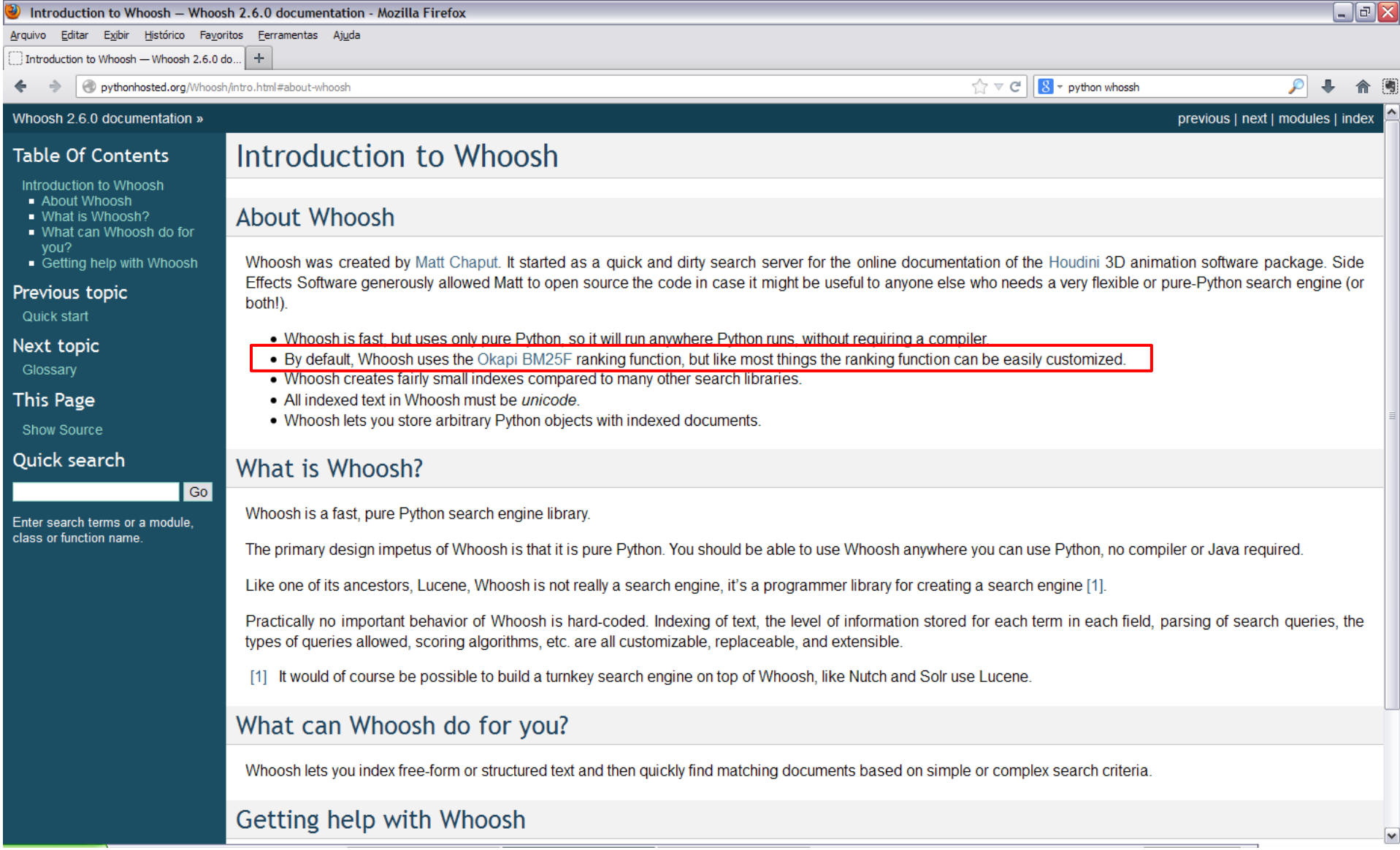


Whoosh – "ranking function"

Okapi BM25 - Wikipedia, the free encyclopedia - Mozilla Firefox

Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda

W Okapi BM25 - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Okapi_BM25

python whoosh

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Español
Français
Русский
Tiếng Việt
Edit links

Article Talk | Read Edit View history | Search

Okapi BM25

From Wikipedia, the free encyclopedia

In information retrieval, **Okapi BM25** is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. It is based on the probabilistic retrieval framework developed in the 1970s and 1980s by Stephen E. Robertson, Karen Spärck Jones, and others.

The name of the actual ranking function is BM25. To set the right context, however, it usually referred to as "Okapi BM25", since the Okapi information retrieval system, implemented at London's City University in the 1980s and 1990s, was the first system to implement this function.

BM25, and its newer variants, e.g. BM25F (a version of BM25 that can take document structure and anchor text into account), represent state-of-the-art TF-IDF-like retrieval functions used in document retrieval, such as Web search.

Contents [hide]

The ranking function [edit]

BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of the inter-relationship between the query terms within a document (e.g., their relative proximity). It is not a single function, but actually a whole family of scoring functions, with slightly different components and parameters. One of the most prominent instantiations of the function is as follows.

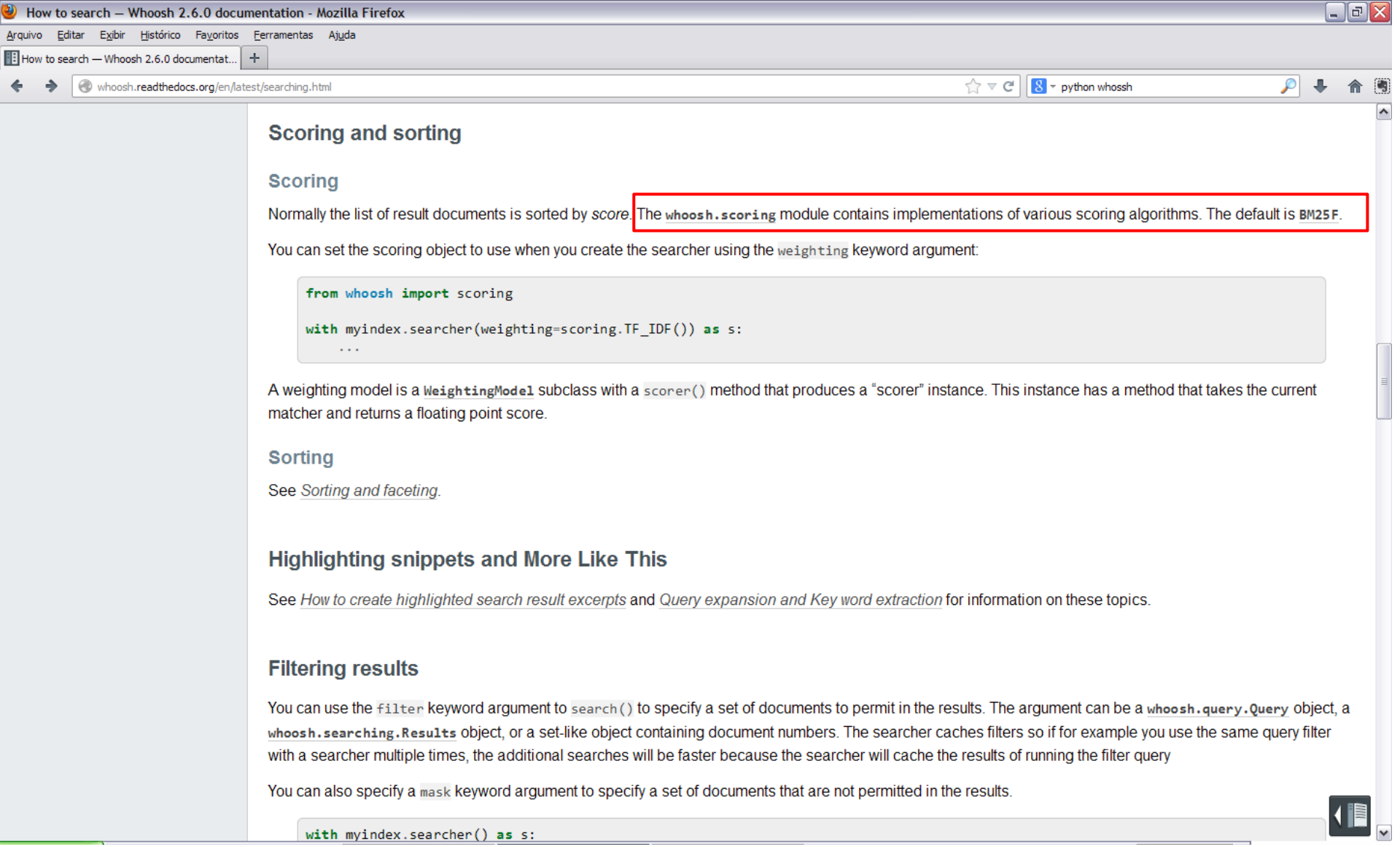
Given a query Q, containing keywords $q_1, \ldots, q_n$, the BM25 score of a document D is:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})},$$

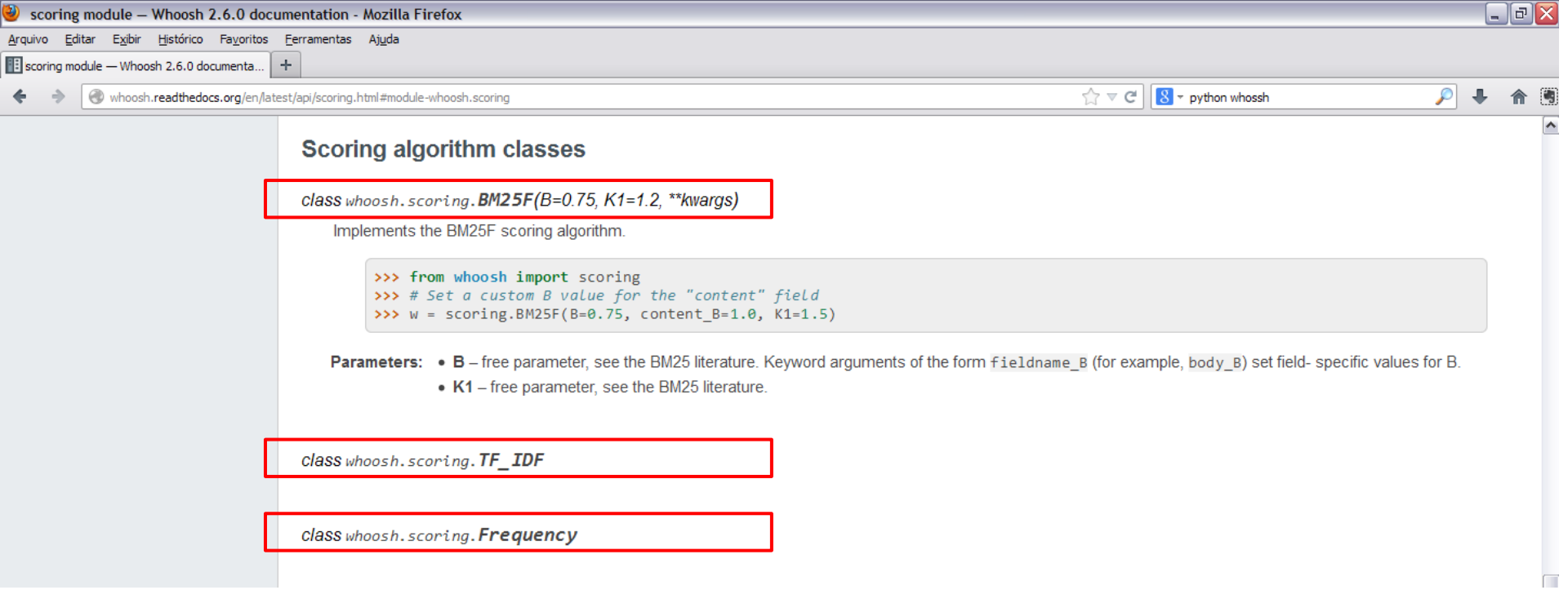
where $f(q_i, D)$ is q_i's term frequency in the document D, $|D|$ is the length of the document D in words, and $avgdl$ is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.[1] $\text{IDF}(q_i)$ is the IDF (inverse document frequency) weight of the query term q_i. It is usually computed as:

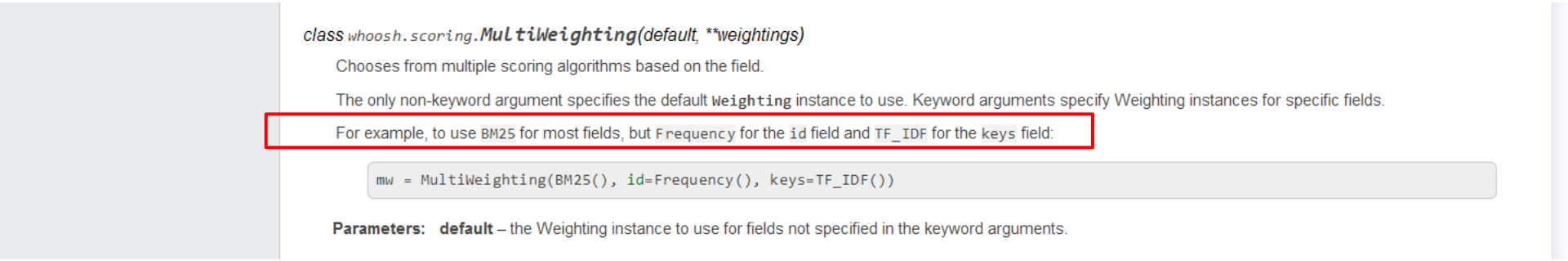
$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

Whoosh – "ranking function" (cont.)

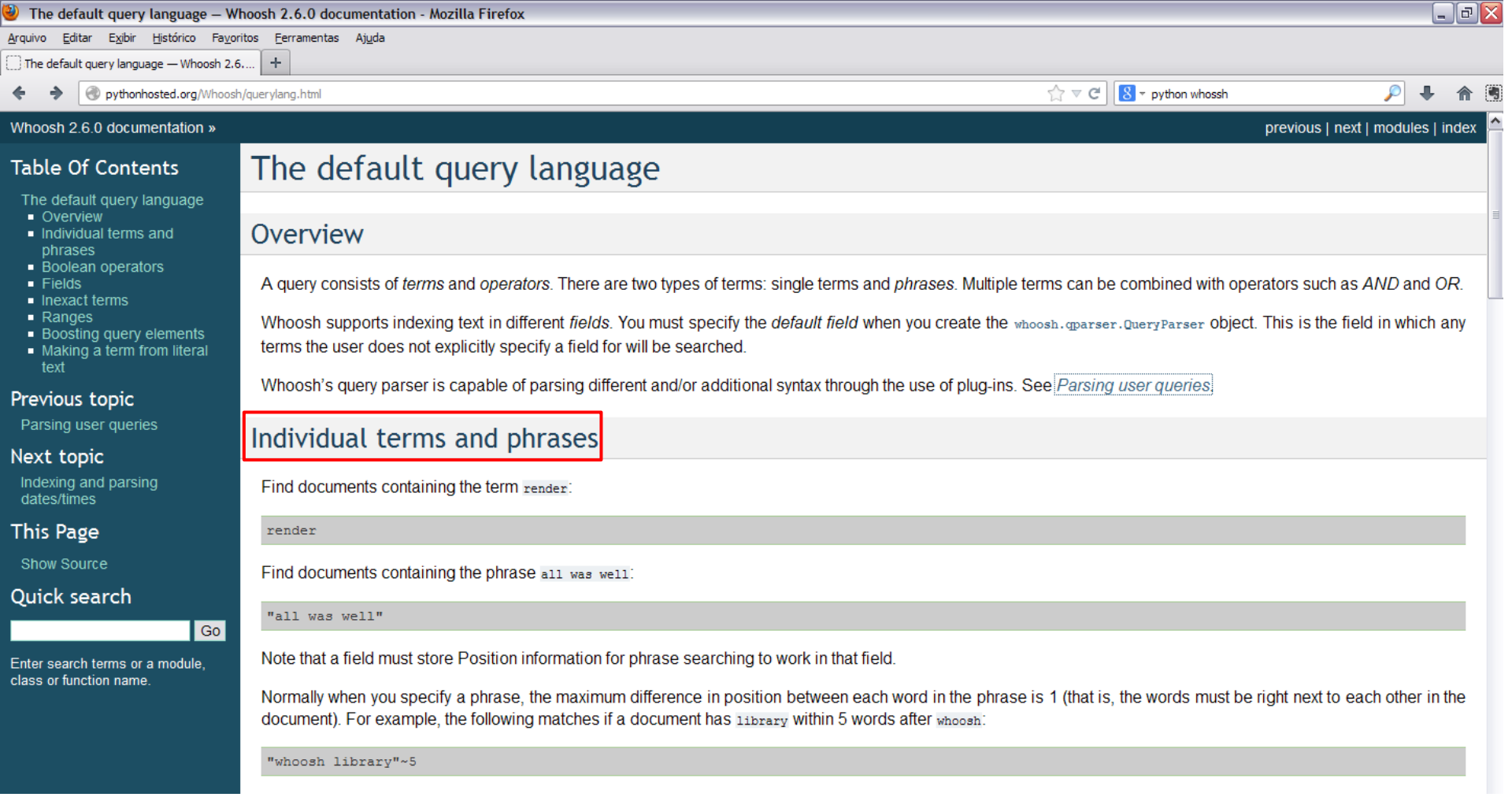


Whoosh – "ranking function" (cont.)





Whoosh – “ranking function” (cont.)



Whoosh – "query language"

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
```

Whoosh – "query language" (cont.)

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.

Whoosh – "query language" (cont.)

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()'
```

Whoosh – "query language" (cont.)

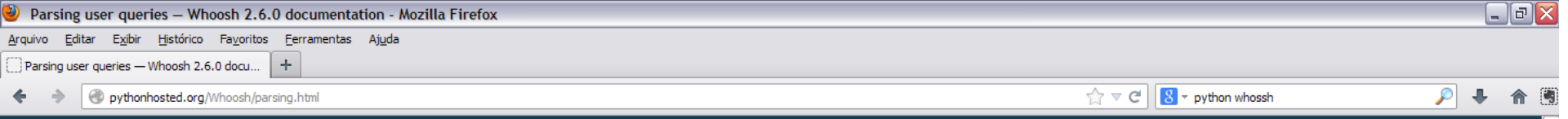


Table Of Contents

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.

Common customizations

http://pythonhosted.org/Whoosh/parsing.html

Whoosh – "query parser"