for this field.
scorable	bool	If True, the length of (number of terms in) the field in each document is stored in the index. Slightly misnamed, since field lengths are not required for all scoring. However, field lengths are required to get proper results from BM25F.
stored	bool	If True, the value of this field is stored in the index.
unique	bool	If True, the value of this field may be used to replace documents with the same value when the user calls `document_update()` on an `IndexWriter`.

The constructors for most of the predefined field types have parameters that let you customize these parts. For example:

- Most of the predefined field types take a stored keyword argument that sets FieldType.stored.
- The `TEXT()` constructor takes an `analyzer` keyword argument that is passed on to the format object.

Formats

A `Format` object defines what kind of information a field records about each term, and how the information is stored on disk.

For example, the `Existence` format would store postings like this:

Doc	
10	
20	
30	

Whereas the `Positions` format would store postings like this:

Doc	Positions
10	`[1,5,23]`
20	`[45]`
30	`[7,12]`

The indexing code passes the unicode string for a field to the field's `Format` object. The `Format` object calls its analyzer (see text analysis) to break the string into tokens, then encodes information about each token.

Whoosh ships with the following pre-defined formats.

Class name	Description
Stored	A "null" format for fields that are stored but not indexed.
Existence	Records only whether a term is in a document or not, i.e. it does not store term frequency. Useful for identifier fields (e.g. path or id) and "tag"-type fields, where the frequency is expected to always be 0 or 1.
Frequency	Stores the number of times each term appears in each document.
Positions	Stores the number of times each term appears in each document, and at what positions.

The `STORED` field type uses the `Stored` format (which does nothing, so `STORED` fields are not indexed). The `ID` type uses the `Existence` format. The `KEYWORD` type uses the `Frequency` format. The `TEXT` type uses the `Positions` format if it is instantiated with `phrase=True` (the default), or `Frequency` if `phrase=False`.

In addition, the following formats are implemented for the possible convenience of expert users, but are not currently used in Whoosh:

Class name	Description
DocBoosts	Like Existence, but also stores per-document boosts
Characters	Like Positions, but also stores the start and end character indices of each term
PositionBoosts	Like Positions, but also stores per-position boosts
Character-Boosts	Like Positions, but also stores the start and end character indices of each term and per-position boosts

Vectors

The main index is an inverted index. It maps terms to the documents they appear in. It is also sometimes useful to store a forward index, also known as a term vector, that maps documents to the terms that appear in them.

For example, imagine an inverted index like this for a field:

Term	Postings
apple	`[(doc=1, freq=2), (doc=2, freq=5), (doc=3, freq=1)]`
bear	`[(doc=2, freq=7)]`

The corresponding forward index, or term vector, would be:

Doc	Postings
1	`[(text=apple, freq=2)]`
2	`[(text=apple, freq=5), (text='bear', freq=7)]`
3	`[(text=apple, freq=1)]`

If you set `FieldType.vector` to a `Format` object, the indexing code will use the `Format` object to store information about the terms in each document. Currently by default Whoosh does not make use of term vectors at all, but they are available to expert users who want to implement their own field types.

How to index documents

Creating an Index object

To create an index in a directory, use `index.create_in`:

```
import os, os.path
from whoosh import index

if not os.path.exists("indexdir"):
    os.mkdir("indexdir")

ix = index.create_in("indexdir", schema)
```

To open an existing index in a directory, use `index.open_dir`:

```
import whoosh.index as index

ix = index.open_dir("indexdir")
```

These are convenience methods for:

```
from whoosh.filedb.filestore import FileStorage
storage = FileStorage("indexdir")

# Create an index
ix = storage.create_index(schema)

# Open an existing index
storage.open_index()
```

The schema you created the index with is pickled and stored with the index.

You can keep multiple indexes in the same directory using the indexname keyword argument:

```
# Using the convenience functions
ix = index.create_in("indexdir", schema=schema, indexname="usages")
ix = index.open_dir("indexdir", indexname="usages")

# Using the Storage object
ix = storage.create_index(schema, indexname="usages")
ix = storage.open_index(indexname="usages")
```

Clearing the index

Calling `index.create_in` on a directory with an existing index will clear the current contents of the index.

To test whether a directory currently contains a valid index, use `index.exists_in`:

```
exists = index.exists_in("indexdir")
usages_exists = index.exists_in("indexdir", indexname="usages")
```

(Alternatively you can simply delete the index's files from the directory, e.g. if you only have one index in the directory, use `shutil.rmtree` to remove the directory and then recreate it.)

Indexing documents

Once you've created an `Index` object, you can add documents to the index with an `IndexWriter` object. The easiest way to get the `IndexWriter` is to call `Index.writer()`:

```
ix = index.open_dir("index")
writer = ix.writer()
```

Creating a writer locks the index for writing, so only one thread/process at a time can have a writer open.

Note: Because opening a writer locks the index for writing, in a multi-threaded or multi-process environment your code needs to be aware that opening a writer may raise an exception (`whoosh.store.LockError`) if a writer is already open. Whoosh includes a couple of example implementations (*`whoosh.writing.AsyncWriter`* and *`whoosh.writing.BufferedWriter`*) of ways to work around the write lock.

Note: While the writer is open and during the commit, the index is still available for reading. Existing readers are unaffected and new readers can open the current index normally. Once the commit is finished, existing readers

continue to see the previous version of the index (that is, they do not automatically see the newly committed changes). New readers will see the updated index.

The IndexWriter's `add_document(**kwargs)` method accepts keyword arguments where the field name is mapped to a value:

```
writer = ix.writer()
writer.add_document(title=u"My document", content=u"This is my document!",
                    path=u"/a", tags=u"first short", icon=u"/icons/star.png")
writer.add_document(title=u"Second try", content=u"This is the second example.",
                    path=u"/b", tags=u"second short", icon=u"/icons/sheep.png")
writer.add_document(title=u"Third time's the charm", content=u"Examples are many.",
                    path=u"/c", tags=u"short", icon=u"/icons/book.png")
writer.commit()
```

You don't have to fill in a value for every field. Whoosh doesn't care if you leave out a field from a document.

Indexed fields must be passed a unicode value. Fields that are stored but not indexed (i.e. the `STORED` field type) can be passed any pickle-able object.

Whoosh will happily allow you to add documents with identical values, which can be useful or annoying depending on what you're using the library for:

```
writer.add_document(path=u"/a", title=u"A", content=u"Hello there")
writer.add_document(path=u"/a", title=u"A", content=u"Deja vu!")
```

This adds two documents to the index with identical path and title fields. See "updating documents" below for information on the `update_document` method, which uses "unique" fields to replace old documents instead of appending.

Indexing and storing different values for the same field

If you have a field that is both indexed and stored, you can index a unicode value but store a different object if necessary (it's usually not, but sometimes this is really useful) using a "special" keyword argument `_stored_<fieldname>`. The normal value will be analyzed and indexed, but the "stored" value will show up in the results:

```
writer.add_document(title=u"Title to be indexed", _stored_title=u"Stored title")
```

Finishing adding documents

An `IndexWriter` object is kind of like a database transaction. You specify a bunch of changes to the index, and then "commit" them all at once.

Calling `commit()` on the `IndexWriter` saves the added documents to the index:

```
writer.commit()
```

Once your documents are in the index, you can search for them.

If you want to close the writer without committing the changes, call `cancel()` instead of `commit()`:

```
writer.cancel()
```

Keep in mind that while you have a writer open (including a writer you opened and is still in scope), no other thread or process can get a writer or modify the index. A writer also keeps several open files. So you should always remember to call either `commit()` or `cancel()` when you're done with a writer object.

Merging segments

A Whoosh `filedb` index is really a container for one or more "sub-indexes" called segments. When you add documents to an index, instead of integrating the new documents with the existing documents (which could potentially be very expensive, since it involves resorting all the indexed terms on disk), Whoosh creates a new segment next to the existing segment. Then when you search the index, Whoosh searches both segments individually and merges the results so the segments appear to be one unified index. (This smart design is copied from Lucene.)

So, having a few segments is more efficient than rewriting the entire index every time you add some documents. But searching multiple segments does slow down searching somewhat, and the more segments you have, the slower it gets. So Whoosh has an algorithm that runs when you call `commit()` that looks for small segments it can merge together to make fewer, bigger segments.

To prevent Whoosh from merging segments during a commit, use the `merge` keyword argument:

```
writer.commit(merge=False)
```

To merge all segments together, optimizing the index into a single segment, use the `optimize` keyword argument:

```
writer.commit(optimize=True)
```

Since optimizing rewrites all the information in the index, it can be slow on a large index. It's generally better to rely on Whoosh's merging algorithm than to optimize all the time.

(The `Index` object also has an `optimize()` method that lets you optimize the index (merge all the segments together). It simply creates a writer and calls `commit(optimize=True)` on it.)

For more control over segment merging, you can write your own merge policy function and use it as an argument to the `commit()` method. See the implementation of the `NO_MERGE`, `MERGE_SMALL`, and `OPTIMIZE` functions in the `whoosh.writing` module.

Deleting documents

You can delete documents using the following methods on an `IndexWriter` object. You then need to call `commit()` on the writer to save the deletions to disk.

`delete_document(docnum)`

Low-level method to delete a document by its internal document number.

`is_deleted(docnum)`

Low-level method, returns `True` if the document with the given internal number is deleted.

`delete_by_term(fieldname, termtext)`

Deletes any documents where the given (indexed) field contains the given term. This is mostly useful for `ID` or `KEYWORD` fields.

`delete_by_query(query)`

Deletes any documents that match the given query.

```
# Delete document by its path -- this field must be indexed
ix.delete_by_term('path', u'/a/b/c')
# Save the deletion to disk
ix.commit()
```

In the `filedb` backend, "deleting" a document simply adds the document number to a list of deleted documents stored with the index. When you search the index, it knows not to return deleted documents in the results. However,

the document's contents are still stored in the index, and certain statistics (such as term document frequencies) are not updated, until you merge the segments containing deleted documents (see merging above). (This is because removing the information immediately from the index would essentially involving rewriting the entire index on disk, which would be very inefficient.)

Updating documents

If you want to "replace" (re-index) a document, you can delete the old document using one of the `delete_*` methods on `Index` or `IndexWriter`, then use `IndexWriter.add_document` to add the new version. Or, you can use `IndexWriter.update_document` to do this in one step.

For `update_document` to work, you must have marked at least one of the fields in the schema as "unique". Whoosh will then use the contents of the "unique" field(s) to search for documents to delete:

```
from whoosh.fields import Schema, ID, TEXT

schema = Schema(path = ID(unique=True), content=TEXT)

ix = index.create_in("index")
writer = ix.writer()
writer.add_document(path=u"/a", content=u"The first document")
writer.add_document(path=u"/b", content=u"The second document")
writer.commit()

writer = ix.writer()
# Because "path" is marked as unique, calling update_document with path="/a"
# will delete any existing documents where the "path" field contains "/a".
writer.update_document(path=u"/a", content="Replacement for the first document")
writer.commit()
```

The "unique" field(s) must be indexed.

If no existing document matches the unique fields of the document you're updating, `update_document` acts just like `add_document`.

"Unique" fields and `update_document` are simply convenient shortcuts for deleting and adding. Whoosh has no inherent concept of a unique identifier, and in no way enforces uniqueness when you use `add_document`.

Incremental indexing

When you're indexing a collection of documents, you'll often want two code paths: one to index all the documents from scratch, and one to only update the documents that have changed (leaving aside web applications where you need to add/update documents according to user actions).

Indexing everything from scratch is pretty easy. Here's a simple example:

```
import os.path
from whoosh import index
from whoosh.fields import Schema, ID, TEXT

def clean_index(dirname):
  # Always create the index from scratch
  ix = index.create_in(dirname, schema=get_schema())
  writer = ix.writer()

  # Assume we have a function that gathers the filenames of the
  # documents to be indexed
```

```
  for path in my_docs():
    add_doc(writer, path)

  writer.commit()


def get_schema()
  return Schema(path=ID(unique=True, stored=True), content=TEXT)


def add_doc(writer, path):
  fileobj = open(path, "rb")
  content = fileobj.read()
  fileobj.close()
  writer.add_document(path=path, content=content)
```

Now, for a small collection of documents, indexing from scratch every time might actually be fast enough. But for large collections, you'll want to have the script only re-index the documents that have changed.

To start we'll need to store each document's last-modified time, so we can check if the file has changed. In this example, we'll just use the mtime for simplicity:

```
def get_schema()
  return Schema(path=ID(unique=True, stored=True), time=STORED, content=TEXT)

def add_doc(writer, path):
  fileobj = open(path, "rb")
  content = fileobj.read()
  fileobj.close()
  modtime = os.path.getmtime(path)
  writer.add_document(path=path, content=content, time=modtime)
```

Now we can modify the script to allow either "clean" (from scratch) or incremental indexing:

```
def index_my_docs(dirname, clean=False):
  if clean:
    clean_index(dirname)
  else:
    incremental_index(dirname)


def incremental_index(dirname)
    ix = index.open_dir(dirname)

    # The set of all paths in the index
    indexed_paths = set()
    # The set of all paths we need to re-index
    to_index = set()

    with ix.searcher() as searcher:
      writer = ix.writer()

      # Loop over the stored fields in the index
      for fields in searcher.all_stored_fields():
        indexed_path = fields['path']
        indexed_paths.add(indexed_path)

        if not os.path.exists(indexed_path):
```

```
          # This file was deleted since it was indexed
          writer.delete_by_term('path', indexed_path)

        else:
          # Check if this file was changed since it
          # was indexed
          indexed_time = fields['time']
          mtime = os.path.getmtime(indexed_path)
          if mtime > indexed_time:
            # The file has changed, delete it and add it to the list of
            # files to reindex
            writer.delete_by_term('path', indexed_path)
            to_index.add(indexed_path)

      # Loop over the files in the filesystem
      # Assume we have a function that gathers the filenames of the
      # documents to be indexed
      for path in my_docs():
        if path in to_index or path not in indexed_paths:
          # This is either a file that's changed, or a new file
          # that wasn't indexed before. So index it!
          add_doc(writer, path)

      writer.commit()
```

The `incremental_index` function:

- Loops through all the paths that are currently indexed.
  - If any of the files no longer exist, delete the corresponding document from the index.
  - If the file still exists, but has been modified, add it to the list of paths to be re-indexed.
  - If the file exists, whether it's been modified or not, add it to the list of all indexed paths.
- Loops through all the paths of the files on disk.
  - If a path is not in the set of all indexed paths, the file is new and we need to index it.
  - If a path is in the set of paths to re-index, we need to index it.
  - Otherwise, we can skip indexing the file.

Clearing the index

In some cases you may want to re-index from scratch. To clear the index without disrupting any existing readers:

```
from whoosh import writing

with myindex.writer() as mywriter:
    # You can optionally add documents to the writer here
    # e.g. mywriter.add_document(...)

    # Using mergetype=CLEAR clears all existing segments so the index will
    # only have any documents you've added to this writer
    mywriter.mergetype = writing.CLEAR
```

Or, if you don't use the writer as a context manager and call `commit()` directly, do it like this:

```
mywriter = myindex.writer()
# ...
mywriter.commit(mergetype=writing.CLEAR)
```

Note: If you don't need to worry about existing readers, a more efficient method is to simply delete the contents of the index directory and start over.

How to search

Once you've created an index and added documents to it, you can search for those documents.

The `Searcher` object

To get a *whoosh.searching.Searcher* object, call `searcher()` on your `Index` object:

```
searcher = myindex.searcher()
```

You'll usually want to open the searcher using a `with` statement so the searcher is automatically closed when you're done with it (searcher objects represent a number of open files, so if you don't explicitly close them and the system is slow to collect them, you can run out of file handles):

```
with ix.searcher() as searcher:
    ...
```

This is of course equivalent to:

```
try:
    searcher = ix.searcher()
    ...
finally:
    searcher.close()
```

The `Searcher` object is the main high-level interface for reading the index. It has lots of useful methods for getting information about the index, such as `lexicon(fieldname)`.

```
>>> list(searcher.lexicon("content"))
[u"document", u"index", u"whoosh"]
```

However, the most important method on the `Searcher` object is *search()*, which takes a *whoosh.query.Query* object and returns a *Results* object:

```
from whoosh.qparser import QueryParser

qp = QueryParser("content", schema=myindex.schema)
q = qp.parse(u"hello world")

with myindex.searcher() as s:
    results = s.search(q)
```

By default the results contains at most the first 10 matching documents. To get more results, use the `limit` keyword:

```
results = s.search(q, limit=20)
```

If you want all results, use `limit=None`. However, setting the limit whenever possible makes searches faster because Whoosh doesn't need to examine and score every document.

Since displaying a page of results at a time is a common pattern, the `search_page` method lets you conveniently retrieve only the results on a given page:

```
results = s.search_page(q, 1)
```

The default page length is 10 hits. You can use the `pagelen` keyword argument to set a different page length:

```
results = s.search_page(q, 5, pagelen=20)
```

Results object

The *`Results`* object acts like a list of the matched documents. You can use it to access the stored fields of each hit document, to display to the user.

```
>>> # Show the best hit's stored fields
>>> results[0]
{"title": u"Hello World in Python", "path": u"/a/b/c"}
>>> results[0:2]
[{"title": u"Hello World in Python", "path": u"/a/b/c"},
{"title": u"Foo", "path": u"/bar"}]
```

By default, `Searcher.search(myquery)` limits the number of hits to 20, So the number of scored hits in the `Results` object may be less than the number of matching documents in the index.

```
>>> # How many documents in the entire index would have matched?
>>> len(results)
27
>>> # How many scored and sorted documents in this Results object?
>>> # This will often be less than len() if the number of hits was limited
>>> # (the default).
>>> results.scored_length()
10
```

Calling `len(Results)` runs a fast (unscored) version of the query again to figure out the total number of matching documents. This is usually very fast but for large indexes it can cause a noticeable delay. If you want to avoid this delay on very large indexes, you can use the *`has_exact_length()`*, *`estimated_length()`*, and *`estimated_min_length()`* methods to estimate the number of matching documents without calling `len()`:

```
found = results.scored_length()
if results.has_exact_length():
    print("Scored", found, "of exactly", len(results), "documents")
else:
    low = results.estimated_min_length()
    high = results.estimated_length()

    print("Scored", found, "of between", low, "and", high, "documents")
```

Scoring and sorting

Scoring

Normally the list of result documents is sorted by *score*. The `whoosh.scoring` module contains implementations of various scoring algorithms. The default is `BM25F`.

You can set the scoring object to use when you create the searcher using the `weighting` keyword argument:

```
from whoosh import scoring

with myindex.searcher(weighting=scoring.TF_IDF()) as s:
    ...
```

A weighting model is a `WeightingModel` subclass with a `scorer()` method that produces a "scorer" instance. This instance has a method that takes the current matcher and returns a floating point score.

Sorting

See *Sorting and faceting*.

Highlighting snippets and More Like This

See *How to create highlighted search result excerpts* and *Query expansion and Key word extraction* for information on these topics.

Filtering results

You can use the `filter` keyword argument to `search()` to specify a set of documents to permit in the results. The argument can be a `whoosh.query.Query` object, a `whoosh.searching.Results` object, or a set-like object containing document numbers. The searcher caches filters so if for example you use the same query filter with a searcher multiple times, the additional searches will be faster because the searcher will cache the results of running the filter query

You can also specify a `mask` keyword argument to specify a set of documents that are not permitted in the results.

```
with myindex.searcher() as s:
    qp = qparser.QueryParser("content", myindex.schema)
    user_q = qp.parse(query_string)

    # Only show documents in the "rendering" chapter
    allow_q = query.Term("chapter", "rendering")
    # Don't show any documents where the "tag" field contains "todo"
    restrict_q = query.Term("tag", "todo")

    results = s.search(user_q, filter=allow_q, mask=restrict_q)
```

(If you specify both a `filter` and a `mask`, and a matching document appears in both, the `mask` "wins" and the document is not permitted.)

To find out how many results were filtered out of the results, use `results.filtered_count` (or `resultspage.results.filtered_count`):

```
with myindex.searcher() as s:
    qp = qparser.QueryParser("content", myindex.schema)
    user_q = qp.parse(query_string)
```

```
    # Filter documents older than 7 days
    old_q = query.DateRange("created", None, datetime.now() - timedelta(days=7))
    results = s.search(user_q, mask=old_q)

    print("Filtered out %d older documents" % results.filtered_count)
```

Which terms from my query matched?

You can use the `terms=True` keyword argument to `search()` to have the search record which terms in the query matched which documents:

```
with myindex.searcher() as s:
    results = s.seach(myquery, terms=True)
```

You can then get information about which terms matched from the *whoosh.searching.Results* and *whoosh.searching.Hit* objects:

```
# Was this results object created with terms=True?
if results.has_matched_terms():
    # What terms matched in the results?
    print(results.matched_terms())

    # What terms matched in each hit?
    for hit in results:
        print(hit.matched_terms())
```

Collapsing results

Whoosh lets you eliminate all but the top N documents with the same facet key from the results. This can be useful in a few situations:

- Eliminating duplicates at search time.
- Restricting the number of matches per source. For example, in a web search application, you might want to show at most three matches from any website.

Whether a document should be collapsed is determined by the value of a "collapse facet". If a document has an empty collapse key, it will never be collapsed, but otherwise only the top N documents with the same collapse key will appear in the results.

See *Sorting and faceting* for information on facets.

```
with myindex.searcher() as s:
    # Set the facet to collapse on and the maximum number of documents per
    # facet value (default is 1)
    results = s.collector(collapse="hostname", collapse_limit=3)

    # Dictionary mapping collapse keys to the number of documents that
    # were filtered out by collapsing on that key
    print(results.collapsed_counts)
```

Collapsing works with both scored and sorted results. You can use any of the facet types available in the *whoosh.sorting* module.

By default, Whoosh uses the results order (score or sort key) to determine the documents to collapse. For example, in scored results, the best scoring documents would be kept. You can optionally specify a `collapse_order` facet to control which documents to keep when collapsing.

For example, in a product search you could display results sorted by decreasing price, and eliminate all but the highest rated item of each product type:

```
from whoosh import sorting

with myindex.searcher() as s:
    price_facet = sorting.FieldFacet("price", reverse=True)
    type_facet = sorting.FieldFacet("type")
    rating_facet = sorting.FieldFacet("rating", reverse=True)

    results = s.collector(sortedby=price_facet,  # Sort by reverse price
                          collapse=type_facet,  # Collapse on product type
                          collapse_order=rating_facet  # Collapse to highest rated
                          )
```

The collapsing happens during the search, so it is usually more efficient than finding everything and post-processing the results. However, if the collapsing eliminates a large number of documents, collapsed search can take longer because the search has to consider more documents and remove many already-collected documents.

Since this collector must sometimes go back and remove already-collected documents, if you use it in combination with *TermsCollector* and/or *FacetCollector*, those collectors may contain information about documents that were filtered out of the final results by collapsing.

Time limited searches

To limit the amount of time a search can take:

```
from whoosh.collectors import TimeLimitCollector, TimeLimit

with myindex.searcher() as s:
    # Get a collector object
    c = s.collector(limit=None, sortedby="title_exact")
    # Wrap it in a TimeLimitedCollector and set the time limit to 10 seconds
    tlc = TimeLimitedCollector(c, timelimit=10.0)

    # Try searching
    try:
        s.search_with_collector(myquery, tlc)
    except TimeLimit:
        print("Search took too long, aborting!")

    # You can still get partial results from the collector
    results = tlc.results()
```

Convenience methods

The *document()* and *documents()* methods on the `Searcher` object let you retrieve the stored fields of documents matching terms you pass in keyword arguments.

This is especially useful for fields such as dates/times, identifiers, paths, and so on.

```
>>> list(searcher.documents(indexeddate=u"20051225"))
[{"title": u"Christmas presents"}, {"title": u"Turkey dinner report"}]
>>> print searcher.document(path=u"/a/b/c")
{"title": "Document C"}
```

These methods have some limitations:

- The results are not scored.
- Multiple keywords are always AND-ed together.
- The entire value of each keyword argument is considered a single term; you can't search for multiple terms in the same field.

Combining Results objects

It is sometimes useful to use the results of another query to influence the order of a *whoosh.searching.Results* object.

For example, you might have a "best bet" field. This field contains hand-picked keywords for documents. When the user searches for those keywords, you want those documents to be placed at the top of the results list. You could try to do this by boosting the "bestbet" field tremendously, but that can have unpredictable effects on scoring. It's much easier to simply run the query twice and combine the results:

```
# Parse the user query
userquery = queryparser.parse(querystring)

# Get the terms searched for
termset = set()
userquery.existing_terms(termset)

# Formulate a "best bet" query for the terms the user
# searched for in the "content" field
bbq = Or([Term("bestbet", text) for fieldname, text
          in termset if fieldname == "content"])

# Find documents matching the searched for terms
results = s.search(bbq, limit=5)

# Find documents that match the original query
allresults = s.search(userquery, limit=10)

# Add the user query results on to the end of the "best bet"
# results. If documents appear in both result sets, push them
# to the top of the combined results.
results.upgrade_and_extend(allresults)
```

The `Results` object supports the following methods:

`Results.extend(results)` Adds the documents in 'results' on to the end of the list of result documents.

`Results.filter(results)` Removes the documents in 'results' from the list of result documents.

`Results.upgrade(results)` Any result documents that also appear in 'results' are moved to the top of the list of result documents.

`Results.upgrade_and_extend(results)` Any result documents that also appear in 'results' are moved to the top of the list of result documents. Then any other documents in 'results' are added on to the list of result documents.

Parsing user queries

Overview

The job of a query parser is to convert a *query string* submitted by a user into *query objects* (objects from the *whoosh.query* module).

For example, the user query:

```
rendering shading
```

might be parsed into query objects like this:

```
And([Term("content", u"rendering"), Term("content", u"shading")])
```

Whoosh includes a powerful, modular parser for user queries in the *whoosh.qparser* module. The default parser implements a query language similar to the one that ships with Lucene. However, by changing plugins or using functions such as *whoosh.qparser.MultifieldParser()*, *whoosh.qparser.SimpleParser()* or *whoosh.qparser.DisMaxParser()*, you can change how the parser works, get a simpler parser or change the query language syntax.

(In previous versions of Whoosh, the query parser was based on `pyparsing`. The new hand-written parser is less brittle and more flexible.)

Note: Remember that you can directly create query objects programmatically using the objects in the *whoosh.query* module. If you are not processing actual user queries, this is preferable to building a query string just to parse it.

Using the default parser

To create a *whoosh.qparser.QueryParser* object, pass it the name of the *default field* to search and the schema of the index you'll be searching.

```
from whoosh.qparser import QueryParser

parser = QueryParser("content", schema=myindex.schema)
```

Tip: You can instantiate a `QueryParser` object without specifying a schema, however the parser will not process the text of the user query. This is useful for debugging, when you want to see how QueryParser will build a query, but don't want to make up a schema just for testing.

Once you have a `QueryParser` object, you can call `parse()` on it to parse a query string into a query object:

```
>>> parser.parse(u"alpha OR beta gamma")
And([Or([Term('content', u'alpha'), Term('content', u'beta')]), Term('content', u
↪'gamma')])
```

See the *query language reference* for the features and syntax of the default parser's query language.

Common customizations

Searching for any terms instead of all terms by default

If the user doesn't explicitly specify `AND` or `OR` clauses:

```
physically based rendering
```

...by default, the parser treats the words as if they were connected by `AND`, meaning all the terms must be present for a document to match:

```
physically AND based AND rendering
```

To change the parser to use `OR` instead, so that any of the terms may be present for a document to match, i.e.:

```
physically OR based OR rendering
```

...configure the QueryParser using the `group` keyword argument like this:

```
from whoosh import qparser

parser = qparser.QueryParser(fieldname, schema=myindex.schema,
                             group=qparser.OrGroup)
```

The Or query lets you specify that documents that contain more of the query terms score higher. For example, if the user searches for `foo bar`, a document with four occurances of `foo` would normally outscore a document that contained one occurance each of `foo` and `bar`. However, users usually expect documents that contain more of the words they searched for to score higher. To configure the parser to produce Or groups with this behavior, use the `factory()` class method of `OrGroup`:

```
og = qparser.OrGroup.factory(0.9)
parser = qparser.QueryParser(fieldname, schema, group=og)
```

where the argument to `factory()` is a scaling factor on the bonus (between 0 and 1).

Letting the user search multiple fields by default

The default QueryParser configuration takes terms without explicit fields and assigns them to the default field you specified when you created the object, so for example if you created the object with:

```
parser = QueryParser("content", schema=myschema)
```

And the user entered the query:

```
three blind mice
```

The parser would treat it as:

```
content:three content:blind content:mice
```

However, you might want to let the user search *multiple* fields by default. For example, you might want "unfielded" terms to search both the `title` and `content` fields.

In that case, you can use a *whoosh.qparser.MultifieldParser*. This is just like the normal QueryParser, but instead of a default field name string, it takes a *sequence* of field names:

```
from whoosh.qparser import MultifieldParser

mparser = MultifieldParser(["title", "content"], schema=myschema)
```

When this MultifieldParser instance parses `three blind mice`, it treats it as:

```
(title:three OR content:three) (title:blind OR content:blind) (title:mice OR␣
↪content:mice)
```

Simplifying the query language

Once you have a parser:

```
parser = qparser.QueryParser("content", schema=myschema)
```

you can remove features from it using the *`remove_plugin_class()`* method.

For example, to remove the ability of the user to specify fields to search:

```
parser.remove_plugin_class(qparser.FieldsPlugin)
```

To remove the ability to search for wildcards, which can be harmful to query performance:

```
parser.remove_plugin_class(qparser.WildcardPlugin)
```

See *qparser module* for information about the plugins included with Whoosh's query parser.

Changing the AND, OR, ANDNOT, ANDMAYBE, and NOT syntax

The default parser uses English keywords for the AND, OR, ANDNOT, ANDMAYBE, and NOT functions:

```
parser = qparser.QueryParser("content", schema=myschema)
```

You can replace the default `OperatorsPlugin` object to replace the default English tokens with your own regular expressions.

The *`whoosh.qparser.OperatorsPlugin`* implements the ability to use AND, OR, NOT, ANDNOT, and ANDMAYBE clauses in queries. You can instantiate a new `OperatorsPlugin` and use the `And`, `Or`, `Not`, `AndNot`, and `AndMaybe` keyword arguments to change the token patterns:

```
# Use Spanish equivalents instead of AND and OR
op = qparser.OperatorsPlugin(And=" Y ", Or=" O ")
parser.replace_plugin(op)
```

Further, you may change the syntax of the `NOT` operator:

```
np = qparser.OperatorsPlugin(Not=' NO ')
parser.replace_plugin(np)
```

The arguments can be pattern strings or precompiled regular expression objects.

For example, to change the default parser to use typographic symbols instead of words for the AND, OR, ANDNOT, ANDMAYBE, and NOT functions:

```
parser = qparser.QueryParser("content", schema=myschema)
# These are regular expressions, so we have to escape the vertical bar
op = qparser.OperatorsPlugin(And="&", Or="\\|", AndNot="&!", AndMaybe="&~", Not="\\-")
parser.replace_plugin(op)
```

Adding less-than, greater-than, etc.

Normally, the way you match all terms in a field greater than "apple" is with an open ended range:

```
field:{apple to]
```

The *whoosh.qparser.GtLtPlugin* lets you specify the same search like this:

```
field:>apple
```

The plugin lets you use >, <, >=, <=, =>, or =< after a field specifier, and translates the expression into the equivalent range:

```
date:>='31 march 2001'

date:[31 march 2001 to]
```

Adding fuzzy term queries

Fuzzy queries are good for catching misspellings and similar words. The `whoosh.qparser.FuzzyTermPlugin` lets you search for "fuzzy" terms, that is, terms that don't have to match exactly. The fuzzy term will match any similar term within a certain number of "edits" (character insertions, deletions, and/or transpositions – this is called the "Damerau-Levenshtein edit distance").

To add the fuzzy plugin:

```
parser = qparser.QueryParser("fieldname", my_index.schema)
parser.add_plugin(qparser.FuzzyTermPlugin())
```

Once you add the fuzzy plugin to the parser, you can specify a fuzzy term by adding a `~` followed by an optional maximum edit distance. If you don't specify an edit distance, the default is `1`.

For example, the following "fuzzy" term query:

```
cat~
```

would match `cat` and all terms in the index within one "edit" of cat, for example `cast` (insert `s`), `at` (delete `c`), and `act` (transpose `c` and `a`).

If you wanted `cat` to match `bat`, it requires two edits (delete `c` and insert `b`) so you would need to set the maximum edit distance to `2`:

```
cat~2
```

Because each additional edit you allow increases the number of possibilities that must be checked, edit distances greater than `2` can be very slow.

It is often useful to require that the first few characters of a fuzzy term match exactly. This is called a prefix. You can set the length of the prefix by adding a slash and a number after the edit distance. For example, to use a maximum edit distance of `2` and a prefix length of `3`:

```
johannson~2/3
```

You can specify a prefix without specifying an edit distance:

```
johannson~/3
```

The default prefix distance is `0`.

Allowing complex phrase queries

The default parser setup allows phrase (proximity) queries such as:

```
"whoosh search library"
```

The default phrase query tokenizes the text between the quotes and creates a search for those terms in proximity.

If you want to do more complex proximity searches, you can replace the phrase plugin with the `whoosh.qparser.SequencePlugin`, which allows any query between the quotes. For example:

```
"(john OR jon OR jonathan~) peters*"
```

The sequence syntax lets you add a "slop" factor just like the regular phrase:

```
"(john OR jon OR jonathan~) peters*"~2
```

To replace the default phrase plugin with the sequence plugin:

```
parser = qparser.QueryParser("fieldname", my_index.schema)
parser.remove_plugin_class(qparser.PhrasePlugin)
parser.add_plugin(qparser.SequencePlugin())
```

Alternatively, you could keep the default phrase plugin and give the sequence plugin different syntax by specifying a regular expression for the start/end marker when you create the sequence plugin. The regular expression should have a named group `slop` for the slop factor. For example:

```
parser = qparser.QueryParser("fieldname", my_index.schema)
parser.add_plugin(qparser.SequencePlugin("!(~(?P<slop>[1-9][0-9]*))?"))
```

This would allow you to use regular phrase queries and sequence queries at the same time:

```
"regular phrase" AND !sequence query~2!
```

Advanced customization

QueryParser arguments

QueryParser supports two extra keyword arguments:

`group` The query class to use to join sub-queries when the user doesn't explicitly specify a boolean operator, such as `AND` or `OR`. This lets you change the default operator from `AND` to `OR`.

This will be the *`whoosh.qparser.AndGroup`* or *`whoosh.qparser.OrGroup`* class (*not* an instantiated object) unless you've written your own custom grouping syntax you want to use.

termclass The query class to use to wrap single terms.

This must be a *whoosh.query.Query* subclass (*not* an instantiated object) that accepts a fieldname string and term text unicode string in its __init__ method. The default is *whoosh.query.Term*.

This is useful if you want to change the default term class to *whoosh.query.Variations*, or if you've written a custom term class you want the parser to use instead of the ones shipped with Whoosh.

```
>>> from whoosh.qparser import QueryParser, OrGroup
>>> orparser = QueryParser("content", schema=myschema, group=OrGroup)
```

Configuring plugins

The query parser's functionality is provided by a set of plugins. You can remove plugins to remove functionality, add plugins to add functionality, or replace default plugins with re-configured or rewritten versions.

The *whoosh.qparser.QueryParser.add_plugin()*, *whoosh.qparser.QueryParser.remove_plugin_class()*, and *whoosh.qparser.QueryParser.replace_plugin()* methods let you manipulate the plugins in a `QueryParser` object.

See *qparser module* for information about the available plugins.

Creating custom operators

- Decide whether you want a `PrefixOperator`, `PostfixOperator`, or `InfixOperator`.
- Create a new `whoosh.qparser.syntax.GroupNode` subclass to hold nodes affected by your operator. This object is responsible for generating a *whoosh.query.Query* object corresponding to the syntax.
- Create a regular expression pattern for the operator's query syntax.
- Create an `OperatorsPlugin.OpTagger` object from the above information.
- Create a new `OperatorsPlugin` instance configured with your custom operator(s).
- Replace the default `OperatorsPlugin` in your parser with your new instance.

For example, if you were creating a `BEFORE` operator:

```
from whoosh import qparser, query

optype = qparser.InfixOperator
pattern = " BEFORE "

class BeforeGroup(qparser.GroupNode):
    merging = True
    qclass = query.Ordered
```

Create an OpTagger for your operator:

```
btagger = qparser.OperatorPlugin.OpTagger(pattern, BeforeGroup,
                                          qparser.InfixOperator)
```

By default, infix operators are left-associative. To make a right-associative infix operator, do this:

```
btagger = qparser.OperatorPlugin.OpTagger(pattern, BeforeGroup,
                                          qparser.InfixOperator,
                                          leftassoc=False)
```

Create an `OperatorsPlugin` instance with your new operator, and replace the default operators plugin in your query parser:

```
qp = qparser.QueryParser("text", myschema)
my_op_plugin = qparser.OperatorsPlugin([(btagger, 0)])
qp.replace_plugin(my_op_plugin)
```

Note that the list of operators you specify with the first argument is IN ADDITION TO the default operators (AND, OR, etc.). To turn off one of the default operators, you can pass None to the corresponding keyword argument:

```
cp = qparser.OperatorsPlugin([(optagger, 0)], And=None)
```

If you want ONLY your list of operators and none of the default operators, use the `clean` keyword argument:

```
cp = qparser.OperatorsPlugin([(optagger, 0)], clean=True)
```

Operators earlier in the list bind more closely than operators later in the list.

The default query language

Overview

A query consists of *terms* and *operators*. There are two types of terms: single terms and *phrases*. Multiple terms can be combined with operators such as *AND* and *OR*.

Whoosh supports indexing text in different *fields*. You must specify the *default field* when you create the *whoosh.qparser.QueryParser* object. This is the field in which any terms the user does not explicitly specify a field for will be searched.

Whoosh's query parser is capable of parsing different and/or additional syntax through the use of plug-ins. See *Parsing user queries*.

Individual terms and phrases

Find documents containing the term `render`:

```
render
```

Find documents containing the phrase `all was well`:

```
"all was well"
```

Note that a field must store Position information for phrase searching to work in that field.

Normally when you specify a phrase, the maximum difference in position between each word in the phrase is 1 (that is, the words must be right next to each other in the document). For example, the following matches if a document has `library` within 5 words after `whoosh`:

```
"whoosh library"~5
```

Boolean operators

Find documents containing `render` *and* `shading`:

```
render AND shading
```

Note that AND is the default relation between terms, so this is the same as:

```
render shading
```

Find documents containing `render`, *and* also either `shading` *or* `modeling`:

```
render AND shading OR modeling
```

Find documents containing `render` but *not* modeling:

```
render NOT modeling
```

Find documents containing `alpha` but not either `beta` or `gamma`:

```
alpha NOT (beta OR gamma)
```

Note that when no boolean operator is specified between terms, the parser will insert one, by default AND. So this query:

```
render shading modeling
```

is equivalent (by default) to:

```
render AND shading AND modeling
```

See *customizing the default parser* for information on how to change the default operator to OR.

Group operators together with parentheses. For example to find documents that contain both `render` and `shading`, or contain `modeling`:

```
(render AND shading) OR modeling
```

Fields

Find the term `ivan` in the `name` field:

```
name:ivan
```

The `field:` prefix only sets the field for the term it directly precedes, so the query:

```
title:open sesame
```

Will search for `open` in the `title` field and `sesame` in the *default* field.

To apply a field prefix to multiple terms, group them with parentheses:

```
title:(open sesame)
```

This is the same as:

```
title:open title:sesame
```

Of course you can specify a field for phrases too:

```
title:"open sesame"
```

Inexact terms

Use "globs" (wildcard expressions using `?` to represent a single character and `*` to represent any number of characters) to match terms:

```
te?t test* *b?g*
```

Note that a wildcard starting with `?` or `*` is very slow. Note also that these wildcards only match *individual terms*. For example, the query:

```
my*life
```

will **not** match an indexed phrase like:

```
my so called life
```

because those are four separate terms.

Ranges

You can match a range of terms. For example, the following query will match documents containing terms in the lexical range from `apple` to `bear` *inclusive*. For example, it will match documents containing `azores` and `be` but not `blur`:

```
[apple TO bear]
```

This is very useful when you've stored, for example, dates in a lexically sorted format (i.e. YYYYMMDD):

```
date:[20050101 TO 20090715]
```

The range is normally *inclusive* (that is, the range will match all terms between the start and end term, *as well as* the start and end terms themselves). You can specify that one or both ends of the range are *exclusive* by using the `{` and/or `}` characters:

```
[0000 TO 0025}
{prefix TO suffix}
```

You can also specify *open-ended* ranges by leaving out the start or end term:

```
[0025 TO]
{TO suffix}
```

Boosting query elements

You can specify that certain parts of a query are more important for calculating the score of a matched document than others. For example, to specify that `ninja` is twice as important as other words, and `bear` is half as important:

```
ninja^2 cowboy bear^0.5
```

You can apply a boost to several terms using grouping parentheses:

```
(open sesame)^2.5 roc
```

Making a term from literal text

If you need to include characters in a term that are normally treated specially by the parser, such as spaces, colons, or brackets, you can enclose the term in single quotes:

```
path:'MacHD:My Documents'
'term with spaces'
title:'function()'
```

Indexing and parsing dates/times

Indexing dates

Whoosh lets you index and search dates/times using the *whoosh.fields.DATETIME* field type. Instead of passing text for the field in `add_document()`, you use a Python `datetime.datetime` object:

```
from datetime import datetime, timedelta
from whoosh import fields, index

schema = fields.Schema(title=fields.TEXT, content=fields.TEXT,
                       date=fields.DATETIME)
ix = index.create_in("indexdir", schema)

w = ix.writer()
w.add_document(title="Document 1", content="Rendering images from the command line",
               date=datetime.utcnow())
w.add_document(title="Document 2", content="Creating shaders using a node network",
               date=datetime.utcnow() + timedelta(days=1))
w.commit()
```

Parsing date queries

Once you've have an indexed `DATETIME` field, you can search it using a rich date parser contained in the `whoosh.qparser.dateparse.DateParserPlugin`:

```
from whoosh import index
from whoosh.qparser import QueryParser
from whoosh.qparser.dateparse import DateParserPlugin

ix = index.open_dir("indexdir")

# Instatiate a query parser
qp = QueryParser("content", ix.schema)

# Add the DateParserPlugin to the parser
qp.add_plugin(DateParserPlugin())
```

With the `DateParserPlugin`, users can use date queries such as:

```
20050912
2005 sept 12th
june 23 1978
23 mar 2005
july 1985
sep 12
today
yesterday
tomorrow
now
next friday
last tuesday
5am
10:25:54
23:12
8 PM
4:46 am oct 31 2010
last tuesday to today
today to next friday
jan 2005 to feb 2008
-1 week to now
now to +2h
-1y6mo to +2 yrs 23d
```

Normally, as with other types of queries containing spaces, the users need to quote date queries containing spaces using single quotes:

```
render date:'last tuesday' command
date:['last tuesday' to 'next friday']
```

If you use the `free` argument to the `DateParserPlugin`, the plugin will try to parse dates from unquoted text following a date field prefix:

```
qp.add_plugin(DateParserPlugin(free=True))
```

This allows the user to type a date query with spaces and special characters following the name of date field and a colon. The date query can be mixed with other types of queries without quotes:

```
date:last tuesday
render date:oct 15th 2001 5:20am command
```

If you don't use the `DateParserPlugin`, users can still search DATETIME fields using a simple numeric form `YYYY[MM[DD[hh[mm[ss]]]]]` that is built into the `DATETIME` field:

```
from whoosh import index
from whoosh.qparser import QueryParser

ix = index.open_dir("indexdir")
qp = QueryParser("content", schema=ix.schema)

# Find all datetimes in 2005
q = qp.parse(u"date:2005")

# Find all datetimes on June 24, 2005
q = qp.parse(u"date:20050624")

# Find all datetimes from 1am-2am on June 24, 2005
q = qp.parse(u"date:2005062401")
```

```
# Find all datetimes from Jan 1, 2005 to June 2, 2010
q = qp.parse(u"date:[20050101 to 20100602]")
```

About time zones and basetime

The best way to deal with time zones is to always index `datetimes` in native UTC form. Any `tzinfo` attribute on the `datetime` object is *ignored* by the indexer. If you are working with local datetimes, you should convert them to native UTC datetimes before indexing.

Date parser notes

Please note that the date parser is still somewhat experimental.

Setting the base datetime

When you create the `DateParserPlugin` you can pass a `datetime` object to the `basedate` argument to set the datetime against which relative queries (such as `last tuesday` and `-2 hours`) are measured. By default, the basedate is `datetime.utcnow()` at the moment the plugin is instantiated:

```
qp.add_plugin(DateParserPlugin(basedate=my_datetime))
```

Registering an error callback

To avoid user queries causing exceptions in your application, the date parser attempts to fail silently when it can't parse a date query. However, you can register a callback function to be notified of parsing failures so you can display feedback to the user. The argument to the callback function is the date text that could not be parsed (this is an experimental feature and may change in future versions):

```
errors = []
def add_error(msg):
    errors.append(msg)
qp.add_plugin(DateParserPlug(callback=add_error))

q = qp.parse(u"date:blarg")
# errors == [u"blarg"]
```

Using free parsing

While the `free` option is easier for users, it may result in ambiguities. As one example, if you want to find documents containing reference to a march and the number 2 in documents from the year 2005, you might type:

```
date:2005 march 2
```

This query would be interpreted correctly as a date query and two term queries when `free=False`, but as a single date query when `free=True`. In this case the user could limit the scope of the date parser with single quotes:

```
date:'2005' march 2
```

Parsable formats

The date parser supports a wide array of date and time formats, however it is not my intention to try to support *all* types of human-readable dates (for example `ten to five the friday after next`). The best idea might be to pick a date format that works and try to train users on it, and if they use one of the other formats that also works consider it a happy accident.

Limitations

- Since it's based on Python's `datetime.datetime` object, the `DATETIME` field shares all the limitations of that class, such as no support for dates before year 1 on the proleptic Gregorian calendar. The `DATETIME` field supports practically unlimited dates, so if the `datetime` object is every improved it could support it. An alternative possibility might be to add support for `mxDateTime` objects someday.
- The `DateParserPlugin` currently only has support for English dates. The architecture supports creation of parsers for other languages, and I hope to add examples for other languages soon.
- `DATETIME` fields do not currently support open-ended ranges. You can simulate an open ended range by using an endpoint far in the past or future.

Query objects

The classes in the *`whoosh.query`* module implement *queries* you can run against the index.

TBD.

See *How to search* for how to search the index using query objects.

About analyzers

Overview

An analyzer is a function or callable class (a class with a `__call__` method) that takes a unicode string and returns a generator of tokens. Usually a "token" is a word, for example the string "Mary had a little lamb" might yield the tokens "Mary", "had", "a", "little", and "lamb". However, tokens do not necessarily correspond to words. For example, you might tokenize Chinese text into individual characters or bi-grams. Tokens are the units of indexing, that is, they are what you are able to look up in the index.

An analyzer is basically just a wrapper for a tokenizer and zero or more filters. The analyzer's `__call__` method will pass its parameters to a tokenizer, and the tokenizer will usually be wrapped in a few filters.

A tokenizer is a callable that takes a unicode string and yields a series of `analysis.Token` objects.

For example, the provided *`whoosh.analysis.RegexTokenizer`* class implements a customizable, regular-expression-based tokenizer that extracts words and ignores whitespace and punctuation.

```
>>> from whoosh.analysis import RegexTokenizer
>>> tokenizer = RegexTokenizer()
>>> for token in tokenizer(u"Hello there my friend!"):
...   print repr(token.text)
u'Hello'
u'there'
u'my'
u'friend'
```

A filter is a callable that takes a generator of Tokens (either a tokenizer or another filter) and in turn yields a series of Tokens.

For example, the provided *whoosh.analysis.LowercaseFilter()* filters tokens by converting their text to lowercase. The implementation is very simple:

```
def LowercaseFilter(tokens):
    """Uses lower() to lowercase token text. For example, tokens
    "This","is","a","TEST" become "this","is","a","test".
    """

    for t in tokens:
        t.text = t.text.lower()
        yield t
```

You can wrap the filter around a tokenizer to see it in operation:

```
>>> from whoosh.analysis import LowercaseFilter
>>> for token in LowercaseFilter(tokenizer(u"These ARE the things I want!")):
...   print repr(token.text)
u'these'
u'are'
u'the'
u'things'
u'i'
u'want'
```

An analyzer is just a means of combining a tokenizer and some filters into a single package.

You can implement an analyzer as a custom class or function, or compose tokenizers and filters together using the `|` character:

```
my_analyzer = RegexTokenizer() | LowercaseFilter() | StopFilter()
```

The first item must be a tokenizer and the rest must be filters (you can't put a filter first or a tokenizer after the first item). Note that this only works if at least the tokenizer is a subclass of `whoosh.analysis.Composable`, as all the tokenizers and filters that ship with Whoosh are.

See the *whoosh.analysis* module for information on the available analyzers, tokenizers, and filters shipped with Whoosh.

Using analyzers

When you create a field in a schema, you can specify your analyzer as a keyword argument to the field object:

```
schema = Schema(content=TEXT(analyzer=StemmingAnalyzer()))
```

Advanced Analysis

Token objects

The `Token` class has no methods. It is merely a place to record certain attributes. A `Token` object actually has two kinds of attributes: *settings* that record what kind of information the `Token` object does or should contain, and *information* about the current token.

Token setting attributes

A `Token` object should always have the following attributes. A tokenizer or filter can check these attributes to see what kind of information is available and/or what kind of information they should be setting on the `Token` object.

These attributes are set by the tokenizer when it creates the Token(s), based on the parameters passed to it from the Analyzer.

Filters **should not** change the values of these attributes.

Type	Attribute name	Description	Default
str	mode	The mode in which the analyzer is being called, e.g. 'index' during indexing or 'query' during query parsing	''
bool	positions	Whether term positions are recorded in the token	False
bool	chars	Whether term start and end character indices are recorded in the token	False
bool	boosts	Whether per-term boosts are recorded in the token	False
bool	removestops	Whether stop-words should be removed from the token stream	True

Token information attributes

A `Token` object may have any of the following attributes. The `text` attribute should always be present. The original attribute may be set by a tokenizer. All other attributes should only be accessed or set based on the values of the "settings" attributes above.

Type	Name	Description
unicode	text	The text of the token (this should always be present)
unicode	original	The original (pre-filtered) text of the token. The tokenizer may record this, and filters are expected not to modify it.
int	pos	The position of the token in the stream, starting at 0 (only set if positions is True)
int	startchar	The character index of the start of the token in the original string (only set if chars is True)
int	endchar	The character index of the end of the token in the original string (only set if chars is True)
float	boost	The boost for this token (only set if boosts is True)
bool	stopped	Whether this token is a "stop" word (only set if removestops is False)

So why are most of the information attributes optional? Different field formats require different levels of information about each token. For example, the `Frequency` format only needs the token text. The `Positions` format records term positions, so it needs them on the `Token`. The `Characters` format records term positions and the start and end character indices of each term, so it needs them on the token, and so on.

The `Format` object that represents the format of each field calls the analyzer for the field, and passes it parameters corresponding to the types of information it needs, e.g.:

```
analyzer(unicode_string, positions=True)
```

The analyzer can then pass that information to a tokenizer so the tokenizer initializes the required attributes on the `Token` object(s) it produces.

Performing different analysis for indexing and query parsing

Whoosh sets the `mode` setting attribute to indicate whether the analyzer is being called by the indexer (`mode='index'`) or the query parser (`mode='query'`). This is useful if there's a transformation that you only

want to apply at indexing or query parsing:

```
class MyFilter(Filter):
    def __call__(self, tokens):
        for t in tokens:
            if t.mode == 'query':
                ...
            else:
                ...
```

The *whoosh.analysis.MultiFilter* filter class lets you specify different filters to use based on the mode setting:

```
intraword = MultiFilter(index=IntraWordFilter(mergewords=True, mergenums=True),
                        query=IntraWordFilter(mergewords=False, mergenums=False))
```

Stop words

"Stop" words are words that are so common it's often counter-productive to index them, such as "and", "or", "if", etc. The provided `analysis.StopFilter` lets you filter out stop words, and includes a default list of common stop words.

```
>>> from whoosh.analysis import StopFilter
>>> stopper = StopFilter()
>>> for token in stopper(LowercaseFilter(tokenizer(u"These ARE the things I want!"))):
...   print repr(token.text)
u'these'
u'things'
u'want'
```

However, this seemingly simple filter idea raises a couple of minor but slightly thorny issues: renumbering term positions and keeping or removing stopped words.

Renumbering term positions

Remember that analyzers are sometimes asked to record the position of each token in the token stream:

Token.text	u'Mary'	u'had'	u'a'	u'lamb'
Token.pos	0	1	2	3

So what happens to the `pos` attribute of the tokens if `StopFilter` removes the words `had` and `a` from the stream? Should it renumber the positions to pretend the "stopped" words never existed? I.e.:

Token.text	u'Mary'	u'lamb'
Token.pos	0	1

or should it preserve the original positions of the words? I.e:

Token.text	u'Mary'	u'lamb'
Token.pos	0	3

It turns out that different situations call for different solutions, so the provided `StopFilter` class supports both of the above behaviors. Renumbering is the default, since that is usually the most useful and is necessary to support phrase searching. However, you can set a parameter in StopFilter's constructor to tell it not to renumber positions:

```
stopper = StopFilter(renumber=False)
```

Removing or leaving stop words

The point of using `StopFilter` is to remove stop words, right? Well, there are actually some situations where you might want to mark tokens as "stopped" but not remove them from the token stream.

For example, if you were writing your own query parser, you could run the user's query through a field's analyzer to break it into tokens. In that case, you might want to know which words were "stopped" so you can provide helpful feedback to the end user (e.g. "The following words are too common to search for:").

In other cases, you might want to leave stopped words in the stream for certain filtering steps (for example, you might have a step that looks at previous tokens, and want the stopped tokens to be part of the process), but then remove them later.

The `analysis` module provides a couple of tools for keeping and removing stop-words in the stream.

The `removestops` parameter passed to the analyzer's `__call__` method (and copied to the `Token` object as an attribute) specifies whether stop words should be removed from the stream or left in.

```
>>> from whoosh.analysis import StandardAnalyzer
>>> analyzer = StandardAnalyzer()
>>> [(t.text, t.stopped) for t in analyzer(u"This is a test")]
[(u'test', False)]
>>> [(t.text, t.stopped) for t in analyzer(u"This is a test", removestops=False)]
[(u'this', True), (u'is', True), (u'a', True), (u'test', False)]
```

The `analysis.unstopped()` filter function takes a token generator and yields only the tokens whose `stopped` attribute is `False`.

Note: Even if you leave stopped words in the stream in an analyzer you use for indexing, the indexer will ignore any tokens where the `stopped` attribute is `True`.

Implementation notes

Because object creation is slow in Python, the stock tokenizers do not create a new `analysis.Token` object for each token. Instead, they create one `Token` object and yield it over and over. This is a nice performance shortcut but can lead to strange behavior if your code tries to remember tokens between loops of the generator.

Because the analyzer only has one `Token` object, of which it keeps changing the attributes, if you keep a copy of the Token you get from a loop of the generator, it will be changed from under you. For example:

```
>>> list(tokenizer(u"Hello there my friend"))
[Token(u"friend"), Token(u"friend"), Token(u"friend"), Token(u"friend")]
```

Instead, do this:

```
>>> [t.text for t in tokenizer(u"Hello there my friend")]
```

That is, save the attributes, not the token object itself.

If you implement your own tokenizer, filter, or analyzer as a class, you should implement an `__eq__` method. This is important to allow comparison of `Schema` objects.

The mixing of persistent "setting" and transient "information" attributes on the `Token` object is not especially elegant. If I ever have a better idea I might change it. ;) Nothing requires that an Analyzer be implemented by calling a tokenizer and filters. Tokenizers and filters are simply a convenient way to structure the code. You're free to write an analyzer any way you want, as long as it implements `__call__`.

Stemming, variations, and accent folding

The problem

The indexed text will often contain words in different form than the one the user searches for. For example, if the user searches for `render`, we would like the search to match not only documents that contain the `render`, but also `renders`, `rendering`, `rendered`, etc.

A related problem is one of accents. Names and loan words may contain accents in the original text but not in the user's query, or vice versa. For example, we want the user to be able to search for `cafe` and find documents containing `café`.

The default analyzer for the *whoosh.fields.TEXT* field does not do stemming or accent folding.

Stemming

Stemming is a heuristic process of removing suffixes (and sometimes prefixes) from words to arrive (hopefully, most of the time) at the base word. Whoosh includes several stemming algorithms such as Porter and Porter2, Paice Husk, and Lovins.

```
>>> from whoosh.lang.porter import stem
>>> stem("rendering")
'render'
```

The stemming filter applies the stemming function to the terms it indexes, and to words in user queries. So in theory all variations of a root word ("render", "rendered", "renders", "rendering", etc.) are reduced to a single term in the index, saving space. And all possible variations users might use in a query are reduced to the root, so stemming enhances "recall".

The *whoosh.analysis.StemFilter* lets you add a stemming filter to an analyzer chain.

```
>>> rext = RegexTokenizer()
>>> stream = rext(u"fundamentally willows")
>>> stemmer = StemFilter()
>>> [token.text for token in stemmer(stream)]
[u"fundament", u"willow"]
```

The *whoosh.analysis.StemmingAnalyzer()* is a pre-packaged analyzer that combines a tokenizer, lowercase filter, optional stop filter, and stem filter:

```
from whoosh import fields
from whoosh.analysis import StemmingAnalyzer

stem_ana = StemmingAnalyzer()
schema = fields.Schema(title=TEXT(analyzer=stem_ana, stored=True),
                       content=TEXT(analyzer=stem_ana))
```

Stemming has pros and cons.

- It allows the user to find documents without worrying about word forms.
- It reduces the size of the index, since it reduces the number of separate terms indexed by "collapsing" multiple word forms into a single base word.
- It's faster than using variations (see below)
- The stemming algorithm can sometimes incorrectly conflate words or change the meaning of a word by removing suffixes.

- The stemmed forms are often not proper words, so the terms in the field are not useful for things like creating a spelling dictionary.

Variations

Whereas stemming encodes the words in the index in a base form, when you use variations you instead index words "as is" and *at query time* expand words in the user query using a heuristic algorithm to generate morphological variations of the word.

```
>>> from whoosh.lang.morph_en import variations
>>> variations("rendered")
set(['rendered', 'rendernesses', 'render', 'renderless', 'rendering',
'renderness', 'renderes', 'renderer', 'renderements', 'rendereless',
'renderenesses', 'rendere', 'renderment', 'renderest', 'renderement',
'rendereful', 'renderers', 'renderful', 'renderings', 'renders', 'renderly',
'renderely', 'rendereness', 'renderments'])
```

Many of the generated variations for a given word will not be valid words, but it's fairly fast for Whoosh to check which variations are actually in the index and only search for those.

The *whoosh.query.Variations* query object lets you search for variations of a word. Whereas the normal *whoosh.query.Term* object only searches for the given term, the `Variations` query acts like an `Or` query for the variations of the given word in the index. For example, the query:

```
query.Variations("content", "rendered")
```

...might act like this (depending on what words are in the index):

```
query.Or([query.Term("content", "render"), query.Term("content", "rendered"),
          query.Term("content", "renders"), query.Term("content", "rendering")])
```

To have the query parser use *whoosh.query.Variations* instead of *whoosh.query.Term* for individual terms, use the `termclass` keyword argument to the parser initialization method:

```
from whoosh import qparser, query

qp = qparser.QueryParser("content", termclass=query.Variations)
```

Variations has pros and cons.

- It allows the user to find documents without worrying about word forms.
- The terms in the field are actual words, not stems, so you can use the field's contents for other purposes such as spell checking queries.
- It increases the size of the index relative to stemming, because different word forms are indexed separately.
- It acts like an `Or` search for all the variations, which is slower than searching for a single term.

Lemmatization

Whereas stemming is a somewhat "brute force", mechanical attempt at reducing words to their base form using simple rules, lemmatization usually refers to more sophisticated methods of finding the base form ("lemma") of a word using language models, often involving analysis of the surrounding context and part-of-speech tagging.

Whoosh does not include any lemmatization functions, but if you have separate lemmatizing code you could write a custom `whoosh.analysis.Filter` to integrate it into a Whoosh analyzer.

Character folding

You can set up an analyzer to treat, for example, á, a, å, and â as equivalent to improve recall. This is often very useful, allowing the user to, for example, type `cafe` or `resume` and find documents containing `café` and `resumé`.

Character folding is especially useful for unicode characters that may appear in Asian language texts that should be treated as equivalent to their ASCII equivalent, such as "half-width" characters.

Character folding is not always a panacea. See this article for caveats on where accent folding can break down.

http://www.alistapart.com/articles/accent-folding-for-auto-complete/

Whoosh includes several mechanisms for adding character folding to an analyzer.

The *whoosh.analysis.CharsetFilter* applies a character map to token text. For example, it will filter the tokens `u'café', u'resumé', ...` to `u'cafe', u'resume', ...`. This is usually the method you'll want to use unless you need to use a charset to tokenize terms:

```
from whoosh.analysis import CharsetFilter, StemmingAnalyzer
from whoosh import fields
from whoosh.support.charset import accent_map

# For example, to add an accent-folding filter to a stemming analyzer:
my_analyzer = StemmingAnalyzer() | CharsetFilter(accent_map)

# To use this analyzer in your schema:
my_schema = fields.Schema(content=fields.TEXT(analyzer=my_analyzer))
```

The *whoosh.analysis.CharsetTokenizer* uses a Sphinx charset table to both separate terms and perform character folding. This tokenizer is slower than the *whoosh.analysis.RegexTokenizer* because it loops over each character in Python. If the language(s) you're indexing can be tokenized using regular expressions, it will be much faster to use `RegexTokenizer` and `CharsetFilter` in combination instead of using `CharsetTokenizer`.

The *whoosh.support.charset* module contains an accent folding map useful for most Western languages, as well as a much more extensive Sphinx charset table and a function to convert Sphinx charset tables into the character maps required by `CharsetTokenizer` and `CharsetFilter`:

```
# To create a filter using an enourmous character map for most languages
# generated from a Sphinx charset table
from whoosh.analysis import CharsetFilter
from whoosh.support.charset import default_charset, charset_table_to_dict
charmap = charset_table_to_dict(default_charset)
my_analyzer = StemmingAnalyzer() | CharsetFilter(charmap)
```

(The Sphinx charset table format is described at http://www.sphinxsearch.com/docs/current.html#conf-charset-table)

Indexing and searching N-grams

Overview

N-gram indexing is a powerful method for getting fast, "search as you type" functionality like iTunes. It is also useful for quick and effective indexing of languages such as Chinese and Japanese without word breaks.

N-grams refers to groups of N characters... bigrams are groups of two characters, trigrams are groups of three characters, and so on.

Whoosh includes two methods for analyzing N-gram fields: an N-gram tokenizer, and a filter that breaks tokens into N-grams.

whoosh.analysis.NgramTokenizer tokenizes the entire field into N-grams. This is more useful for Chinese/Japanese/Korean languages, where it's useful to index bigrams of characters rather than individual characters. Using this tokenizer with roman languages leads to spaces in the tokens.

```
>>> ngt = NgramTokenizer(minsize=2, maxsize=4)
>>> [token.text for token in ngt(u"hi there")]
[u'hi', u'hi ', u'hi t',u'i ', u'i t', u'i th', u' t', u' th', u' the', u'th',
u'the', u'ther', u'he', u'her', u'here', u'er', u'ere', u're']
```

whoosh.analysis.NgramFilter breaks individual tokens into N-grams as part of an analysis pipeline. This is more useful for languages with word separation.

```
>>> my_analyzer = StandardAnalyzer() | NgramFilter(minsize=2, maxsize=4)
>>> [token.text for token in my_analyzer(u"rendering shaders")]
[u'ren', u'rend', u'end', u'ende', u'nde', u'nder', u'der', u'deri', u'eri',
u'erin', u'rin', u'ring', u'ing', u'sha', u'shad', u'had', u'hade', u'ade',
u'ader', u'der', u'ders', u'ers']
```

Whoosh includes two pre-configured field types for N-grams: *whoosh.fields.NGRAM* and *whoosh.fields.NGRAMWORDS*. The only difference is that `NGRAM` runs all text through the N-gram filter, including whitespace and punctuation, while `NGRAMWORDS` extracts words from the text using a tokenizer, then runs each word through the N-gram filter.

TBD.

Sorting and faceting

Note: The API for sorting and faceting changed in Whoosh 3.0.

Overview

Sorting and faceting search results in Whoosh is based on **facets**. Each facet associates a value with each document in the search results, allowing you to sort by the keys or use them to group the documents. Whoosh includes a variety of **facet types** you can use for sorting and grouping (see below).

Sorting

By default, the results of a search are sorted with the highest-scoring documents first. You can use the `sortedby` keyword argument to order the results by some other criteria instead, such as the value of a field.

Making fields sortable

In order to sort on a field, you should create the field using the `sortable=True` keyword argument:

```
schema = fields.Schema(title=fields.TEXT(sortable=True),
                       content=fields.TEXT,
                       modified=fields.DATETIME(sortable=True)
                       )
```

It's possible to sort on a field that doesn't have `sortable=True`, but this requires Whoosh to load the unique terms in the field into memory. Using `sortable` is much more efficient.

About column types

When you create a field using `sortable=True`, you are telling Whoosh to store per-document values for that field in a *column*. A column object specifies the format to use to store the per-document values on disk.

The *whoosh.columns* module contains several different column object implementations. Each field type specifies a reasonable default column type (for example, the default for text fields is *whoosh.columns.VarBytesColumn*, the default for numeric fields is *whoosh.columns.NumericColumn*). However, if you want maximum efficiency you may want to use a different column type for a field.

For example, if all document values in a field are a fixed length, you can use a *whoosh.columns.FixedBytesColumn*. If you have a field where many documents share a relatively small number of possible values (an example might be a "category" field, or "month" or other enumeration type fields), you might want to use *whoosh.columns.RefBytesColumn* (which can handle both variable and fixed-length values). There are column types for storing per-document bit values, structs, pickled objects, and compressed byte values.

To specify a custom column object for a field, pass it as the `sortable` keyword argument instead of `True`:

```
from whoosh import columns, fields

category_col = columns.RefBytesColumn()
schema = fields.Schema(title=fields.TEXT(sortable=True),
                       category=fields.KEYWORD(sortable=category_col)
```

Using a COLUMN field for custom sort keys

When you add a document with a sortable field, Whoosh uses the value you pass for the field as the sortable value. For example, if "title" is a sortable field, and you add this document:

```
writer.add_document(title="Mr. Palomar")
```

...then `Mr. Palomar` is stored in the field column as the sorting key for the document.

This is usually good, but sometimes you need to "massage" the sortable key so it's different from the value the user searches and/or sees in the interface. For example, if you allow the user to sort by title, you might want to use different values for the visible title and the value used for sorting:

```
# Visible title
title = "The Unbearable Lightness of Being"

# Sortable title: converted to lowercase (to prevent different ordering
# depending on uppercase/lowercase), with initial article moved to the end
sort_title = "unbearable lightness of being, the"
```

The best way to do this is to use an additional field just for sorting. You can use the `whoosh.fields.COLUMN` field type to create a field that is not indexed or stored, it only holds per-document column values:

```
schema = fields.Schema(title=fields.TEXT(stored=True),
                       sort_title=fields.COLUMN(columns.VarBytesColumn())
                       )
```

The single argument to the `whoosh.fields.COLUMN` initializer is a `whoosh.columns.ColumnType` object. You can use any of the various column types in the *whoosh.columns* module.

As another example, say you are indexing documents that have a custom sorting order associated with each document, such as a "priority" number:

```
name=Big Wheel
price=100
priority=1

name=Toss Across
price=40
priority=3

name=Slinky
price=25
priority=2
...
```

You can use a column field with a numeric column object to hold the "priority" and use it for sorting:

```
schema = fields.Schema(name=fields.TEXT(stored=True),
                       price=fields.NUMERIC(stored=True),
                       priority=fields.COLUMN(columns.NumericColumn("i"),
                       )
```

(Note that `columns.NumericColumn` takes a type code character like the codes used by Python's `struct` and `array` modules.)

Making existing fields sortable

If you have an existing index from before the `sortable` argument was added in Whoosh 3.0, or you didn't think you needed a field to be sortable but now you find that you need to sort it, you can add "sortability" to an existing index using the `whoosh.sorting.add_sortable()` utility function:

```
from whoosh import columns, fields, index, sorting

# Say we have an existing index with this schema
schema = fields.Schema(title=fields.TEXT,
                       price=fields.NUMERIC)

# To use add_sortable, first open a writer for the index
ix = index.open_dir("indexdir")
with ix.writer() as w:
    # Add sortable=True to the "price" field using field terms as the
    # sortable values
    sorting.add_sortable(w, "price", sorting.FieldFacet("price"))

    # Add sortable=True to the "title" field using the
    # stored field values as the sortable value
    sorting.add_sortable(w, "title", sorting.StoredFieldFacet("title"))
```

You can specify a custom column type when you call `add_sortable` using the `column` keyword argument:

```
add_sortable(w, "chapter", sorting.FieldFacet("chapter"),
             column=columns.RefBytesColumn())
```

See the documentation for `add_sortable()` for more information.

Sorting search results

When you tell Whoosh to sort by a field (or fields), it uses the per-document values in the field's column as sorting keys for the documents.

Normally search results are sorted by descending relevance score. You can tell Whoosh to use a different ordering by passing the `sortedby` keyword argument to the *`search()`* method:

```
from whoosh import fields, index, qparser

schema = fields.Schema(title=fields.TEXT(stored=True),
                       price=fields.NUMERIC(sortable=True))
ix = index.create_in("indexdir", schema)

with ix.writer() as w:
    w.add_document(title="Big Deal", price=20)
    w.add_document(title="Mr. Big", price=10)
    w.add_document(title="Big Top", price=15)

with ix.searcher() as s:
    qp = qparser.QueryParser("big", ix.schema)
    q = qp.parse(user_query_string)

    # Sort search results from lowest to highest price
    results = s.search(q, sortedby="price")
    for hit in results:
        print(hit["title"])
```

You can use any of the following objects as `sortedby` values:

A `FacetType` object Uses this object to sort the documents. See below for the available facet types.

A field name string Converts the field name into a `FieldFacet` (see below) and uses it to sort the documents.

A list of `FacetType` objects and/or field name strings Bundles the facets together into a `MultiFacet` so you can sort by multiple keys. Note that this shortcut does not allow you to reverse the sort direction of individual facets. To do that, you need to construct the `MultiFacet` object yourself.

Note: You can use the `reverse=True` keyword argument to the `Searcher.search()` method to reverse the overall sort direction. This is more efficient than reversing each individual facet.

Examples

Sort by the value of the size field:

```
results = searcher.search(myquery, sortedby="size")
```

Sort by the reverse (highest-to-lowest) order of the "price" field:

```
facet = sorting.FieldFacet("price", reverse=True)
results = searcher.search(myquery, sortedby=facet)
```

Sort by ascending size and then descending price:

```
mf = sorting.MultiFacet()
mf.add_field("size")
```

```
mf.add_field("price", reverse=True)
results = searcher.search(myquery, sortedby=mf)

# or...
sizes = sorting.FieldFacet("size")
prices = sorting.FieldFacet("price", reverse=True)
results = searcher.search(myquery, sortedby=[sizes, prices])
```

Sort by the "category" field, then by the document's score:

```
cats = sorting.FieldFacet("category")
scores = sorting.ScoreFacet()
results = searcher.search(myquery, sortedby=[cats, scores])
```

Accessing column values

Per-document column values are available in *Hit* objects just like stored field values:

```
schema = fields.Schema(title=fields.TEXT(stored=True),
                       price=fields.NUMERIC(sortable=True))

...

results = searcher.search(myquery)
for hit in results:
    print(hit["title"], hit["price"])
```

ADVANCED: if you want to access abitrary per-document values quickly you can get a column reader object:

```
with ix.searcher() as s:
    reader = s.reader()

    colreader = s.reader().column_reader("price")
    for docnum in reader.all_doc_ids():
        print(colreader[docnum])
```

Grouping

It is often very useful to present "faceted" search results to the user. Faceting is dynamic grouping of search results into categories. The categories let users view a slice of the total results based on the categories they're interested in.

For example, if you are programming a shopping website, you might want to display categories with the search results such as the manufacturers and price ranges.

Manufacturer	Price
Apple (5)	$0 - $100 (2)
Sanyo (1)	$101 - $500 (10)
Sony (2)	$501 - $1000 (1)
Toshiba (5)	

You can let your users click the different facet values to only show results in the given categories.

Another useful UI pattern is to show, say, the top 5 results for different types of found documents, and let the user click to see more results from a category they're interested in, similarly to how the Spotlight quick results work on Mac OS X.

The `groupedby` keyword argument

You can use the following objects as `groupedby` values:

A `FacetType` object Uses this object to group the documents. See below for the available facet types.

A field name string Converts the field name into a `FieldFacet` (see below) and uses it to sort the documents. The name of the field is used as the facet name.

A list or tuple of field name strings Sets up multiple field grouping criteria.

A dictionary mapping facet names to `FacetType` objects Sets up multiple grouping criteria.

A `Facets` object This object is a lot like using a dictionary, but has some convenience methods to make setting up multiple groupings a little easier.

Examples

Group by the value of the "category" field:

```
results = searcher.search(myquery, groupedby="category")
```

Group by the value of the "category" field and also by the value of the "tags" field and a date range:

```
cats = sorting.FieldFacet("category")
tags = sorting.FieldFacet("tags", allow_overlap=True)
results = searcher.search(myquery, groupedby={"category": cats, "tags": tags})

# ...or, using a Facets object has a little less duplication
facets = sorting.Facets()
facets.add_field("category")
facets.add_field("tags", allow_overlap=True)
results = searcher.search(myquery, groupedby=facets)
```

To group results by the *intersected values of multiple fields*, use a `MultiFacet` object (see below). For example, if you have two fields named `tag` and `size`, you could group the results by all combinations of the `tag` and `size` field, such as `('tag1', 'small')`, `('tag2', 'small')`, `('tag1', 'medium')`, and so on:

```
# Generate a grouping from the combination of the "tag" and "size" fields
mf = MultiFacet(["tag", "size"])
results = searcher.search(myquery, groupedby={"tag/size": mf})
```

Getting the faceted groups

The `Results.groups("facetname")` method returns a dictionary mapping category names to lists of **document IDs**:

```
myfacets = sorting.Facets().add_field("size").add_field("tag")
results = mysearcher.search(myquery, groupedby=myfacets)
results.groups("size")
# {"small": [8, 5, 1, 2, 4], "medium": [3, 0, 6], "large": [7, 9]}
```

If there is only one facet, you can just use `Results.groups()` with no argument to access its groups:

```
results = mysearcher.search(myquery, groupedby=myfunctionfacet)
results.groups()
```

By default, the values in the dictionary returned by `groups()` are lists of document numbers in the same relative order as in the results. You can use the `Searcher` object's `stored_fields()` method to take a document number and return the document's stored fields as a dictionary:

```
for category_name in categories:
    print "Top 5 documents in the %s category" % category_name
    doclist = categories[category_name]
    for docnum, score in doclist[:5]:
        print "  ", searcher.stored_fields(docnum)
    if len(doclist) > 5:
        print "  (%s more)" % (len(doclist) - 5)
```

If you want different information about the groups, for example just the count of documents in each group, or you don't need the groups to be ordered, you can specify a *whoosh.sorting.FacetMap* type or instance with the `maptype` keyword argument when creating the `FacetType`:

```
# This is the same as the default
myfacet = FieldFacet("size", maptype=sorting.OrderedList)
results = mysearcher.search(myquery, groupedby=myfacet)
results.groups()
# {"small": [8, 5, 1, 2, 4], "medium": [3, 0, 6], "large": [7, 9]}

# Don't sort the groups to match the order of documents in the results
# (faster)
myfacet = FieldFacet("size", maptype=sorting.UnorderedList)
results = mysearcher.search(myquery, groupedby=myfacet)
results.groups()
# {"small": [1, 2, 4, 5, 8], "medium": [0, 3, 6], "large": [7, 9]}

# Only count the documents in each group
myfacet = FieldFacet("size", maptype=sorting.Count)
results = mysearcher.search(myquery, groupedby=myfacet)
results.groups()
# {"small": 5, "medium": 3, "large": 2}

# Only remember the "best" document in each group
myfacet = FieldFacet("size", maptype=sorting.Best)
results = mysearcher.search(myquery, groupedby=myfacet)
results.groups()
# {"small": 8, "medium": 3, "large": 7}
```

Alternatively you can specify a `maptype` argument in the `Searcher.search()` method call which applies to all facets:

```
results = mysearcher.search(myquery, groupedby=["size", "tag"],
                            maptype=sorting.Count)
```

(You can override this overall `maptype` argument on individual facets by specifying the `maptype` argument for them as well.)

Facet types

FieldFacet

This is the most common facet type. It sorts or groups based on the value in a certain field in each document. This generally works best (or at all) if each document has only one term in the field (e.g. an ID field):

```
# Sort search results by the value of the "path" field
facet = sorting.FieldFacet("path")
results = searcher.search(myquery, sortedby=facet)

# Group search results by the value of the "parent" field
facet = sorting.FieldFacet("parent")
results = searcher.search(myquery, groupedby=facet)
parent_groups = results.groups("parent")
```

By default, `FieldFacet` only supports **non-overlapping** grouping, where a document cannot belong to multiple facets at the same time (each document will be sorted into one category arbitrarily.) To get overlapping groups with multi-valued fields, use the `allow_overlap=True` keyword argument:

```
facet = sorting.FieldFacet(fieldname, allow_overlap=True)
```

This supports overlapping group membership where documents have more than one term in a field (e.g. KEYWORD fields). If you don't need overlapping, don't use `allow_overlap` because it's *much* slower and uses more memory (see the secion on `allow_overlap` below).

QueryFacet

You can set up categories defined by arbitrary queries. For example, you can group names using prefix queries:

```
# Use queries to define each category
# (Here I'll assume "price" is a NUMERIC field, so I'll use
# NumericRange)
qdict = {}
qdict["A-D"] = query.TermRange("name", "a", "d")
qdict["E-H"] = query.TermRange("name", "e", "h")
qdict["I-L"] = query.TermRange("name", "i", "l")
# ...

qfacet = sorting.QueryFacet(qdict)
r = searcher.search(myquery, groupedby={"firstltr": qfacet})
```

By default, `QueryFacet` only supports **non-overlapping** grouping, where a document cannot belong to multiple facets at the same time (each document will be sorted into one category arbitrarily). To get overlapping groups with multi-valued fields, use the `allow_overlap=True` keyword argument:

```
facet = sorting.QueryFacet(querydict, allow_overlap=True)
```

RangeFacet

The `RangeFacet` is for NUMERIC field types. It divides a range of possible values into groups. For example, to group documents based on price into buckets $100 "wide":

```
pricefacet = sorting.RangeFacet("price", 0, 1000, 100)
```

The first argument is the name of the field. The next two arguments are the full range to be divided. Value outside this range (in this example, values below 0 and above 1000) will be sorted into the "missing" (None) group. The fourth argument is the "gap size", the size of the divisions in the range.

The "gap" can be a list instead of a single value. In that case, the values in the list will be used to set the size of the initial divisions, with the last value in the list being the size for all subsequent divisions. For example:

```
pricefacet = sorting.RangeFacet("price", 0, 1000, [5, 10, 35, 50])
```

...will set up divisions of 0-5, 5-15, 15-50, 50-100, and then use 50 as the size for all subsequent divisions (i.e. 100-150, 150-200, and so on).

The `hardend` keyword argument controls whether the last division is clamped to the end of the range or allowed to go past the end of the range. For example, this:

```
facet = sorting.RangeFacet("num", 0, 10, 4, hardend=False)
```

...gives divisions 0-4, 4-8, and 8-12, while this:

```
facet = sorting.RangeFacet("num", 0, 10, 4, hardend=True)
```

...gives divisions 0-4, 4-8, and 8-10. (The default is `hardend=False`.)

Note: The ranges/buckets are always **inclusive** at the start and **exclusive** at the end.

DateRangeFacet

This is like `RangeFacet` but for DATETIME fields. The start and end values must be `datetime.datetime` objects, and the gap(s) is/are `datetime.timedelta` objects.

For example:

```
from datetime import datetime, timedelta

start = datetime(2000, 1, 1)
end = datetime.now()
gap = timedelta(days=365)
bdayfacet = sorting.DateRangeFacet("birthday", start, end, gap)
```

As with `RangeFacet`, you can use a list of gaps and the `hardend` keyword argument.

ScoreFacet

This facet is sometimes useful for sorting.

For example, to sort by the "category" field, then for documents with the same category, sort by the document's score:

```
cats = sorting.FieldFacet("category")
scores = sorting.ScoreFacet()
results = searcher.search(myquery, sortedby=[cats, scores])
```

The `ScoreFacet` always sorts higher scores before lower scores.

Note: While using `sortedby=ScoreFacet()` should give the same results as using the default scored ordering (`sortedby=None`), using the facet will be slower because Whoosh automatically turns off many optimizations when sorting.

FunctionFacet

This facet lets you pass a custom function to compute the sorting/grouping key for documents. (Using this facet type may be easier than subclassing FacetType and Categorizer to set up some custom behavior.)

The function will be called with the index searcher and index document ID as arguments. For example, if you have an index with term vectors:

```
schema = fields.Schema(id=fields.STORED,
                       text=fields.TEXT(stored=True, vector=True))
ix = RamStorage().create_index(schema)
```

...you could use a function to sort documents higher the closer they are to having equal occurances of two terms:

```
def fn(searcher, docnum):
    v = dict(searcher.vector_as("frequency", docnum, "text"))
    # Sort documents that have equal number of "alfa" and "bravo" first
    return 0 - (1.0 / (abs(v.get("alfa", 0) - v.get("bravo", 0)) + 1.0))

facet = sorting.FunctionFacet(fn)
results = searcher.search(myquery, sortedby=facet)
```

StoredFieldFacet

This facet lets you use stored field values as the sorting/grouping key for documents. This is usually slower than using an indexed field, but when using `allow_overlap` it can actually be faster for large indexes just because it avoids the overhead of reading posting lists.

`StoredFieldFacet` supports `allow_overlap` by splitting the stored value into separate keys. By default it calls the value's `split()` method (since most stored values are strings), but you can supply a custom split function. See the section on `allow_overlap` below.

MultiFacet

This facet type returns a composite of the keys returned by two or more sub-facets, allowing you to sort/group by the intersected values of multiple facets.

`MultiFacet` has methods for adding facets:

```
myfacet = sorting.RangeFacet(0, 1000, 10)

mf = sorting.MultiFacet()
mf.add_field("category")
mf.add_field("price", reverse=True)
mf.add_facet(myfacet)
mf.add_score()
```

You can also pass a list of field names and/or `FacetType` objects to the initializer:

```
prices = sorting.FieldFacet("price", reverse=True)
scores = sorting.ScoreFacet()
mf = sorting.MultiFacet(["category", prices, myfacet, scores])
```

Missing values

- When sorting, documents without any terms in a given field, or whatever else constitutes "missing" for different facet types, will always sort to the end.
- When grouping, "missing" documents will appear in a group with the key `None`.

Using overlapping groups

The common supported workflow for grouping and sorting is where the given field has *one value for document*, for example a `path` field containing the file path of the original document. By default, facets are set up to support this single-value approach.

Of course, there are situations where you want documents to be sorted into multiple groups based on a field with multiple terms per document. The most common example would be a `tags` field. The `allow_overlap` keyword argument to the *`FieldFacet`*, *`QueryFacet`*, and *`StoredFieldFacet`* allows this multi-value approach.

However, there is an important caveat: using `allow_overlap=True` is slower than the default, potentially *much* slower for very large result sets. This is because Whoosh must read every posting of every term in the field to create a temporary "forward index" mapping documents to terms.

If a field is indexed with *term vectors*, `FieldFacet` will use them to speed up `allow_overlap` faceting for small result sets, but for large result sets, where Whoosh has to open the vector list for every matched document, this can still be very slow.

For very large indexes and result sets, if a field is stored, you can get faster overlapped faceting using *`StoredFieldFacet`* instead of `FieldFacet`. While reading stored values is usually slower than using the index, in this case avoiding the overhead of opening large numbers of posting readers can make it worthwhile.

`StoredFieldFacet` supports `allow_overlap` by loading the stored value for the given field and splitting it into multiple values. The default is to call the value's `split()` method.

For example, if you've stored the `tags` field as a string like `"tag1 tag2 tag3"`:

```
schema = fields.Schema(name=fields.TEXT(stored=True),
                       tags=fields.KEYWORD(stored=True))
ix = index.create_in("indexdir")
with ix.writer() as w:
    w.add_document(name="A Midsummer Night's Dream", tags="comedy fairies")
    w.add_document(name="Hamlet", tags="tragedy denmark")
    # etc.
```

...Then you can use a `StoredFieldFacet` like this:

```
ix = index.open_dir("indexdir")
with ix.searcher() as s:
    sff = sorting.StoredFieldFacet("tags", allow_overlap=True)
    results = s.search(myquery, groupedby={"tags": sff})
```

For stored Python objects other than strings, you can supply a split function (using the `split_fn` keyword argument to `StoredFieldFacet`). The function should accept a single argument (the stored value) and return a list or tuple of grouping keys.

Using a custom sort order

It is sometimes useful to have a custom sort order per-search. For example, different languages use different sort orders. If you have a function to return the sorting order you want for a given field value, such as an implementation

of the Unicode Collation Algorithm (UCA), you can customize the sort order for the user's language.

The `whoosh.sorting.TranslateFacet` lets you apply a function to the value of another facet. This lets you "translate" a field value into an arbitrary sort key, such as with UCA:

```
from pyuca import Collator

# The Collator object has a sort_key() method which takes a unicode
# string and returns a sort key
c = Collator("allkeys.txt")

# Make a facet object for the field you want to sort on
nf = sorting.FieldFacet("name")

# Wrap the facet in a TranslateFacet with the translation function
# (the Collator object's sort_key method)
tf = sorting.TranslateFacet(facet, c.sort_key)

# Use the facet to sort the search results
results = searcher.search(myquery, sortedby=tf)
```

(You can pass multiple "wrapped" facets to the `TranslateFacet`, and it will call the function with the values of the facets as multiple arguments.)

The TranslateFacet can also be very useful with numeric fields to sort on the output of some formula:

```
# Sort based on the average of two numeric fields
def average(a, b):
    return (a + b) / 2.0

# Create two facets for the fields and pass them with the function to
# TranslateFacet
af = sorting.FieldFacet("age")
wf = sorting.FieldFacet("weight")
facet = sorting.TranslateFacet(average, af, wf)

results = searcher.search(myquery. sortedby=facet)
```

Remember that you can still sort by multiple facets. For example, you could sort by a numeric value transformed by a quantizing function first, and then if that is equal sort by the value of another field:

```
# Sort by a quantized size first, then by name
tf = sorting.TranslateFacet(quantize, sorting.FieldFacet("size"))
results = searcher.search(myquery, sortedby=[tf, "name"])
```

Expert: writing your own facet

TBD.

How to create highlighted search result excerpts

Overview

The highlighting system works as a pipeline, with four component types.

- **Fragmenters** chop up the original text into __fragments__, based on the locations of matched terms in the text.
- **Scorers** assign a score to each fragment, allowing the system to rank the best fragments by whatever criterion.
- **Order functions** control in what order the top-scoring fragments are presented to the user. For example, you can show the fragments in the order they appear in the document (FIRST) or show higher-scoring fragments first (SCORE)
- **Formatters** turn the fragment objects into human-readable output, such as an HTML string.

Requirements

Highlighting requires that you have the text of the indexed document available. You can keep the text in a stored field, or if the original text is available in a file, database column, etc, just reload it on the fly. Note that you might need to process the text to remove e.g. HTML tags, wiki markup, etc.

How to

Get search results:

```
results = mysearcher.search(myquery)
for hit in results:
    print(hit["title"])
```

You can use the *highlights()* method on the *whoosh.searching.Hit* object to get highlighted snippets from the document containing the search terms.

The first argument is the name of the field to highlight. If the field is stored, this is the only argument you need to supply:

```
results = mysearcher.search(myquery)
for hit in results:
    print(hit["title"])
    # Assume "content" field is stored
    print(hit.highlights("content"))
```

If the field is not stored, you need to retrieve the text of the field some other way. For example, reading it from the original file or a database. Then you can supply the text to highlight with the `text` argument:

```
results = mysearcher.search(myquery)
for hit in results:
    print(hit["title"])

    # Assume the "path" stored field contains a path to the original file
    with open(hit["path"]) as fileobj:
        filecontents = fileobj.read()

    print(hit.highlights("content", text=filecontents))
```

The character limit

By default, Whoosh only pulls fragments from the first 32K characters of the text. This prevents very long texts from bogging down the highlighting process too much, and is usually justified since important/summary information is usually at the start of a document. However, if you find the highlights are missing information (for example, very long encyclopedia articles where the terms appear in a later section), you can increase the fragmenter's character limit.

You can change the character limit on the results object like this:

```
results = mysearcher.search(myquery)
results.fragmenter.charlimit = 100000
```

To turn off the character limit:

```
results.fragmenter.charlimit = None
```

If you instantiate a custom fragmenter, you can set the character limit on it directly:

```
sf = highlight.SentenceFragmenter(charlimit=100000)
results.fragmenter = sf
```

See below for information on customizing the highlights.

If you increase or disable the character limit to highlight long documents, you may need to use the tips in the "speeding up highlighting" section below to make highlighting faster.

Customizing the highlights

Number of fragments

You can use the `top` keyword argument to control the number of fragments returned in each snippet:

```
# Show a maximum of 5 fragments from the document
print hit.highlights("content", top=5)
```

Fragment size

The default fragmenter has a `maxchars` attribute (default 200) controlling the maximum length of a fragment, and a `surround` attribute (default 20) controlling the maximum number of characters of context to add at the beginning and end of a fragment:

```
# Allow larger fragments
results.fragmenter.maxchars = 300

# Show more context before and after
results.fragmenter.surround = 50
```

Fragmenter

A fragmenter controls how to extract excerpts from the original text.

The `highlight` module has the following pre-made fragmenters:

whoosh.highlight.ContextFragmenter **(the default)** This is a "smart" fragmenter that finds matched terms and then pulls in surround text to form fragments. This fragmenter only yields fragments that contain matched terms.

whoosh.highlight.SentenceFragmenter Tries to break the text into fragments based on sentence punctuation (".", "!", and "?"). This object works by looking in the original text for a sentence end as the next character after each token's 'endchar'. Can be fooled by e.g. source code, decimals, etc.

whoosh.highlight.WholeFragmenter Returns the entire text as one "fragment". This can be useful if you are highlighting a short bit of text and don't need to fragment it.

The different fragmenters have different options. For example, the default *ContextFragmenter* lets you set the maximum fragment size and the size of the context to add on either side:

```
my_cf = highlight.ContextFragmenter(maxchars=100, surround=30)
```

See the *whoosh.highlight* docs for more information.

To use a different fragmenter:

```
results.fragmenter = my_cf
```

Scorer

A scorer is a callable that takes a *whoosh.highlight.Fragment* object and returns a sortable value (where higher values represent better fragments). The default scorer adds up the number of matched terms in the fragment, and adds a "bonus" for the number of __different__ matched terms. The highlighting system uses this score to select the best fragments to show to the user.

As an example of a custom scorer, to rank fragments by lowest standard deviation of the positions of matched terms in the fragment:

```
def StandardDeviationScorer(fragment):
    """Gives higher scores to fragments where the matched terms are close
    together.
    """

    # Since lower values are better in this case, we need to negate the
    # value
    return 0 - stddev([t.pos for t in fragment.matched])
```

To use a different scorer:

```
results.scorer = StandardDeviationScorer
```

Order

The order is a function that takes a fragment and returns a sortable value used to sort the highest-scoring fragments before presenting them to the user (where fragments with lower values appear before fragments with higher values).

The `highlight` module has the following order functions.

`FIRST` (the default) Show fragments in the order they appear in the document.

`SCORE` Show highest scoring fragments first.

The `highlight` module also includes `LONGER` (longer fragments first) and `SHORTER` (shorter fragments first), but they probably aren't as generally useful.

To use a different order:

```
results.order = highlight.SCORE
```

Formatter

A formatter contols how the highest scoring fragments are turned into a formatted bit of text for display to the user. It can return anything (e.g. plain text, HTML, a Genshi event stream, a SAX event generator, or anything else useful to the calling system).

The `highlight` module contains the following pre-made formatters.

whoosh.highlight.HtmlFormatter Outputs a string containing HTML tags (with a class attribute) around the matched terms.

whoosh.highlight.UppercaseFormatter Converts the matched terms to UPPERCASE.

whoosh.highlight.GenshiFormatter Outputs a Genshi event stream, with the matched terms wrapped in a configurable element.

The easiest way to create a custom formatter is to subclass `highlight.Formatter` and override the `format_token` method:

```
class BracketFormatter(highlight.Formatter):
    """Puts square brackets around the matched terms.
    """

    def format_token(self, text, token, replace=False):
        # Use the get_text function to get the text corresponding to the
        # token
        tokentext = highlight.get_text(text, token, replace)

        # Return the text as you want it to appear in the highlighted
        # string
        return "[%s]" % tokentext
```

To use a different formatter:

```
brf = BracketFormatter()
results.formatter = brf
```

If you need more control over the formatting (or want to output something other than strings), you will need to override other methods. See the documentation for the `whoosh.highlight.Formatter` class.

Highlighter object

Rather than setting attributes on the results object, you can create a reusable *whoosh.highlight.Highlighter* object. Keyword arguments let you change the `fragmenter`, `scorer`, `order`, and/or `formatter`:

```
hi = highlight.Highlighter(fragmenter=my_cf, scorer=sds)
```

You can then use the `whoosh.highlight.Highlighter.highlight_hit()` method to get highlights for a `Hit` object:

```
for hit in results:
    print(hit["title"])
    print(hi.highlight_hit(hit))
```

(When you assign to a `Results` object's `fragmenter`, `scorer`, `order`, or `formatter` attributes, you're actually changing the values on the results object's default `Highlighter` object.)

Speeding up highlighting

Recording which terms matched in which documents during the search may make highlighting faster, since it will skip documents it knows don't contain any matching terms in the given field:

```
# Record per-document term matches
results = searcher.search(myquery, terms=True)
```

PinpointFragmenter

Usually the highlighting system uses the field's analyzer to re-tokenize the document's text to find the matching terms in context. If you have long documents and have increased/disabled the character limit, and/or if the field has a very complex analyzer, re-tokenizing may be slow.

Instead of retokenizing, Whoosh can look up the character positions of the matched terms in the index. Looking up the character positions is not instantaneous, but is usually faster than analyzing large amounts of text.

To use *whoosh.highlight.PinpointFragmenter* and avoid re-tokenizing the document text, you must do all of the following:

Index the field with character information (this will require re-indexing an existing index):

```
# Index the start and end chars of each term
schema = fields.Schema(content=fields.TEXT(stored=True, chars=True))
```

Record per-document term matches in the results:

```
# Record per-document term matches
results = searcher.search(myquery, terms=True)
```

Set a *whoosh.highlight.PinpointFragmenter* as the fragmenter:

```
results.fragmenter = highlight.PinpointFragmenter()
```

PinpointFragmenter limitations

When the highlighting system does not re-tokenize the text, it doesn't know where any other words are in the text except the matched terms it looked up in the index. Therefore when the fragmenter adds surrounding context, it just adds or a certain number of characters blindly, and so doesn't distinguish between content and whitespace, or break on word boundaries, for example:

```
>>> hit.highlights("content")
're when the <b>fragmenter</b>\n       ad'
```

(This can be embarassing when the word fragments form dirty words!)

One way to avoid this is to not show any surrounding context, but then fragments containing one matched term will contain ONLY that matched term:

```
>>> hit.highlights("content")
'<b>fragmenter</b>'
```

Alternatively, you can normalize whitespace in the text before passing it to the highlighting system:

```
>>> text = searcher.stored_
>>> re.sub("[\t\r\n ]+", " ", text)
>>> hit.highlights("content", text=text)
```

...and use the `autotrim` option of `PinpointFragmenter` to automatically strip text before the first space and after the last space in the fragments:

```
>>> results.fragmenter = highlight.PinpointFragmenter(autotrim=True)
>>> hit.highlights("content")
'when the <b>fragmenter</b>'
```

Using the low-level API

Usage

The following function lets you retokenize and highlight a piece of text using an analyzer:

```
from whoosh.highlight import highlight

excerpts = highlight(text, terms, analyzer, fragmenter, formatter, top=3,
                     scorer=BasicFragmentScorer, minscore=1, order=FIRST)
```

text The original text of the document.

terms A sequence or set containing the query words to match, e.g. ("render", "shader").

analyzer The analyzer to use to break the document text into tokens for matching against the query terms. This is usually the analyzer for the field the query terms are in.

fragmenter A *whoosh.highlight.Fragmenter* object, see below.

formatter A `whoosh.highlight.Formatter` object, see below.

top The number of fragments to include in the output.

scorer A *whoosh.highlight.FragmentScorer* object. The only scorer currently included with Whoosh is *BasicFragmentScorer*, the default.

minscore The minimum score a fragment must have to be considered for inclusion.

order An ordering function that determines the order of the "top" fragments in the output text.

Query expansion and Key word extraction

Overview

Whoosh provides methods for computing the "key terms" of a set of documents. For these methods, "key terms" basically means terms that are frequent in the given documents, but relatively infrequent in the indexed collection as a whole.

Because this is a purely statistical operation, not a natural language processing or AI function, the quality of the results will vary based on the content, the size of the document collection, and the number of documents for which you extract keywords.

These methods can be useful for providing the following features to users:

- Search term expansion. You can extract key terms for the top N results from a query and suggest them to the user as additional/alternate query terms to try.
- Tag suggestion. Extracting the key terms for a single document may yield useful suggestions for tagging the document.
- "More like this". You can extract key terms for the top ten or so results from a query (and removing the original query terms), and use those key words as the basis for another query that may find more documents using terms the user didn't think of.

Usage

- Get more documents like a certain search hit. *This requires that the field you want to match on is vectored or stored, or that you have access to the original text (such as from a database).*

  Use *more_like_this()*:

```
results = mysearcher.search(myquery)
first_hit = results[0]
more_results = first_hit.more_like_this("content")
```

- Extract keywords for the top N documents in a *whoosh.searching.Results* object. *This requires that the field is either vectored or stored.*

  Use the *key_terms()* method of the *whoosh.searching.Results* object to extract keywords from the top N documents of the result set.

  For example, to extract *five* key terms from the `content` field of the top *ten* documents of a results object:

```
keywords = [keyword for keyword, score
            in results.key_terms("content", docs=10, numterms=5)
```

- Extract keywords for an arbitrary set of documents. *This requires that the field is either vectored or stored.*

  Use the *document_number()* or *document_numbers()* methods of the *whoosh.searching.Searcher* object to get the document numbers for the document(s) you want to extract keywords from.

  Use the *key_terms()* method of a *whoosh.searching.Searcher* to extract the keywords, given the list of document numbers.

  For example, let's say you have an index of emails. To extract key terms from the `content` field of emails whose `emailto` field contains `matt@whoosh.ca`:

```
with email_index.searcher() as s:
    docnums = s.document_numbers(emailto=u"matt@whoosh.ca")
    keywords = [keyword for keyword, score
                in s.key_terms(docnums, "body")]
```

- Extract keywords from arbitrary text not in the index.

  Use the *key_terms_from_text()* method of a *whoosh.searching.Searcher* to extract the keywords, given the text:

```
with email_index.searcher() as s:
    keywords = [keyword for keyword, score
                in s.key_terms_from_text("body", mytext)]
```

Expansion models

The `ExpansionModel` subclasses in the `whoosh.classify` module implement different weighting functions for key words. These models are translated into Python from original Java implementations in Terrier.

"Did you mean... ?" Correcting errors in user queries

Overview

Whoosh can quickly suggest replacements for mis-typed words by returning a list of words from the index (or a dictionary) that are close to the mis-typed word:

```
with ix.searcher() as s:
    corrector = s.corrector("text")
    for mistyped_word in mistyped_words:
        print corrector.suggest(mistyped_word, limit=3)
```

See the *whoosh.spelling.Corrector.suggest()* method documentation for information on the arguments.

Currently the suggestion engine is more like a "typo corrector" than a real "spell checker" since it doesn't do the kind of sophisticated phonetic matching or semantic/contextual analysis a good spell checker might. However, it is still very useful.

There are two main strategies for correcting words:

- Use the terms from an index field.
- Use words from a word list.

Pulling suggestions from an indexed field

In Whoosh 2.7 and later, spelling suggestions are available on all fields. However, if you have an analyzer that modifies the indexed words (such as stemming), you can add `spelling=True` to a field to have it store separate unmodified versions of the terms for spelling suggestions:

```
ana = analysis.StemmingAnalyzer()
schema = fields.Schema(text=TEXT(analyzer=ana, spelling=True))
```

You can then use the `whoosh.searching.Searcher.corrector()` method to get a corrector for a field:

```
corrector = searcher.corrector("content")
```

The advantage of using the contents of an index field is that when you are spell checking queries on that index, the suggestions are tailored to the contents of the index. The disadvantage is that if the indexed documents contain spelling errors, then the spelling suggestions will also be erroneous.

Pulling suggestions from a word list

There are plenty of word lists available on the internet you can use to populate the spelling dictionary.

(In the following examples, `word_list` can be a list of unicode strings, or a file object with one word on each line.)

To create a *whoosh.spelling.Corrector* object from a sorted word list:

```
from whoosh.spelling import ListCorrector

# word_list must be a sorted list of unicocde strings
corrector = ListCorrector(word_list)
```

Merging two or more correctors

You can combine suggestions from two sources (for example, the contents of an index field and a word list) using a *whoosh.spelling.MultiCorrector*:

```
c1 = searcher.corrector("content")
c2 = spelling.ListCorrector(word_list)
corrector = MultiCorrector([c1, c2])
```

Correcting user queries

You can spell-check a user query using the *whoosh.searching.Searcher.correct_query()* method:

```
from whoosh import qparser

# Parse the user query string
qp = qparser.QueryParser("content", myindex.schema)
q = qp.parse(qstring)

# Try correcting the query
with myindex.searcher() as s:
    corrected = s.correct_query(q, qstring)
    if corrected.query != q:
        print("Did you mean:", corrected.string)
```

The `correct_query` method returns an object with the following attributes:

query A corrected *whoosh.query.Query* tree. You can test whether this is equal (==) to the original parsed query to check if the corrector actually changed anything.

string A corrected version of the user's query string.

tokens A list of corrected token objects representing the corrected terms. You can use this to reformat the user query (see below).

You can use a `whoosh.highlight.Formatter` object to format the corrected query string. For example, use the *HtmlFormatter* to format the corrected string as HTML:

```
from whoosh import highlight

hf = highlight.HtmlFormatter()
corrected = s.correct_query(q, qstring, formatter=hf)
```

See the documentation for *whoosh.searching.Searcher.correct_query()* for information on the defaults and arguments.

Field caches

The default (`filedb`) backend uses *field caches* in certain circumstances. The field cache basically pre-computes the order of documents in the index to speed up sorting and faceting.

Generating field caches can take time the first time you sort/facet on a large index. The field cache is kept in memory (and by default written to disk when it is generated) so subsequent sorted/faceted searches should be faster.

The default caching policy never expires field caches, so reused searchers and/or sorting a lot of different fields could use up quite a bit of memory with large indexes.

Customizing cache behaviour

(The following API examples refer to the default `filedb` backend.)

By default, Whoosh saves field caches to disk. To prevent a reader or searcher from writing out field caches, do this before you start using it:

```
searcher.set_caching_policy(save=False)
```

By default, if caches are written to disk they are saved in the index directory. To tell a reader or searcher to save cache files to a different location, create a storage object and pass it to the `storage` keyword argument:

```
from whoosh.filedb.filestore import FileStorage

mystorage = FileStorage("path/to/cachedir")
reader.set_caching_policy(storage=mystorage)
```

Creating a custom caching policy

Expert users who want to implement a custom caching policy (for example, to add cache expiration) should subclass `whoosh.filedb.fieldcache.FieldCachingPolicy`. Then you can pass an instance of your policy object to the `set_caching_policy` method:

```
searcher.set_caching_policy(MyPolicy())
```

Tips for speeding up batch indexing

Overview

Indexing documents tends to fall into two general patterns: adding documents one at a time as they are created (as in a web application), and adding a bunch of documents at once (batch indexing).

The following settings and alternate workflows can make batch indexing faster.

StemmingAnalyzer cache

The stemming analyzer by default uses a least-recently-used (LRU) cache to limit the amount of memory it uses, to prevent the cache from growing very large if the analyzer is reused for a long period of time. However, the LRU cache can slow down indexing by almost 200% compared to a stemming analyzer with an "unbounded" cache.

When you're indexing in large batches with a one-shot instance of the analyzer, consider using an unbounded cache:

```
w = myindex.writer()
# Get the analyzer object from a text field
stem_ana = w.schema["content"].format.analyzer
# Set the cachesize to -1 to indicate unbounded caching
stem_ana.cachesize = -1
# Reset the analyzer to pick up the changed attribute
stem_ana.clear()

# Use the writer to index documents...
```

The `limitmb` parameter

The `limitmb` parameter to *`whoosh.index.Index.writer()`* controls the *maximum* memory (in megabytes) the writer will use for the indexing pool. The higher the number, the faster indexing will be.

The default value of `128` is actually somewhat low, considering many people have multiple gigabytes of RAM these days. Setting it higher can speed up indexing considerably:

```
from whoosh import index

ix = index.open_dir("indexdir")
writer = ix.writer(limitmb=256)
```

Note: The actual memory used will be higher than this value because of interpreter overhead (up to twice as much!). It is very useful as a tuning parameter, but not for trying to exactly control the memory usage of Whoosh.

The `procs` parameter

The `procs` parameter to *`whoosh.index.Index.writer()`* controls the number of processors the writer will use for indexing (via the `multiprocessing` module):

```
from whoosh import index

ix = index.open_dir("indexdir")
writer = ix.writer(procs=4)
```

Note that when you use multiprocessing, the `limitmb` parameter controls the amount of memory used by *each process*, so the actual memory used will be `limitmb * procs`:

```
# Each process will use a limit of 128, for a total of 512
writer = ix.writer(procs=4, limitmb=128)
```

The `multisegment` parameter

The `procs` parameter causes the default writer to use multiple processors to do much of the indexing, but then still uses a single process to merge the pool of each sub-writer into a single segment.

You can get much better indexing speed by also using the `multisegment=True` keyword argument, which instead of merging the results of each sub-writer, simply has them each just write out a new segment:

```
from whoosh import index

ix = index.open_dir("indexdir")
writer = ix.writer(procs=4, multisegment=True)
```

The drawback is that instead of creating a single new segment, this option creates a number of new segments **at least** equal to the number of processes you use.

For example, if you use `procs=4`, the writer will create four new segments. (If you merge old segments or call `add_reader` on the parent writer, the parent writer will also write a segment, meaning you'll get five new segments.)

So, while `multisegment=True` is much faster than a normal writer, you should only use it for large batch indexing jobs (or perhaps only for indexing from scratch). It should not be the only method you use for indexing, because otherwise the number of segments will tend to increase forever!

Concurrency, locking, and versioning

Concurrency

The `FileIndex` object is "stateless" and should be share-able between threads.

A `Reader` object (which underlies the `Searcher` object) wraps open files and often individual methods rely on consistent file cursor positions (e.g. they do two `file.read()`s in a row, so if another thread moves the cursor between the two read calls Bad Things would happen). You should use one Reader/Searcher per thread in your code.

Readers/Searchers tend to cache information (such as field caches for sorting), so if you can share one across multiple search requests, it's a big performance win.

Locking

Only one thread/process can write to an index at a time. When you open a writer, it locks the index. If you try to open a writer on the same index in another thread/process, it will raise `whoosh.store.LockError`.

In a multi-threaded or multi-process environment your code needs to be aware that opening a writer may raise this exception if a writer is already open. Whoosh includes a couple of example implementations (*`whoosh.writing.AsyncWriter`* and *`whoosh.writing.BufferedWriter`*) of ways to work around the write lock.

While the writer is open and during the commit, **the index is still available for reading**. Existing readers are unaffected and new readers can open the current index normally.

Lock files

Locking the index is accomplished by acquiring an exclusive file lock on the `<indexname>_WRITELOCK` file in the index directory. The file is not deleted after the file lock is released, so the fact that the file exists **does not** mean the index is locked.

Versioning

When you open a reader/searcher, the reader represents a view of the **current version** of the index. If someone writes changes to the index, any readers that are already open **will not** pick up the changes automatically. A reader always sees the index as it existed when the reader was opened.

If you are re-using a Searcher across multiple search requests, you can check whether the Searcher is a view of the latest version of the index using *whoosh.searching.Searcher.up_to_date()*. If the searcher is not up to date, you can get an up-to-date copy of the searcher using *whoosh.searching.Searcher.refresh()*:

```
# If 'searcher' is not up-to-date, replace it
searcher = searcher.refresh()
```

(If the searcher has the latest version of the index, `refresh()` simply returns it.)

Calling `Searcher.refresh()` is more efficient that closing the searcher and opening a new one, since it will re-use any underlying readers and caches that haven't changed.

Indexing and searching document hierarchies

Overview

Whoosh's full-text index is essentially a flat database of documents. However, Whoosh supports two techniques for simulating the indexing and querying of hierarchical documents, that is, sets of documents that form a parent-child hierarchy, such as "Chapter - Section - Paragraph" or "Module - Class - Method".

You can specify parent-child relationships *at indexing time*, by grouping documents in the same hierarchy, and then use the *whoosh.query.NestedParent* and/or *whoosh.query.NestedChildren* to find parents based on their children or vice-versa.

Alternatively, you can use *query time joins*, essentially like external key joins in a database, where you perform one search to find a relevant document, then use a stored value on that document (for example, a `parent` field) to look up another document.

Both methods have pros and cons.

Using nested document indexing

Indexing

This method works by indexing a "parent" document and all its "child" documents *as a "group"* so they are guaranteed to end up in the same segment. You can use the context manager returned by `IndexWriter.group()` to group documents:

```
with ix.writer() as w:
    with w.group():
        w.add_document(kind="class", name="Index")
        w.add_document(kind="method", name="add document")
        w.add_document(kind="method", name="add reader")
        w.add_document(kind="method", name="close")
    with w.group():
        w.add_document(kind="class", name="Accumulator")
        w.add_document(kind="method", name="add")
        w.add_document(kind="method", name="get result")
    with w.group():
        w.add_document(kind="class", name="Calculator")
        w.add_document(kind="method", name="add")
        w.add_document(kind="method", name="add all")
        w.add_document(kind="method", name="add some")
        w.add_document(kind="method", name="multiply")
        w.add_document(kind="method", name="close")
```

```
    with w.group():
        w.add_document(kind="class", name="Deleter")
        w.add_document(kind="method", name="add")
        w.add_document(kind="method", name="delete")
```

Alternatively you can use the `start_group()` and `end_group()` methods:

```
with ix.writer() as w:
    w.start_group()
    w.add_document(kind="class", name="Index")
    w.add_document(kind="method", name="add document")
    w.add_document(kind="method", name="add reader")
    w.add_document(kind="method", name="close")
    w.end_group()
```

Each level of the hierarchy should have a query that distinguishes it from other levels (for example, in the above index, you can use `kind:class` or `kind:method` to match different levels of the hierarchy).

Once you've indexed the hierarchy of documents, you can use two query types to find parents based on children or vice-versa.

(There is currently no support in the default query parser for nested queries.)

NestedParent query

The *whoosh.query.NestedParent* query type lets you specify a query for child documents, but have the query return an "ancestor" document from higher in the hierarchy:

```
# First, we need a query that matches all the documents in the "parent"
# level we want of the hierarchy
all_parents = query.Term("kind", "class")

# Then, we need a query that matches the children we want to find
wanted_kids = query.Term("name", "close")

# Now we can make a query that will match documents where "name" is
# "close", but the query will return the "parent" documents of the matching
# children
q = query.NestedParent(all_parents, wanted_kids)
# results = Index, Calculator
```

Note that in a hierarchy with more than two levels, you can specify a "parents" query that matches any level of the hierarchy, so you can return the top-level ancestors of the matching children, or the second level, third level, etc.

The query works by first building a bit vector representing which documents are "parents":

```
Index
|      Calculator
|      |
1000100100000100
    |        |
    |        Deleter
    Accumulator
```

Then for each match of the "child" query, it calculates the previous parent from the bit vector and returns it as a match (it only returns each parent once no matter how many children match). This parent lookup is very efficient:

```
1000100100000100
   |
|<-+ close
```

NestedChildren query

The opposite of `NestedParent` is *`whoosh.query.NestedChildren`*. This query lets you match parents but return their children. This is useful, for example, to search for an album title and return the songs in the album:

```
# Query that matches all documents in the "parent" level we want to match
# at
all_parents = query.Term("kind", "album")

# Parent documents we want to match
wanted_parents = query.Term("album_title", "heaven")

# Now we can make a query that will match parent documents where "album_title"
# contains "heaven", but the query will return the "child" documents of the
# matching parents
q1 = query.NestedChildren(all_parents, wanted_parents)
```

You can then combine that query with an `AND` clause, for example to find songs with "hell" in the song title that occur on albums with "heaven" in the album title:

```
q2 = query.And([q1, query.Term("song_title", "hell")])
```

Deleting and updating hierarchical documents

The drawback of the index-time method is *updating and deleting*. Because the implementation of the queries depends on the parent and child documents being contiguous in the segment, you can't update/delete just one child document. You can only update/delete an entire top-level document at once (for example, if your hierarchy is "Chapter - Section - Paragraph", you can only update or delete entire chapters, not a section or paragraph). If the top-level of the hierarchy represents very large blocks of text, this can involve a lot of deleting and reindexing.

Currently `Writer.update_document()` does not automatically work with nested documents. You must manually delete and re-add document groups to update them.

To delete nested document groups, use the `Writer.delete_by_query()` method with a `NestedParent` query:

```
# Delete the "Accumulator" class
all_parents = query.Term("kind", "class")
to_delete = query.Term("name", "Accumulator")
q = query.NestedParent(all_parents, to_delete)
with myindex.writer() as w:
    w.delete_by_query(q)
```

Using query-time joins

A second technique for simulating hierarchical documents in Whoosh involves using a stored field on each document to point to its parent, and then using the value of that field at query time to find parents and children.

For example, if we index a hierarchy of classes and methods using pointers to parents instead of nesting:

```
# Store a pointer to the parent on each "method" document
with ix.writer() as w:
    w.add_document(kind="class", c_name="Index", docstring="...")
    w.add_document(kind="method", m_name="add document", parent="Index")
    w.add_document(kind="method", m_name="add reader", parent="Index")
    w.add_document(kind="method", m_name="close", parent="Index")

    w.add_document(kind="class", c_name="Accumulator", docstring="...")
    w.add_document(kind="method", m_name="add", parent="Accumulator")
    w.add_document(kind="method", m_name="get result", parent="Accumulator")

    w.add_document(kind="class", c_name="Calculator", docstring="...")
    w.add_document(kind="method", m_name="add", parent="Calculator")
    w.add_document(kind="method", m_name="add all", parent="Calculator")
    w.add_document(kind="method", m_name="add some", parent="Calculator")
    w.add_document(kind="method", m_name="multiply", parent="Calculator")
    w.add_document(kind="method", m_name="close", parent="Calculator")

    w.add_document(kind="class", c_name="Deleter", docstring="...")
    w.add_document(kind="method", m_name="add", parent="Deleter")
    w.add_document(kind="method", m_name="delete", parent="Deleter")

# Now do manual joins at query time
with ix.searcher() as s:
    # Tip: Searcher.document() and Searcher.documents() let you look up
    # documents by field values more easily than using Searcher.search()

    # Children to parents:
    # Print the docstrings of classes on which "close" methods occur
    for child_doc in s.documents(m_name="close"):
        # Use the stored value of the "parent" field to look up the parent
        # document
        parent_doc = s.document(c_name=child_doc["parent"])
        # Print the parent document's stored docstring field
        print(parent_doc["docstring"])

    # Parents to children:
    # Find classes with "big" in the docstring and print their methods
    q = query.Term("kind", "class") & query.Term("docstring", "big")
    for hit in s.search(q, limit=None):
        print("Class name=", hit["c_name"], "methods:")
        for child_doc in s.documents(parent=hit["c_name"]):
            print("  Method name=", child_doc["m_name"])
```

This technique is more flexible than index-time nesting in that you can delete/update individual documents in the hierarchy piece by piece, although it doesn't support finding different parent levels as easily. It is also slower than index-time nesting (potentially much slower), since you must perform additional searches for each found document.

Future versions of Whoosh may include "join" queries to make this process more efficient (or at least more automatic).

Whoosh recipes

General

Get the stored fields for a document from the document number

```
stored_fields = searcher.stored_fields(docnum)
```

Analysis

Eliminate words shorter/longer than N

Use a *StopFilter* and the `minsize` and `maxsize` keyword arguments. If you just want to filter based on size and not common words, set the `stoplist` to `None`:

```
sf = analysis.StopFilter(stoplist=None, minsize=2, maxsize=40)
```

Allow optional case-sensitive searches

A quick and easy way to do this is to index both the original and lowercased versions of each word. If the user searches for an all-lowercase word, it acts as a case-insensitive search, but if they search for a word with any uppercase characters, it acts as a case-sensitive search:

```
class CaseSensitivizer(analysis.Filter):
    def __call__(self, tokens):
        for t in tokens:
            yield t
            if t.mode == "index":
                low = t.text.lower()
                if low != t.text:
                    t.text = low
                    yield t

ana = analysis.RegexTokenizer() | CaseSensitivizer()
[t.text for t in ana("The new SuperTurbo 5000", mode="index")]
# ["The", "the", "new", "SuperTurbo", "superturbo", "5000"]
```

Searching

Find every document

```
myquery = query.Every()
```

iTunes-style search-as-you-type

Use the *whoosh.analysis.NgramWordAnalyzer* as the analyzer for the field you want to search as the user types. You can save space in the index by turning off positions in the field using `phrase=False`, since phrase searching on N-gram fields usually doesn't make much sense:

```
# For example, to search the "title" field as the user types
analyzer = analysis.NgramWordAnalyzer()
title_field = fields.TEXT(analyzer=analyzer, phrase=False)
schema = fields.Schema(title=title_field)
```

See the documentation for the *NgramWordAnalyzer* class for information on the available options.

Shortcuts

Look up documents by a field value

```
# Single document (unique field value)
stored_fields = searcher.document(id="bacon")

# Multiple documents
for stored_fields in searcher.documents(tag="cake"):
    ...
```

Sorting and scoring

See *Sorting and faceting*.

Score results based on the position of the matched term

The following scoring function uses the position of the first occurance of a term in each document to calculate the score, so documents with the given term earlier in the document will score higher:

```
from whoosh import scoring

def pos_score_fn(searcher, fieldname, text, matcher):
    poses = matcher.value_as("positions")
    return 1.0 / (poses[0] + 1)

pos_weighting = scoring.FunctionWeighting(pos_score_fn)
with myindex.searcher(weighting=pos_weighting) as s:
    ...
```

Results

How many hits were there?

The number of *scored* hits:

```
found = results.scored_length()
```

Depending on the arguments to the search, the exact total number of hits may be known:

```
if results.has_exact_length():
    print("Scored", found, "of exactly", len(results), "documents")
```

Usually, however, the exact number of documents that match the query is not known, because the searcher can skip over blocks of documents it knows won't show up in the "top N" list. If you call `len(results)` on a query where the exact length is unknown, Whoosh will run an unscored version of the original query to get the exact number. This is faster than the scored search, but may still be noticeably slow on very large indexes or complex queries.

As an alternative, you might display the *estimated* total hits:

```
found = results.scored_length()
if results.has_exact_length():
    print("Scored", found, "of exactly", len(results), "documents")
else:
    low = results.estimated_min_length()
    high = results.estimated_length()

    print("Scored", found, "of between", low, "and", high, "documents")
```

Which terms matched in each hit?

```
# Use terms=True to record term matches for each hit
results = searcher.search(myquery, terms=True)

for hit in results:
    # Which terms matched in this hit?
    print("Matched:", hit.matched_terms())

    # Which terms from the query didn't match in this hit?
    print("Didn't match:", myquery.all_terms() - hit.matched_terms())
```

Global information

How many documents are in the index?

```
# Including documents that are deleted but not yet optimized away
numdocs = searcher.doc_count_all()

# Not including deleted documents
numdocs = searcher.doc_count()
```

What fields are in the index?

```
return myindex.schema.names()
```

Is term X in the index?

```
return ("content", "wobble") in searcher
```

How many times does term X occur in the index?

```
# Number of times content:wobble appears in all documents
freq = searcher.frequency("content", "wobble")

# Number of documents containing content:wobble
docfreq = searcher.doc_frequency("content", "wobble")
```

Is term X in document Y?

```
# Check if the "content" field of document 500 contains the term "wobble"

# Without term vectors, skipping through list...
postings = searcher.postings("content", "wobble")
postings.skip_to(500)
return postings.id() == 500

# ...or the slower but easier way
docset = set(searcher.postings("content", "wobble").all_ids())
return 500 in docset

# If field has term vectors, skipping through list...
vector = searcher.vector(500, "content")
vector.skip_to("wobble")
return vector.id() == "wobble"

# ...or the slower but easier way
wordset = set(searcher.vector(500, "content").all_ids())
return "wobble" in wordset
```

Whoosh API

`analysis` module

Classes and functions for turning a piece of text into an indexable stream of "tokens" (usually equivalent to words). There are three general classes involved in analysis:

- Tokenizers are always at the start of the text processing pipeline. They take a string and yield Token objects (actually, the same token object over and over, for performance reasons) corresponding to the tokens (words) in the text.

  Every tokenizer is a callable that takes a string and returns an iterator of tokens.

- Filters take the tokens from the tokenizer and perform various transformations on them. For example, the LowercaseFilter converts all tokens to lowercase, which is usually necessary when indexing regular English text.

  Every filter is a callable that takes a token generator and returns a token generator.

- Analyzers are convenience functions/classes that "package up" a tokenizer and zero or more filters into a single unit. For example, the StandardAnalyzer combines a RegexTokenizer, LowercaseFilter, and StopFilter.

  Every analyzer is a callable that takes a string and returns a token iterator. (So Tokenizers can be used as Analyzers if you don't need any filtering).

You can compose tokenizers and filters together using the `|` character:

```
my_analyzer = RegexTokenizer() | LowercaseFilter() | StopFilter()
```

The first item must be a tokenizer and the rest must be filters (you can't put a filter first or a tokenizer after the first item).

Analyzers

whoosh.analysis.**IDAnalyzer**(*lowercase=False*)
Deprecated, just use an IDTokenizer directly, with a LowercaseFilter if desired.

whoosh.analysis.**KeywordAnalyzer**(*lowercase=False*, *commas=False*)
Parses whitespace- or comma-separated tokens.

```
>>> ana = KeywordAnalyzer()
>>> [token.text for token in ana("Hello there, this is a TEST")]
["Hello", "there,", "this", "is", "a", "TEST"]
```

Parameters

- **lowercase** – whether to lowercase the tokens.
- **commas** – if True, items are separated by commas rather than whitespace.

whoosh.analysis.**RegexAnalyzer**(*expression='\\w+(\\.?\\w+)*'*, *gaps=False*)
Deprecated, just use a RegexTokenizer directly.

whoosh.analysis.**SimpleAnalyzer**(*expression=<_sre.SRE_Pattern object>*, *gaps=False*)
Composes a RegexTokenizer with a LowercaseFilter.

```
>>> ana = SimpleAnalyzer()
>>> [token.text for token in ana("Hello there, this is a TEST")]
["hello", "there", "this", "is", "a", "test"]
```

Parameters

- **expression** – The regular expression pattern to use to extract tokens.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.

whoosh.analysis.**StandardAnalyzer**(*expression=<_sre.SRE_Pattern object>*, *stoplist=frozenset(['and', 'is', 'it', 'an', 'as', 'at', 'have', 'in', 'yet', 'if', 'from', 'for', 'when', 'by', 'to', 'you', 'be', 'we', 'that', 'may', 'not', 'with', 'tbd', 'a', 'on', 'your', 'this', 'of', 'us', 'will', 'can', 'the', 'or', 'are'])*, *minsize=2*, *maxsize=None*, *gaps=False*)
Composes a RegexTokenizer with a LowercaseFilter and optional StopFilter.

```
>>> ana = StandardAnalyzer()
>>> [token.text for token in ana("Testing is testing and testing")]
["testing", "testing", "testing"]
```

Parameters

- **expression** – The regular expression pattern to use to extract tokens.
- **stoplist** – A list of stop words. Set this to None to disable the stop word filter.
- **minsize** – Words smaller than this are removed from the stream.
- **maxsize** – Words longer that this are removed from the stream.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.

whoosh.analysis.**StemmingAnalyzer**(*expression=<_sre.SRE_Pattern object>*, *stoplist=frozenset(['and', 'is', 'it', 'an', 'as', 'at', 'have', 'in', 'yet', 'if', 'from', 'for', 'when', 'by', 'to', 'you', 'be', 'we', 'that', 'may', 'not', 'with', 'tbd', 'a', 'on', 'your', 'this', 'of', 'us', 'will', 'can', 'the', 'or', 'are'])*, *minsize=2*, *maxsize=None*, *gaps=False*, *stemfn=<function stem>*, *ignore=None*, *cachesize=50000*)

Composes a RegexTokenizer with a lower case filter, an optional stop filter, and a stemming filter.

```
>>> ana = StemmingAnalyzer()
>>> [token.text for token in ana("Testing is testing and testing")]
["test", "test", "test"]
```

Parameters

- **expression** – The regular expression pattern to use to extract tokens.
- **stoplist** – A list of stop words. Set this to None to disable the stop word filter.
- **minsize** – Words smaller than this are removed from the stream.
- **maxsize** – Words longer that this are removed from the stream.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.
- **ignore** – a set of words to not stem.
- **cachesize** – the maximum number of stemmed words to cache. The larger this number, the faster stemming will be but the more memory it will use. Use None for no cache, or -1 for an unbounded cache.

whoosh.analysis.**FancyAnalyzer**(*expression='\\s+'*, *stoplist=frozenset(['and', 'is', 'it', 'an', 'as', 'at', 'have', 'in', 'yet', 'if', 'from', 'for', 'when', 'by', 'to', 'you', 'be', 'we', 'that', 'may', 'not', 'with', 'tbd', 'a', 'on', 'your', 'this', 'of', 'us', 'will', 'can', 'the', 'or', 'are'])*, *minsize=2*, *maxsize=None*, *gaps=True*, *splitwords=True*, *splitnums=True*, *mergewords=False*, *mergenums=False*)

Composes a RegexTokenizer with an IntraWordFilter, LowercaseFilter, and StopFilter.

```
>>> ana = FancyAnalyzer()
>>> [token.text for token in ana("Should I call getInt or get_real?")]
["should", "call", "getInt", "get", "int", "get_real", "get", "real"]
```

Parameters

- **expression** – The regular expression pattern to use to extract tokens.
- **stoplist** – A list of stop words. Set this to None to disable the stop word filter.
- **minsize** – Words smaller than this are removed from the stream.
- **maxsize** – Words longer that this are removed from the stream.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.

whoosh.analysis.**NgramAnalyzer**(*minsize*, *maxsize=None*)

Composes an NgramTokenizer and a LowercaseFilter.

```
>>> ana = NgramAnalyzer(4)
>>> [token.text for token in ana("hi there")]
["hi t", "i th", " the", "ther", "here"]
```

whoosh.analysis.**NgramWordAnalyzer**(*minsize*, *maxsize=None*, *tokenizer=None*, *at=None*)

whoosh.analysis.**LanguageAnalyzer**(*lang*, *expression=<_sre.SRE_Pattern object>*, *gaps=False*, *cachesize=50000*)

Configures a simple analyzer for the given language, with a LowercaseFilter, StopFilter, and StemFilter.

```
>>> ana = LanguageAnalyzer("es")
>>> [token.text for token in ana("Por el mar corren las liebres")]
['mar', 'corr', 'liebr']
```

The list of available languages is in *whoosh.lang.languages*. You can use `whoosh.lang.has_stemmer()` and `whoosh.lang.has_stopwords()` to check if a given language has a stemming function and/or stop word list available.

Parameters

- **expression** – The regular expression pattern to use to extract tokens.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.
- **cachesize** – the maximum number of stemmed words to cache. The larger this number, the faster stemming will be but the more memory it will use.

Tokenizers

class whoosh.analysis.**IDTokenizer**

Yields the entire input string as a single token. For use in indexed but untokenized fields, such as a document's path.

```
>>> idt = IDTokenizer()
>>> [token.text for token in idt("/a/b 123 alpha")]
["/a/b 123 alpha"]
```

class whoosh.analysis.**RegexTokenizer**(*expression=<_sre.SRE_Pattern object>*, *gaps=False*)

Uses a regular expression to extract tokens from text.

```
>>> rex = RegexTokenizer()
>>> [token.text for token in rex(u("hi there 3.141 big-time under_score"))]
["hi", "there", "3.141", "big", "time", "under_score"]
```

Parameters

- **expression** – A regular expression object or string. Each match of the expression equals a token. Group 0 (the entire matched text) is used as the text of the token. If you require more complicated handling of the expression match, simply write your own tokenizer.
- **gaps** – If True, the tokenizer *splits* on the expression, rather than matching on the expression.

class whoosh.analysis.**CharsetTokenizer**(*charmap*)

Tokenizes and translates text according to a character mapping object. Characters that map to None are considered token break characters. For all other characters the map is used to translate the character. This is useful for case and accent folding.

This tokenizer loops character-by-character and so will likely be much slower than *RegexTokenizer*.

One way to get a character mapping object is to convert a Sphinx charset table file using *whoosh.support.charset.charset_table_to_dict()*.

```
>>> from whoosh.support.charset import charset_table_to_dict
>>> from whoosh.support.charset import default_charset
>>> charmap = charset_table_to_dict(default_charset)
>>> chtokenizer = CharsetTokenizer(charmap)
>>> [t.text for t in chtokenizer(u'Stra\xdfe ABC')]
[u'strase', u'abc']
```

The Sphinx charset table format is described at http://www.sphinxsearch.com/docs/current.html#conf-charset-table.

Parameters **charmap** – a mapping from integer character numbers to unicode characters, as used by the unicode.translate() method.

whoosh.analysis.**SpaceSeparatedTokenizer**()
Returns a RegexTokenizer that splits tokens by whitespace.

```
>>> sst = SpaceSeparatedTokenizer()
>>> [token.text for token in sst("hi there big-time, what's up")]
["hi", "there", "big-time,", "what's", "up"]
```

whoosh.analysis.**CommaSeparatedTokenizer**()
Splits tokens by commas.

Note that the tokenizer calls unicode.strip() on each match of the regular expression.

```
>>> cst = CommaSeparatedTokenizer()
>>> [token.text for token in cst("hi there, what's , up")]
["hi there", "what's", "up"]
```

class whoosh.analysis.**NgramTokenizer**(*minsize*, *maxsize=None*)
Splits input text into N-grams instead of words.

```
>>> ngt = NgramTokenizer(4)
>>> [token.text for token in ngt("hi there")]
["hi t", "i th", " the", "ther", "here"]
```

Note that this tokenizer does NOT use a regular expression to extract words, so the grams emitted by it will contain whitespace, punctuation, etc. You may want to massage the input or add a custom filter to this tokenizer's output.

Alternatively, if you only want sub-word grams without whitespace, you could combine a RegexTokenizer with NgramFilter instead.

Parameters

- **minsize** – The minimum size of the N-grams.
- **maxsize** – The maximum size of the N-grams. If you omit this parameter, maxsize == minsize.

class whoosh.analysis.**PathTokenizer**(*expression='[^/]+'*)
A simple tokenizer that given a string `"/a/b/c"` yields tokens `["/a", "/a/b", "/a/b/c"]`.

Filters

class whoosh.analysis.**PassFilter**
An identity filter: passes the tokens through untouched.

class whoosh.analysis.**LoggingFilter**(*logger=None*)
Prints the contents of every filter that passes through as a debug log entry.

Parameters **target** – the logger to use. If omitted, the "whoosh.analysis" logger is used.

class whoosh.analysis.**MultiFilter**(***kwargs*)
Chooses one of two or more sub-filters based on the 'mode' attribute of the token stream.

Use keyword arguments to associate mode attribute values with instantiated filters.

```
>>> iwf_for_index = IntraWordFilter(mergewords=True, mergenums=False)
>>> iwf_for_query = IntraWordFilter(mergewords=False, mergenums=False)
>>> mf = MultiFilter(index=iwf_for_index, query=iwf_for_query)
```

This class expects that the value of the mode attribute is consistent among all tokens in a token stream.

class whoosh.analysis.**TeeFilter**(**filters*)
Interleaves the results of two or more filters (or filter chains).

NOTE: because it needs to create copies of each token for each sub-filter, this filter is quite slow.

```
>>> target = "ALFA BRAVO CHARLIE"
>>> # In one branch, we'll lower-case the tokens
>>> f1 = LowercaseFilter()
>>> # In the other branch, we'll reverse the tokens
>>> f2 = ReverseTextFilter()
>>> ana = RegexTokenizer(r"\S+") | TeeFilter(f1, f2)
>>> [token.text for token in ana(target)]
["alfa", "AFLA", "bravo", "OVARB", "charlie", "EILRAHC"]
```

To combine the incoming token stream with the output of a filter chain, use `TeeFilter` and make one of the filters a *`PassFilter`*.

```
>>> f1 = PassFilter()
>>> f2 = BiWordFilter()
>>> ana = RegexTokenizer(r"\S+") | TeeFilter(f1, f2) | LowercaseFilter()
>>> [token.text for token in ana(target)]
["alfa", "alfa-bravo", "bravo", "bravo-charlie", "charlie"]
```

class whoosh.analysis.**ReverseTextFilter**
Reverses the text of each token.

```
>>> ana = RegexTokenizer() | ReverseTextFilter()
>>> [token.text for token in ana("hello there")]
["olleh", "ereht"]
```

class whoosh.analysis.**LowercaseFilter**
Uses unicode.lower() to lowercase token text.

```
>>> rext = RegexTokenizer()
>>> stream = rext("This is a TEST")
>>> [token.text for token in LowercaseFilter(stream)]
["this", "is", "a", "test"]
```

class whoosh.analysis.**StripFilter**
Calls unicode.strip() on the token text.

class whoosh.analysis.**StopFilter**(*stoplist=frozenset(['and', 'is', 'it', 'an', 'as', 'at', 'have', 'in', 'yet', 'if', 'from', 'for', 'when', 'by', 'to', 'you', 'be', 'we', 'that', 'may', 'not', 'with', 'tbd', 'a', 'on', 'your', 'this', 'of', 'us', 'will', 'can', 'the', 'or', 'are'])*, *minsize=2*, *maxsize=None*, *renumber=True*, *lang=None*)
Marks "stop" words (words too common to index) in the stream (and by default removes them).

Make sure you precede this filter with a *LowercaseFilter*.

```
>>> stopper = RegexTokenizer() | StopFilter()
>>> [token.text for token in stopper(u"this is a test")]
["test"]
>>> es_stopper = RegexTokenizer() | StopFilter(lang="es")
>>> [token.text for token in es_stopper(u"el lapiz es en la mesa")]
["lapiz", "mesa"]
```

The list of available languages is in *whoosh.lang.languages*. You can use whoosh.lang.has_stopwords() to check if a given language has a stop word list available.

Parameters

- **stoplist** – A collection of words to remove from the stream. This is converted to a frozenset. The default is a list of common English stop words.
- **minsize** – The minimum length of token texts. Tokens with text smaller than this will be stopped. The default is 2.
- **maxsize** – The maximum length of token texts. Tokens with text larger than this will be stopped. Use None to allow any length.
- **renumber** – Change the 'pos' attribute of unstopped tokens to reflect their position with the stopped words removed.
- **lang** – Automatically get a list of stop words for the given language

class whoosh.analysis.**StemFilter**(*stemfn=<function stem>*, *lang=None*, *ignore=None*, *cachesize=50000*)
Stems (removes suffixes from) the text of tokens using the Porter stemming algorithm. Stemming attempts to reduce multiple forms of the same root word (for example, "rendering", "renders", "rendered", etc.) to a single word in the index.

```
>>> stemmer = RegexTokenizer() | StemFilter()
>>> [token.text for token in stemmer("fundamentally willows")]
["fundament", "willow"]
```

You can pass your own stemming function to the StemFilter. The default is the Porter stemming algorithm for English.

```
>>> stemfilter = StemFilter(stem_function)
```

You can also use one of the Snowball stemming functions by passing the *lang* keyword argument.

```
>>> stemfilter = StemFilter(lang="ru")
```

The list of available languages is in *whoosh.lang.languages*. You can use whoosh.lang.has_stemmer() to check if a given language has a stemming function available.

By default, this class wraps an LRU cache around the stemming function. The `cachesize` keyword argument sets the size of the cache. To make the cache unbounded (the class caches every input), use `cachesize=-1`. To disable caching, use `cachesize=None`.

If you compile and install the py-stemmer library, the `PyStemmerFilter` provides slightly easier access to the language stemmers in that library.

Parameters

- **stemfn** – the function to use for stemming.
- **lang** – if not None, overrides the stemfn with a language stemmer from the `whoosh.lang.snowball` package.
- **ignore** – a set/list of words that should not be stemmed. This is converted into a frozenset. If you omit this argument, all tokens are stemmed.
- **cachesize** – the maximum number of words to cache. Use `-1` for an unbounded cache, or `None` for no caching.

class whoosh.analysis.**CharsetFilter**(*charmap*)

Translates the text of tokens by calling unicode.translate() using the supplied character mapping object. This is useful for case and accent folding.

The `whoosh.support.charset` module has a useful map for accent folding.

```
>>> from whoosh.support.charset import accent_map
>>> retokenizer = RegexTokenizer()
>>> chfilter = CharsetFilter(accent_map)
>>> [t.text for t in chfilter(retokenizer(u'café'))]
[u'cafe']
```

Another way to get a character mapping object is to convert a Sphinx charset table file using *whoosh.support.charset.charset_table_to_dict()*.

```
>>> from whoosh.support.charset import charset_table_to_dict
>>> from whoosh.support.charset import default_charset
>>> retokenizer = RegexTokenizer()
>>> charmap = charset_table_to_dict(default_charset)
>>> chfilter = CharsetFilter(charmap)
>>> [t.text for t in chfilter(retokenizer(u'Stra\xdfe'))]
[u'strase']
```

The Sphinx charset table format is described at http://www.sphinxsearch.com/docs/current.html#conf-charset-table.

Parameters **charmap** – a dictionary mapping from integer character numbers to unicode characters, as required by the unicode.translate() method.

class whoosh.analysis.**NgramFilter**(*minsize*, *maxsize=None*, *at=None*)

Splits token text into N-grams.

```
>>> rext = RegexTokenizer()
>>> stream = rext("hello there")
>>> ngf = NgramFilter(4)
>>> [token.text for token in ngf(stream)]
["hell", "ello", "ther", "here"]
```

Parameters

- **minsize** – The minimum size of the N-grams.

- **maxsize** – The maximum size of the N-grams. If you omit this parameter, maxsize == minsize.
- **at** – If 'start', only take N-grams from the start of each word. if 'end', only take N-grams from the end of each word. Otherwise, take all N-grams from the word (the default).

class whoosh.analysis.**IntraWordFilter**(*delims=u'-_"'()!@#$%^&*[]{}<>\|;:, ./?'~=+'*, *splitwords=True*, *splitnums=True*, *mergewords=False*, *mergenums=False*)

Splits words into subwords and performs optional transformations on subword groups. This filter is funtionally based on yonik's WordDelimiterFilter in Solr, but shares no code with it.

•Split on intra-word delimiters, e.g. *Wi-Fi* -> *Wi*, *Fi*.

•When splitwords=True, split on case transitions, e.g. *PowerShot* -> *Power*, *Shot*.

•When splitnums=True, split on letter-number transitions, e.g. *SD500* -> *SD*, *500*.

•Leading and trailing delimiter characters are ignored.

•Trailing possesive "'s" removed from subwords, e.g. *O'Neil's* -> *O*, *Neil*.

The mergewords and mergenums arguments turn on merging of subwords.

When the merge arguments are false, subwords are not merged.

•*PowerShot* -> *0*:*Power*, *1*:*Shot* (where *0* and *1* are token positions).

When one or both of the merge arguments are true, consecutive runs of alphabetic and/or numeric subwords are merged into an additional token with the same position as the last sub-word.

•*PowerShot* -> *0*:*Power*, *1*:*Shot*, *1*:*PowerShot*

•*A's+B's&C's* -> *0*:*A*, *1*:*B*, *2*:*C*, *2*:*ABC*

•*Super-Duper-XL500-42-AutoCoder!* -> *0*:*Super*, *1*:*Duper*, *2*:*XL*, *2*:*SuperDuperXL*, *3*:*500*, *4*:*42*, *4*:*50042*, *5*:*Auto*, *6*:*Coder*, *6*:*AutoCoder*

When using this filter you should use a tokenizer that only splits on whitespace, so the tokenizer does not remove intra-word delimiters before this filter can see them, and put this filter before any use of LowercaseFilter.

```
>>> rt = RegexTokenizer(r"\S+")
>>> iwf = IntraWordFilter()
>>> lcf = LowercaseFilter()
>>> analyzer = rt | iwf | lcf
```

One use for this filter is to help match different written representations of a concept. For example, if the source text contained *wi-fi*, you probably want *wifi*, *WiFi*, *wi-fi*, etc. to match. One way of doing this is to specify mergewords=True and/or mergenums=True in the analyzer used for indexing, and mergewords=False / mergenums=False in the analyzer used for querying.

```
>>> iwf_i = IntraWordFilter(mergewords=True, mergenums=True)
>>> iwf_q = IntraWordFilter(mergewords=False, mergenums=False)
>>> iwf = MultiFilter(index=iwf_i, query=iwf_q)
>>> analyzer = RegexTokenizer(r"\S+") | iwf | LowercaseFilter()
```

(See *MultiFilter*.)

Parameters

- **delims** – a string of delimiter characters.
- **splitwords** – if True, split at case transitions, e.g. *PowerShot* -> *Power*, *Shot*

- **splitnums** – if True, split at letter-number transitions, e.g. *SD500* -> *SD*, *500*
- **mergewords** – merge consecutive runs of alphabetic subwords into an additional token with the same position as the last subword.
- **mergenums** – merge consecutive runs of numeric subwords into an additional token with the same position as the last subword.

class whoosh.analysis.**CompoundWordFilter**(*wordset*, *keep_compound=True*)

Given a set of words (or any object with a __contains__ method), break any tokens in the stream that are composites of words in the word set into their individual parts.

Given the correct set of words, this filter can break apart run-together words and trademarks (e.g. "turbosquid", "applescript"). It can also be useful for agglutinative languages such as German.

The keep_compound argument lets you decide whether to keep the compound word in the token stream along with the word segments.

```
>>> cwf = CompoundWordFilter(wordset, keep_compound=True)
>>> analyzer = RegexTokenizer(r"\S+") | cwf
>>> [t.text for t in analyzer("I do not like greeneggs and ham")
["I", "do", "not", "like", "greeneggs", "green", "eggs", "and", "ham"]
>>> cwf.keep_compound = False
>>> [t.text for t in analyzer("I do not like greeneggs and ham")
["I", "do", "not", "like", "green", "eggs", "and", "ham"]
```

Parameters

- **wordset** – an object with a __contains__ method, such as a set, containing strings to look for inside the tokens.
- **keep_compound** – if True (the default), the original compound token will be retained in the stream before the subwords.

class whoosh.analysis.**BiWordFilter**(*sep='-'*)

Merges adjacent tokens into "bi-word" tokens, so that for example:

```
"the", "sign", "of", "four"
```

becomes:

```
"the-sign", "sign-of", "of-four"
```

This can be used to create fields for pseudo-phrase searching, where if all the terms match the document probably contains the phrase, but the searching is faster than actually doing a phrase search on individual word terms.

The BiWordFilter is much faster than using the otherwise equivalent ShingleFilter(2).

class whoosh.analysis.**ShingleFilter**(*size=2*, *sep='-'*)

Merges a certain number of adjacent tokens into multi-word tokens, so that for example:

```
"better", "a", "witty", "fool", "than", "a", "foolish", "wit"
```

with ShingleFilter(3, ' ') becomes:

```
'better a witty', 'a witty fool', 'witty fool than', 'fool than a',
'than a foolish', 'a foolish wit'
```

This can be used to create fields for pseudo-phrase searching, where if all the terms match the document probably contains the phrase, but the searching is faster than actually doing a phrase search on individual word terms.

If you're using two-word shingles, you should use the functionally equivalent `BiWordFilter` instead because it's faster than `ShingleFilter`.

class whoosh.analysis.**DelimitedAttributeFilter**(*delimiter='^'*, *attribute='boost'*, *default=1.0*, *type=<type 'float'>*)

Looks for delimiter characters in the text of each token and stores the data after the delimiter in a named attribute on the token.

The defaults are set up to use the ^ character as a delimiter and store the value after the ^ as the boost for the token.

```
>>> daf = DelimitedAttributeFilter(delimiter="^", attribute="boost")
>>> ana = RegexTokenizer("\\S+") | DelimitedAttributeFilter()
>>> for t in ana(u("image render^2 file^0.5"))
...    print("%r %f" % (t.text, t.boost))
'image' 1.0
'render' 2.0
'file' 0.5
```

Note that you need to make sure your tokenizer includes the delimiter and data as part of the token!

Parameters

- **delimiter** – a string that, when present in a token's text, separates the actual text from the "data" payload.
- **attribute** – the name of the attribute in which to store the data on the token.
- **default** – the value to use for the attribute for tokens that don't have delimited data.
- **type** – the type of the data, for example `str` or `float`. This is used to convert the string value of the data before storing it in the attribute.

class whoosh.analysis.**DoubleMetaphoneFilter**(*primary_boost=1.0*, *secondary_boost=0.5*, *combine=False*)

Transforms the text of the tokens using Lawrence Philips's Double Metaphone algorithm. This algorithm attempts to encode words in such a way that similar-sounding words reduce to the same code. This may be useful for fields containing the names of people and places, and other uses where tolerance of spelling differences is desireable.

Parameters

- **primary_boost** – the boost to apply to the token containing the primary code.
- **secondary_boost** – the boost to apply to the token containing the secondary code, if any.
- **combine** – if True, the original unencoded tokens are kept in the stream, preceding the encoded tokens.

class whoosh.analysis.**SubstitutionFilter**(*pattern*, *replacement*)

Performs a regular expression substitution on the token text.

This is especially useful for removing text from tokens, for example hyphens:

```
ana = RegexTokenizer(r"\S+") | SubstitutionFilter("-", "")
```

Because it has the full power of the re.sub() method behind it, this filter can perform some fairly complex transformations. For example, to take tokens like `'a=b'`, `'c=d'`, `'e=f'` and change them to `'b=a'`, `'d=c'`, `'f=e'`:

```
# Analyzer that swaps the text on either side of an equal sign
rt = RegexTokenizer(r"\S+")
sf = SubstitutionFilter("([^/]*)/(./*)", r"\2/\1")
ana = rt | sf
```

Parameters

- **pattern** – a pattern string or compiled regular expression object describing the text to replace.
- **replacement** – the substitution text.

Token classes and functions

class whoosh.analysis.**Token**(*positions=False*, *chars=False*, *removestops=True*, *mode=''*, ***kwargs*)
Represents a "token" (usually a word) extracted from the source text being indexed.

See "Advanced analysis" in the user guide for more information.

Because object instantiation in Python is slow, tokenizers should create ONE SINGLE Token object and YIELD IT OVER AND OVER, changing the attributes each time.

This trick means that consumers of tokens (i.e. filters) must never try to hold onto the token object between loop iterations, or convert the token generator into a list. Instead, save the attributes between iterations, not the object:

```
def RemoveDuplicatesFilter(self, stream):
    # Removes duplicate words.
    lasttext = None
    for token in stream:
        # Only yield the token if its text doesn't
        # match the previous token.
        if lasttext != token.text:
            yield token
        lasttext = token.text
```

...or, call token.copy() to get a copy of the token object.

Parameters

- **positions** – Whether tokens should have the token position in the 'pos' attribute.
- **chars** – Whether tokens should have character offsets in the 'startchar' and 'endchar' attributes.
- **removestops** – whether to remove stop words from the stream (if the tokens pass through a stop filter).
- **mode** – contains a string describing the purpose for which the analyzer is being called, i.e. 'index' or 'query'.

whoosh.analysis.**unstopped**(*tokenstream*)
Removes tokens from a token stream where token.stopped = True.

codec.base module

This module contains base classes/interfaces for "codec" objects.

Classes

class whoosh.codec.base.**Codec**

class whoosh.codec.base.**PerDocumentWriter**

class whoosh.codec.base.**FieldWriter**

class whoosh.codec.base.**PostingsWriter**

written()
Returns True if this object has already written to disk.

class whoosh.codec.base.**TermsReader**

class whoosh.codec.base.**PerDocumentReader**

all_doc_ids()
Returns an iterator of all (undeleted) document IDs in the reader.

class whoosh.codec.base.**Segment**(*indexname*)
Do not instantiate this object directly. It is used by the Index object to hold information about a segment. A list of objects of this class are pickled as part of the TOC file.

The TOC file stores a minimal amount of information – mostly a list of Segment objects. Segments are the real reverse indexes. Having multiple segments allows quick incremental indexing: just create a new segment for the new documents, and have the index overlay the new segment over previous ones for purposes of reading/search. "Optimizing" the index combines the contents of existing segments into one (removing any deleted documents along the way).

create_file(*storage*, *ext*, ***kwargs*)
Convenience method to create a new file in the given storage named with this segment's ID and the given extension. Any keyword arguments are passed to the storage's create_file method.

delete_document(*docnum*, *delete=True*)
Deletes the given document number. The document is not actually removed from the index until it is optimized.

Parameters

- **docnum** – The document number to delete.
- **delete** – If False, this undeletes a deleted document.

deleted_count()
Returns the total number of deleted documents in this segment.

doc_count()
Returns the number of (undeleted) documents in this segment.

doc_count_all()
Returns the total number of documents, DELETED OR UNDELETED, in this segment.

has_deletions()
Returns True if any documents in this segment are deleted.

is_deleted(*docnum*)
Returns True if the given document number is deleted.

open_file(*storage*, *ext*, ***kwargs*)
Convenience method to open a file in the given storage named with this segment's ID and the given extension. Any keyword arguments are passed to the storage's open_file method.

collectors module

This module contains "collector" objects. Collectors provide a way to gather "raw" results from a *whoosh.matching.Matcher* object, implement sorting, filtering, collation, etc., and produce a *whoosh.searching.Results* object.

The basic collectors are:

TopCollector Returns the top N matching results sorted by score, using block-quality optimizations to skip blocks of documents that can't contribute to the top N. The *whoosh.searching.Searcher.search()* method uses this type of collector by default or when you specify a `limit`.

UnlimitedCollector Returns all matching results sorted by score. The *whoosh.searching.Searcher.search()* method uses this type of collector when you specify `limit=None` or you specify a limit equal to or greater than the number of documents in the searcher.

SortingCollector Returns all matching results sorted by a `whoosh.sorting.Facet` object. The *whoosh.searching.Searcher.search()* method uses this type of collector when you use the `sortedby` parameter.

Here's an example of a simple collector that instead of remembering the matched documents just counts up the number of matches:

```
class CountingCollector(Collector):
    def prepare(self, top_searcher, q, context):
        # Always call super method in prepare
        Collector.prepare(self, top_searcher, q, context)

        self.count = 0

    def collect(self, sub_docnum):
        self.count += 1

c = CountingCollector()
mysearcher.search_with_collector(myquery, c)
print(c.count)
```

There are also several wrapping collectors that extend or modify the functionality of other collectors. The meth:*whoosh.searching.Searcher.search* method uses many of these when you specify various parameters.

NOTE: collectors are not designed to be reentrant or thread-safe. It is generally a good idea to create a new collector for each search.

Base classes

class `whoosh.collectors.`**`Collector`**

Base class for collectors.

`all_ids``()`

Returns a sequence of docnums matched in this collector. (Only valid after the collector is run.)

The default implementation is based on the docset. If a collector does not maintain the docset, it will need to override this method.

`collect``(`*sub_docnum*`)`

This method is called for every matched document. It should do the work of adding a matched document to the results, and it should return an object to use as a "sorting key" for the given document (such as the document's score, a key generated by a facet, or just None). Subclasses must implement this method.

If you want the score for the current document, use `self.matcher.score()`.

Overriding methods should add the current document offset (`self.offset`) to the `sub_docnum` to get the top-level document number for the matching document to add to results.

Parameters **sub_docnum** – the document number of the current match within the current subsearcher. You must add `self.offset` to this number to get the document's top-level document number.

collect_matches()
This method calls *Collector.matches()* and then for each matched document calls *Collector.collect()*. Sub-classes that want to intervene between finding matches and adding them to the collection (for example, to filter out certain documents) can override this method.

computes_count()
Returns True if the collector naturally computes the exact number of matching documents. Collectors that use block optimizations will return False since they might skip blocks containing matching documents.

Note that if this method returns False you can still call *count()*, but it means that method might have to do more work to calculate the number of matching documents.

count()
Returns the total number of documents matched in this collector. (Only valid after the collector is run.)

The default implementation is based on the docset. If a collector does not maintain the docset, it will need to override this method.

finish()
This method is called after a search.

Subclasses can override this to perform set-up work, but they should still call the superclass's method because it sets several necessary attributes on the collector object:

self.runtime The time (in seconds) the search took.

matches()
Yields a series of relative document numbers for matches in the current subsearcher.

prepare(*top_searcher*, *q*, *context*)
This method is called before a search.

Subclasses can override this to perform set-up work, but they should still call the superclass's method because it sets several necessary attributes on the collector object:

self.top_searcher The top-level searcher.

self.q The query object

self.context `context.needs_current` controls whether a wrapping collector requires that this collector's matcher be in a valid state at every call to `collect()`. If this is `False`, the collector is free to use faster methods that don't necessarily keep the matcher updated, such as `matcher.all_ids()`.

Parameters

- **top_searcher** – the top-level *whoosh.searching.Searcher* object.
- **q** – the *whoosh.query.Query* object being searched for.
- **context** – a `whoosh.searching.SearchContext` object containing information about the search.

remove(*global_docnum*)
Removes a document from the collector. Not that this method uses the global document number as opposed to `Collector.collect()` which takes a segment-relative docnum.

results()
Returns a `Results` object containing the results of the search. Subclasses must implement this method

set_subsearcher(*subsearcher*, *offset*)
This method is called each time the collector starts on a new sub-searcher.

Subclasses can override this to perform set-up work, but they should still call the superclass's method because it sets several necessary attributes on the collector object:

self.subsearcher The current sub-searcher. If the top-level searcher is atomic, this is the same as the top-level searcher.

self.offset The document number offset of the current searcher. You must add this number to the document number passed to `Collector.collect()` to get the top-level document number for use in results.

self.matcher A `whoosh.matching.Matcher` object representing the matches for the query in the current sub-searcher.

sort_key(*sub_docnum*)
Returns a sorting key for the current match. This should return the same value returned by `Collector.collect()`, but without the side effect of adding the current document to the results.

If the collector has been prepared with `context.needs_current=True`, this method can use `self.matcher` to get information, for example the score. Otherwise, it should only use the provided `sub_docnum`, since the matcher may be in an inconsistent state.

Subclasses must implement this method.

class whoosh.collectors.**ScoredCollector**(*replace=10*)
Base class for collectors that sort the results based on document score.

Parameters **replace** – Number of matches between attempts to replace the matcher with a more efficient version.

class whoosh.collectors.**WrappingCollector**(*child*)
Base class for collectors that wrap other collectors.

Basic collectors

class whoosh.collectors.**TopCollector**(*limit=10*, *usequality=True*, ***kwargs*)
A collector that only returns the top "N" scored results.

Parameters

- **limit** – the maximum number of results to return.
- **usequality** – whether to use block-quality optimizations. This may be useful for debugging.

class whoosh.collectors.**UnlimitedCollector**(*reverse=False*)
A collector that returns **all** scored results.

class whoosh.collectors.**SortingCollector**(*sortedby*, *limit=10*, *reverse=False*)
A collector that returns results sorted by a given `whoosh.sorting.Facet` object. See *Sorting and faceting* for more information.

Parameters

- **sortedby** – see *Sorting and faceting*.

- **reverse** – If True, reverse the overall results. Note that you can reverse individual facets in a multi-facet sort key as well.

Wrappers

class whoosh.collectors.**FilterCollector**(*child*, *allow=None*, *restrict=None*)

A collector that lets you allow and/or restrict certain document numbers in the results:

```
uc = collectors.UnlimitedCollector()

ins = query.Term("chapter", "rendering")
outs = query.Term("status", "restricted")
fc = FilterCollector(uc, allow=ins, restrict=outs)

mysearcher.search_with_collector(myquery, fc)
print(fc.results())
```

This collector discards a document if:

•The allowed set is not None and a document number is not in the set, or

•The restrict set is not None and a document number is in the set.

(So, if the same document number is in both sets, that document will be discarded.)

If you have a reference to the collector, you can use `FilterCollector.filtered_count` to get the number of matching documents filtered out of the results by the collector.

Parameters

- **child** – the collector to wrap.
- **allow** – a query, Results object, or set-like object containing docnument numbers that are allowed in the results, or None (meaning everything is allowed).
- **restrict** – a query, Results object, or set-like object containing document numbers to disallow from the results, or None (meaning nothing is disallowed).

class whoosh.collectors.**FacetCollector**(*child*, *groupedby*, *maptype=None*)

A collector that creates groups of documents based on `whoosh.sorting.Facet` objects. See *Sorting and faceting* for more information.

This collector is used if you specify a `groupedby` parameter in the *whoosh.searching.Searcher.search()* method. You can use the *whoosh.searching.Results.groups()* method to access the facet groups.

If you have a reference to the collector can also use `FacetedCollector.facetmaps` to access the groups directly:

```
uc = collectors.UnlimitedCollector()
fc = FacetedCollector(uc, sorting.FieldFacet("category"))
mysearcher.search_with_collector(myquery, fc)
print(fc.facetmaps)
```

Parameters

- **groupedby** – see *Sorting and faceting*.
- **maptype** – a *whoosh.sorting.FacetMap* type to use for any facets that don't specify their own.

class whoosh.collectors.**CollapseCollector**(*child*, *keyfacet*, *limit=1*, *order=None*)

A collector that collapses results based on a facet. That is, it eliminates all but the top N results that share the same facet key. Documents with an empty key for the facet are never eliminated.

The "top" results within each group is determined by the result ordering (e.g. highest score in a scored search) or an optional second "ordering" facet.

If you have a reference to the collector you can use `CollapseCollector.collapsed_counts` to access the number of documents eliminated based on each key:

```
tc = TopCollector(limit=20)
cc = CollapseCollector(tc, "group", limit=3)
mysearcher.search_with_collector(myquery, cc)
print(cc.collapsed_counts)
```

See *Collapsing results* for more information.

Parameters

- **child** – the collector to wrap.
- **keyfacet** – a `whoosh.sorting.Facet` to use for collapsing. All but the top N documents that share a key will be eliminated from the results.
- **limit** – the maximum number of documents to keep for each key.
- **order** – an optional `whoosh.sorting.Facet` to use to determine the "top" document(s) to keep when collapsing. The default (`orderfaceet=None`) uses the results order (e.g. the highest score in a scored search).

class whoosh.collectors.**TimeLimitCollector**(*child*, *timelimit*, *greedy=False*, *use_alarm=True*)

A collector that raises a `TimeLimit` exception if the search does not complete within a certain number of seconds:

```
uc = collectors.UnlimitedCollector()
tlc = TimeLimitedCollector(uc, timelimit=5.8)
try:
    mysearcher.search_with_collector(myquery, tlc)
except collectors.TimeLimit:
    print("The search ran out of time!")

# We can still get partial results from the collector
print(tlc.results())
```

IMPORTANT: On Unix systems (systems where signal.SIGALRM is defined), the code uses signals to stop searching immediately when the time limit is reached. On Windows, the OS does not support this functionality, so the search only checks the time between each found document, so if a matcher is slow the search could exceed the time limit.

Parameters

- **child** – the collector to wrap.
- **timelimit** – the maximum amount of time (in seconds) to allow for searching. If the search takes longer than this, it will raise a `TimeLimit` exception.
- **greedy** – if `True`, the collector will finish adding the most recent hit before raising the `TimeLimit` exception.
- **use_alarm** – if `True` (the default), the collector will try to use signal.SIGALRM (on UNIX).

class whoosh.collectors.**TermsCollector**(*child*, *settype=<type 'set'>*)
A collector that remembers which terms appeared in which terms appeared in each matched document.

This collector is used if you specify `terms=True` in the *whoosh.searching.Searcher.search()* method.

If you have a reference to the collector can also use `TermsCollector.termslist` to access the term lists directly:

```
uc = collectors.UnlimitedCollector()
tc = TermsCollector(uc)
mysearcher.search_with_collector(myquery, tc)
# tc.termdocs is a dictionary mapping (fieldname, text) tuples to
# sets of document numbers
print(tc.termdocs)
# tc.docterms is a dictionary mapping docnums to lists of
# (fieldname, text) tuples
print(tc.docterms)
```

`columns` module

The API and implementation of columns may change in the next version of Whoosh!

This module contains "Column" objects which you can use as the argument to a Field object's `sortable=` keyword argument. Each field defines a default column type for when the user specifies `sortable=True` (the object returned by the field's `default_column()` method).

The default column type for most fields is `VarBytesColumn`, although numeric and date fields use `NumericColumn`. Expert users may use other field types that may be faster or more storage efficient based on the field contents. For example, if a field always contains one of a limited number of possible values, a `RefBytesColumn` will save space by only storing the values once. If a field's values are always a fixed length, the `FixedBytesColumn` saves space by not storing the length of each value.

A `Column` object basically exists to store configuration information and provides two important methods: `writer()` to return a `ColumnWriter` object and `reader()` to return a `ColumnReader` object.

Base classes

class whoosh.columns.**Column**
Represents a "column" of rows mapping docnums to document values.

The interface requires that you store the start offset of the column, the length of the column data, and the number of documents (rows) separately, and pass them to the reader object.

default_value(*reverse=False*)
Returns the default value for this column type.

reader(*dbfile*, *basepos*, *length*, *doccount*)
Returns a *ColumnReader* object you can use to read a column of this type from disk.

Parameters

- **dbfile** – the *StructFile* to read from.
- **basepos** – the offset within the file at which the column starts.
- **length** – the length in bytes of the column occupies in the file.
- **doccount** – the number of rows (documents) in the column.

stores_lists()
Returns True if the column stores a list of values for each document instead of a single value.

writer(*dbfile*)
Returns a *ColumnWriter* object you can use to use to create a column of this type on disk.

Parameters **dbfile** – the *StructFile* to write to.

class whoosh.columns.**ColumnWriter**(*dbfile*)

class whoosh.columns.**ColumnReader**(*dbfile*, *basepos*, *length*, *doccount*)

Basic columns

class whoosh.columns.**VarBytesColumn**(*allow_offsets=True*, *write_offsets_cutoff=32768*)
Stores variable length byte strings. See also *RefBytesColumn*.

The current implementation limits the total length of all document values a segment to 2 GB.

The default value (the value returned for a document that didn't have a value assigned to it at indexing time) is an empty bytestring (`b''`).

Parameters

- **allow_offsets** – Whether the column should write offsets when there are many rows in the column (this makes opening the column much faster). This argument is mostly for testing.
- **write_offsets_cutoff** – Write offsets (for speed) when there are more than this many rows in the column. This argument is mostly for testing.

class whoosh.columns.**FixedBytesColumn**(*fixedlen*, *default=None*)
Stores fixed-length byte strings.

Parameters

- **fixedlen** – the fixed length of byte strings in this column.
- **default** – the default value to use for documents that don't specify a value. If you don't specify a default, the column will use `b'\x00' * fixedlen`.

class whoosh.columns.**RefBytesColumn**(*fixedlen=0*, *default=None*)
Stores variable-length or fixed-length byte strings, similar to *VarBytesColumn* and *FixedBytesColumn*. However, where those columns stores a value for each document, this column keeps a list of all the unique values in the field, and for each document stores a short pointer into the unique list. For fields where the number of possible values is smaller than the number of documents (for example, "category" or "chapter"), this saves significant space.

This column type supports a maximum of 65535 unique values across all documents in a segment. You should generally use this column type where the number of unique values is in no danger of approaching that number (for example, a "tags" field). If you try to index too many unique values, the column will convert additional unique values to the default value and issue a warning using the `warnings` module (this will usually be preferable to crashing the indexer and potentially losing indexed documents).

Parameters

- **fixedlen** – an optional fixed length for the values. If you specify a number other than 0, the column will require all values to be the specified length.
- **default** – a default value to use for documents that don't specify one. If you don't specify a default, the column will use an empty bytestring (`b''`), or if you specify a fixed length, `b'\x00' * fixedlen`.

class whoosh.columns.**NumericColumn**(*typecode*, *default=0*)
Stores numbers (integers and floats) as compact binary.

Parameters

- **typecode** – a typecode character (as used by the struct module) specifying the number type. For example, "i" for signed integers.
- **default** – the default value to use for documents that don't specify one.

Technical columns

class whoosh.columns.**BitColumn**(*compress_at=2048*)
Stores a column of True/False values compactly.

Parameters **compress_at** – columns with this number of values or fewer will be saved compressed on disk, and loaded into RAM for reading. Set this to 0 to disable compression.

class whoosh.columns.**CompressedBytesColumn**(*level=3*, *module='zlib'*)
Stores variable-length byte strings compressed using deflate (by default).

Parameters

- **level** – the compression level to use.
- **module** – a string containing the name of the compression module to use. The default is "zlib". The module should export "compress" and "decompress" functions.

class whoosh.columns.**StructColumn**(*spec*, *default*)

class whoosh.columns.**PickleColumn**(*child*)
Converts arbitrary objects to pickled bytestrings and stores them using the wrapped column (usually a *VarBytesColumn* or *CompressedBytesColumn*).

If you can express the value you want to store as a number or bytestring, you should use the appropriate column type to avoid the time and size overhead of pickling and unpickling.

Experimental columns

class whoosh.columns.**ClampedNumericColumn**(*child*)
An experimental wrapper type for NumericColumn that clamps out-of-range values instead of raising an exception.

fields module

Contains functions and classes related to fields.

Schema class

class whoosh.fields.**Schema**(***fields*)
Represents the collection of fields in an index. Maps field names to FieldType objects which define the behavior of each field.

Low-level parts of the index use field numbers instead of field names for compactness. This class has several methods for converting between the field name, field number, and field object itself.

All keyword arguments to the constructor are treated as fieldname = fieldtype pairs. The fieldtype can be an instantiated FieldType object, or a FieldType sub-class (in which case the Schema will instantiate it with the default constructor before adding it).

For example:

```
s = Schema(content = TEXT,
           title = TEXT(stored = True),
           tags = KEYWORD(stored = True))
```

add(*name*, *fieldtype*, *glob=False*)
Adds a field to this schema.

Parameters

- **name** – The name of the field.
- **fieldtype** – An instantiated fields.FieldType object, or a FieldType subclass. If you pass an instantiated object, the schema will use that as the field configuration for this field. If you pass a FieldType subclass, the schema will automatically instantiate it with the default constructor.

copy()
Returns a shallow copy of the schema. The field instances are not deep copied, so they are shared between schema copies.

items()
Returns a list of ("fieldname", field_object) pairs for the fields in this schema.

names(*check_names=None*)
Returns a list of the names of the fields in this schema.

Parameters **check_names** – (optional) sequence of field names to check whether the schema accepts them as (dynamic) field names - acceptable names will also be in the result list. Note: You may also have static field names in check_names, that won't create duplicates in the result list. Unsupported names will not be in the result list.

scorable_names()
Returns a list of the names of fields that store field lengths.

stored_names()
Returns a list of the names of fields that are stored.

class whoosh.fields.**SchemaClass**(***fields*)
Allows you to define a schema using declarative syntax, similar to Django models:

```
class MySchema(SchemaClass):
    path = ID
    date = DATETIME
    content = TEXT
```

You can use inheritance to share common fields between schemas:

```
class Parent(SchemaClass):
    path = ID(stored=True)
    date = DATETIME

class Child1(Parent):
    content = TEXT(positions=False)
```

```
class Child2(Parent):
    tags = KEYWORD
```

This class overrides __new__ so instantiating your sub-class always results in an instance of `Schema`.

```
>>> class MySchema(SchemaClass):
...     title = TEXT(stored=True)
...     content = TEXT
...
>>> s = MySchema()
>>> type(s)
<class 'whoosh.fields.Schema'>
```

All keyword arguments to the constructor are treated as fieldname = fieldtype pairs. The fieldtype can be an instantiated FieldType object, or a FieldType sub-class (in which case the Schema will instantiate it with the default constructor before adding it).

For example:

```
s = Schema(content = TEXT,
           title = TEXT(stored = True),
           tags = KEYWORD(stored = True))
```

FieldType base class

class whoosh.fields.**FieldType**(*format*, *analyzer*, *scorable=False*, *stored=False*, *unique=False*, *multitoken_query='default'*, *sortable=False*, *vector=None*)

Represents a field configuration.

The FieldType object supports the following attributes:

- format (formats.Format): the storage format for posting blocks.
- analyzer (analysis.Analyzer): the analyzer to use to turn text into terms.
- scorable (boolean): whether searches against this field may be scored. This controls whether the index stores per-document field lengths for this field.
- stored (boolean): whether the content of this field is stored for each document. For example, in addition to indexing the title of a document, you usually want to store the title so it can be presented as part of the search results.
- unique (boolean): whether this field's value is unique to each document. For example, 'path' or 'ID'. IndexWriter.update_document() will use fields marked as 'unique' to find the previous version of a document being updated.
- multitoken_query is a string indicating what kind of query to use when a "word" in a user query parses into multiple tokens. The string is interpreted by the query parser. The strings understood by the default query parser are "first" (use first token only), "and" (join the tokens with an AND query), "or" (join the tokens with OR), "phrase" (join the tokens with a phrase query), and "default" (use the query parser's default join type).
- **vector (formats.Format or boolean): the format to use to store term** vectors. If not a `Format` object, any true value means to use the index format as the term vector format. Any flase value means don't store term vectors for this field.

The constructor for the base field type simply lets you supply your own attribute values. Subclasses may configure some or all of this for you.

clean()
Clears any cached information in the field and any child objects.

index(*value*, ***kwargs*)
Returns an iterator of (btext, frequency, weight, encoded_value) tuples for each unique word in the input value.

The default implementation uses the `analyzer` attribute to tokenize the value into strings, then encodes them into bytes using UTF-8.

parse_query(*fieldname*, *qstring*, *boost=1.0*)
When `self_parsing()` returns True, the query parser will call this method to parse basic query text.

parse_range(*fieldname*, *start*, *end*, *startexcl*, *endexcl*, *boost=1.0*)
When `self_parsing()` returns True, the query parser will call this method to parse range query text. If this method returns None instead of a query object, the parser will fall back to parsing the start and end terms using process_text().

process_text(*qstring*, *mode=''*, ***kwargs*)
Analyzes the given string and returns an iterator of token texts.

```
>>> field = fields.TEXT()
>>> list(field.process_text("The ides of March"))
["ides", "march"]
```

self_parsing()
Subclasses should override this method to return True if they want the query parser to call the field's `parse_query()` method instead of running the analyzer on text in this field. This is useful where the field needs full control over how queries are interpreted, such as in the numeric field type.

separate_spelling()
Returns True if the field stores unstemmed words in a separate field for spelling suggestions.

sortable_terms(*ixreader*, *fieldname*)
Returns an iterator of the "sortable" tokens in the given reader and field. These values can be used for sorting. The default implementation simply returns all tokens in the field.

This can be overridden by field types such as NUMERIC where some values in a field are not useful for sorting.

spellable_words(*value*)
Returns an iterator of each unique word (in sorted order) in the input value, suitable for inclusion in the field's word graph.

The default behavior is to call the field analyzer with the keyword argument `no_morph=True`, which should make the analyzer skip any morphological transformation filters (e.g. stemming) to preserve the original form of the words. Exotic field types may need to override this behavior.

spelling_fieldname(*fieldname*)
Returns the name of a field to use for spelling suggestions instead of this field.

Parameters **fieldname** – the name of this field.

subfields()
Returns an iterator of `(name_prefix, fieldobject)` pairs for the fields that need to be indexed when content is put in this field. The default implementation simply yields `("", self)`.

supports(*name*)
Returns True if the underlying format supports the given posting value type.

```
>>> field = TEXT()
>>> field.supports("positions")
True
>>> field.supports("chars")
False
```

to_bytes(*value*)
Returns a bytes representation of the given value, appropriate to be written to disk. The default implementation assumes a unicode value and encodes it using UTF-8.

to_column_value(*value*)
Returns an object suitable to be inserted into the document values column for this field. The default implementation simply calls `self.to_bytes(value)`.

tokenize(*value*, ***kwargs*)
Analyzes the given string and returns an iterator of Token objects (note: for performance reasons, actually the same token yielded over and over with different attributes).

Pre-made field types

class whoosh.fields.**ID**(*stored=False*, *unique=False*, *field_boost=1.0*, *sortable=False*, *analyzer=None*)
Configured field type that indexes the entire value of the field as one token. This is useful for data you don't want to tokenize, such as the path of a file.

Parameters **stored** – Whether the value of this field is stored with the document.

class whoosh.fields.**IDLIST**(*stored=False*, *unique=False*, *expression=None*, *field_boost=1.0*)
Configured field type for fields containing IDs separated by whitespace and/or punctuation (or anything else, using the expression param).

Parameters

- **stored** – Whether the value of this field is stored with the document.
- **unique** – Whether the value of this field is unique per-document.
- **expression** – The regular expression object to use to extract tokens. The default expression breaks tokens on CRs, LFs, tabs, spaces, commas, and semicolons.

class whoosh.fields.**STORED**
Configured field type for fields you want to store but not index.

class whoosh.fields.**KEYWORD**(*stored=False*, *lowercase=False*, *commas=False*, *scorable=False*, *unique=False*, *field_boost=1.0*, *sortable=False*, *vector=None*, *analyzer=None*)
Configured field type for fields containing space-separated or comma-separated keyword-like data (such as tags). The default is to not store positional information (so phrase searching is not allowed in this field) and to not make the field scorable.

Parameters

- **stored** – Whether to store the value of the field with the document.
- **commas** – Whether this is a comma-separated field. If this is False (the default), it is treated as a space-separated field.
- **scorable** – Whether this field is scorable.

class whoosh.fields.**TEXT**(*analyzer=None*, *phrase=True*, *chars=False*, *stored=False*, *field_boost=1.0*, *multitoken_query='default'*, *spelling=False*, *sortable=False*, *lang=None*, *vector=None*, *spelling_prefix='spell_'*)

Configured field type for text fields (for example, the body text of an article). The default is to store positional information to allow phrase searching. This field type is always scorable.

Parameters

- **analyzer** – The analysis.Analyzer to use to index the field contents. See the analysis module for more information. If you omit this argument, the field uses analysis.StandardAnalyzer.
- **phrase** – Whether the store positional information to allow phrase searching.
- **chars** – Whether to store character ranges along with positions. If this is True, "phrase" is also implied.
- **stored** – Whether to store the value of this field with the document. Since this field type generally contains a lot of text, you should avoid storing it with the document unless you need to, for example to allow fast excerpts in the search results.
- **spelling** – if True, and if the field's analyzer changes the form of term text (such as a stemming analyzer), this field will store extra information in a separate field (named using the `spelling_prefix` keyword argument) to allow spelling suggestions to use the unchanged word forms as spelling suggestions.
- **sortable** – If True, make this field sortable using the default column type. If you pass a *whoosh.columns.Column* instance instead of True, the field will use the given column type.
- **lang** – automaticaly configure a *whoosh.analysis.LanguageAnalyzer* for the given language. This is ignored if you also specify an `analyzer`.
- **vector** – if this value evaluates to true, store a list of the terms in this field in each document. If the value is an instance of *whoosh.formats.Format*, the index will use the object to store the term vector. Any other true value (e.g. `vector=True`) will use the field's index format to store the term vector as well.

class whoosh.fields.**NUMERIC**(*numtype=<type 'int'>*, *bits=32*, *stored=False*, *unique=False*, *field_boost=1.0*, *decimal_places=0*, *shift_step=4*, *signed=True*, *sortable=False*, *default=None*)

Special field type that lets you index integer or floating point numbers in relatively short fixed-width terms. The field converts numbers to sortable bytes for you before indexing.

You specify the numeric type of the field (`int` or `float`) when you create the `NUMERIC` object. The default is `int`. For `int`, you can specify a size in bits (`32` or `64`). For both `int` and `float` you can specify a `signed` keyword argument (default is `True`).

```
>>> schema = Schema(path=STORED, position=NUMERIC(int, 64, signed=False))
>>> ix = storage.create_index(schema)
>>> with ix.writer() as w:
...     w.add_document(path="/a", position=5820402204)
...
```

You can also use the NUMERIC field to store Decimal instances by specifying a type of `int` or `long` and the `decimal_places` keyword argument. This simply multiplies each number by `(10 ** decimal_places)` before storing it as an integer. Of course this may throw away decimal prcesision (by truncating, not rounding) and imposes the same maximum value limits as `int`/`long`, but these may be acceptable for certain applications.

```
>>> from decimal import Decimal
>>> schema = Schema(path=STORED, position=NUMERIC(int, decimal_places=4))
>>> ix = storage.create_index(schema)
>>> with ix.writer() as w:
...     w.add_document(path="/a", position=Decimal("123.45")
...
```

Parameters

- **numtype** – the type of numbers that can be stored in this field, either `int`, `float`. If you use `Decimal`, use the `decimal_places` argument to control how many decimal places the field will store.
- **bits** – When `numtype` is `int`, the number of bits to use to store the number: 8, 16, 32, or 64.
- **stored** – Whether the value of this field is stored with the document.
- **unique** – Whether the value of this field is unique per-document.
- **decimal_places** – specifies the number of decimal places to save when storing Decimal instances. If you set this, you will always get Decimal instances back from the field.
- **shift_steps** – The number of bits of precision to shift away at each tiered indexing level. Values should generally be 1-8. Lower values yield faster searches but take up more space. A value of *0* means no tiered indexing.
- **signed** – Whether the numbers stored in this field may be negative.

class whoosh.fields.**DATETIME**(*stored=False*, *unique=False*, *sortable=False*)

Special field type that lets you index datetime objects. The field converts the datetime objects to sortable text for you before indexing.

Since this field is based on Python's datetime module it shares all the limitations of that module, such as the inability to represent dates before year 1 in the proleptic Gregorian calendar. However, since this field stores datetimes as an integer number of microseconds, it could easily represent a much wider range of dates if the Python datetime implementation ever supports them.

```
>>> schema = Schema(path=STORED, date=DATETIME)
>>> ix = storage.create_index(schema)
>>> w = ix.writer()
>>> w.add_document(path="/a", date=datetime.now())
>>> w.commit()
```

Parameters

- **stored** – Whether the value of this field is stored with the document.
- **unique** – Whether the value of this field is unique per-document.

class whoosh.fields.**BOOLEAN**(*stored=False*, *field_boost=1.0*)

Special field type that lets you index boolean values (True and False). The field converts the boolean values to text for you before indexing.

```
>>> schema = Schema(path=STORED, done=BOOLEAN)
>>> ix = storage.create_index(schema)
>>> w = ix.writer()
>>> w.add_document(path="/a", done=False)
>>> w.commit()
```

Parameters **stored** – Whether the value of this field is stored with the document.

class whoosh.fields.**NGRAM**(*minsize=2*, *maxsize=4*, *stored=False*, *field_boost=1.0*, *queryor=False*, *phrase=False*, *sortable=False*)

Configured field that indexes text as N-grams. For example, with a field type NGRAM(3,4), the value "hello" will be indexed as tokens "hel", "hell", "ell", "ello", "llo". This field type chops the entire text into N-grams, including whitespace and punctuation. See *NGRAMWORDS* for a field type that breaks the text into words first before chopping the words into N-grams.

Parameters

- **minsize** – The minimum length of the N-grams.
- **maxsize** – The maximum length of the N-grams.
- **stored** – Whether to store the value of this field with the document. Since this field type generally contains a lot of text, you should avoid storing it with the document unless you need to, for example to allow fast excerpts in the search results.
- **queryor** – if True, combine the N-grams with an Or query. The default is to combine N-grams with an And query.
- **phrase** – store positions on the N-grams to allow exact phrase searching. The default is off.

class whoosh.fields.**NGRAMWORDS**(*minsize=2*, *maxsize=4*, *stored=False*, *field_boost=1.0*, *tokenizer=None*, *at=None*, *queryor=False*, *sortable=False*)

Configured field that chops text into words using a tokenizer, lowercases the words, and then chops the words into N-grams.

Parameters

- **minsize** – The minimum length of the N-grams.
- **maxsize** – The maximum length of the N-grams.
- **stored** – Whether to store the value of this field with the document. Since this field type generally contains a lot of text, you should avoid storing it with the document unless you need to, for example to allow fast excerpts in the search results.
- **tokenizer** – an instance of whoosh.analysis.Tokenizer used to break the text into words.
- **at** – if 'start', only takes N-grams from the start of the word. If 'end', only takes N-grams from the end. Otherwise the default is to take all N-grams from each word.
- **queryor** – if True, combine the N-grams with an Or query. The default is to combine N-grams with an And query.

Exceptions

exception whoosh.fields.**FieldConfigurationError**

exception whoosh.fields.**UnknownFieldError**

filedb.filestore module

Base class

class whoosh.filedb.filestore.**Storage**

Abstract base class for storage objects.

A storage object is a virtual flat filesystem, allowing the creation and retrieval of file-like objects (*StructFile* objects). The default implementation (*FileStorage*) uses actual files in a directory.

All access to files in Whoosh goes through this object. This allows more different forms of storage (for example, in RAM, in a database, in a single file) to be used transparently.

For example, to create a *FileStorage* object:

```
# Create a storage object
st = FileStorage("indexdir")
# Create the directory if it doesn't already exist
st.create()
```

The *Storage.create()* method makes it slightly easier to swap storage implementations. The `create()` method handles set-up of the storage object. For example, `FileStorage.create()` creates the directory. A database implementation might create tables. This is designed to let you avoid putting implementation-specific setup code in your application.

close()

Closes any resources opened by this storage object. For some storage implementations this will be a no-op, but for others it is necessary to release locks and/or prevent leaks, so it's a good idea to call it when you're done with a storage object.

create()

Creates any required implementation-specific resources. For example, a filesystem-based implementation might create a directory, while a database implementation might create tables. For example:

```
from whoosh.filedb.filestore import FileStorage
# Create a storage object
st = FileStorage("indexdir")
# Create any necessary resources
st.create()
```

This method returns `self` so you can also say:

```
st = FileStorage("indexdir").create()
```

Storage implementations should be written so that calling create() a second time on the same storage

Returns a *Storage* instance.

create_file(*name*)

Creates a file with the given name in this storage.

Parameters **name** – the name for the new file.

Returns a *whoosh.filedb.structfile.StructFile* instance.

create_index(*schema*, *indexname='MAIN'*, *indexclass=None*)

Creates a new index in this storage.

```
>>> from whoosh import fields
>>> from whoosh.filedb.filestore import FileStorage
>>> schema = fields.Schema(content=fields.TEXT)
>>> # Create the storage directory
>>> st = FileStorage.create("indexdir")
```

```
>>> # Create an index in the storage
>>> ix = st.create_index(schema)
```

Parameters

- **schema** – the *whoosh.fields.Schema* object to use for the new index.
- **indexname** – the name of the index within the storage object. You can use this option to store multiple indexes in the same storage.
- **indexclass** – an optional custom `Index` sub-class to use to create the index files. The default is *whoosh.index.FileIndex*. This method will call the `create` class method on the given class to create the index.

Returns a *whoosh.index.Index* instance.

delete_file(*name*)
Removes the given file from this storage.

Parameters **name** – the name to delete.

destroy(**args*, ***kwargs*)
Removes any implementation-specific resources related to this storage object. For example, a filesystem-based implementation might delete a directory, and a database implementation might drop tables.

The arguments are implementation-specific.

file_exists(*name*)
Returns True if the given file exists in this storage.

Parameters **name** – the name to check.

Return type bool

file_length(*name*)
Returns the size (in bytes) of the given file in this storage.

Parameters **name** – the name to check.

Return type int

file_modified(*name*)
Returns the last-modified time of the given file in this storage (as a "ctime" UNIX timestamp).

Parameters **name** – the name to check.

Returns a "ctime" number.

index_exists(*indexname=None*)
Returns True if a non-empty index exists in this storage.

Parameters **indexname** – the name of the index within the storage object. You can use this option to store multiple indexes in the same storage.

Return type bool

list()
Returns a list of file names in this storage.

Returns a list of strings

lock(*name*)
Return a named lock object (implementing `.acquire()` and `.release()` methods). Different storage implementations may use different lock types with different guarantees. For example, the RamStorage

object uses Python thread locks, while the FileStorage object uses filesystem-based locks that are valid across different processes.

Parameters **name** – a name for the lock.

Returns a lock-like object.

open_file(*name*, **args*, ***kwargs*)
Opens a file with the given name in this storage.

Parameters **name** – the name for the new file.

Returns a *whoosh.filedb.structfile.StructFile* instance.

open_index(*indexname='MAIN'*, *schema=None*, *indexclass=None*)
Opens an existing index (created using *create_index()*) in this storage.

```
>>> from whoosh.filedb.filestore import FileStorage
>>> st = FileStorage("indexdir")
>>> # Open an index in the storage
>>> ix = st.open_index()
```

Parameters

- **indexname** – the name of the index within the storage object. You can use this option to store multiple indexes in the same storage.
- **schema** – if you pass in a *whoosh.fields.Schema* object using this argument, it will override the schema that was stored with the index.
- **indexclass** – an optional custom `Index` sub-class to use to open the index files. The default is *whoosh.index.FileIndex*. This method will instantiate the class with this storage object.

Returns a *whoosh.index.Index* instance.

optimize()
Optimizes the storage object. The meaning and cost of "optimizing" will vary by implementation. For example, a database implementation might run a garbage collection procedure on the underlying database.

rename_file(*frm*, *to*, *safe=False*)
Renames a file in this storage.

Parameters

- **frm** – The current name of the file.
- **to** – The new name for the file.
- **safe** – if True, raise an exception if a file with the new name already exists.

temp_storage(*name=None*)
Creates a new storage object for temporary files. You can call *Storage.destroy()* on the new storage when you're finished with it.

Parameters **name** – a name for the new storage. This may be optional or required depending on the storage implementation.

Return type *Storage*

Implementation classes

class whoosh.filedb.filestore.**FileStorage**(*path*, *supports_mmap=True*, *readonly=False*, *debug=False*)
Storage object that stores the index as files in a directory on disk.

Prior to version 3, the initializer would raise an IOError if the directory did not exist. As of version 3, the object does not check if the directory exists at initialization. This change is to support using the `FileStorage.create()` method.

Parameters

- **path** – a path to a directory.
- **supports_mmap** – if True (the default), use the `mmap` module to open memory mapped files. You can open the storage object with `supports_mmap=False` to force Whoosh to open files normally instead of with `mmap`.
- **readonly** – If `True`, the object will raise an exception if you attempt to create or rename a file.

class whoosh.filedb.filestore.**RamStorage**
Storage object that keeps the index in memory.

Helper functions

whoosh.filedb.filestore.**copy_storage**(*sourcestore*, *deststore*)
Copies the files from the source storage object to the destination storage object using `shutil.copyfileobj`.

whoosh.filedb.filestore.**copy_to_ram**(*storage*)
Copies the given FileStorage object into a new RamStorage object.

Return type *RamStorage*

Exceptions

exception whoosh.filedb.filestore.**ReadOnlyError**

filedb.filetables module

This module defines writer and reader classes for a fast, immutable on-disk key-value database format. The current format is based heavily on D. J. Bernstein's CDB format (http://cr.yp.to/cdb.html).

Hash file

class whoosh.filedb.filetables.**HashWriter**(*dbfile*, *magic='HSH3'*, *hashtype=0*)
Implements a fast on-disk key-value store. This hash uses a two-level hashing scheme, where a key is hashed, the low eight bits of the hash value are used to index into one of 256 hash tables. This is basically the CDB algorithm, but unlike CDB this object writes all data serially (it doesn't seek backwards to overwrite information at the end).

Also unlike CDB, this format uses 64-bit file pointers, so the file length is essentially unlimited. However, each key and value must be less than 2 GB in length.

Parameters

- **dbfile** – a *StructFile* object to write to.

- **magic** – the format tag bytes to write at the start of the file.
- **hashtype** – an integer indicating which hashing algorithm to use. Possible values are 0 (MD5), 1 (CRC32), or 2 (CDB hash).

add(*key*, *value*)
Adds a key/value pair to the file. Note that keys DO NOT need to be unique. You can store multiple values under the same key and retrieve them using `HashReader.all()`.

add_all(*items*)
Convenience method to add a sequence of `(key, value)` pairs. This is the same as calling `HashWriter.add()` on each pair in the sequence.

class whoosh.filedb.filetables.**HashReader**(*dbfile*, *length=None*, *magic='HSH3'*, *startoffset=0*)
Reader for the fast on-disk key-value files created by `HashWriter`.

Parameters

- **dbfile** – a `StructFile` object to read from.
- **length** – the length of the file data. This is necessary since the hashing information is written at the end of the file.
- **magic** – the format tag bytes to look for at the start of the file. If the file's format tag does not match these bytes, the object raises a `FileFormatError` exception.
- **startoffset** – the starting point of the file data.

all(*key*)
Yields a sequence of values associated with the given key.

classmethod open(*storage*, *name*)
Convenience method to open a hash file given a `whoosh.filedb.filestore.Storage` object and a name. This takes care of opening the file and passing its length to the initializer.

ranges_for_key(*key*)
Yields a sequence of `(datapos, datalength)` tuples associated with the given key.

Ordered Hash file

class whoosh.filedb.filetables.**OrderedHashWriter**(*dbfile*)
Implements an on-disk hash, but requires that keys be added in order. An `OrderedHashReader` can then look up "nearest keys" based on the ordering.

class whoosh.filedb.filetables.**OrderedHashReader**(*dbfile*, *length=None*, *magic='HSH3'*, *startoffset=0*)

Parameters

- **dbfile** – a `StructFile` object to read from.
- **length** – the length of the file data. This is necessary since the hashing information is written at the end of the file.
- **magic** – the format tag bytes to look for at the start of the file. If the file's format tag does not match these bytes, the object raises a `FileFormatError` exception.
- **startoffset** – the starting point of the file data.

filedb.structfile module

Classes

class whoosh.filedb.structfile.**StructFile**(*fileobj*, *name=None*, *onclose=None*)
Returns a "structured file" object that wraps the given file object and provides numerous additional methods for writing structured data, such as "write_varint" and "write_long".

close()
Closes the wrapped file.

flush()
Flushes the buffer of the wrapped file. This is a no-op if the wrapped file does not have a flush method.

read_pickle()
Reads a pickled object from the wrapped file.

read_string()
Reads a string from the wrapped file.

read_svarint()
Reads a variable-length encoded signed integer from the wrapped file.

read_tagint()
Reads a sometimes-compressed unsigned integer from the wrapped file. This is similar to the varint methods but uses a less compressed but faster format.

read_varint()
Reads a variable-length encoded unsigned integer from the wrapped file.

write_byte(*n*)
Writes a single byte to the wrapped file, shortcut for `file.write(chr(n))`.

write_pickle(*obj*, *protocol=-1*)
Writes a pickled representation of obj to the wrapped file.

write_string(*s*)
Writes a string to the wrapped file. This method writes the length of the string first, so you can read the string back without having to know how long it was.

write_svarint(*i*)
Writes a variable-length signed integer to the wrapped file.

write_tagint(*i*)
Writes a sometimes-compressed unsigned integer to the wrapped file. This is similar to the varint methods but uses a less compressed but faster format.

write_varint(*i*)
Writes a variable-length unsigned integer to the wrapped file.

class whoosh.filedb.structfile.**BufferFile**(*buf*, *name=None*, *onclose=None*)

class whoosh.filedb.structfile.**ChecksumFile**(**args*, ***kwargs*)

formats module

The classes in this module encode and decode posting information for a field. The field format essentially determines what information is stored about each occurance of a term.

Base class

class whoosh.formats.**Format**(*field_boost=1.0*, ***options*)
Abstract base class representing a storage format for a field or vector. Format objects are responsible for writing and reading the low-level representation of a field. It controls what kind/level of information to store about the indexed fields.

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

decode_as(*astype*, *valuestring*)
Interprets the encoded value string as 'astype', where 'astype' is for example "frequency" or "positions". This object must have a corresponding decode_<astype>() method.

decoder(*name*)
Returns the bound method for interpreting value as 'name', where 'name' is for example "frequency" or "positions". This object must have a corresponding Format.decode_<name>() method.

supports(*name*)
Returns True if this format supports interpreting its posting value as 'name' (e.g. "frequency" or "positions").

word_values(*value*, *analyzer*, ***kwargs*)
Takes the text value to be indexed and yields a series of ("tokentext", frequency, weight, valuestring) tuples, where frequency is the number of times "tokentext" appeared in the value, weight is the weight (a float usually equal to frequency in the absence of per-term boosts) and valuestring is encoded field-specific posting value for the token. For example, in a Frequency format, the value string would be the same as frequency; in a Positions format, the value string would encode a list of token positions at which "tokentext" occured.

Parameters

- **value** – The unicode text to index.
- **analyzer** – The analyzer to use to process the text.

Formats

class whoosh.formats.**Existence**(*field_boost=1.0*, ***options*)
Only indexes whether a given term occurred in a given document; it does not store frequencies or positions. This is useful for fields that should be searchable but not scorable, such as file path.

Supports: frequency, weight (always reports frequency = 1).

class whoosh.formats.**Frequency**(*field_boost=1.0*, *boost_as_freq=False*, ***options*)
Stores frequency information for each posting.

Supports: frequency, weight.

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

class whoosh.formats.**Positions**(*field_boost=1.0*, ***options*)
Stores position information in each posting, to allow phrase searching and "near" queries.

Supports: frequency, weight, positions, position_boosts (always reports position boost = 1.0).

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

class whoosh.formats.**Characters**(*field_boost=1.0*, ***options*)
Stores token position and character start and end information for each posting.

Supports: frequency, weight, positions, position_boosts (always reports position boost = 1.0), characters.

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

class whoosh.formats.**PositionBoosts**(*field_boost=1.0*, ***options*)
A format that stores positions and per-position boost information in each posting.

Supports: frequency, weight, positions, position_boosts.

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

class whoosh.formats.**CharacterBoosts**(*field_boost=1.0*, ***options*)
A format that stores positions, character start and end, and per-position boost information in each posting.

Supports: frequency, weight, positions, position_boosts, characters, character_boosts.

Parameters **field_boost** – A constant boost factor to scale to the score of all queries matching terms in this field.

highlight module

The highlight module contains classes and functions for displaying short excerpts from hit documents in the search results you present to the user, with query terms highlighted.

The highlighting system has four main elements.

- **Fragmenters** chop up the original text into __fragments__, based on the locations of matched terms in the text.
- **Scorers** assign a score to each fragment, allowing the system to rank the best fragments by whatever criterion.
- **Order functions** control in what order the top-scoring fragments are presented to the user. For example, you can show the fragments in the order they appear in the document (FIRST) or show higher-scoring fragments first (SCORE)
- **Formatters** turn the fragment objects into human-readable output, such as an HTML string.

See *How to create highlighted search result excerpts* for more information.

See *how to highlight terms in search results*.

Manual highlighting

class whoosh.highlight.**Highlighter**(*fragmenter=None*, *scorer=None*, *formatter=None*, *always_retokenize=False*, *order=<function FIRST>*)

whoosh.highlight.**highlight**(*text*, *terms*, *analyzer*, *fragmenter*, *formatter*, *top=3*, *scorer=None*, *minscore=1*, *order=<function FIRST>*, *mode='query'*)

Fragmenters

class whoosh.highlight.**Fragmenter**

fragment_matches(*text*, *matched_tokens*)
Yields `Fragment` objects based on the text and the matched terms.

Parameters

- **text** – the string being highlighted.
- **matched_tokens** – a list of `analysis.Token` objects representing the term matches in the string.

fragment_tokens(*text*, *all_tokens*)
Yields *Fragment* objects based on the tokenized text.

Parameters

- **text** – the string being highlighted.
- **all_tokens** – an iterator of `analysis.Token` objects from the string.

must_retokenize()
Returns True if this fragmenter requires retokenized text.

If this method returns True, the fragmenter's `fragment_tokens` method will be called with an iterator of ALL tokens from the text, with the tokens for matched terms having the `matched` attribute set to True.

If this method returns False, the fragmenter's `fragment_matches` method will be called with a LIST of matching tokens.

class whoosh.highlight.**WholeFragmenter**(*charlimit=32768*)
Doesn't fragment the token stream. This object just returns the entire entire stream as one "fragment". This is useful if you want to highlight the entire text.

Note that even if you use the *WholeFragmenter*, the highlight code will return no fragment if no terms matched in the given field. To return the whole fragment even in that case, call *highlights()* with *minscore=0*:

```
# Query where no terms match in the "text" field
q = query.Term("tag", "new")

r = mysearcher.search(q)
r.fragmenter = highlight.WholeFragmenter()
r.formatter = highlight.UppercaseFormatter()
# Since no terms in the "text" field matched, we get no fragments back
assert r[0].highlights("text") == ""

# If we lower the minimum score to 0, we get a fragment even though it
# has no matching terms
assert r[0].highlights("text", minscore=0) == "This is the text field."
```

class whoosh.highlight.**SentenceFragmenter**(*maxchars=200*, *sentencechars='.!?'*, *charlimit=32768*)
Breaks the text up on sentence end punctuation characters (".", "!", or "?"). This object works by looking in the original text for a sentence end as the next character after each token's 'endchar'.

When highlighting with this fragmenter, you should use an analyzer that does NOT remove stop words, for example:

```
sa = StandardAnalyzer(stoplist=None)
```

Parameters **maxchars** – The maximum number of characters allowed in a fragment.

class whoosh.highlight.**ContextFragmenter**(*maxchars=200*, *surround=20*, *charlimit=32768*)
Looks for matched terms and aggregates them with their surrounding context.

Parameters

- **maxchars** – The maximum number of characters allowed in a fragment.
- **surround** – The number of extra characters of context to add both before the first matched term and after the last matched term.

class whoosh.highlight.**PinpointFragmenter**(*maxchars=200*, *surround=20*, *autotrim=False*, *charlimit=32768*)

This is a NON-RETOKENIZING fragmenter. It builds fragments from the positions of the matched terms.

Parameters

- **maxchars** – The maximum number of characters allowed in a fragment.
- **surround** – The number of extra characters of context to add both before the first matched term and after the last matched term.
- **autotrim** – automatically trims text before the first space and after the last space in the fragments, to try to avoid truncated words at the start and end. For short fragments or fragments with long runs between spaces this may give strange results.

Scorers

class whoosh.highlight.**FragmentScorer**

class whoosh.highlight.**BasicFragmentScorer**

Formatters

class whoosh.highlight.**UppercaseFormatter**(*between='...'*)

Returns a string in which the matched terms are in UPPERCASE.

Parameters **between** – the text to add between fragments.

class whoosh.highlight.**HtmlFormatter**(*tagname='strong'*, *between='...'*, *classname='match'*, *termclass='term'*, *maxclasses=5*, *attrquote=""*)

Returns a string containing HTML formatting around the matched terms.

This formatter wraps matched terms in an HTML element with two class names. The first class name (set with the constructor argument `classname`) is the same for each match. The second class name (set with the constructor argument `termclass` is different depending on which term matched. This allows you to give different formatting (for example, different background colors) to the different terms in the excerpt.

```
>>> hf = HtmlFormatter(tagname="span", classname="match", termclass="term")
>>> hf(mytext, myfragments)
"The <span class="match term0">template</span> <span class="match term1">geometry
↪</span> is..."
```

This object maintains a dictionary mapping terms to HTML class names (e.g. `term0` and `term1` above), so that multiple excerpts will use the same class for the same term. If you want to re-use the same HtmlFormatter object with different searches, you should call HtmlFormatter.clear() between searches to clear the mapping.

Parameters

- **tagname** – the tag to wrap around matching terms.
- **between** – the text to add between fragments.
- **classname** – the class name to add to the elements wrapped around matching terms.
- **termclass** – the class name prefix for the second class which is different for each matched term.

- **maxclasses** – the maximum number of term classes to produce. This limits the number of classes you have to define in CSS by recycling term class names. For example, if you set maxclasses to 3 and have 5 terms, the 5 terms will use the CSS classes `term0`, `term1`, `term2`, `term0`, `term1`.

class `whoosh.highlight.`**GenshiFormatter**(*qname='strong'*, *between='...'*)
Returns a Genshi event stream containing HTML formatting around the matched terms.

Parameters

- **qname** – the QName for the tag to wrap around matched terms.
- **between** – the text to add between fragments.

Utility classes

class `whoosh.highlight.`**Fragment**(*text*, *matches*, *startchar=0*, *endchar=-1*)
Represents a fragment (extract) from a hit document. This object is mainly used to keep track of the start and end points of the fragment and the "matched" character ranges inside; it does not contain the text of the fragment or do much else.

The useful attributes are:

Fragment.text The entire original text from which this fragment is taken.

Fragment.matches An ordered list of objects representing the matched terms in the fragment. These objects have `startchar` and `endchar` attributes.

Fragment.startchar The index of the first character in the fragment.

Fragment.endchar The index of the last character in the fragment.

Fragment.matched_terms A `set` of the `text` of the matched terms in the fragment (if available).

Parameters

- **text** – the source text of the fragment.
- **matches** – a list of objects which have `startchar` and `endchar` attributes, and optionally a `text` attribute.
- **startchar** – the index into `text` at which the fragment starts. The default is 0.
- **endchar** – the index into `text` at which the fragment ends. The default is -1, which is interpreted as the length of `text`.

`support.bitvector` module

An implementation of an object that acts like a collection of on/off bits.

Base classes

class `whoosh.idsets.`**DocIdSet**
Base class for a set of positive integers, implementing a subset of the built-in `set` type's interface with extra docid-related methods.

This is a superclass for alternative set implementations to the built-in `set` which are more memory-efficient and specialized toward storing sorted lists of positive integers, though they will inevitably be slower than `set` for most operations since they're pure Python.

after(*i*)
Returns the next integer in the set after `i`, or None.

before(*i*)
Returns the previous integer in the set before `i`, or None.

first()
Returns the first (lowest) integer in the set.

invert_update(*size*)
Updates the set in-place to contain numbers in the range `[0 - size)` except numbers that are in this set.

last()
Returns the last (highest) integer in the set.

class whoosh.idsets.**BaseBitSet**

Implementation classes

class whoosh.idsets.**BitSet**(*source=None*, *size=0*)
A DocIdSet backed by an array of bits. This can also be useful as a bit array (e.g. for a Bloom filter). It is much more memory efficient than a large built-in set of integers, but wastes memory for sparse sets.

Parameters

- **maxsize** – the maximum size of the bit array.
- **source** – an iterable of positive integers to add to this set.
- **bits** – an array of unsigned bytes ("B") to use as the underlying bit array. This is used by some of the object's methods.

class whoosh.idsets.**OnDiskBitSet**(*dbfile*, *basepos*, *bytecount*)
A DocIdSet backed by an array of bits on disk.

```
>>> st = RamStorage()
>>> f = st.create_file("test.bin")
>>> bs = BitSet([1, 10, 15, 7, 2])
>>> bytecount = bs.to_disk(f)
>>> f.close()
>>> # ...
>>> f = st.open_file("test.bin")
>>> odbs = OnDiskBitSet(f, bytecount)
>>> list(odbs)
[1, 2, 7, 10, 15]
```

Parameters

- **dbfile** – a *StructFile* object to read from.
- **basepos** – the base position of the bytes in the given file.
- **bytecount** – the number of bytes to use for the bit array.

class whoosh.idsets.**SortedIntSet**(*source=None*, *typecode='I'*)
A DocIdSet backed by a sorted array of integers.

class whoosh.idsets.**MultiIdSet**(*idsets*, *offsets*)
Wraps multiple SERIAL sub-DocIdSet objects and presents them as an aggregated, read-only set.

Parameters

- **idsets** – a list of DocIdSet objects.
- **offsets** – a list of offsets corresponding to the DocIdSet objects in `idsets`.

`index` module

Contains the main functions/classes for creating, maintaining, and using an index.

Functions

`whoosh.index.`**`create_in`**(*dirname*, *schema*, *indexname=None*)
Convenience function to create an index in a directory. Takes care of creating a FileStorage object for you.

Parameters

- **dirname** – the path string of the directory in which to create the index.
- **schema** – a *`whoosh.fields.Schema`* object describing the index's fields.
- **indexname** – the name of the index to create; you only need to specify this if you are creating multiple indexes within the same storage object.

Returns *`Index`*

`whoosh.index.`**`open_dir`**(*dirname*, *indexname=None*, *readonly=False*, *schema=None*)
Convenience function for opening an index in a directory. Takes care of creating a FileStorage object for you. dirname is the filename of the directory in containing the index. indexname is the name of the index to create; you only need to specify this if you have multiple indexes within the same storage object.

Parameters

- **dirname** – the path string of the directory in which to create the index.
- **indexname** – the name of the index to create; you only need to specify this if you have multiple indexes within the same storage object.

`whoosh.index.`**`exists_in`**(*dirname*, *indexname=None*)
Returns True if dirname contains a Whoosh index.

Parameters

- **dirname** – the file path of a directory.
- **indexname** – the name of the index. If None, the default index name is used.

`whoosh.index.`**`exists`**(*storage*, *indexname=None*)
Deprecated; use `storage.index_exists()`.

Parameters

- **storage** – a store.Storage object.
- **indexname** – the name of the index. If None, the default index name is used.

`whoosh.index.`**`version_in`**(*dirname*, *indexname=None*)
Returns a tuple of (release_version, format_version), where release_version is the release version number of the Whoosh code that created the index – e.g. (0, 1, 24) – and format_version is the version number of the on-disk format used for the index – e.g. -102.

You should avoid attaching significance to the second number (the index version). This is simply a version number for the TOC file and probably should not have been exposed in a public interface. The best way to

check if the current version of Whoosh can open an index is to actually try to open it and see if it raises a `whoosh.index.IndexVersionError` exception.

Note that the release and format version are available as attributes on the Index object in Index.release and Index.version.

Parameters

- **dirname** – the file path of a directory containing an index.
- **indexname** – the name of the index. If None, the default index name is used.

Returns ((major_ver, minor_ver, build_ver), format_ver)

`whoosh.index.`**version**(*storage*, *indexname=None*)

Returns a tuple of (release_version, format_version), where release_version is the release version number of the Whoosh code that created the index – e.g. (0, 1, 24) – and format_version is the version number of the on-disk format used for the index – e.g. -102.

You should avoid attaching significance to the second number (the index version). This is simply a version number for the TOC file and probably should not have been exposed in a public interface. The best way to check if the current version of Whoosh can open an index is to actually try to open it and see if it raises a `whoosh.index.IndexVersionError` exception.

Note that the release and format version are available as attributes on the Index object in Index.release and Index.version.

Parameters

- **storage** – a store.Storage object.
- **indexname** – the name of the index. If None, the default index name is used.

Returns ((major_ver, minor_ver, build_ver), format_ver)

Base class

class `whoosh.index.`**Index**

Represents an indexed collection of documents.

add_field(*fieldname*, *fieldspec*)

Adds a field to the index's schema.

Parameters

- **fieldname** – the name of the field to add.
- **fieldspec** – an instantiated *whoosh.fields.FieldType* object.

close()

Closes any open resources held by the Index object itself. This may not close all resources being used everywhere, for example by a Searcher object.

doc_count()

Returns the total number of UNDELETED documents in this index.

doc_count_all()

Returns the total number of documents, DELETED OR UNDELETED, in this index.

field_length(*fieldname*)

Returns the total length of the field across all documents.

is_empty()

Returns True if this index is empty (that is, it has never had any documents successfully written to it.

last_modified()
Returns the last modified time of the index, or -1 if the backend doesn't support last-modified times.

latest_generation()
Returns the generation number of the latest generation of this index, or -1 if the backend doesn't support versioning.

max_field_length(*fieldname*)
Returns the maximum length of the field across all documents.

optimize()
Optimizes this index, if necessary.

reader(*reuse=None*)
Returns an IndexReader object for this index.

Parameters **reuse** – an existing reader. Some implementations may recycle resources from this existing reader to create the new reader. Note that any resources in the "recycled" reader that are not used by the new reader will be CLOSED, so you CANNOT use it afterward.

Return type *whoosh.reading.IndexReader*

refresh()
Returns a new Index object representing the latest generation of this index (if this object is the latest generation, or the backend doesn't support versioning, returns self).

Returns *Index*

remove_field(*fieldname*)
Removes the named field from the index's schema. Depending on the backend implementation, this may or may not actually remove existing data for the field from the index. Optimizing the index should always clear out existing data for a removed field.

searcher(***kwargs*)
Returns a Searcher object for this index. Keyword arguments are passed to the Searcher object's constructor.

Return type *whoosh.searching.Searcher*

up_to_date()
Returns True if this object represents the latest generation of this index. Returns False if this object is not the latest generation (that is, someone else has updated the index since you opened this object).

writer(***kwargs*)
Returns an IndexWriter object for this index.

Return type *whoosh.writing.IndexWriter*

Implementation

class whoosh.index.**FileIndex**(*storage*, *schema=None*, *indexname='MAIN'*)

Exceptions

exception whoosh.index.**LockError**

exception whoosh.index.**IndexError**
Generic index error.

exception whoosh.index.**IndexVersionError**(*msg*, *version*, *release=None*)
Raised when you try to open an index using a format that the current version of Whoosh cannot read. That is, when the index you're trying to open is either not backward or forward compatible with this version of Whoosh.

exception whoosh.index.**OutOfDateError**
Raised when you try to commit changes to an index which is not the latest generation.

exception whoosh.index.**EmptyIndexError**
Raised when you try to work with an index that has no indexed terms.

lang.morph_en module

Contains the variations() function for expanding an English word into multiple variations by programatically adding and removing suffixes.

Translated to Python from the com.sun.labs.minion.lexmorph.LiteMorph_en class of Sun's Minion search engine.

whoosh.lang.morph_en.**variations**(*word*)
Given an English word, returns a collection of morphological variations on the word by algorithmically adding and removing suffixes. The variation list may contain non-words (e.g. render -> renderment).

```
>>> variations("pull")
set(['pull', 'pullings', 'pullnesses', 'pullful', 'pullment', 'puller', ... ])
```

lang.porter module

Reimplementation of the Porter stemming algorithm in Python.

In my quick tests, this implementation about 3.5 times faster than the seriously weird Python linked from the official page.

whoosh.lang.porter.**stem**(*w*)
Uses the Porter stemming algorithm to remove suffixes from English words.

```
>>> stem("fundamentally")
"fundament"
```

lang.wordnet module

This module contains low-level functions and a high-level class for parsing the prolog file "wn_s.pl" from the WordNet prolog download into an object suitable for looking up synonyms and performing query expansion.

http://wordnetcode.princeton.edu/3.0/WNprolog-3.0.tar.gz

Thesaurus

class whoosh.lang.wordnet.**Thesaurus**
Represents the WordNet synonym database, either loaded into memory from the wn_s.pl Prolog file, or stored on disk in a Whoosh index.

This class allows you to parse the prolog file "wn_s.pl" from the WordNet prolog download into an object suitable for looking up synonyms and performing query expansion.

http://wordnetcode.princeton.edu/3.0/WNprolog-3.0.tar.gz

To load a Thesaurus object from the wn_s.pl file...

```
>>> t = Thesaurus.from_filename("wn_s.pl")
```

To save the in-memory Thesaurus to a Whoosh index...

```
>>> from whoosh.filedb.filestore import FileStorage
>>> fs = FileStorage("index")
>>> t.to_storage(fs)
```

To load a Thesaurus object from a Whoosh index...

```
>>> t = Thesaurus.from_storage(fs)
```

The Thesaurus object is thus usable in two ways:

- Parse the wn_s.pl file into memory (Thesaurus.from_*) and then look up synonyms in memory. This has a startup cost for parsing the file, and uses quite a bit of memory to store two large dictionaries, however synonym look-ups are very fast.
- Parse the wn_s.pl file into memory (Thesaurus.from_filename) then save it to an index (to_storage). From then on, open the thesaurus from the saved index (Thesaurus.from_storage). This has a large cost for storing the index, but after that it is faster to open the Thesaurus (than re-parsing the file) but slightly slower to look up synonyms.

Here are timings for various tasks on my (fast) Windows machine, which might give an idea of relative costs for in-memory vs. on-disk.

Task	Approx. time (s)
Parsing the wn_s.pl file	1.045
Saving to an on-disk index	13.084
Loading from an on-disk index	0.082
Look up synonyms for “light” (in memory)	0.0011
Look up synonyms for “light” (loaded from disk)	0.0028

Basically, if you can afford spending the memory necessary to parse the Thesaurus and then cache it, it’s faster. Otherwise, use an on-disk index.

classmethod `from_file` (*fileobj*)
Creates a Thesaurus object from the given file-like object, which should contain the WordNet wn_s.pl file.

```
>>> f = open("wn_s.pl")
>>> t = Thesaurus.from_file(f)
>>> t.synonyms("hail")
['acclaim', 'come', 'herald']
```

classmethod `from_filename` (*filename*)
Creates a Thesaurus object from the given filename, which should contain the WordNet wn_s.pl file.

```
>>> t = Thesaurus.from_filename("wn_s.pl")
>>> t.synonyms("hail")
['acclaim', 'come', 'herald']
```

classmethod `from_storage` (*storage*, *indexname='THES'*)
Creates a Thesaurus object from the given storage object, which should contain an index created by Thesaurus.to_storage().

```
>>> from whoosh.filedb.filestore import FileStorage
>>> fs = FileStorage("index")
```

```
>>> t = Thesaurus.from_storage(fs)
>>> t.synonyms("hail")
['acclaim', 'come', 'herald']
```

Parameters

- **storage** – A `whoosh.store.Storage` object from which to load the index.
- **indexname** – A name for the index. This allows you to store multiple indexes in the same storage object.

synonyms(*word*)
Returns a list of synonyms for the given word.

```
>>> thesaurus.synonyms("hail")
['acclaim', 'come', 'herald']
```

to_storage(*storage*, *indexname='THES'*)
Creates am index in the given storage object from the synonyms loaded from a WordNet file.

```
>>> from whoosh.filedb.filestore import FileStorage
>>> fs = FileStorage("index")
>>> t = Thesaurus.from_filename("wn_s.pl")
>>> t.to_storage(fs)
```

Parameters

- **storage** – A `whoosh.store.Storage` object in which to save the index.
- **indexname** – A name for the index. This allows you to store multiple indexes in the same storage object.

Low-level functions

whoosh.lang.wordnet.**parse_file**(*f*)
Parses the WordNet wn_s.pl prolog file and returns two dictionaries: word2nums and num2words.

whoosh.lang.wordnet.**synonyms**(*word2nums*, *num2words*, *word*)
Uses the word2nums and num2words dicts to look up synonyms for the given word. Returns a list of synonym strings.

whoosh.lang.wordnet.**make_index**(*storage*, *indexname*, *word2nums*, *num2words*)
Creates a Whoosh index in the given storage object containing synonyms taken from word2nums and num2words. Returns the Index object.

`matching` module

Matchers

class whoosh.matching.**Matcher**
Base class for all matchers.

all_ids()
Returns a generator of all IDs in the matcher.

What this method returns for a matcher that has already read some postings (whether it only yields the remaining postings or all postings from the beginning) is undefined, so it's best to only use this method on fresh matchers.

all_items()
Returns a generator of all (ID, encoded value) pairs in the matcher.

What this method returns for a matcher that has already read some postings (whether it only yields the remaining postings or all postings from the beginning) is undefined, so it's best to only use this method on fresh matchers.

block_quality()
Returns a quality measurement of the current block of postings, according to the current weighting algorithm. Raises `NoQualityAvailable` if the matcher or weighting do not support quality measurements.

children()
Returns an (possibly empty) list of the submatchers of this matcher.

copy()
Returns a copy of this matcher.

depth()
Returns the depth of the tree under this matcher, or 0 if this matcher does not have any children.

id()
Returns the ID of the current posting.

is_active()
Returns True if this matcher is still "active", that is, it has not yet reached the end of the posting list.

items_as(*astype*)
Returns a generator of all (ID, decoded value) pairs in the matcher.

What this method returns for a matcher that has already read some postings (whether it only yields the remaining postings or all postings from the beginning) is undefined, so it's best to only use this method on fresh matchers.

matching_terms(*id=None*)
Returns an iterator of `("fieldname", "termtext")` tuples for the **currently matching** term matchers in this tree.

max_quality()
Returns the maximum possible quality measurement for this matcher, according to the current weighting algorithm. Raises `NoQualityAvailable` if the matcher or weighting do not support quality measurements.

next()
Moves this matcher to the next posting.

replace(*minquality=0*)
Returns a possibly-simplified version of this matcher. For example, if one of the children of a UnionMatcher is no longer active, calling this method on the UnionMatcher will return the other child.

reset()
Returns to the start of the posting list.

Note that reset() may not do what you expect after you call *Matcher.replace()*, since this can mean calling reset() not on the original matcher, but on an optimized replacement.

score()
Returns the score of the current posting.

skip_to(*id*)
Moves this matcher to the first posting with an ID equal to or greater than the given ID.

skip_to_quality(*minquality*)
Moves this matcher to the next block with greater than the given minimum quality value.

spans()
Returns a list of `Span` objects for the matches in this document. Raises an exception if the field being searched does not store positions.

supports(*astype*)
Returns True if the field's format supports the named data type, for example 'frequency' or 'characters'.

supports_block_quality()
Returns True if this matcher supports the use of `quality` and `block_quality`.

term()
Returns a `("fieldname", "termtext")` tuple for the term this matcher matches, or None if this matcher is not a term matcher.

term_matchers()
Returns an iterator of term matchers in this tree.

value()
Returns the encoded value of the current posting.

value_as(*astype*)
Returns the value(s) of the current posting as the given type.

weight()
Returns the weight of the current posting.

whoosh.matching.**NullMatcher**

class whoosh.matching.**ListMatcher**(*ids*, *weights=None*, *values=None*, *format=None*, *scorer=None*, *position=0*, *all_weights=None*, *term=None*, *terminfo=None*)
Synthetic matcher backed by a list of IDs.

Parameters

- **ids** – a list of doc IDs.
- **weights** – a list of weights corresponding to the list of IDs. If this argument is not supplied, a list of 1.0 values is used.
- **values** – a list of encoded values corresponding to the list of IDs.
- **format** – a *whoosh.formats.Format* object representing the format of the field.
- **scorer** – a *whoosh.scoring.BaseScorer* object for scoring the postings.
- **term** – a `("fieldname", "text")` tuple, or None if this is not a term matcher.

class whoosh.matching.**WrappingMatcher**(*child*, *boost=1.0*)
Base class for matchers that wrap sub-matchers.

class whoosh.matching.**MultiMatcher**(*matchers*, *idoffsets*, *scorer=None*, *current=0*)
Serializes the results of a list of sub-matchers.

Parameters

- **matchers** – a list of Matcher objects.
- **idoffsets** – a list of offsets corresponding to items in the `matchers` list.

class whoosh.matching.**FilterMatcher**(*child*, *ids*, *exclude=False*, *boost=1.0*)
Filters the postings from the wrapped based on whether the IDs are present in or absent from a set.

Parameters

- **child** – the child matcher.
- **ids** – a set of IDs to filter by.
- **exclude** – by default, only IDs from the wrapped matcher that are **in** the set are used. If this argument is True, only IDs from the wrapped matcher that are **not in** the set are used.

class whoosh.matching.**BiMatcher**(*a*, *b*)
Base class for matchers that combine the results of two sub-matchers in some way.

class whoosh.matching.**AdditiveBiMatcher**(*a*, *b*)
Base class for binary matchers where the scores of the sub-matchers are added together.

class whoosh.matching.**UnionMatcher**(*a*, *b*)
Matches the union (OR) of the postings in the two sub-matchers.

class whoosh.matching.**DisjunctionMaxMatcher**(*a*, *b*, *tiebreak=0.0*)
Matches the union (OR) of two sub-matchers. Where both sub-matchers match the same posting, returns the weight/score of the higher-scoring posting.

class whoosh.matching.**IntersectionMatcher**(*a*, *b*)
Matches the intersection (AND) of the postings in the two sub-matchers.

class whoosh.matching.**AndNotMatcher**(*a*, *b*)
Matches the postings in the first sub-matcher that are NOT present in the second sub-matcher.

class whoosh.matching.**InverseMatcher**(*child*, *limit*, *missing=None*, *weight=1.0*, *id=0*)
Synthetic matcher, generates postings that are NOT present in the wrapped matcher.

class whoosh.matching.**RequireMatcher**(*a*, *b*)
Matches postings that are in both sub-matchers, but only uses scores from the first.

class whoosh.matching.**AndMaybeMatcher**(*a*, *b*)
Matches postings in the first sub-matcher, and if the same posting is in the second sub-matcher, adds their scores.

class whoosh.matching.**ConstantScoreMatcher**(*score=1.0*)

Exceptions

exception whoosh.matching.**ReadTooFar**
Raised when *next()* or *skip_to()* are called on an inactive matcher.

exception whoosh.matching.**NoQualityAvailable**
Raised when quality methods are called on a matcher that does not support block quality optimizations.

qparser module

Parser object

class whoosh.qparser.**QueryParser**(*fieldname*, *schema*, *plugins=None*, *termclass=<class 'whoosh.query.terms.Term'>*, *phraseclass=<class 'whoosh.query.positional.Phrase'>*, *group=<class 'whoosh.qparser.syntax.AndGroup'>*)
A hand-written query parser built on modular plug-ins. The default configuration implements a powerful fielded query language similar to Lucene's.

You can use the `plugins` argument when creating the object to override the default list of plug-ins, and/or use `add_plugin()` and/or `remove_plugin_class()` to change the plug-ins included in the parser.

```
>>> from whoosh import qparser
>>> parser = qparser.QueryParser("content", schema)
>>> parser.remove_plugin_class(qparser.WildcardPlugin)
>>> parser.add_plugin(qparser.PrefixPlugin())
>>> parser.parse(u"hello there")
And([Term("content", u"hello"), Term("content", u"there")])
```

Parameters

- **fieldname** – the default field – the parser uses this as the field for any terms without an explicit field.
- **schema** – a *whoosh.fields.Schema* object to use when parsing. The appropriate fields in the schema will be used to tokenize terms/phrases before they are turned into query objects. You can specify None for the schema to create a parser that does not analyze the text of the query, usually for testing purposes.
- **plugins** – a list of plugins to use. WhitespacePlugin is automatically included, do not put it in this list. This overrides the default list of plugins. Classes in the list will be automatically instantiated.
- **termclass** – the query class to use for individual search terms. The default is *whoosh.query.Term*.
- **phraseclass** – the query class to use for phrases. The default is *whoosh.query.Phrase*.
- **group** – the default grouping. `AndGroup` makes terms required by default. `OrGroup` makes terms optional by default.

add_plugin(*pin*)
Adds the given plugin to the list of plugins in this parser.

add_plugins(*pins*)
Adds the given list of plugins to the list of plugins in this parser.

default_set()
Returns the default list of plugins to use.

filterize(*nodes*, *debug=False*)
Takes a group of nodes and runs the filters provided by the parser's plugins.

filters()
Returns a priorized list of filter functions provided by the parser's currently configured plugins.

multitoken_query(*spec*, *texts*, *fieldname*, *termclass*, *boost*)
Returns a query for multiple texts. This method implements the intention specified in the field's `multitoken_query` attribute, which specifies what to do when strings that look like single terms to the parser turn out to yield multiple tokens when analyzed.

Parameters

- **spec** – a string describing how to join the text strings into a query. This is usually the value of the field's `multitoken_query` attribute.
- **texts** – a list of token strings.
- **fieldname** – the name of the field.

- **termclass** – the query class to use for single terms.
- **boost** – the original term's boost in the query string, should be applied to the returned query object.

parse(*text*, *normalize=True*, *debug=False*)
Parses the input string and returns a *whoosh.query.Query* object/tree.

Parameters

- **text** – the unicode string to parse.
- **normalize** – whether to call normalize() on the query object/tree before returning it. This should be left on unless you're trying to debug the parser output.

Return type *whoosh.query.Query*

process(*text*, *pos=0*, *debug=False*)
Returns a group of syntax nodes corresponding to the given text, tagged by the plugin Taggers and filtered by the plugin filters.

Parameters

- **text** – the text to tag.
- **pos** – the position in the text to start tagging at.

remove_plugin(*pi*)
Removes the given plugin object from the list of plugins in this parser.

remove_plugin_class(*cls*)
Removes any plugins of the given class from this parser.

replace_plugin(*plugin*)
Removes any plugins of the class of the given plugin and then adds it. This is a convenience method to keep from having to call `remove_plugin_class` followed by `add_plugin` each time you want to reconfigure a default plugin.

```
>>> qp = qparser.QueryParser("content", schema)
>>> qp.replace_plugin(qparser.NotPlugin("(^| )-"))
```

tag(*text*, *pos=0*, *debug=False*)
Returns a group of syntax nodes corresponding to the given text, created by matching the Taggers provided by the parser's plugins.

Parameters

- **text** – the text to tag.
- **pos** – the position in the text to start tagging at.

taggers()
Returns a priorized list of tagger objects provided by the parser's currently configured plugins.

term_query(*fieldname*, *text*, *termclass*, *boost=1.0*, *tokenize=True*, *removestops=True*)
Returns the appropriate query object for a single term in the query string.

Pre-made configurations

The following functions return pre-configured QueryParser objects.

whoosh.qparser.**MultifieldParser**(*fieldnames*, *schema*, *fieldboosts=None*, ***kwargs*)
Returns a QueryParser configured to search in multiple fields.

Instead of assigning unfielded clauses to a default field, this parser transforms them into an OR clause that searches a list of fields. For example, if the list of multi-fields is "f1", "f2" and the query string is "hello there", the class will parse "(f1:hello OR f2:hello) (f1:there OR f2:there)". This is very useful when you have two textual fields (e.g. "title" and "content") you want to search by default.

Parameters

- **fieldnames** – a list of field names to search.
- **fieldboosts** – an optional dictionary mapping field names to boosts.

whoosh.qparser.**SimpleParser**(*fieldname*, *schema*, ***kwargs*)
Returns a QueryParser configured to support only +, -, and phrase syntax.

whoosh.qparser.**DisMaxParser**(*fieldboosts*, *schema*, *tiebreak=0.0*, ***kwargs*)
Returns a QueryParser configured to support only +, -, and phrase syntax, and which converts individual terms into DisjunctionMax queries across a set of fields.

Parameters **fieldboosts** – a dictionary mapping field names to boosts.

Plug-ins

class whoosh.qparser.**Plugin**
Base class for parser plugins.

filters(*parser*)
Should return a list of `(filter_function, priority)` tuples to add to parser. Lower priority numbers run first.

Filter functions will be called with `(parser, groupnode)` and should return a group node.

taggers(*parser*)
Should return a list of `(Tagger, priority)` tuples to add to the syntax the parser understands. Lower priorities run first.

class whoosh.qparser.**SingleQuotePlugin**(*expr=None*)
Adds the ability to specify single "terms" containing spaces by enclosing them in single quotes.

class whoosh.qparser.**PrefixPlugin**(*expr=None*)
Adds the ability to specify prefix queries by ending a term with an asterisk.

This plugin is useful if you want the user to be able to create prefix but not wildcard queries (for performance reasons). If you are including the wildcard plugin, you should not include this plugin as well.

```
>>> qp = qparser.QueryParser("content", myschema)
>>> qp.remove_plugin_class(qparser.WildcardPlugin)
>>> qp.add_plugin(qparser.PrefixPlugin())
>>> q = qp.parse("pre*")
```

class whoosh.qparser.**WildcardPlugin**(*expr=None*)

class whoosh.qparser.**RegexPlugin**(*expr=None*)
Adds the ability to specify regular expression term queries.

The default syntax for a regular expression term is `r"termexpr"`.

```
>>> qp = qparser.QueryParser("content", myschema)
>>> qp.add_plugin(qparser.RegexPlugin())
>>> q = qp.parse('foo title:r"bar+"')
```

class whoosh.qparser.**BoostPlugin**(*expr=None*)
Adds the ability to boost clauses of the query using the circumflex.

```
>>> qp = qparser.QueryParser("content", myschema)
>>> q = qp.parse("hello there^2")
```

class whoosh.qparser.**GroupPlugin**(*openexpr='[(]'*, *closeexpr='[)]'*)
Adds the ability to group clauses using parentheses.

class whoosh.qparser.**EveryPlugin**(*expr=None*)

class whoosh.qparser.**FieldsPlugin**(*expr='(?P<text>\w+|[*]):'*, *remove_unknown=True*)
Adds the ability to specify the field of a clause.

Parameters

- **expr** – the regular expression to use for tagging fields.
- **remove_unknown** – if True, converts field specifications for fields that aren't in the schema into regular text.

class whoosh.qparser.**PhrasePlugin**(*expr='"(?P<text>.*?)"(~(?P<slop>[1-9][0-9]*))?'*)
Adds the ability to specify phrase queries inside double quotes.

class whoosh.qparser.**RangePlugin**(*expr=None*, *excl_start='{'*, *excl_end='}'*)
Adds the ability to specify term ranges.

class whoosh.qparser.**OperatorsPlugin**(*ops=None*, *clean=False*, *And='(?<=\s)AND(?=\s)'*, *Or='(?<=\s)OR(?=\s)'*, *AndNot='(?<=\s)ANDNOT(?=\s)'*, *AndMaybe='(?<=\s)ANDMAYBE(?=\s)'*, *Not='(^|(?<=(\s|[()])))NOT(?=\s)'*, *Require='(^|(?<=\s))REQUIRE(?=\s)'*)
By default, adds the AND, OR, ANDNOT, ANDMAYBE, and NOT operators to the parser syntax. This plugin scans the token stream for subclasses of *Operator* and calls their Operator.make_group() methods to allow them to manipulate the stream.

There are two levels of configuration available.

The first level is to change the regular expressions of the default operators, using the And, Or, AndNot, AndMaybe, and/or Not keyword arguments. The keyword value can be a pattern string or a compiled expression, or None to remove the operator:

```
qp = qparser.QueryParser("content", schema)
cp = qparser.OperatorsPlugin(And="&", Or="\|", AndNot="&!",
                             AndMaybe="&~", Not=None)
qp.replace_plugin(cp)
```

You can also specify a list of (OpTagger, priority) pairs as the first argument to the initializer to use custom operators. See *Creating custom operators* for more information on this.

class whoosh.qparser.**PlusMinusPlugin**(*plusexpr='\+'*, *minusexpr='-'*)
Adds the ability to use + and - in a flat OR query to specify required and prohibited terms.

This is the basis for the parser configuration returned by SimpleParser().

class whoosh.qparser.**GtLtPlugin**(*expr=None*)
Allows the user to use greater than/less than symbols to create range queries:

```
a:>100 b:<=z c:>=-1.4 d:<mz
```

This is the equivalent of:

```
a:{100 to] b:[to z] c:[-1.4 to] d:[to mz}
```

The plugin recognizes >, <, >=, <=, =>, and =< after a field specifier. The field specifier is required. You cannot do the following:

```
>100
```

This plugin requires the FieldsPlugin and RangePlugin to work.

class whoosh.qparser.**MultifieldPlugin**(*fieldnames*, *fieldboosts=None*, *group=<class 'whoosh.qparser.syntax.OrGroup'>*)

Converts any unfielded terms into OR clauses that search for the term in a specified list of fields.

```
>>> qp = qparser.QueryParser(None, myschema)
>>> qp.add_plugin(qparser.MultifieldPlugin(["a", "b"])
>>> qp.parse("alfa c:bravo")
And([Or([Term("a", "alfa"), Term("b", "alfa")]), Term("c", "bravo")])
```

This plugin is the basis for the `MultifieldParser`.

Parameters

- **fieldnames** – a list of fields to search.
- **fieldboosts** – an optional dictionary mapping field names to a boost to use for that field.
- **group** – the group to use to relate the fielded terms to each other.

class whoosh.qparser.**FieldAliasPlugin**(*fieldmap*)

Adds the ability to use "aliases" of fields in the query string.

This plugin is useful for allowing users of languages that can't be represented in ASCII to use field names in their own language, and translate them into the "real" field names, which must be valid Python identifiers.

```
>>> # Allow users to use 'body' or 'text' to refer to the 'content' field
>>> parser.add_plugin(FieldAliasPlugin({"content": ["body", "text"]}))
>>> parser.parse("text:hello")
Term("content", "hello")
```

class whoosh.qparser.**CopyFieldPlugin**(*map*, *group=<class 'whoosh.qparser.syntax.OrGroup'>*, *mirror=False*)

Looks for basic syntax nodes (terms, prefixes, wildcards, phrases, etc.) occurring in a certain field and replaces it with a group (by default OR) containing the original token and the token copied to a new field.

For example, the query:

```
hello name:matt
```

could be automatically converted by `CopyFieldPlugin({"name", "author"})` to:

```
hello (name:matt OR author:matt)
```

This is useful where one field was indexed with a differently-analyzed copy of another, and you want the query to search both fields.

You can specify a different group type with the `group` keyword. You can also specify `group=None`, in which case the copied node is inserted "inline" next to the original, instead of in a new group:

```
hello name:matt author:matt
```

Parameters

- **map** – a dictionary mapping names of fields to copy to the names of the destination fields.
- **group** – the type of group to create in place of the original token. You can specify `group=None` to put the copied node "inline" next to the original node instead of in a new group.
- **two_way** – if True, the plugin copies both ways, so if the user specifies a query in the 'toname' field, it will be copied to the 'fromname' field.

Syntax node objects

Base nodes

class whoosh.qparser.**SyntaxNode**

Base class for nodes that make up the abstract syntax tree (AST) of a parsed user query string. The AST is an intermediate step, generated from the query string, then converted into a *whoosh.query.Query* tree by calling the `query()` method on the nodes.

Instances have the following required attributes:

has_fieldname True if this node has a `fieldname` attribute.

has_text True if this node has a `text` attribute

has_boost True if this node has a `boost` attribute.

startchar The character position in the original text at which this node started.

endchar The character position in the original text at which this node ended.

is_ws()
Returns True if this node is ignorable whitespace.

query(*parser*)
Returns a *whoosh.query.Query* instance corresponding to this syntax tree node.

r()
Returns a basic representation of this node. The base class's `__repr__` method calls this, then does the extra busy work of adding fieldname and boost where appropriate.

set_boost(*boost*)
Sets the boost associated with this node.

For nodes that don't have a boost, this is a no-op.

set_fieldname(*name*, *override=False*)
Sets the fieldname associated with this node. If `override` is False (the default), the fieldname will only be replaced if this node does not already have a fieldname set.

For nodes that don't have a fieldname, this is a no-op.

set_range(*startchar*, *endchar*)
Sets the character range associated with this node.

Nodes

class whoosh.qparser.**FieldnameNode**(*fieldname*, *original*)
Abstract syntax tree node for field name assignments.

class whoosh.qparser.**TextNode**(*text*)
Intermediate base class for basic nodes that search for text, such as term queries, wildcards, prefixes, etc.

Instances have the following attributes:

qclass If a subclass does not override `query()`, the base class will use this class to construct the query.

tokenize If True and the subclass does not override `query()`, the node's text will be tokenized before constructing the query

removestops If True and the subclass does not override `query()`, and the field's analyzer has a stop word filter, stop words will be removed from the text before constructing the query.

class whoosh.qparser.**WordNode**(*text*)
Syntax node for term queries.

class whoosh.qparser.**RangeNode**(*start*, *end*, *startexcl*, *endexcl*)
Syntax node for range queries.

class whoosh.qparser.**MarkerNode**
Base class for nodes that only exist to mark places in the tree.

Group nodes

class whoosh.qparser.**GroupNode**(*nodes=None*, *boost=1.0*, ***kwargs*)
Base class for abstract syntax tree node types that group together sub-nodes.

Instances have the following attributes:

merging True if side-by-side instances of this group can be merged into a single group.

qclass If a subclass doesn't override `query()`, the base class will simply wrap this class around the queries returned by the subnodes.

This class implements a number of list methods for operating on the subnodes.

class whoosh.qparser.**BinaryGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)
Intermediate base class for group nodes that have two subnodes and whose `qclass` initializer takes two arguments instead of a list.

class whoosh.qparser.**ErrorNode**(*message*, *node=None*)

class whoosh.qparser.**AndGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**OrGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**AndNotGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**AndMaybeGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**DisMaxGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**RequireGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

class whoosh.qparser.**NotGroup**(*nodes=None*, *boost=1.0*, ***kwargs*)

Operators

class whoosh.qparser.**Operator**(*text*, *grouptype*, *leftassoc=True*)
Base class for PrefixOperator, PostfixOperator, and InfixOperator.

Operators work by moving the nodes they apply to (e.g. for prefix operator, the previous node, for infix operator, the nodes on either side, etc.) into a group node. The group provides the code for what to do with the nodes.

Parameters

- **text** – the text of the operator in the query string.
- **grouptype** – the type of group to create in place of the operator and the node(s) it operates on.
- **leftassoc** – for infix opeators, whether the operator is left associative. use `leftassoc=False` for right-associative infix operators.

class whoosh.qparser.**PrefixOperator**(*text*, *grouptype*, *leftassoc=True*)

Parameters

- **text** – the text of the operator in the query string.
- **grouptype** – the type of group to create in place of the operator and the node(s) it operates on.
- **leftassoc** – for infix opeators, whether the operator is left associative. use `leftassoc=False` for right-associative infix operators.

class whoosh.qparser.**PostfixOperator**(*text*, *grouptype*, *leftassoc=True*)

Parameters

- **text** – the text of the operator in the query string.
- **grouptype** – the type of group to create in place of the operator and the node(s) it operates on.
- **leftassoc** – for infix opeators, whether the operator is left associative. use `leftassoc=False` for right-associative infix operators.

class whoosh.qparser.**InfixOperator**(*text*, *grouptype*, *leftassoc=True*)

Parameters

- **text** – the text of the operator in the query string.
- **grouptype** – the type of group to create in place of the operator and the node(s) it operates on.
- **leftassoc** – for infix opeators, whether the operator is left associative. use `leftassoc=False` for right-associative infix operators.

query module

See also *whoosh.qparser* which contains code for parsing user queries into query objects.

Base classes

The following abstract base classes are subclassed to create the "real" query operations.

class whoosh.query.**Query**

Abstract base class for all queries.

Note that this base class implements __or__, __and__, and __sub__ to allow slightly more convenient composition of query objects:

```
>>> Term("content", u"a") | Term("content", u"b")
Or([Term("content", u"a"), Term("content", u"b")])

>>> Term("content", u"a") & Term("content", u"b")
And([Term("content", u"a"), Term("content", u"b")])

>>> Term("content", u"a") - Term("content", u"b")
And([Term("content", u"a"), Not(Term("content", u"b"))])
```

accept(*fn*)

Applies the given function to this query's subqueries (if any) and then to this query itself:

```
def boost_phrases(q):
    if isintance(q, Phrase):
        q.boost *= 2.0
    return q

myquery = myquery.accept(boost_phrases)
```

This method automatically creates copies of the nodes in the original tree before passing them to your function, so your function can change attributes on nodes without altering the original tree.

This method is less flexible than using *Query.apply()* (in fact it's implemented using that method) but is often more straightforward.

all_terms(*phrases=True*)

Returns a set of all terms in this query tree.

This method exists for backwards-compatibility. Use iter_all_terms() instead.

Parameters **phrases** – Whether to add words found in Phrase queries.

Return type set

all_tokens(*boost=1.0*)

Returns an iterator of analysis.Token objects corresponding to all terms in this query tree. The Token objects will have the fieldname, text, and boost attributes set. If the query was built by the query parser, they Token objects will also have startchar and endchar attributes indexing into the original user query.

apply(*fn*)

If this query has children, calls the given function on each child and returns a new copy of this node with the new children returned by the function. If this is a leaf node, simply returns this object.

This is useful for writing functions that transform a query tree. For example, this function changes all Term objects in a query tree into Variations objects:

```
def term2var(q):
    if isinstance(q, Term):
        return Variations(q.fieldname, q.text)
    else:
        return q.apply(term2var)

q = And([Term("f", "alfa"),
         Or([Term("f", "bravo"),
```

```
                Not(Term("f", "charlie"))])])
q = term2var(q)
```

Note that this method does not automatically create copies of nodes. To avoid modifying the original tree, your function should call the *Query.copy()* method on nodes before changing their attributes.

children()
Returns an iterator of the subqueries of this object.

copy()
Deprecated, just use `copy.deepcopy`.

deletion_docs(*searcher*)
Returns an iterator of docnums matching this query for the purpose of deletion. The *delete_by_query()* method will use this method when deciding what documents to delete, allowing special queries (e.g. nested queries) to override what documents are deleted. The default implementation just forwards to *Query.docs()*.

docs(*searcher*)
Returns an iterator of docnums matching this query.

```
>>> with my_index.searcher() as searcher:
...     list(my_query.docs(searcher))
[10, 34, 78, 103]
```

Parameters **searcher** – A *whoosh.searching.Searcher* object.

estimate_min_size(*ixreader*)
Returns an estimate of the minimum number of documents this query could potentially match.

estimate_size(*ixreader*)
Returns an estimate of how many documents this query could potentially match (for example, the estimated size of a simple term query is the document frequency of the term). It is permissible to overestimate, but not to underestimate.

existing_terms(*ixreader*, *phrases=True*, *expand=False*, *fieldname=None*)
Returns a set of all byteterms in this query tree that exist in the given ixreader.

Parameters

- **ixreader** – A *whoosh.reading.IndexReader* object.
- **phrases** – Whether to add words found in Phrase queries.
- **expand** – If True, queries that match multiple terms will return all matching expansions.

Return type set

field()
Returns the field this query matches in, or None if this query does not match in a single field.

has_terms()
Returns True if this specific object represents a search for a specific term (as opposed to a pattern, as in Wildcard and Prefix) or terms (i.e., whether the `replace()` method does something meaningful on this instance).

is_leaf()
Returns True if this is a leaf node in the query tree, or False if this query has sub-queries.

is_range()
Returns True if this object searches for values within a range.

iter_all_terms(*phrases=True*)
Returns an iterator of (fieldname, text) pairs for all terms in this query tree.

```
>>> qp = qparser.QueryParser("text", myindex.schema)
>>> q = myparser.parse("alfa bravo title:charlie")
>>> # List the terms in a query
>>> list(q.iter_all_terms())
[("text", "alfa"), ("text", "bravo"), ("title", "charlie")]
>>> # Get a set of all terms in the query that don't exist in the index
>>> r = myindex.reader()
>>> missing = set(t for t in q.iter_all_terms() if t not in r)
set([("text", "alfa"), ("title", "charlie")])
>>> # All terms in the query that occur in fewer than 5 documents in
>>> # the index
>>> [t for t in q.iter_all_terms() if r.doc_frequency(t[0], t[1]) < 5]
[("title", "charlie")]
```

Parameters **phrases** – Whether to add words found in Phrase queries.

leaves()
Returns an iterator of all the leaf queries in this query tree as a flat series.

matcher(*searcher*, *context=None*)
Returns a *Matcher* object you can use to retrieve documents and scores matching this query.

Return type *whoosh.matching.Matcher*

normalize()
Returns a recursively "normalized" form of this query. The normalized form removes redundancy and empty queries. This is called automatically on query trees created by the query parser, but you may want to call it yourself if you're writing your own parser or building your own queries.

```
>>> q = And([And([Term("f", u"a"),
...               Term("f", u"b")]),
...               Term("f", u"c"), Or([])])
>>> q.normalize()
And([Term("f", u"a"), Term("f", u"b"), Term("f", u"c")])
```

Note that this returns a *new, normalized* query. It *does not* modify the original query "in place".

replace(*fieldname*, *oldtext*, *newtext*)
Returns a copy of this query with oldtext replaced by newtext (if oldtext was anywhere in this query).

Note that this returns a *new* query with the given text replaced. It *does not* modify the original query "in place".

requires()
Returns a set of queries that are *known* to be required to match for the entire query to match. Note that other queries might also turn out to be required but not be determinable by examining the static query.

```
>>> a = Term("f", u"a")
>>> b = Term("f", u"b")
>>> And([a, b]).requires()
set([Term("f", u"a"), Term("f", u"b")])
>>> Or([a, b]).requires()
set([])
>>> AndMaybe(a, b).requires()
set([Term("f", u"a")])
```

```
>>> a.requires()
set([Term("f", u"a")])
```

simplify(*ixreader*)
Returns a recursively simplified form of this query, where "second-order" queries (such as Prefix and Variations) are re-written into lower-level queries (such as Term and Or).

terms(*phrases=False*)
Yields zero or more (fieldname, text) pairs queried by this object. You can check whether a query object targets specific terms before you call this method using *Query.has_terms()*.

To get all terms in a query tree, use *Query.iter_all_terms()*.

tokens(*boost=1.0*, *exreader=None*)
Yields zero or more `analysis.Token` objects corresponding to the terms searched for by this query object. You can check whether a query object targets specific terms before you call this method using *Query.has_terms()*.

The Token objects will have the `fieldname`, `text`, and `boost` attributes set. If the query was built by the query parser, they Token objects will also have `startchar` and `endchar` attributes indexing into the original user query.

To get all tokens for a query tree, use *Query.all_tokens()*.

Parameters **exreader** – a reader to use to expand multiterm queries such as prefixes and wildcards. The default is None meaning do not expand.

with_boost(*boost*)
Returns a COPY of this query with the boost set to the given value.

If a query type does not accept a boost itself, it will try to pass the boost on to its children, if any.

class whoosh.query.**CompoundQuery**(*subqueries*, *boost=1.0*)
Abstract base class for queries that combine or manipulate the results of multiple sub-queries .

class whoosh.query.**MultiTerm**
Abstract base class for queries that operate on multiple terms in the same field.

class whoosh.query.**ExpandingTerm**
Intermediate base class for queries such as FuzzyTerm and Variations that expand into multiple queries, but come from a single term.

class whoosh.query.**WrappingQuery**(*child*)

Query classes

class whoosh.query.**Term**(*fieldname*, *text*, *boost=1.0*, *minquality=None*)
Matches documents containing the given term (fieldname+text pair).

```
>>> Term("content", u"render")
```

class whoosh.query.**Variations**(*fieldname*, *text*, *boost=1.0*)
Query that automatically searches for morphological variations of the given word in the same field.

class whoosh.query.**FuzzyTerm**(*fieldname*, *text*, *boost=1.0*, *maxdist=1*, *prefixlength=1*, *constantscore=True*)
Matches documents containing words similar to the given term.

Parameters

- **fieldname** – The name of the field to search.

- **text** – The text to search for.
- **boost** – A boost factor to apply to scores of documents matching this query.
- **maxdist** – The maximum edit distance from the given text.
- **prefixlength** – The matched terms must share this many initial characters with 'text'. For example, if text is "light" and prefixlength is 2, then only terms starting with "li" are checked for similarity.

class whoosh.query.**Phrase**(*fieldname*, *words*, *slop=1*, *boost=1.0*, *char_ranges=None*)
Matches documents containing a given phrase.

Parameters

- **fieldname** – the field to search.
- **words** – a list of words (unicode strings) in the phrase.
- **slop** – the number of words allowed between each "word" in the phrase; the default of 1 means the phrase must match exactly.
- **boost** – a boost factor that to apply to the raw score of documents matched by this query.
- **char_ranges** – if a Phrase object is created by the query parser, it will set this attribute to a list of (startchar, endchar) pairs corresponding to the words in the phrase

class whoosh.query.**And**(*subqueries*, *boost=1.0*)
Matches documents that match ALL of the subqueries.

```
>>> And([Term("content", u"render"),
...      Term("content", u"shade"),
...      Not(Term("content", u"texture"))])
>>> # You can also do this
>>> Term("content", u"render") & Term("content", u"shade")
```

class whoosh.query.**Or**(*subqueries*, *boost=1.0*, *minmatch=0*, *scale=None*)
Matches documents that match ANY of the subqueries.

```
>>> Or([Term("content", u"render"),
...     And([Term("content", u"shade"), Term("content", u"texture")]),
...     Not(Term("content", u"network"))])
>>> # You can also do this
>>> Term("content", u"render") | Term("content", u"shade")
```

Parameters

- **subqueries** – a list of *Query* objects to search for.
- **boost** – a boost factor to apply to the scores of all matching documents.
- **minmatch** – not yet implemented.
- **scale** – a scaling factor for a "coordination bonus". If this value is not None, it should be a floating point number greater than 0 and less than 1. The scores of the matching documents are boosted/penalized based on the number of query terms that matched in the document. This number scales the effect of the bonuses.

class whoosh.query.**DisjunctionMax**(*subqueries*, *boost=1.0*, *tiebreak=0.0*)
Matches all documents that match any of the subqueries, but scores each document using the maximum score from the subqueries.

class whoosh.query.**Not**(*query*, *boost=1.0*)
Excludes any documents that match the subquery.

```
>>> # Match documents that contain 'render' but not 'texture'
>>> And([Term("content", u"render"),
...      Not(Term("content", u"texture"))])
>>> # You can also do this
>>> Term("content", u"render") - Term("content", u"texture")
```

Parameters

- **query** – A *Query* object. The results of this query are *excluded* from the parent query.
- **boost** – Boost is meaningless for excluded documents but this keyword argument is accepted for the sake of a consistent interface.

class whoosh.query.**Prefix**(*fieldname*, *text*, *boost=1.0*, *constantscore=True*)
Matches documents that contain any terms that start with the given text.

```
>>> # Match documents containing words starting with 'comp'
>>> Prefix("content", u"comp")
```

class whoosh.query.**Wildcard**(*fieldname*, *text*, *boost=1.0*, *constantscore=True*)
Matches documents that contain any terms that match a "glob" pattern. See the Python `fnmatch` module for information about globs.

```
>>> Wildcard("content", u"in*f?x")
```

class whoosh.query.**Regex**(*fieldname*, *text*, *boost=1.0*, *constantscore=True*)
Matches documents that contain any terms that match a regular expression. See the Python `re` module for information about regular expressions.

class whoosh.query.**TermRange**(*fieldname*, *start*, *end*, *startexcl=False*, *endexcl=False*, *boost=1.0*, *constantscore=True*)
Matches documents containing any terms in a given range.

```
>>> # Match documents where the indexed "id" field is greater than or equal
>>> # to 'apple' and less than or equal to 'pear'.
>>> TermRange("id", u"apple", u"pear")
```

Parameters

- **fieldname** – The name of the field to search.
- **start** – Match terms equal to or greater than this.
- **end** – Match terms equal to or less than this.
- **startexcl** – If True, the range start is exclusive. If False, the range start is inclusive.
- **endexcl** – If True, the range end is exclusive. If False, the range end is inclusive.
- **boost** – Boost factor that should be applied to the raw score of results matched by this query.

class whoosh.query.**NumericRange**(*fieldname*, *start*, *end*, *startexcl=False*, *endexcl=False*, *boost=1.0*, *constantscore=True*)
A range query for NUMERIC fields. Takes advantage of tiered indexing to speed up large ranges by matching at a high resolution at the edges of the range and a low resolution in the middle.

```
>>> # Match numbers from 10 to 5925 in the "number" field.
>>> nr = NumericRange("number", 10, 5925)
```

Parameters

- **fieldname** – The name of the field to search.
- **start** – Match terms equal to or greater than this number. This should be a number type, not a string.
- **end** – Match terms equal to or less than this number. This should be a number type, not a string.
- **startexcl** – If True, the range start is exclusive. If False, the range start is inclusive.
- **endexcl** – If True, the range end is exclusive. If False, the range end is inclusive.
- **boost** – Boost factor that should be applied to the raw score of results matched by this query.
- **constantscore** – If True, the compiled query returns a constant score (the value of the `boost` keyword argument) instead of actually scoring the matched terms. This gives a nice speed boost and won't affect the results in most cases since numeric ranges will almost always be used as a filter.

class `whoosh.query.`**DateRange**(*fieldname*, *start*, *end*, *startexcl=False*, *endexcl=False*, *boost=1.0*, *constantscore=True*)

This is a very thin subclass of *NumericRange* that only overrides the initializer and `__repr__()` methods to work with datetime objects instead of numbers. Internally this object converts the datetime objects it's created with to numbers and otherwise acts like a `NumericRange` query.

```
>>> DateRange("date", datetime(2010, 11, 3, 3, 0),
...           datetime(2010, 11, 3, 17, 59))
```

class `whoosh.query.`**Every**(*fieldname=None*, *boost=1.0*)

A query that matches every document containing any term in a given field. If you don't specify a field, the query matches every document.

```
>>> # Match any documents with something in the "path" field
>>> q = Every("path")
>>> # Matcher every document
>>> q = Every()
```

The unfielded form (matching every document) is efficient.

The fielded is more efficient than a prefix query with an empty prefix or a '*' wildcard, but it can still be very slow on large indexes. It requires the searcher to read the full posting list of every term in the given field.

Instead of using this query it is much more efficient when you create the index to include a single term that appears in all documents that have the field you want to match.

For example, instead of this:

```
# Match all documents that have something in the "path" field
q = Every("path")
```

Do this when indexing:

```
# Add an extra field that indicates whether a document has a path
schema = fields.Schema(path=fields.ID, has_path=fields.ID)

# When indexing, set the "has_path" field based on whether the document
# has anything in the "path" field
writer.add_document(text=text_value1)
writer.add_document(text=text_value2, path=path_value2, has_path="t")
```

Then to find all documents with a path:

```
q = Term("has_path", "t")
```

Parameters **fieldname** – the name of the field to match, or `None` or `*` to match all documents.

whoosh.query.**NullQuery**

Binary queries

class whoosh.query.**Require**(*a*, *b*)
Binary query returns results from the first query that also appear in the second query, but only uses the scores from the first query. This lets you filter results without affecting scores.

class whoosh.query.**AndMaybe**(*a*, *b*)
Binary query takes results from the first query. If and only if the same document also appears in the results from the second query, the score from the second query will be added to the score from the first query.

class whoosh.query.**AndNot**(*a*, *b*)
Binary boolean query of the form 'a ANDNOT b', where documents that match b are removed from the matches for a.

class whoosh.query.**Otherwise**(*a*, *b*)
A binary query that only matches the second clause if the first clause doesn't match any documents.

Span queries

class whoosh.query.**Span**(*start*, *end=None*, *startchar=None*, *endchar=None*, *boost=1.0*)

classmethod **merge**(*spans*)
Merges overlapping and touches spans in the given list of spans.

Note that this modifies the original list.

```
>>> spans = [Span(1,2), Span(3)]
>>> Span.merge(spans)
>>> spans
[<1-3>]
```

class whoosh.query.**SpanQuery**
Abstract base class for span-based queries. Each span query type wraps a "regular" query that implements the basic document-matching functionality (for example, SpanNear wraps an And query, because SpanNear requires that the two sub-queries occur in the same documents. The wrapped query is stored in the `q` attribute.

Subclasses usually only need to implement the initializer to set the wrapped query, and `matcher()` to return a span-aware matcher object.

class whoosh.query.**SpanFirst**(*q*, *limit=0*)

Matches spans that end within the first N positions. This lets you for example only match terms near the beginning of the document.

Parameters

- **q** – the query to match.
- **limit** – the query must match within this position at the start of a document. The default is 0, which means the query must match at the first position.

class whoosh.query.**SpanNear**(*a*, *b*, *slop=1*, *ordered=True*, *mindist=1*)

Note: for new code, use *SpanNear2* instead of this class. SpanNear2 takes a list of sub-queries instead of requiring you to create a binary tree of query objects.

Matches queries that occur near each other. By default, only matches queries that occur right next to each other (slop=1) and in order (ordered=True).

For example, to find documents where "whoosh" occurs next to "library" in the "text" field:

```
from whoosh import query, spans
t1 = query.Term("text", "whoosh")
t2 = query.Term("text", "library")
q = spans.SpanNear(t1, t2)
```

To find documents where "whoosh" occurs at most 5 positions before "library":

```
q = spans.SpanNear(t1, t2, slop=5)
```

To find documents where "whoosh" occurs at most 5 positions before or after "library":

```
q = spans.SpanNear(t1, t2, slop=5, ordered=False)
```

You can use the `phrase()` class method to create a tree of SpanNear queries to match a list of terms:

```
q = spans.SpanNear.phrase("text", ["whoosh", "search", "library"],
                          slop=2)
```

Parameters

- **a** – the first query to match.
- **b** – the second query that must occur within "slop" positions of the first query.
- **slop** – the number of positions within which the queries must occur. Default is 1, meaning the queries must occur right next to each other.
- **ordered** – whether a must occur before b. Default is True.

Pram mindist the minimum distance allowed between the queries.

class whoosh.query.**SpanNear2**(*qs*, *slop=1*, *ordered=True*, *mindist=1*)

Matches queries that occur near each other. By default, only matches queries that occur right next to each other (slop=1) and in order (ordered=True).

New code should use this query type instead of *SpanNear*.

(Unlike *SpanNear*, this query takes a list of subqueries instead of requiring you to build a binary tree of query objects. This query should also be slightly faster due to less overhead.)

For example, to find documents where "whoosh" occurs next to "library" in the "text" field:

```
from whoosh import query, spans
t1 = query.Term("text", "whoosh")
t2 = query.Term("text", "library")
q = spans.SpanNear2([t1, t2])
```

To find documents where "whoosh" occurs at most 5 positions before "library":

```
q = spans.SpanNear2([t1, t2], slop=5)
```

To find documents where "whoosh" occurs at most 5 positions before or after "library":

```
q = spans.SpanNear2(t1, t2, slop=5, ordered=False)
```

Parameters

- **qs** – a sequence of sub-queries to match.
- **slop** – the number of positions within which the queries must occur. Default is 1, meaning the queries must occur right next to each other.
- **ordered** – whether a must occur before b. Default is True.

Pram mindist the minimum distance allowed between the queries.

class whoosh.query.**SpanNot**(*a*, *b*)

Matches spans from the first query only if they don't overlap with spans from the second query. If there are no non-overlapping spans, the document does not match.

For example, to match documents that contain "bear" at most 2 places after "apple" in the "text" field but don't have "cute" between them:

```
from whoosh import query, spans
t1 = query.Term("text", "apple")
t2 = query.Term("text", "bear")
near = spans.SpanNear(t1, t2, slop=2)
q = spans.SpanNot(near, query.Term("text", "cute"))
```

Parameters

- **a** – the query to match.
- **b** – do not match any spans that overlap with spans from this query.

class whoosh.query.**SpanOr**(*subqs*)

Matches documents that match any of a list of sub-queries. Unlike query.Or, this class merges together matching spans from the different sub-queries when they overlap.

Parameters **subqs** – a list of queries to match.

class whoosh.query.**SpanContains**(*a*, *b*)

Matches documents where the spans of the first query contain any spans of the second query.

For example, to match documents where "apple" occurs at most 10 places before "bear" in the "text" field and "cute" is between them:

```
from whoosh import query, spans
t1 = query.Term("text", "apple")
t2 = query.Term("text", "bear")
```

```
near = spans.SpanNear(t1, t2, slop=10)
q = spans.SpanContains(near, query.Term("text", "cute"))
```

Parameters

- **a** – the query to match.
- **b** – the query whose spans must occur within the matching spans of the first query.

class whoosh.query.**SpanBefore**(*a*, *b*)

Matches documents where the spans of the first query occur before any spans of the second query.

For example, to match documents where "apple" occurs anywhere before "bear":

```
from whoosh import query, spans
t1 = query.Term("text", "apple")
t2 = query.Term("text", "bear")
q = spans.SpanBefore(t1, t2)
```

Parameters

- **a** – the query that must occur before the second.
- **b** – the query that must occur after the first.

class whoosh.query.**SpanCondition**(*a*, *b*)

Matches documents that satisfy both subqueries, but only uses the spans from the first subquery.

This is useful when you want to place conditions on matches but not have those conditions affect the spans returned.

For example, to get spans for the term `alfa` in documents that also must contain the term `bravo`:

```
SpanCondition(Term("text", u"alfa"), Term("text", u"bravo"))
```

Special queries

class whoosh.query.**NestedParent**(*parents*, *subq*, *per_parent_limit=None*, *score_fn=<built-in function sum>*)

A query that allows you to search for "nested" documents, where you can index (possibly multiple levels of) "parent" and "child" documents using the *group()* and/or *start_group()* methods of a *whoosh.writing.IndexWriter* to indicate that hierarchically related documents should be kept together:

```
schema = fields.Schema(type=fields.ID, text=fields.TEXT(stored=True))

with ix.writer() as w:
    # Say we're indexing chapters (type=chap) and each chapter has a
    # number of paragraphs (type=p)
    with w.group():
        w.add_document(type="chap", text="Chapter 1")
        w.add_document(type="p", text="Able baker")
        w.add_document(type="p", text="Bright morning")
    with w.group():
        w.add_document(type="chap", text="Chapter 2")
        w.add_document(type="p", text="Car trip")
        w.add_document(type="p", text="Dog eared")
        w.add_document(type="p", text="Every day")
```

```
    with w.group():
        w.add_document(type="chap", text="Chapter 3")
        w.add_document(type="p", text="Fine day")
```

The `NestedParent` query wraps two sub-queries: the "parent query" matches a class of "parent documents". The "sub query" matches nested documents you want to find. For each "sub document" the "sub query" finds, this query acts as if it found the corresponding "parent document".

```
>>> with ix.searcher() as s:
...   r = s.search(query.Term("text", "day"))
...   for hit in r:
...     print(hit["text"])
...
Chapter 2
Chapter 3
```

Parameters

- **parents** – a query, DocIdSet object, or Results object representing the documents you want to use as the "parent" documents. Where the sub-query matches, the corresponding document in these results will be returned as the match.
- **subq** – a query matching the information you want to find.
- **per_parent_limit** – a maximum number of "sub documents" to search per parent. The default is None, meaning no limit.
- **score_fn** – a function to use to combine the scores of matching sub-documents to calculate the score returned for the parent document. The default is `sum`, that is, add up the scores of the sub-documents.

class whoosh.query.**NestedChildren**(*parents*, *subq*, *boost=1.0*)

This is the reverse of a *NestedParent* query: instead of taking a query that matches children but returns the parent, this query matches parents but returns the children.

This is useful, for example, to search for an album title and return the songs in the album:

```
schema = fields.Schema(type=fields.ID(stored=True),
                       album_name=fields.TEXT(stored=True),
                       track_num=fields.NUMERIC(stored=True),
                       track_name=fields.TEXT(stored=True),
                       lyrics=fields.TEXT)
ix = RamStorage().create_index(schema)

# Indexing
with ix.writer() as w:
    # For each album, index a "group" of a parent "album" document and
    # multiple child "track" documents.
    with w.group():
        w.add_document(type="album",
                       artist="The Cure", album_name="Disintegration")
        w.add_document(type="track", track_num=1,
                       track_name="Plainsong")
        w.add_document(type="track", track_num=2,
                       track_name="Pictures of You")
        # ...
    # ...
```

```
# Find songs where the song name has "heaven" in the title and the
# album the song is on has "hell" in the title
qp = QueryParser("lyrics", ix.schema)
with ix.searcher() as s:
    # A query that matches all parents
    all_albums = qp.parse("type:album")

    # A query that matches the parents we want
    albums_with_hell = qp.parse("album_name:hell")

    # A query that matches the desired albums but returns the tracks
    songs_on_hell_albums = NestedChildren(all_albums, albums_with_hell)

    # A query that matches tracks with heaven in the title
    songs_with_heaven = qp.parse("track_name:heaven")

    # A query that finds tracks with heaven in the title on albums
    # with hell in the title
    q = query.And([songs_on_hell_albums, songs_with_heaven])
```

class whoosh.query.**ConstantScoreQuery**(*child*, *score=1.0*)
Wraps a query and uses a matcher that always gives a constant score to all matching documents. This is a useful optimization when you don't care about scores from a certain branch of the query tree because it is simply acting as a filter. See also the *AndMaybe* query.

Exceptions

exception whoosh.query.**QueryError**
Error encountered while running a query.

reading module

This module contains classes that allow reading from an index.

Classes

class whoosh.reading.**IndexReader**
Do not instantiate this object directly. Instead use Index.reader().

all_doc_ids()
Returns an iterator of all (undeleted) document IDs in the reader.

all_stored_fields()
Yields the stored fields for all non-deleted documents.

all_terms()
Yields (fieldname, text) tuples for every term in the index.

close()
Closes the open files associated with this reader.

codec()
Returns the *whoosh.codec.base.Codec* object used to read this reader's segment. If this reader is not atomic (`reader.is_atomic() == True`), returns None.

column_reader(*fieldname*, *column=None*, *reverse=False*, *translate=False*)

Parameters

- **fieldname** – the name of the field for which to get a reader.
- **column** – if passed, use this Column object instead of the one associated with the field in the Schema.
- **reverse** – if passed, reverses the order of keys returned by the reader's `sort_key()` method. If the column type is not reversible, this will raise a `NotImplementedError`.
- **translate** – if True, wrap the reader to call the field's `from_bytes()` method on the returned values.

Returns a *whoosh.columns.ColumnReader* object.

corrector(*fieldname*)
Returns a *whoosh.spelling.Corrector* object that suggests corrections based on the terms in the given field.

doc_count()
Returns the total number of UNDELETED documents in this reader.

doc_count_all()
Returns the total number of documents, DELETED OR UNDELETED, in this reader.

doc_field_length(*docnum*, *fieldname*, *default=0*)
Returns the number of terms in the given field in the given document. This is used by some scoring algorithms.

doc_frequency(*fieldname*, *text*)
Returns how many documents the given term appears in.

expand_prefix(*fieldname*, *prefix*)
Yields terms in the given field that start with the given prefix.

field_length(*fieldname*)
Returns the total number of terms in the given field. This is used by some scoring algorithms.

field_terms(*fieldname*)
Yields all term values (converted from on-disk bytes) in the given field.

first_id(*fieldname*, *text*)
Returns the first ID in the posting list for the given term. This may be optimized in certain backends.

frequency(*fieldname*, *text*)
Returns the total number of instances of the given term in the collection.

generation()
Returns the generation of the index being read, or -1 if the backend is not versioned.

has_deletions()
Returns True if the underlying index/segment has deleted documents.

has_vector(*docnum*, *fieldname*)
Returns True if the given document has a term vector for the given field.

indexed_field_names()
Returns an iterable of strings representing the names of the indexed fields. This may include additional names not explicitly listed in the Schema if you use "glob" fields.

is_deleted(*docnum*)
Returns True if the given document number is marked deleted.

iter_docs()
Yields a series of `(docnum, stored_fields_dict)` tuples for the undeleted documents in the reader.

iter_field(*fieldname*, *prefix=''*)
Yields (text, terminfo) tuples for all terms in the given field.

iter_from(*fieldname*, *text*)
Yields ((fieldname, text), terminfo) tuples for all terms in the reader, starting at the given term.

iter_postings()
Low-level method, yields all postings in the reader as `(fieldname, text, docnum, weight, valuestring)` tuples.

iter_prefix(*fieldname*, *prefix*)
Yields (text, terminfo) tuples for all terms in the given field with a certain prefix.

leaf_readers()
Returns a list of (IndexReader, docbase) pairs for the child readers of this reader if it is a composite reader. If this is not a composite reader, it returns *[(self, 0)]*.

lexicon(*fieldname*)
Yields all bytestrings in the given field.

max_field_length(*fieldname*)
Returns the minimum length of the field across all documents. This is used by some scoring algorithms.

min_field_length(*fieldname*)
Returns the minimum length of the field across all documents. This is used by some scoring algorithms.

most_distinctive_terms(*fieldname*, *number=5*, *prefix=''*)
Returns the top 'number' terms with the highest *tf*idf* scores as a list of (score, text) tuples.

most_frequent_terms(*fieldname*, *number=5*, *prefix=''*)
Returns the top 'number' most frequent terms in the given field as a list of (frequency, text) tuples.

postings(*fieldname*, *text*)
Returns a *Matcher* for the postings of the given term.

```
>>> pr = reader.postings("content", "render")
>>> pr.skip_to(10)
>>> pr.id
12
```

Parameters

- **fieldname** – the field name or field number of the term.
- **text** – the text of the term.

Return type *whoosh.matching.Matcher*

segment()
Returns the `whoosh.index.Segment` object used by this reader. If this reader is not atomic (`reader.is_atomic() == True`), returns None.

storage()
Returns the *whoosh.filedb.filestore.Storage* object used by this reader to read its files. If the reader is not atomic, (`reader.is_atomic() == True`), returns None.

stored_fields(*docnum*)
Returns the stored fields for the given document number.

Parameters **numerickeys** – use field numbers as the dictionary keys instead of field names.

term_info(*fieldname*, *text*)
Returns a *TermInfo* object allowing access to various statistics about the given term.

terms_from(*fieldname*, *prefix*)
Yields (fieldname, text) tuples for every term in the index starting at the given prefix.

terms_within(*fieldname*, *text*, *maxdist*, *prefix=0*)
Returns a generator of words in the given field within `maxdist` Damerau-Levenshtein edit distance of the given text.

Important: the terms are returned in **no particular order**. The only criterion is that they are within `maxdist` edits of `text`. You may want to run this method multiple times with increasing `maxdist` values to ensure you get the closest matches first. You may also have additional information (such as term frequency or an acoustic matching algorithm) you can use to rank terms with the same edit distance.

Parameters

- **maxdist** – the maximum edit distance.
- **prefix** – require suggestions to share a prefix of this length with the given word. This is often justifiable since most misspellings do not involve the first letter of the word. Using a prefix dramatically decreases the time it takes to generate the list of words.
- **seen** – an optional set object. Words that appear in the set will not be yielded.

vector(*docnum*, *fieldname*, *format_=None*)
Returns a *Matcher* object for the given term vector.

```
>>> docnum = searcher.document_number(path=u'/a/b/c')
>>> v = searcher.vector(docnum, "content")
>>> v.all_as("frequency")
[(u"apple", 3), (u"bear", 2), (u"cab", 2)]
```

Parameters

- **docnum** – the document number of the document for which you want the term vector.
- **fieldname** – the field name or field number of the field for which you want the term vector.

Return type *whoosh.matching.Matcher*

vector_as(*astype*, *docnum*, *fieldname*)
Returns an iterator of (termtext, value) pairs for the terms in the given term vector. This is a convenient shortcut to calling vector() and using the Matcher object when all you want are the terms and/or values.

```
>>> docnum = searcher.document_number(path=u'/a/b/c')
>>> searcher.vector_as("frequency", docnum, "content")
[(u"apple", 3), (u"bear", 2), (u"cab", 2)]
```

Parameters

- **docnum** – the document number of the document for which you want the term vector.
- **fieldname** – the field name or field number of the field for which you want the term vector.
- **astype** – a string containing the name of the format you want the term vector's data in, for example "weights".

class whoosh.reading.**MultiReader**(*readers*, *generation=None*)
Do not instantiate this object directly. Instead use Index.reader().

class whoosh.reading.**TermInfo**(*weight=0*, *df=0*, *minlength=None*, *maxlength=0*, *maxweight=0*, *minid=None*, *maxid=0*)
Represents a set of statistics about a term. This object is returned by *IndexReader.term_info()*. These statistics may be useful for optimizations and scoring algorithms.

doc_frequency()
Returns the number of documents the term appears in.

max_id()
Returns the highest document ID this term appears in.

max_length()
Returns the length of the longest field value the term appears in.

max_weight()
Returns the number of times the term appears in the document in which it appears the most.

min_id()
Returns the lowest document ID this term appears in.

min_length()
Returns the length of the shortest field value the term appears in.

weight()
Returns the total frequency of the term across all documents.

Exceptions

exception whoosh.reading.**TermNotFound**

scoring module

This module contains classes for scoring (and sorting) search results.

Base classes

class whoosh.scoring.**WeightingModel**
Abstract base class for scoring models. A WeightingModel object provides a method, `scorer`, which returns an instance of `whoosh.scoring.Scorer`.

Basically, WeightingModel objects store the configuration information for the model (for example, the values of B and K1 in the BM25F model), and then creates a scorer instance based on additional run-time information (the searcher, the fieldname, and term text) to do the actual scoring.

final(*searcher*, *docnum*, *score*)
Returns a final score for each document. You can use this method in subclasses to apply document-level adjustments to the score, for example using the value of stored field to influence the score (although that would be slow).

WeightingModel sub-classes that use `final()` should have the attribute `use_final` set to `True`.

Parameters

- **searcher** – *whoosh.searching.Searcher* for the index.
- **docnum** – the doc number of the document being scored.

- **score** – the document's accumulated term score.

Return type float

idf(*searcher*, *fieldname*, *text*)
Returns the inverse document frequency of the given term.

scorer(*searcher*, *fieldname*, *text*, *qf=1*)
Returns an instance of `whoosh.scoring.Scorer` configured for the given searcher, fieldname, and term text.

class `whoosh.scoring.`**BaseScorer**
Base class for "scorer" implementations. A scorer provides a method for scoring a document, and sometimes methods for rating the "quality" of a document and a matcher's current "block", to implement quality-based optimizations.

Scorer objects are created by WeightingModel objects. Basically, WeightingModel objects store the configuration information for the model (for example, the values of B and K1 in the BM25F model), and then creates a scorer instance.

block_quality(*matcher*)
Returns the *maximum limit* on the possible score the matcher can give **in its current "block"** (whatever concept of "block" the backend might use). This can be an estimate and not necessarily the actual maximum score possible, but it must never be less than the actual maximum score.

If this score is less than the minimum score required to make the "top N" results, then we can tell the matcher to skip ahead to another block with better "quality".

max_quality()
Returns the *maximum limit* on the possible score the matcher can give. This can be an estimate and not necessarily the actual maximum score possible, but it must never be less than the actual maximum score.

score(*matcher*)
Returns a score for the current document of the matcher.

supports_block_quality()
Returns True if this class supports quality optimizations.

class `whoosh.scoring.`**WeightScorer**(*maxweight*)
A scorer that simply returns the weight as the score. This is useful for more complex weighting models to return when they are asked for a scorer for fields that aren't scorable (don't store field lengths).

class `whoosh.scoring.`**WeightLengthScorer**
Base class for scorers where the only per-document variables are term weight and field length.

Subclasses should override the `_score(weight, length)` method to return the score for a document with the given weight and length, and call the `setup()` method at the end of the initializer to set up common attributes.

Scoring algorithm classes

class `whoosh.scoring.`**BM25F**(*B=0.75*, *K1=1.2*, ***kwargs*)
Implements the BM25F scoring algorithm.

```
>>> from whoosh import scoring
>>> # Set a custom B value for the "content" field
>>> w = scoring.BM25F(B=0.75, content_B=1.0, K1=1.5)
```

Parameters

- **B** – free parameter, see the BM25 literature. Keyword arguments of the form `fieldname_B` (for example, `body_B`) set field- specific values for B.
- **K1** – free parameter, see the BM25 literature.

class whoosh.scoring.**TF_IDF**

class whoosh.scoring.**Frequency**

Scoring utility classes

class whoosh.scoring.**FunctionWeighting**(*fn*)

Uses a supplied function to do the scoring. For simple scoring functions and experiments this may be simpler to use than writing a full weighting model class and scorer class.

The function should accept the arguments `searcher, fieldname, text, matcher`.

For example, the following function will score documents based on the earliest position of the query term in the document:

```
def pos_score_fn(searcher, fieldname, text, matcher):
    poses = matcher.value_as("positions")
    return 1.0 / (poses[0] + 1)

pos_weighting = scoring.FunctionWeighting(pos_score_fn)
with myindex.searcher(weighting=pos_weighting) as s:
    results = s.search(q)
```

Note that the searcher passed to the function may be a per-segment searcher for performance reasons. If you want to get global statistics inside the function, you should use `searcher.get_parent()` to get the top-level searcher. (However, if you are using global statistics, you should probably write a real model/scorer combo so you can cache them on the object.)

class whoosh.scoring.**MultiWeighting**(*default*, ***weightings*)

Chooses from multiple scoring algorithms based on the field.

The only non-keyword argument specifies the default `Weighting` instance to use. Keyword arguments specify Weighting instances for specific fields.

For example, to use `BM25` for most fields, but `Frequency` for the `id` field and `TF_IDF` for the `keys` field:

```
mw = MultiWeighting(BM25(), id=Frequency(), keys=TF_IDF())
```

Parameters **default** – the Weighting instance to use for fields not specified in the keyword arguments.

class whoosh.scoring.**ReverseWeighting**(*weighting*)

Wraps a weighting object and subtracts the wrapped model's scores from 0, essentially reversing the weighting model.

`searching` module

This module contains classes and functions related to searching the index.

Searching classes

class whoosh.searching.**Searcher**(*reader*, *weighting=<class 'whoosh.scoring.BM25F'>*, *closereader=True*, *fromindex=None*, *parent=None*)

Wraps an *IndexReader* object and provides methods for searching the index.

Parameters

- **reader** – An *IndexReader* object for the index to search.
- **weighting** – A whoosh.scoring.Weighting object to use to score found documents.
- **closereader** – Whether the underlying reader will be closed when the searcher is closed.
- **fromindex** – An optional reference to the index of the underlying reader. This is required for *Searcher.up_to_date()* and *Searcher.refresh()* to work.

boolean_context()

Shortcut returns a SearchContext set for unscored (boolean) searching.

collector(*limit=10*, *sortedby=None*, *reverse=False*, *groupedby=None*, *collapse=None*, *collapse_limit=1*, *collapse_order=None*, *optimize=True*, *filter=None*, *mask=None*, *terms=False*, *maptype=None*, *scored=True*)

Low-level method: returns a configured *whoosh.collectors.Collector* object based on the given arguments. You can use this object with *Searcher.search_with_collector()* to search.

See the documentation for the *Searcher.search()* method for a description of the parameters.

This method may be useful to get a basic collector object and then wrap it with another collector from whoosh.collectors or with a custom collector of your own:

```
# Equivalent of
# results = mysearcher.search(myquery, limit=10)
# but with a time limt...

# Create a TopCollector
c = mysearcher.collector(limit=10)

# Wrap it with a TimeLimitedCollector with a time limit of
# 10.5 seconds
from whoosh.collectors import TimeLimitedCollector
c = TimeLimitCollector(c, 10.5)

# Search using the custom collector
results = mysearcher.search_with_collector(myquery, c)
```

context(***kwargs*)

Generates a SearchContext for this searcher.

correct_query(*q*, *qstring*, *correctors=None*, *terms=None*, *maxdist=2*, *prefix=0*, *aliases=None*)

Returns a corrected version of the given user query using a default *whoosh.spelling.ReaderCorrector*.

The default:

•Corrects any words that don't appear in the index.

•Takes suggestions from the words in the index. To make certain fields use custom correctors, use the correctors argument to pass a dictionary mapping field names to *whoosh.spelling.Corrector* objects.

Expert users who want more sophisticated correction behavior can create a custom *whoosh.spelling.QueryCorrector* and use that instead of this method.

Returns a *whoosh.spelling.Correction* object with a `query` attribute containing the corrected *whoosh.query.Query* object and a `string` attributes containing the corrected query string.

```
>>> from whoosh import qparser, highlight
>>> qtext = 'mary "litle lamb"'
>>> q = qparser.QueryParser("text", myindex.schema)
>>> mysearcher = myindex.searcher()
>>> correction = mysearcher().correct_query(q, qtext)
>>> correction.query
<query.And ...>
>>> correction.string
'mary "little lamb"'
>>> mysearcher.close()
```

You can use the `Correction` object's `format_string` method to format the corrected query string using a `whoosh.highlight.Formatter` object. For example, you can format the corrected string as HTML, emphasizing the changed words.

```
>>> hf = highlight.HtmlFormatter(classname="change")
>>> correction.format_string(hf)
'mary "<strong class="change term0">little</strong> lamb"'
```

Parameters

- **q** – the *whoosh.query.Query* object to correct.
- **qstring** – the original user query from which the query object was created. You can pass None instead of a string, in which the second item in the returned tuple will also be None.
- **correctors** – an optional dictionary mapping fieldnames to *whoosh.spelling.Corrector* objects. By default, this method uses the contents of the index to spell check the terms in the query. You can use this argument to "override" some fields with a different correct, for example a `whoosh.spelling.GraphCorrector`.
- **terms** – a sequence of `("fieldname", "text")` tuples to correct in the query. By default, this method corrects terms that don't appear in the index. You can use this argument to override that behavior and explicitly specify the terms that should be corrected.
- **maxdist** – the maximum number of "edits" (insertions, deletions, subsitutions, or transpositions of letters) allowed between the original word and any suggestion. Values higher than `2` may be slow.
- **prefix** – suggested replacement words must share this number of initial characters with the original word. Increasing this even to just `1` can dramatically speed up suggestions, and may be justifiable since spellling mistakes rarely involve the first letter of a word.
- **aliases** – an optional dictionary mapping field names in the query to different field names to use as the source of spelling suggestions. The mappings in `correctors` are applied after this.

Return type *whoosh.spelling.Correction*

doc_count()
Returns the number of UNDELETED documents in the index.

doc_count_all()
Returns the total number of documents, DELETED OR UNDELETED, in the index.

docs_for_query(*q*, *for_deletion=False*)
Returns an iterator of document numbers for documents matching the given *whoosh.query.Query* object.

document(**kw)
Convenience method returns the stored fields of a document matching the given keyword arguments, where the keyword keys are field names and the values are terms that must appear in the field.

This method is equivalent to:

```
searcher.stored_fields(searcher.document_number(<keyword args>))
```

Where Searcher.documents() returns a generator, this function returns either a dictionary or None. Use it when you assume the given keyword arguments either match zero or one documents (i.e. at least one of the fields is a unique key).

```
>>> stored_fields = searcher.document(path=u"/a/b")
>>> if stored_fields:
...   print(stored_fields['title'])
... else:
...   print("There is no document with the path /a/b")
```

document_number(**kw)
Returns the document number of the document matching the given keyword arguments, where the keyword keys are field names and the values are terms that must appear in the field.

```
>>> docnum = searcher.document_number(path=u"/a/b")
```

Where Searcher.document_numbers() returns a generator, this function returns either an int or None. Use it when you assume the given keyword arguments either match zero or one documents (i.e. at least one of the fields is a unique key).

Return type int

document_numbers(**kw)
Returns a generator of the document numbers for documents matching the given keyword arguments, where the keyword keys are field names and the values are terms that must appear in the field. If you do not specify any arguments (`Searcher.document_numbers()`), this method will yield **all** document numbers.

```
>>> docnums = list(searcher.document_numbers(emailto="matt@whoosh.ca"))
```

documents(**kw)
Convenience method returns the stored fields of a document matching the given keyword arguments, where the keyword keys are field names and the values are terms that must appear in the field.

Returns a generator of dictionaries containing the stored fields of any documents matching the keyword arguments. If you do not specify any arguments (`Searcher.documents()`), this method will yield **all** documents.

```
>>> for stored_fields in searcher.documents(emailto=u"matt@whoosh.ca"):
...   print("Email subject:", stored_fields['subject'])
```

get_parent()
Returns the parent of this searcher (if has_parent() is True), or else self.

idf(*fieldname*, *text*)
Calculates the Inverse Document Frequency of the current term (calls idf() on the searcher's Weighting object).

key_terms(*docnums*, *fieldname*, *numterms=5*, *model=<class 'whoosh.classify.Bo1Model'>*, *normalize=True*)
Returns the 'numterms' most important terms from the documents listed (by number) in 'docnums'. You can get document numbers for the documents your interested in with the document_number() and document_numbers() methods.

"Most important" is generally defined as terms that occur frequently in the top hits but relatively infrequently in the collection as a whole.

```
>>> docnum = searcher.document_number(path=u"/a/b")
>>> keywords_and_scores = searcher.key_terms([docnum], "content")
```

This method returns a list of ("term", score) tuples. The score may be useful if you want to know the "strength" of the key terms, however to just get the terms themselves you can just do this:

```
>>> kws = [kw for kw, score in searcher.key_terms([docnum], "content")]
```

Parameters

- **fieldname** – Look at the terms in this field. This field must store vectors.
- **docnums** – A sequence of document numbers specifying which documents to extract key terms from.
- **numterms** – Return this number of important terms.
- **model** – The classify.ExpansionModel to use. See the classify module.
- **normalize** – normalize the scores.

Returns a list of ("term", score) tuples.

key_terms_from_text(*fieldname*, *text*, *numterms=5*, *model=<class 'whoosh.classify.Bo1Model'>*, *normalize=True*)
Return the 'numterms' most important terms from the given text.

Parameters

- **numterms** – Return this number of important terms.
- **model** – The classify.ExpansionModel to use. See the classify module.

more_like(*docnum*, *fieldname*, *text=None*, *top=10*, *numterms=5*, *model=<class 'whoosh.classify.Bo1Model'>*, *normalize=False*, *filter=None*)
Returns a *Results* object containing documents similar to the given document, based on "key terms" in the given field:

```
# Get the ID for the document you're interested in
docnum = search.document_number(path=u"/a/b/c")

r = searcher.more_like(docnum)

print("Documents like", searcher.stored_fields(docnum)["title"])
for hit in r:
    print(hit["title"])
```

Parameters

- **fieldname** – the name of the field to use to test similarity.
- **text** – by default, the method will attempt to load the contents of the field from the stored fields for the document, or from a term vector. If the field isn't stored or vectored in the index, but you have access to the text another way (for example, loading from a file or a database), you can supply it using the `text` parameter.
- **top** – the number of results to return.
- **numterms** – the number of "key terms" to extract from the hit and search for. Using more terms is slower but gives potentially more and more accurate results.
- **model** – (expert) a `whoosh.classify.ExpansionModel` to use to compute "key terms".
- **normalize** – whether to normalize term weights.
- **filter** – a query, Results object, or set of docnums. The results will only contain documents that are also in the filter object.

postings(*fieldname*, *text*, *weighting=None*, *qf=1*)
Returns a *whoosh.matching.Matcher* for the postings of the given term. Unlike the *whoosh.reading.IndexReader.postings()* method, this method automatically sets the scoring functions on the matcher from the searcher's weighting object.

reader()
Returns the underlying *IndexReader*.

refresh()
Returns a fresh searcher for the latest version of the index:

```
my_searcher = my_searcher.refresh()
```

If the index has not changed since this searcher was created, this searcher is simply returned.

This method may CLOSE underlying resources that are no longer needed by the refreshed searcher, so you CANNOT continue to use the original searcher after calling `refresh()` on it.

search(*q*, ***kwargs*)
Runs a *whoosh.query.Query* object on this searcher and returns a *Results* object. See *How to search* for more information.

This method takes many keyword arguments (documented below).

See *Sorting and faceting* for information on using `sortedby` and/or `groupedby`. See *Collapsing results* for more information on using `collapse`, `collapse_limit`, and `collapse_order`.

Parameters

- **query** – a *whoosh.query.Query* object to use to match documents.
- **limit** – the maximum number of documents to score. If you're only interested in the top N documents, you can set limit=N to limit the scoring for a faster search. Default is 10.
- **scored** – whether to score the results. Overriden by `sortedby`. If both `scored=False` and `sortedby=None`, the results will be in arbitrary order, but will usually be computed faster than scored or sorted results.
- **sortedby** – see *Sorting and faceting*.
- **reverse** – Reverses the direction of the sort. Default is False.
- **groupedby** – see *Sorting and faceting*.

- **optimize** – use optimizations to get faster results when possible. Default is True.
- **filter** – a query, Results object, or set of docnums. The results will only contain documents that are also in the filter object.
- **mask** – a query, Results object, or set of docnums. The results will not contain any documents that are in the mask object.
- **terms** – if True, record which terms were found in each matching document. See *How to search* for more information. Default is False.
- **maptype** – by default, the results of faceting with `groupedby` is a dictionary mapping group names to ordered lists of document numbers in the group. You can pass a *whoosh.sorting.FacetMap* subclass to this keyword argument to specify a different (usually faster) method for grouping. For example, `maptype=sorting.Count` would store only the count of documents in each group, instead of the full list of document IDs.
- **collapse** – a *facet* to use to collapse the results. See *Collapsing results* for more information.
- **collapse_limit** – the maximum number of documents to allow with the same collapse key. See *Collapsing results* for more information.
- **collapse_order** – an optional ordering *facet* to control which documents are kept when collapsing. The default (`collapse_order=None`) uses the results order (e.g. the highest scoring documents in a scored search).

Return type *Results*

search_page(*query*, *pagenum*, *pagelen=10*, ***kwargs*)

This method is Like the *Searcher.search()* method, but returns a *ResultsPage* object. This is a convenience function for getting a certain "page" of the results for the given query, which is often useful in web search interfaces.

For example:

```
querystring = request.get("q")
query = queryparser.parse("content", querystring)

pagenum = int(request.get("page", 1))
pagelen = int(request.get("perpage", 10))

results = searcher.search_page(query, pagenum, pagelen=pagelen)
print("Page %d of %d" % (results.pagenum, results.pagecount))
print("Showing results %d-%d of %d"
      % (results.offset + 1, results.offset + results.pagelen + 1,
         len(results)))
for hit in results:
    print("%d: %s" % (hit.rank + 1, hit["title"]))
```

(Note that results.pagelen might be less than the pagelen argument if there aren't enough results to fill a page.)

Any additional keyword arguments you supply are passed through to *Searcher.search()*. For example, you can get paged results of a sorted search:

```
results = searcher.search_page(q, 2, sortedby="date", reverse=True)
```

Currently, searching for page 100 with pagelen of 10 takes the same amount of time as using *Searcher.search()* to find the first 1000 results. That is, this method does not have any special optimizations or

efficiencies for getting a page from the middle of the full results list. (A future enhancement may allow using previous page results to improve the efficiency of finding the next page.)

This method will raise a `ValueError` if you ask for a page number higher than the number of pages in the resulting query.

Parameters

- **query** – the *whoosh.query.Query* object to match.
- **pagenum** – the page number to retrieve, starting at `1` for the first page.
- **pagelen** – the number of results per page.

Returns *ResultsPage*

search_with_collector(*q*, *collector*, *context=None*)

Low-level method: runs a *whoosh.query.Query* object on this searcher using the given *whoosh.collectors.Collector* object to collect the results:

```
myquery = query.Term("content", "cabbage")

uc = collectors.UnlimitedCollector()
tc = TermsCollector(uc)

mysearcher.search_with_collector(myquery, tc)
print(tc.docterms)
print(tc.results())
```

Note that this method does not return a *Results* object. You need to access the collector to get a results object or other information the collector might hold after the search.

Parameters

- **q** – a *whoosh.query.Query* object to use to match documents.
- **collector** – a *whoosh.collectors.Collector* object to feed the results into.

suggest(*fieldname*, *text*, *limit=5*, *maxdist=2*, *prefix=0*)

Returns a sorted list of suggested corrections for the given mis-typed word `text` based on the contents of the given field:

```
>>> searcher.suggest("content", "specail")
["special"]
```

This is a convenience method. If you are planning to get suggestions for multiple words in the same field, it is more efficient to get a *Corrector* object and use it directly:

```
corrector = searcher.corrector("fieldname")
for word in words:
    print(corrector.suggest(word))
```

Parameters

- **limit** – only return up to this many suggestions. If there are not enough terms in the field within `maxdist` of the given word, the returned list will be shorter than this number.
- **maxdist** – the largest edit distance from the given word to look at. Numbers higher than 2 are not very effective or efficient.

- **prefix** – require suggestions to share a prefix of this length with the given word. This is often justifiable since most misspellings do not involve the first letter of the word. Using a prefix dramatically decreases the time it takes to generate the list of words.

up_to_date()
Returns True if this Searcher represents the latest version of the index, for backends that support versioning.

Results classes

class whoosh.searching.**Results**(*searcher*, *q*, *top_n*, *docset=None*, *facetmaps=None*, *runtime=0*, *highlighter=None*)
This object is returned by a Searcher. This object represents the results of a search query. You can mostly use it as if it was a list of dictionaries, where each dictionary is the stored fields of the document at that position in the results.

Note that a Results object keeps a reference to the Searcher that created it, so keeping a reference to a Results object keeps the Searcher alive and so keeps all files used by it open.

Parameters

- **searcher** – the *Searcher* object that produced these results.
- **query** – the original query that created these results.
- **top_n** – a list of (score, docnum) tuples representing the top N search results.

copy()
Returns a deep copy of this results object.

docnum(*n*)
Returns the document number of the result at position n in the list of ranked documents.

docs()
Returns a set-like object containing the document numbers that matched the query.

estimated_length()
The estimated maximum number of matching documents, or the exact number of matching documents if it's known.

estimated_min_length()
The estimated minimum number of matching documents, or the exact number of matching documents if it's known.

extend(*results*)
Appends hits from 'results' (that are not already in this results object) to the end of these results.

Parameters **results** – another results object.

facet_names()
Returns the available facet names, for use with the groups() method.

fields(*n*)
Returns the stored fields for the document at the n th position in the results. Use *Results.docnum()* if you want the raw document number instead of the stored fields.

filter(*results*)
Removes any hits that are not also in the other results object.

groups(*name=None*)
If you generated facet groupings for the results using the *groupedby* keyword argument to the search()

method, you can use this method to retrieve the groups. You can use the `facet_names()` method to get the list of available facet names.

```
>>> results = searcher.search(my_query, groupedby=["tag", "price"])
>>> results.facet_names()
["tag", "price"]
>>> results.groups("tag")
{"new": [12, 1, 4], "apple": [3, 10, 5], "search": [11]}
```

If you only used one facet, you can call the method without a facet name to get the groups for the facet.

```
>>> results = searcher.search(my_query, groupedby="tag")
>>> results.groups()
{"new": [12, 1, 4], "apple": [3, 10, 5, 0], "search": [11]}
```

By default, this returns a dictionary mapping category names to a list of document numbers, in the same relative order as they appear in the results.

```
>>> results = mysearcher.search(myquery, groupedby="tag")
>>> docnums = results.groups()
>>> docnums['new']
[12, 1, 4]
```

You can then use `Searcher.stored_fields()` to get the stored fields associated with a document ID.

If you specified a different `maptype` for the facet when you searched, the values in the dictionary depend on the *whoosh.sorting.FacetMap*.

```
>>> myfacet = sorting.FieldFacet("tag", maptype=sorting.Count)
>>> results = mysearcher.search(myquery, groupedby=myfacet)
>>> counts = results.groups()
{"new": 3, "apple": 4, "search": 1}
```

has_exact_length()
Returns True if this results object already knows the exact number of matching documents.

has_matched_terms()
Returns True if the search recorded which terms matched in which documents.

```
>>> r = searcher.search(myquery)
>>> r.has_matched_terms()
False
>>>
```

is_empty()
Returns True if not documents matched the query.

items()
Returns an iterator of (docnum, score) pairs for the scored documents in the results.

key_terms(*fieldname*, *docs=10*, *numterms=5*, *model=<class 'whoosh.classify.Bo1Model'>*, *normalize=True*)
Returns the 'numterms' most important terms from the top 'docs' documents in these results. "Most important" is generally defined as terms that occur frequently in the top hits but relatively infrequently in the collection as a whole.

Parameters

- **fieldname** – Look at the terms in this field. This field must store vectors.

- **docs** – Look at this many of the top documents of the results.
- **numterms** – Return this number of important terms.
- **model** – The classify.ExpansionModel to use. See the classify module.

Returns list of unicode strings.

matched_terms()

Returns the set of `("fieldname", "text")` tuples representing terms from the query that matched one or more of the TOP N documents (this does not report terms for documents that match the query but did not score high enough to make the top N results). You can compare this set to the terms from the original query to find terms which didn't occur in any matching documents.

This is only valid if you used `terms=True` in the search call to record matching terms. Otherwise it will raise an exception.

```
>>> q = myparser.parse("alfa OR bravo OR charlie")
>>> results = searcher.search(q, terms=True)
>>> results.terms()
set([("content", "alfa"), ("content", "charlie")])
>>> q.all_terms() - results.terms()
set([("content", "bravo")])
```

score(*n*)

Returns the score for the document at the Nth position in the list of ranked documents. If the search was not scored, this may return None.

scored_length()

Returns the number of scored documents in the results, equal to or less than the `limit` keyword argument to the search.

```
>>> r = mysearcher.search(myquery, limit=20)
>>> len(r)
1246
>>> r.scored_length()
20
```

This may be fewer than the total number of documents that match the query, which is what `len(Results)` returns.

upgrade(*results*, *reverse=False*)

Re-sorts the results so any hits that are also in 'results' appear before hits not in 'results', otherwise keeping their current relative positions. This does not add the documents in the other results object to this one.

Parameters

- **results** – another results object.
- **reverse** – if True, lower the position of hits in the other results object instead of raising them.

upgrade_and_extend(*results*)

Combines the effects of extend() and upgrade(): hits that are also in 'results' are raised. Then any hits from the other results object that are not in this results object are appended to the end.

Parameters **results** – another results object.

class whoosh.searching.**Hit**(*results*, *docnum*, *pos=None*, *score=None*)

Represents a single search result ("hit") in a Results object.

This object acts like a dictionary of the matching document's stored fields. If for some reason you need an actual `dict` object, use `Hit.fields()` to get one.

```
>>> r = searcher.search(query.Term("content", "render"))
>>> r[0]
< Hit {title = u"Rendering the scene"} >
>>> r[0].rank
0
>>> r[0].docnum == 4592
True
>>> r[0].score
2.52045682
>>> r[0]["title"]
"Rendering the scene"
>>> r[0].keys()
["title"]
```

Parameters

- **results** – the Results object this hit belongs to.
- **pos** – the position in the results list of this hit, for example pos = 0 means this is the first (highest scoring) hit.
- **docnum** – the document number of this hit.
- **score** – the score of this hit.

fields()
Returns a dictionary of the stored fields of the document this object represents.

highlights(*fieldname*, *text=None*, *top=3*, *minscore=1*)
Returns highlighted snippets from the given field:

```
r = searcher.search(myquery)
for hit in r:
    print(hit["title"])
    print(hit.highlights("content"))
```

See *How to create highlighted search result excerpts*.

To change the fragmeter, formatter, order, or scorer used in highlighting, you can set attributes on the results object:

```
from whoosh import highlight

results = searcher.search(myquery, terms=True)
results.fragmenter = highlight.SentenceFragmenter()
```

...or use a custom *whoosh.highlight.Highlighter* object:

```
hl = highlight.Highlighter(fragmenter=sf)
results.highlighter = hl
```

Parameters

- **fieldname** – the name of the field you want to highlight.
- **text** – by default, the method will attempt to load the contents of the field from the stored fields for the document. If the field you want to highlight isn't stored in the index, but you

have access to the text another way (for example, loading from a file or a database), you can supply it using the `text` parameter.

- **top** – the maximum number of fragments to return.
- **minscore** – the minimum score for fragments to appear in the highlights.

matched_terms()

Returns the set of `("fieldname", "text")` tuples representing terms from the query that matched in this document. You can compare this set to the terms from the original query to find terms which didn't occur in this document.

This is only valid if you used `terms=True` in the search call to record matching terms. Otherwise it will raise an exception.

```
>>> q = myparser.parse("alfa OR bravo OR charlie")
>>> results = searcher.search(q, terms=True)
>>> for hit in results:
...   print(hit["title"])
...   print("Contains:", hit.matched_terms())
...   print("Doesn't contain:", q.all_terms() - hit.matched_terms())
```

more_like_this(*fieldname*, *text=None*, *top=10*, *numterms=5*, *model=<class 'whoosh.classify.Bo1Model'>*, *normalize=True*, *filter=None*)

Returns a new Results object containing documents similar to this hit, based on "key terms" in the given field:

```
r = searcher.search(myquery)
for hit in r:
    print(hit["title"])
    print("Top 3 similar documents:")
    for subhit in hit.more_like_this("content", top=3):
      print("  ", subhit["title"])
```

Parameters

- **fieldname** – the name of the field to use to test similarity.
- **text** – by default, the method will attempt to load the contents of the field from the stored fields for the document, or from a term vector. If the field isn't stored or vectored in the index, but you have access to the text another way (for example, loading from a file or a database), you can supply it using the `text` parameter.
- **top** – the number of results to return.
- **numterms** – the number of "key terms" to extract from the hit and search for. Using more terms is slower but gives potentially more and more accurate results.
- **model** – (expert) a `whoosh.classify.ExpansionModel` to use to compute "key terms".
- **normalize** – whether to normalize term weights.

class whoosh.searching.**ResultsPage**(*results*, *pagenum*, *pagelen=10*)

Represents a single page out of a longer list of results, as returned by *whoosh.searching.Searcher.search_page()*. Supports a subset of the interface of the *Results* object, namely getting stored fields with __getitem__ (square brackets), iterating, and the `score()` and `docnum()` methods.

The `offset` attribute contains the results number this page starts at (numbered from 0). For example, if the page length is 10, the `offset` attribute on the second page will be `10`.

The `pagecount` attribute contains the number of pages available.

The `pagenum` attribute contains the page number. This may be less than the page you requested if the results had too few pages. For example, if you do:

```
ResultsPage(results, 5)
```

but the results object only contains 3 pages worth of hits, `pagenum` will be 3.

The `pagelen` attribute contains the number of results on this page (which may be less than the page length you requested if this is the last page of the results).

The `total` attribute contains the total number of hits in the results.

```
>>> mysearcher = myindex.searcher()
>>> pagenum = 2
>>> page = mysearcher.find_page(pagenum, myquery)
>>> print("Page %s of %s, results %s to %s of %s" %
...       (pagenum, page.pagecount, page.offset+1,
...        page.offset+page.pagelen, page.total))
>>> for i, fields in enumerate(page):
...   print("%s. %r" % (page.offset + i + 1, fields))
>>> mysearcher.close()
```

To set highlighter attributes (for example `formatter`), access the underlying *`Results`* object:

```
page.results.formatter = highlight.UppercaseFormatter()
```

Parameters

- **results** – a *`Results`* object.
- **pagenum** – which page of the results to use, numbered from `1`.
- **pagelen** – the number of hits per page.

docnum(*n*)
Returns the document number of the hit at the nth position on this page.

is_last_page()
Returns True if this object represents the last page of results.

score(*n*)
Returns the score of the hit at the nth position on this page.

Exceptions

exception `whoosh.searching.`**NoTermsException**
Exception raised you try to access matched terms on a *`Results`* object was created without them. To record which terms matched in which document, you need to call the *`Searcher.search()`* method with `terms=True`.

exception `whoosh.searching.`**TimeLimit**
Raised by `TimeLimitedCollector` if the time limit is reached before the search finishes. If you have a reference to the collector, you can get partial results by calling `TimeLimitedCollector.results()`.

sorting module

Base types

class whoosh.sorting.**FacetType**
Base class for "facets", aspects that can be sorted/faceted.

categorizer(*global_searcher*)
Returns a *Categorizer* corresponding to this facet.

Parameters **global_searcher** – A parent searcher. You can use this searcher if you need global document ID references.

class whoosh.sorting.**Categorizer**
Base class for categorizer objects which compute a key value for a document based on certain criteria, for use in sorting/faceting.

Categorizers are created by FacetType objects through the *FacetType.categorizer()* method. The *whoosh.searching.Searcher* object passed to the `categorizer` method may be a composite searcher (that is, wrapping a multi-reader), but categorizers are always run **per-segment**, with segment-relative document numbers.

The collector will call a categorizer's `set_searcher` method as it searches each segment to let the cateogorizer set up whatever segment- specific data it needs.

`Collector.allow_overlap` should be `True` if the caller can use the `keys_for` method instead of `key_for` to group documents into potentially overlapping groups. The default is `False`.

If a categorizer subclass can categorize the document using only the document number, it should set `Collector.needs_current` to `False` (this is the default) and NOT USE the given matcher in the `key_for` or `keys_for` methods, since in that case `segment_docnum` is not guaranteed to be consistent with the given matcher. If a categorizer subclass needs to access information on the matcher, it should set `needs_current` to `True`. This will prevent the caller from using optimizations that might leave the matcher in an inconsistent state.

key_for(*matcher*, *segment_docnum*)
Returns a key for the current match.

Parameters

- **matcher** – a *whoosh.matching.Matcher* object. If `self.needs_current` is `False`, DO NOT use this object, since it may be inconsistent. Use the given `segment_docnum` instead.
- **segment_docnum** – the segment-relative document number of the current match.

key_to_name(*key*)
Returns a representation of the key to be used as a dictionary key in faceting. For example, the sorting key for date fields is a large integer; this method translates it into a `datetime` object to make the groupings clearer.

keys_for(*matcher*, *segment_docnum*)
Yields a series of keys for the current match.

This method will be called instead of `key_for` if `self.allow_overlap` is `True`.

Parameters

- **matcher** – a *whoosh.matching.Matcher* object. If `self.needs_current` is `False`, DO NOT use this object, since it may be inconsistent. Use the given `segment_docnum` instead.

- **segment_docnum** – the segment-relative document number of the current match.

set_searcher(*segment_searcher*, *docoffset*)
Called by the collector when the collector moves to a new segment. The `segment_searcher` will be atomic. The `docoffset` is the offset of the segment's document numbers relative to the entire index. You can use the offset to get absolute index docnums by adding the offset to segment-relative docnums.

Facet types

class `whoosh.sorting.`**FieldFacet**(*fieldname*, *reverse=False*, *allow_overlap=False*, *maptype=None*)
Sorts/facets by the contents of a field.

For example, to sort by the contents of the "path" field in reverse order, and facet by the contents of the "tag" field:

```
paths = FieldFacet("path", reverse=True)
tags = FieldFacet("tag")
results = searcher.search(myquery, sortedby=paths, groupedby=tags)
```

This facet returns different categorizers based on the field type.

Parameters

- **fieldname** – the name of the field to sort/facet on.
- **reverse** – if True, when sorting, reverse the sort order of this facet.
- **allow_overlap** – if True, when grouping, allow documents to appear in multiple groups when they have multiple terms in the field.

class `whoosh.sorting.`**QueryFacet**(*querydict*, *other=None*, *allow_overlap=False*, *maptype=None*)
Sorts/facets based on the results of a series of queries.

Parameters

- **querydict** – a dictionary mapping keys to *whoosh.query.Query* objects.
- **other** – the key to use for documents that don't match any of the queries.

class `whoosh.sorting.`**RangeFacet**(*fieldname*, *start*, *end*, *gap*, *hardend=False*, *maptype=None*)
Sorts/facets based on numeric ranges. For textual ranges, use *QueryFacet*.

For example, to facet the "price" field into $100 buckets, up to $1000:

```
prices = RangeFacet("price", 0, 1000, 100)
results = searcher.search(myquery, groupedby=prices)
```

The ranges/buckets are always **inclusive** at the start and **exclusive** at the end.

Parameters

- **fieldname** – the numeric field to sort/facet on.
- **start** – the start of the entire range.
- **end** – the end of the entire range.
- **gap** – the size of each "bucket" in the range. This can be a sequence of sizes. For example, `gap=[1,5,10]` will use 1 as the size of the first bucket, 5 as the size of the second bucket, and 10 as the size of all subsequent buckets.

- **hardend** – if True, the end of the last bucket is clamped to the value of `end`. If False (the default), the last bucket is always `gap` sized, even if that means the end of the last bucket is after `end`.

class `whoosh.sorting.`**DateRangeFacet**(*fieldname*, *start*, *end*, *gap*, *hardend=False*, *maptype=None*)

Sorts/facets based on date ranges. This is the same as RangeFacet except you are expected to use `daterange` objects as the start and end of the range, and `timedelta` or `relativedelta` objects as the gap(s), and it generates *`DateRange`* queries instead of *`TermRange`* queries.

For example, to facet a "birthday" range into 5 year buckets:

```
from datetime import datetime
from whoosh.support.relativedelta import relativedelta

startdate = datetime(1920, 0, 0)
enddate = datetime.now()
gap = relativedelta(years=5)
bdays = DateRangeFacet("birthday", startdate, enddate, gap)
results = searcher.search(myquery, groupedby=bdays)
```

The ranges/buckets are always **inclusive** at the start and **exclusive** at the end.

Parameters

- **fieldname** – the numeric field to sort/facet on.
- **start** – the start of the entire range.
- **end** – the end of the entire range.
- **gap** – the size of each "bucket" in the range. This can be a sequence of sizes. For example, `gap=[1,5,10]` will use 1 as the size of the first bucket, 5 as the size of the second bucket, and 10 as the size of all subsequent buckets.
- **hardend** – if True, the end of the last bucket is clamped to the value of `end`. If False (the default), the last bucket is always `gap` sized, even if that means the end of the last bucket is after `end`.

class `whoosh.sorting.`**ScoreFacet**

Uses a document's score as a sorting criterion.

For example, to sort by the `tag` field, and then within that by relative score:

```
tag_score = MultiFacet(["tag", ScoreFacet()])
results = searcher.search(myquery, sortedby=tag_score)
```

class `whoosh.sorting.`**FunctionFacet**(*fn*, *maptype=None*)

This facet type is low-level. In most cases you should use `TranslateFacet` instead.

This facet type ets you pass an arbitrary function that will compute the key. This may be easier than subclassing FacetType and Categorizer to set up the desired behavior.

The function is called with the arguments `(searcher, docid)`, where the `searcher` may be a composite searcher, and the `docid` is an absolute index document number (not segment-relative).

For example, to use the number of words in the document's "content" field as the sorting/faceting key:

```
fn = lambda s, docid: s.doc_field_length(docid, "content")
lengths = FunctionFacet(fn)
```

class `whoosh.sorting.`**MultiFacet**(*items=None*, *maptype=None*)

Sorts/facets by the combination of multiple "sub-facets".

For example, to sort by the value of the "tag" field, and then (for documents where the tag is the same) by the value of the "path" field:

```
facet = MultiFacet([FieldFacet("tag"), FieldFacet("path")])
results = searcher.search(myquery, sortedby=facet)
```

As a shortcut, you can use strings to refer to field names, and they will be assumed to be field names and turned into FieldFacet objects:

```
facet = MultiFacet(["tag", "path"])
```

You can also use the `add_*` methods to add criteria to the multifacet:

```
facet = MultiFacet()
facet.add_field("tag")
facet.add_field("path", reverse=True)
facet.add_query({"a-m": TermRange("name", "a", "m"),
                 "n-z": TermRange("name", "n", "z")})
```

class whoosh.sorting.**StoredFieldFacet**(*fieldname*, *allow_overlap=False*, *split_fn=None*, *maptype=None*)

Lets you sort/group using the value in an unindexed, stored field (e.g. *whoosh.fields.STORED*). This is usually slower than using an indexed field.

For fields where the stored value is a space-separated list of keywords, (e.g. `"tag1 tag2 tag3"`), you can use the `allow_overlap` keyword argument to allow overlapped faceting on the result of calling the `split()` method on the field value (or calling a custom split function if one is supplied).

Parameters

- **fieldname** – the name of the stored field.
- **allow_overlap** – if True, when grouping, allow documents to appear in multiple groups when they have multiple terms in the field. The categorizer uses `string.split()` or the custom `split_fn` to convert the stored value into a list of facet values.
- **split_fn** – a custom function to split a stored field value into multiple facet values when `allow_overlap` is True. If not supplied, the categorizer simply calls the value's `split()` method.

Facets object

class whoosh.sorting.**Facets**(*x=None*)

Maps facet names to *FacetType* objects, for creating multiple groupings of documents.

For example, to group by tag, and **also** group by price range:

```
facets = Facets()
facets.add_field("tag")
facets.add_facet("price", RangeFacet("price", 0, 1000, 100))
results = searcher.search(myquery, groupedby=facets)

tag_groups = results.groups("tag")
price_groups = results.groups("price")
```

(To group by the combination of multiple facets, use *MultiFacet*.)

add_facet(*name*, *facet*)

Adds a *FacetType* object under the given `name`.

add_facets(*facets*, *replace=True*)
Adds the contents of the given `Facets` or `dict` object to this object.

add_field(*fieldname*, ***kwargs*)
Adds a *FieldFacet* for the given field name (the field name is automatically used as the facet name).

add_query(*name*, *querydict*, ***kwargs*)
Adds a *QueryFacet* under the given `name`.

Parameters

- **name** – a name for the facet.
- **querydict** – a dictionary mapping keys to *whoosh.query.Query* objects.

items()
Returns a list of (facetname, facetobject) tuples for the facets in this object.

names()
Returns an iterator of the facet names in this object.

FacetType objects

class whoosh.sorting.**FacetMap**
Base class for objects holding the results of grouping search results by a Facet. Use an object's `as_dict()` method to access the results.

You can pass a subclass of this to the `maptype` keyword argument when creating a `FacetType` object to specify what information the facet should record about the group. For example:

```
# Record each document in each group in its sorted order
myfacet = FieldFacet("size", maptype=OrderedList)

# Record only the count of documents in each group
myfacet = FieldFacet("size", maptype=Count)
```

add(*groupname*, *docid*, *sortkey*)
Adds a document to the facet results.

Parameters

- **groupname** – the name of the group to add this document to.
- **docid** – the document number of the document to add.
- **sortkey** – a value representing the sort position of the document in the full results.

as_dict()
Returns a dictionary object mapping group names to implementation-specific values. For example, the value might be a list of document numbers, or a integer representing the number of documents in the group.

class whoosh.sorting.**OrderedList**
Stores a list of document numbers for each group, in the same order as they appear in the search results.

The `as_dict` method returns a dictionary mapping group names to lists of document numbers.

class whoosh.sorting.**UnorderedList**
Stores a list of document numbers for each group, in arbitrary order. This is slightly faster and uses less memory than `OrderedListResult` if you don't care about the ordering of the documents within groups.

The `as_dict` method returns a dictionary mapping group names to lists of document numbers.

class whoosh.sorting.**Count**
Stores the number of documents in each group.

The as_dict method returns a dictionary mapping group names to integers.

class whoosh.sorting.**Best**
Stores the "best" document in each group (that is, the one with the highest sort key).

The as_dict method returns a dictionary mapping group names to docnument numbers.

spelling module

See *correcting errors in user queries*. This module contains helper functions for correcting typos in user queries.

Corrector objects

class whoosh.spelling.**Corrector**
Base class for spelling correction objects. Concrete sub-classes should implement the _suggestions method.

suggest(*text*, *limit=5*, *maxdist=2*, *prefix=0*)

Parameters

- **text** – the text to check. This word will **not** be added to the suggestions, even if it appears in the word graph.
- **limit** – only return up to this many suggestions. If there are not enough terms in the field within maxdist of the given word, the returned list will be shorter than this number.
- **maxdist** – the largest edit distance from the given word to look at. Values higher than 2 are not very effective or efficient.
- **prefix** – require suggestions to share a prefix of this length with the given word. This is often justifiable since most misspellings do not involve the first letter of the word. Using a prefix dramatically decreases the time it takes to generate the list of words.

class whoosh.spelling.**ReaderCorrector**(*reader*, *fieldname*, *fieldobj*)
Suggests corrections based on the content of a field in a reader.

Ranks suggestions by the edit distance, then by highest to lowest frequency.

class whoosh.spelling.**MultiCorrector**(*correctors*, *op*)
Merges suggestions from a list of sub-correctors.

QueryCorrector objects

class whoosh.spelling.**QueryCorrector**(*fieldname*)
Base class for objects that correct words in a user query.

correct_query(*q*, *qstring*)
Returns a *Correction* object representing the corrected form of the given query.

Parameters

- **q** – the original *whoosh.query.Query* tree to be corrected.
- **qstring** – the original user query. This may be None if the original query string is not available, in which case the Correction.string attribute will also be None.

Return type *Correction*

class whoosh.spelling.**SimpleQueryCorrector**(*correctors*, *terms*, *aliases=None*, *prefix=0*, *maxdist=2*)

A simple query corrector based on a mapping of field names to *Corrector* objects, and a list of `("fieldname", "text")` tuples to correct. And terms in the query that appear in list of term tuples are corrected using the appropriate corrector.

Parameters

- **correctors** – a dictionary mapping field names to *Corrector* objects.
- **terms** – a sequence of `("fieldname", "text")` tuples representing terms to be corrected.
- **aliases** – a dictionary mapping field names in the query to field names for spelling suggestions.
- **prefix** – suggested replacement words must share this number of initial characters with the original word. Increasing this even to just `1` can dramatically speed up suggestions, and may be justifiable since spellling mistakes rarely involve the first letter of a word.
- **maxdist** – the maximum number of "edits" (insertions, deletions, subsitutions, or transpositions of letters) allowed between the original word and any suggestion. Values higher than `2` may be slow.

class whoosh.spelling.**Correction**(*q*, *qstring*, *corr_q*, *tokens*)

Represents the corrected version of a user query string. Has the following attributes:

query The corrected *whoosh.query.Query* object.

string The corrected user query string.

original_query The original *whoosh.query.Query* object that was corrected.

original_string The original user query string.

tokens A list of token objects representing the corrected words.

You can also use the `Correction.format_string()` method to reformat the corrected query string using a `whoosh.highlight.Formatter` class. For example, to display the corrected query string as HTML with the changed words emphasized:

```
from whoosh import highlight

correction = mysearcher.correct_query(q, qstring)

hf = highlight.HtmlFormatter(classname="change")
html = correction.format_string(hf)
```

support.charset module

This module contains tools for working with Sphinx charset table files. These files are useful for doing case and accent folding. See *whoosh.analysis.CharsetTokenizer* and *whoosh.analysis.CharsetFilter*.

whoosh.support.charset.**default_charset**

An extensive case- and accent folding charset table. Taken from http://speeple.com/unicode-maps.txt

whoosh.support.charset.**charset_table_to_dict**(*tablestring*)

Takes a string with the contents of a Sphinx charset table file and returns a mapping object (a defaultdict, actually) of the kind expected by the unicode.translate() method: that is, it maps a character number to a unicode character or None if the character is not a valid word character.

The Sphinx charset table format is described at http://www.sphinxsearch.com/docs/current.html#conf-charset-table.

support.levenshtein module

Contains functions implementing edit distance algorithms.

whoosh.support.levenshtein.**relative**(*a*, *b*)
Returns the relative distance between two strings, in the range [0-1] where 1 means total equality.

whoosh.support.levenshtein.**distance**(*seq1*, *seq2*, *limit=None*)
Returns the Damerau-Levenshtein edit distance between two strings.

util module

whoosh.util.**fib**(*n*)
Returns the nth value in the Fibonacci sequence.

whoosh.util.**make_binary_tree**(*fn*, *args*, ***kwargs*)
Takes a function/class that takes two positional arguments and a list of arguments and returns a binary tree of results/instances.

```
>>> make_binary_tree(UnionMatcher, [matcher1, matcher2, matcher3])
UnionMatcher(matcher1, UnionMatcher(matcher2, matcher3))
```

Any keyword arguments given to this function are passed to the class initializer.

whoosh.util.**make_weighted_tree**(*fn*, *ls*, ***kwargs*)
Takes a function/class that takes two positional arguments and a list of (weight, argument) tuples and returns a huffman-like weighted tree of results/instances.

whoosh.util.**synchronized**(*func*)
Decorator for storage-access methods, which synchronizes on a threading lock. The parent object must have 'is_closed' and '_sync_lock' attributes.

whoosh.util.**unclosed**(*method*)
Decorator to check if the object is closed.

writing module

Writer

class whoosh.writing.**IndexWriter**
High-level object for writing to an index.

To get a writer for a particular index, call *writer()* on the Index object.

```
>>> writer = myindex.writer()
```

You can use this object as a context manager. If an exception is thrown from within the context it calls *cancel()* to clean up temporary files, otherwise it calls *commit()* when the context exits.

```
>>> with myindex.writer() as w:
...     w.add_document(title="First document", content="Hello there.")
...     w.add_document(title="Second document", content="This is easy!")
```

add_document(***fields*)

The keyword arguments map field names to the values to index/store:

```
w = myindex.writer()
w.add_document(path=u"/a", title=u"First doc", text=u"Hello")
w.commit()
```

Depending on the field type, some fields may take objects other than unicode strings. For example, NUMERIC fields take numbers, and DATETIME fields take `datetime.datetime` objects:

```
from datetime import datetime, timedelta
from whoosh import index
from whoosh.fields import *

schema = Schema(date=DATETIME, size=NUMERIC(float), content=TEXT)
myindex = index.create_in("indexdir", schema)

w = myindex.writer()
w.add_document(date=datetime.now(), size=5.5, content=u"Hello")
w.commit()
```

Instead of a single object (i.e., unicode string, number, or datetime), you can supply a list or tuple of objects. For unicode strings, this bypasses the field's analyzer. For numbers and dates, this lets you add multiple values for the given field:

```
date1 = datetime.now()
date2 = datetime(2005, 12, 25)
date3 = datetime(1999, 1, 1)
w.add_document(date=[date1, date2, date3], size=[9.5, 10],
               content=[u"alfa", u"bravo", u"charlie"])
```

For fields that are both indexed and stored, you can specify an alternate value to store using a keyword argument in the form "_stored_<fieldname>". For example, if you have a field named "title" and you want to index the text "a b c" but store the text "e f g", use keyword arguments like this:

```
writer.add_document(title=u"a b c", _stored_title=u"e f g")
```

You can boost the weight of all terms in a certain field by specifying a `_<fieldname>_boost` keyword argument. For example, if you have a field named "content", you can double the weight of this document for searches in the "content" field like this:

```
writer.add_document(content="a b c", _title_boost=2.0)
```

You can boost every field at once using the `_boost` keyword. For example, to boost fields "a" and "b" by 2.0, and field "c" by 3.0:

```
writer.add_document(a="alfa", b="bravo", c="charlie",
                    _boost=2.0, _c_boost=3.0)
```

Note that some scoring algroithms, including Whoosh's default BM25F, do not work with term weights less than 1, so you should generally not use a boost factor less than 1.

See also `Writer.update_document()`.

add_field(*fieldname*, *fieldtype*, ***kwargs*)

Adds a field to the index's schema.

Parameters

- **fieldname** – the name of the field to add.
- **fieldtype** – an instantiated *whoosh.fields.FieldType* object.

cancel()
Cancels any documents/deletions added by this object and unlocks the index.

commit()
Finishes writing and unlocks the index.

delete_by_query(*q*, *searcher=None*)
Deletes any documents matching a query object.

Returns the number of documents deleted.

delete_by_term(*fieldname*, *text*, *searcher=None*)
Deletes any documents containing "term" in the "fieldname" field. This is useful when you have an indexed field containing a unique ID (such as "pathname") for each document.

Returns the number of documents deleted.

delete_document(*docnum*, *delete=True*)
Deletes a document by number.

end_group()
Finish indexing a group of hierarchical documents. See *start_group()*.

group()
Returns a context manager that calls *start_group()* and *end_group()* for you, allowing you to use a `with` statement to group hierarchical documents:

```
with myindex.writer() as w:
    with w.group():
        w.add_document(kind="class", name="Accumulator")
        w.add_document(kind="method", name="add")
        w.add_document(kind="method", name="get_result")
        w.add_document(kind="method", name="close")

    with w.group():
        w.add_document(kind="class", name="Calculator")
        w.add_document(kind="method", name="add")
        w.add_document(kind="method", name="multiply")
        w.add_document(kind="method", name="get_result")
        w.add_document(kind="method", name="close")
```

reader(***kwargs*)
Returns a reader for the existing index.

remove_field(*fieldname*, ***kwargs*)
Removes the named field from the index's schema. Depending on the backend implementation, this may or may not actually remove existing data for the field from the index. Optimizing the index should always clear out existing data for a removed field.

start_group()
Start indexing a group of hierarchical documents. The backend should ensure that these documents are all added to the same segment:

```
with myindex.writer() as w:
    w.start_group()
    w.add_document(kind="class", name="Accumulator")
    w.add_document(kind="method", name="add")
    w.add_document(kind="method", name="get_result")
```

```
    w.add_document(kind="method", name="close")
    w.end_group()

    w.start_group()
    w.add_document(kind="class", name="Calculator")
    w.add_document(kind="method", name="add")
    w.add_document(kind="method", name="multiply")
    w.add_document(kind="method", name="get_result")
    w.add_document(kind="method", name="close")
    w.end_group()
```

A more convenient way to group documents is to use the *group()* method and the `with` statement.

update_document(***fields*)

The keyword arguments map field names to the values to index/store.

This method adds a new document to the index, and automatically deletes any documents with the same values in any fields marked "unique" in the schema:

```
schema = fields.Schema(path=fields.ID(unique=True, stored=True),
                       content=fields.TEXT)
myindex = index.create_in("index", schema)

w = myindex.writer()
w.add_document(path=u"/", content=u"Mary had a lamb")
w.commit()

w = myindex.writer()
w.update_document(path=u"/", content=u"Mary had a little lamb")
w.commit()

assert myindex.doc_count() == 1
```

It is safe to use `update_document` in place of `add_document`; if there is no existing document to replace, it simply does an add.

You cannot currently pass a list or tuple of values to a "unique" field.

Because this method has to search for documents with the same unique fields and delete them before adding the new document, it is slower than using `add_document`.

- Marking more fields "unique" in the schema will make each `update_document` call slightly slower.
- When you are updating multiple documents, it is faster to batch delete all changed documents and then use `add_document` to add the replacements instead of using `update_document`.

Note that this method will only replace a *committed* document; currently it cannot replace documents you've added to the IndexWriter but haven't yet committed. For example, if you do this:

```
>>> writer.update_document(unique_id=u"1", content=u"Replace me")
>>> writer.update_document(unique_id=u"1", content=u"Replacement")
```

...this will add two documents with the same value of `unique_id`, instead of the second document replacing the first.

See `Writer.add_document()` for information on `_stored_<fieldname>`, `_<fieldname>_boost`, and `_boost` keyword arguments.

Utility writers

class whoosh.writing.**BufferedWriter**(*index*, *period=60*, *limit=10*, *writerargs=None*, *committargs=None*)

Convenience class that acts like a writer but buffers added documents before dumping the buffered documents as a batch into the actual index.

In scenarios where you are continuously adding single documents very rapidly (for example a web application where lots of users are adding content simultaneously), using a BufferedWriter is *much* faster than opening and committing a writer for each document you add. If you're adding batches of documents at a time, you can just use a regular writer.

(This class may also be useful for batches of `update_document` calls. In a normal writer, `update_document` calls cannot update documents you've added *in that writer*. With `BufferedWriter`, this will work.)

To use this class, create it from your index and *keep it open*, sharing it between threads.

```
>>> from whoosh.writing import BufferedWriter
>>> writer = BufferedWriter(myindex, period=120, limit=20)
>>> # Then you can use the writer to add and update documents
>>> writer.add_document(...)
>>> writer.add_document(...)
>>> writer.add_document(...)
>>> # Before the writer goes out of scope, call close() on it
>>> writer.close()
```

Note: This object stores documents in memory and may keep an underlying writer open, so you must explicitly call the `close()` method on this object before it goes out of scope to release the write lock and make sure any uncommitted changes are saved.

You can read/search the combination of the on-disk index and the buffered documents in memory by calling `BufferedWriter.reader()` or `BufferedWriter.searcher()`. This allows quasi-real-time search, where documents are available for searching as soon as they are buffered in memory, before they are committed to disk.

Tip: By using a searcher from the shared writer, multiple *threads* can search the buffered documents. Of course, other *processes* will only see the documents that have been written to disk. If you want indexed documents to become available to other processes as soon as possible, you have to use a traditional writer instead of a `BufferedWriter`.

You can control how often the `BufferedWriter` flushes the in-memory index to disk using the `period` and `limit` arguments. `period` is the maximum number of seconds between commits. `limit` is the maximum number of additions to buffer between commits.

You don't need to call `commit()` on the `BufferedWriter` manually. Doing so will just flush the buffered documents to disk early. You can continue to make changes after calling `commit()`, and you can call `commit()` multiple times.

Parameters

- **index** – the *whoosh.index.Index* to write to.
- **period** – the maximum amount of time (in seconds) between commits. Set this to `0` or `None` to not use a timer. Do not set this any lower than a few seconds.

- **limit** – the maximum number of documents to buffer before committing.
- **writerargs** – dictionary specifying keyword arguments to be passed to the index's `writer()` method when creating a writer.

class `whoosh.writing.`**AsyncWriter**(*index*, *delay=0.25*, *writerargs=None*)

Convenience wrapper for a writer object that might fail due to locking (i.e. the `filedb` writer). This object will attempt once to obtain the underlying writer, and if it's successful, will simply pass method calls on to it.

If this object *can't* obtain a writer immediately, it will *buffer* delete, add, and update method calls in memory until you call `commit()`. At that point, this object will start running in a separate thread, trying to obtain the writer over and over, and once it obtains it, "replay" all the buffered method calls on it.

In a typical scenario where you're adding a single or a few documents to the index as the result of a Web transaction, this lets you just create the writer, add, and commit, without having to worry about index locks, retries, etc.

For example, to get an aynchronous writer, instead of this:

```
>>> writer = myindex.writer()
```

Do this:

```
>>> from whoosh.writing import AsyncWriter
>>> writer = AsyncWriter(myindex)
```

Parameters

- **index** – the *whoosh.index.Index* to write to.
- **delay** – the delay (in seconds) between attempts to instantiate the actual writer.
- **writerargs** – an optional dictionary specifying keyword arguments to to be passed to the index's `writer()` method.

Exceptions

exception `whoosh.writing.`**IndexingError**

Technical notes

How to implement a new backend

Index

- Subclass *whoosh.index.Index*.
- Indexes must implement the following methods.
  - *whoosh.index.Index.is_empty()*
  - *whoosh.index.Index.doc_count()*
  - *whoosh.index.Index.reader()*
  - *whoosh.index.Index.writer()*
- Indexes that require/support locking must implement the following methods.

  - whoosh.index.Index.lock()
  - whoosh.index.Index.unlock()
- Indexes that support deletion must implement the following methods.
  - whoosh.index.Index.delete_document()
  - *whoosh.index.Index.doc_count_all()* – if the backend has delayed deletion.
- Indexes that require/support versioning/transactions *may* implement the following methods.
  - *whoosh.index.Index.latest_generation()*
  - *whoosh.index.Index.up_to_date()*
  - *whoosh.index.Index.last_modified()*
- Index *may* implement the following methods (the base class's versions are no-ops).
  - *whoosh.index.Index.optimize()*
  - *whoosh.index.Index.close()*

IndexWriter

- Subclass *whoosh.writing.IndexWriter*.
- IndexWriters must implement the following methods.
  - *whoosh.writing.IndexWriter.add_document()*
  - whoosh.writing.IndexWriter.add_reader()
- Backends that support deletion must implement the following methods.
  - *whoosh.writing.IndexWriter.delete_document()*
- IndexWriters that work as transactions must implement the following methods.
  - whoosh.reading.IndexWriter.commit() – Save the additions/deletions done with this IndexWriter to the main index, and release any resources used by the IndexWriter.
  - whoosh.reading.IndexWriter.cancel() – Throw away any additions/deletions done with this IndexWriter, and release any resources used by the IndexWriter.

IndexReader

- Subclass *whoosh.reading.IndexReader*.
- IndexReaders must implement the following methods.
  - whoosh.reading.IndexReader.__contains__()
  - whoosh.reading.IndexReader.__iter__()
  - *whoosh.reading.IndexReader.iter_from()*
  - *whoosh.reading.IndexReader.stored_fields()*
  - *whoosh.reading.IndexReader.doc_count_all()*
  - *whoosh.reading.IndexReader.doc_count()*
  - *whoosh.reading.IndexReader.doc_field_length()*

  - *whoosh.reading.IndexReader.field_length()*
  - *whoosh.reading.IndexReader.max_field_length()*
  - *whoosh.reading.IndexReader.postings()*
  - *whoosh.reading.IndexReader.has_vector()*
  - *whoosh.reading.IndexReader.vector()*
  - *whoosh.reading.IndexReader.doc_frequency()*
  - *whoosh.reading.IndexReader.frequency()*
- Backends that support deleting documents should implement the following methods.
  - *whoosh.reading.IndexReader.has_deletions()*
  - *whoosh.reading.IndexReader.is_deleted()*
- Backends that support versioning should implement the following methods.
  - *whoosh.reading.IndexReader.generation()*
- If the IndexReader object does not keep the schema in the `self.schema` attribute, it needs to override the following methods.
  - `whoosh.reading.IndexReader.field()`
  - `whoosh.reading.IndexReader.field_names()`
  - `whoosh.reading.IndexReader.scorable_names()`
  - `whoosh.reading.IndexReader.vector_names()`
- IndexReaders *may* implement the following methods.
  - `whoosh.reading.DocReader.close()` – closes any open resources associated with the reader.

Matcher

The *whoosh.reading.IndexReader.postings()* method returns a *whoosh.matching.Matcher* object. You will probably need to implement a custom Matcher class for reading from your posting lists.

- Subclass *whoosh.matching.Matcher*.
- Implement the following methods at minimum.
  - *whoosh.matching.Matcher.is_active()*
  - *whoosh.matching.Matcher.copy()*
  - *whoosh.matching.Matcher.id()*
  - *whoosh.matching.Matcher.next()*
  - *whoosh.matching.Matcher.value()*
  - *whoosh.matching.Matcher.value_as()*
  - *whoosh.matching.Matcher.score()*
- Depending on the implementation, you *may* implement the following methods more efficiently.
  - *whoosh.matching.Matcher.skip_to()*
  - *whoosh.matching.Matcher.weight()*
- If the implementation supports quality, you should implement the following methods.

- whoosh.matching.Matcher.supports_quality()
- whoosh.matching.Matcher.quality()
- *whoosh.matching.Matcher.block_quality()*
- *whoosh.matching.Matcher.skip_to_quality()*

filedb notes

TBD.

Files created

<revision_number>.toc The "master" file containing information about the index and its segments.

The index directory will contain a set of files for each segment. A segment is like a mini-index – when you add documents to the index, whoosh creates a new segment and then searches the old segment(s) and the new segment to avoid having to do a big merge every time you add a document. When you get enough small segments whoosh will merge them into larger segments or a single segment.

<segment_number>.dci Contains per-document information (e.g. field lengths). This will grow linearly with the number of documents.

<segment_number>.dcz Contains the stored fields for each document.

<segment_number>.tiz Contains per-term information. The size of file will vary based on the number of unique terms.

<segment_number>.pst Contains per-term postings. The size of this file depends on the size of the collection and the formats used for each field (e.g. storing term positions takes more space than storing frequency only).

<segment_number>.fvz contains term vectors (forward indexes) for each document. This file is only created if at least one field in the schema stores term vectors. The size will vary based on the number of documents, field length, the formats used for each vector (e.g. storing term positions takes more space than storing frequency only), etc.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

D

E

F

G

N

O

P

T

U

V

W