# Elasticsearch DSL Documentation

*Release 7.1.0*

**Honza Král**

**Nov 16, 2019**

# Contents

Elasticsearch DSL is a high-level library whose aim is to help with writing and running queries against Elasticsearch. It is built on top of the official low-level client (`elasticsearch-py`).

It provides a more convenient and idiomatic way to write and manipulate queries. It stays close to the Elasticsearch JSON DSL, mirroring its terminology and structure. It exposes the whole range of the DSL from Python either directly using defined classes or a queryset-like expressions.

It also provides an optional wrapper for working with documents as Python objects: defining mappings, retrieving and saving documents, wrapping the document data in user-defined classes.

To use the other Elasticsearch APIs (eg. cluster health) just use the underlying client.

CHAPTER 1

# Examples

Please see the examples directory to see some complex examples using `elasticsearch-dsl`.

# CHAPTER 2

## Compatibility

The library is compatible with all Elasticsearch versions since `2.x` but you **have to use a matching major version**:

For **Elasticsearch 7.0** and later, use the major version 7 (`7.x.y`) of the library.

For **Elasticsearch 6.0** and later, use the major version 6 (`6.x.y`) of the library.

For **Elasticsearch 5.0** and later, use the major version 5 (`5.x.y`) of the library.

For **Elasticsearch 2.0** and later, use the major version 2 (`2.x.y`) of the library.

The recommended way to set your requirements in your *setup.py* or *requirements.txt* is:

```
# Elasticsearch 7.x
elasticsearch-dsl>=7.0.0,<8.0.0

# Elasticsearch 6.x
elasticsearch-dsl>=6.0.0,<7.0.0

# Elasticsearch 5.x
elasticsearch-dsl>=5.0.0,<6.0.0

# Elasticsearch 2.x
elasticsearch-dsl>=2.0.0,<3.0.0
```

The development is happening on `master`, older branches only get bugfix releases

# CHAPTER 3

## Search Example

Let's have a typical search request written directly as a `dict`:

```
from elasticsearch import Elasticsearch
client = Elasticsearch()

response = client.search(
    index="my-index",
    body={
      "query": {
        "filtered": {
          "query": {
            "bool": {
              "must": [{"match": {"title": "python"}}],
              "must_not": [{"match": {"description": "beta"}}]
            }
          },
          "filter": {"term": {"category": "search"}}
        }
      },
      "aggs" : {
        "per_tag": {
          "terms": {"field": "tags"},
          "aggs": {
            "max_lines": {"max": {"field": "lines"}}
          }
        }
      }
    }
)

for hit in response['hits']['hits']:
    print(hit['_score'], hit['_source']['title'])

for tag in response['aggregations']['per_tag']['buckets']:
    print(tag['key'], tag['max_lines']['value'])
```

The problem with this approach is that it is very verbose, prone to syntax mistakes like incorrect nesting, hard to modify (eg. adding another filter) and definitely not fun to write.

Let's rewrite the example using the Python DSL:

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

s = Search(using=client, index="my-index") \
    .filter("term", category="search") \
    .query("match", title="python")   \
    .exclude("match", description="beta")

s.aggs.bucket('per_tag', 'terms', field='tags') \
    .metric('max_lines', 'max', field='lines')

response = s.execute()

for hit in response:
    print(hit.meta.score, hit.title)

for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

As you see, the library took care of:

- creating appropriate `Query` objects by name (eq. "match")
- composing queries into a compound `bool` query
- creating a `filtered` query since `.filter()` was used
- providing a convenient access to response data
- no curly or square brackets everywhere

# CHAPTER 4

## Persistence Example

Let's have a simple Python class representing an article in a blogging system:

```
from datetime import datetime
from elasticsearch_dsl import Document, Date, Integer, Keyword, Text
from elasticsearch_dsl.connections import connections

# Define a default Elasticsearch client
connections.create_connection(hosts=['localhost'])

class Article(Document):
    title = Text(analyzer='snowball', fields={'raw': Keyword()})
    body = Text(analyzer='snowball')
    tags = Keyword()
    published_from = Date()
    lines = Integer()

    class Index:
        name = 'blog'
        settings = {
          "number_of_shards": 2,
        }

    def save(self, ** kwargs):
        self.lines = len(self.body.split())
        return super(Article, self).save(** kwargs)

    def is_published(self):
        return datetime.now() >= self.published_from

# create the mappings in elasticsearch
Article.init()

# create and save and article
article = Article(meta={'id': 42}, title='Hello world!', tags=['test'])
```

(continues on next page)

(continued from previous page)

```
article.body = ''' looong text '''
article.published_from = datetime.now()
article.save()

article = Article.get(id=42)
print(article.is_published())

# Display cluster health
print(connections.get_connection().cluster.health())
```

In this example you can see:

- providing a *Default connection*
- defining fields with mapping configuration
- setting index name
- defining custom methods
- overriding the built-in `.save()` method to hook into the persistence life cycle
- retrieving and saving the object into Elasticsearch
- accessing the underlying client for other APIs

You can see more in the *Persistence* chapter.

# CHAPTER 5

## Pre-built Faceted Search

If you have your `Documents` defined you can very easily create a faceted search class to simplify searching and filtering.

**Note:** This feature is experimental and may be subject to change.

```
from elasticsearch_dsl import FacetedSearch, TermsFacet, DateHistogramFacet

class BlogSearch(FacetedSearch):
    doc_types = [Article, ]
    # fields that should be searched
    fields = ['tags', 'title', 'body']

    facets = {
        # use bucket aggregations to define facets
        'tags': TermsFacet(field='tags'),
        'publishing_frequency': DateHistogramFacet(field='published_from', interval=
↪'month')
    }

# empty search
bs = BlogSearch()
response = bs.execute()

for hit in response:
    print(hit.meta.score, hit.title)

for (tag, count, selected) in response.facets.tags:
    print(tag, ' (SELECTED):' if selected else ':', count)

for (month, count, selected) in response.facets.publishing_frequency:
    print(month.strftime('%B %Y'), ' (SELECTED):' if selected else ':', count)
```

You can find more details in the *Faceted Search* chapter.

CHAPTER 6

# Update By Query Example

Let's resume the simple example of articles on a blog, and let's assume that each article has a number of likes. For this example, imagine we want to increment the number of likes by 1 for all articles that match a certain tag and do not match a certain description. Writing this as a `dict`, we would have the following code:

```
from elasticsearch import Elasticsearch
client = Elasticsearch()

response = client.update_by_query(
    index="my-index",
    body={
      "query": {
        "bool": {
          "must": [{"match": {"tag": "python"}}],
          "must_not": [{"match": {"description": "beta"}}]
        }
      },
      "script"={
        "source": "ctx._source.likes++",
        "lang": "painless"
      }
    },
  )
```

Using the DSL, we can now express this query as such:

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search, UpdateByQuery

client = Elasticsearch()
ubq = UpdateByQuery(using=client, index="my-index") \
      .query("match", title="python")   \
      .exclude("match", description="beta") \
      .script(source="ctx._source.likes++", lang="painless")
```

(continues on next page)

(continued from previous page)

```
response = ubq.execute()
```

As you can see, the `Update By Query` object provides many of the savings offered by the `Search` object, and additionally allows one to update the results of the search based on a script assigned in the same manner.

# CHAPTER 7

## Migration from `elasticsearch-py`

You don't have to port your entire application to get the benefits of the Python DSL, you can start gradually by creating a `Search` object from your existing `dict`, modifying it using the API and serializing it back to a `dict`:

```
body = {...} # insert complicated query here

# Convert to Search object
s = Search.from_dict(body)

# Add some filters, aggregations, queries, ...
s.filter("term", tags="python")

# Convert back to dict to plug back into existing code
body = s.to_dict()
```

CHAPTER 8

# License

CHAPTER 9

# Contents

## 9.1 Configuration

There are several ways to configure connections for the library. The easiest and most useful approach is to define one default connection that can be used every time an API call is made without explicitly passing in other connections.

**Note:** Unless you want to access multiple clusters from your application, it is highly recommended that you use the `create_connection` method and all operations will use that connection automatically.

### 9.1.1 Default connection

To define a default connection that can be used globally, use the `connections` module and the `create_connection` method like this:

```
from elasticsearch_dsl import connections

connections.create_connection(hosts=['localhost'], timeout=20)
```

#### Single connection with an alias

You can define the `alias` or name of a connection so you can easily refer to it later. The default value for `alias` is `default`.

```
from elasticsearch_dsl import connections

connections.create_connection(alias='my_new_connection', hosts=['localhost'],␣
↪timeout=60)
```

Additional keyword arguments (`hosts` and `timeout` in our example) will be passed to the `Elasticsearch` class from `elasticsearch-py`.

To see all possible configuration options refer to the documentation.

### 9.1.2 Multiple clusters

You can define multiple connections to multiple clusters at the same time using the `configure` method:

```
from elasticsearch_dsl import connections

connections.configure(
    default={'hosts': 'localhost'},
    dev={
        'hosts': ['esdev1.example.com:9200'],
        'sniff_on_start': True
    }
)
```

Such connections will be constructed lazily when requested for the first time.

You can alternatively define multiple connections by adding them one by one as shown in the following example:

```
# if you have configuration options to be passed to Elasticsearch.__init__
# this also shows creating a connection with the alias 'qa'
connections.create_connection('qa', hosts=['esqa1.example.com'], sniff_on_start=True)

# if you already have an Elasticsearch instance ready
connections.add_connection('another_qa', my_client)
```

#### Using aliases

When using multiple connections, you can refer to them using the string alias specified when you created the connection.

This example shows how to use an alias to a connection:

```
s = Search(using='qa')
```

A `KeyError` will be raised if there is no connection registered with that alias.

### 9.1.3 Manual

If you don't want to supply a global configuration, you can always pass in your own connection as an instance of `elasticsearch.Elasticsearch` with the parameter `using` wherever it is accepted like this:

```
s = Search(using=Elasticsearch('localhost'))
```

You can even use this approach to override any connection the object might be already associated with:

```
s = s.using(Elasticsearch('otherhost:9200'))
```

**Note:** When using `elasticsearch_dsl`, it is highly recommended that you use the built-in serializer (`elasticsearch_dsl.serializer.serializer`) to ensure your objects are correctly serialized into `JSON` every time. The `create_connection` method that is described here (and that the `configure` method uses under the hood) will do that automatically for you, unless you explicitly specify your own serializer. The built-in serializer

also allows you to serialize your own objects - just define a `to_dict()` method on your objects and that method will be automatically called when serializing your custom objects to `JSON`.

## 9.2 Search DSL

### 9.2.1 The `Search` object

The `Search` object represents the entire search request:

- queries
- filters
- aggregations
- sort
- pagination
- additional parameters
- associated client

The API is designed to be chainable. With the exception of the aggregations functionality this means that the `Search` object is immutable - all changes to the object will result in a shallow copy being created which contains the changes. This means you can safely pass the `Search` object to foreign code without fear of it modifying your objects as long as it sticks to the `Search` object APIs.

You can pass an instance of the low-level elasticsearch client when instantiating the `Search` object:

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

s = Search(using=client)
```

You can also define the client at a later time (for more options see the *Configuration* chapter):

```
s = s.using(client)
```

**Note:** All methods return a *copy* of the object, making it safe to pass to outside code.

The API is chainable, allowing you to combine multiple method calls in one statement:

```
s = Search().using(client).query("match", title="python")
```

To send the request to Elasticsearch:

```
response = s.execute()
```

If you just want to iterate over the hits returned by your search you can iterate over the `Search` object:

```
for hit in s:
    print(hit.title)
```

Search results will be cached. Subsequent calls to `execute` or trying to iterate over an already executed `Search` object will not trigger additional requests being sent to Elasticsearch. To force a request specify `ignore_cache=True` when calling `execute`.

For debugging purposes you can serialize the `Search` object to a `dict` explicitly:

```
print(s.to_dict())
```

### Delete By Query

You can delete the documents matching a search by calling `delete` on the `Search` object instead of `execute` like this:

```
s = Search(index='i').query("match", title="python")
response = s.delete()
```

### Queries

The library provides classes for all Elasticsearch query types. Pass all the parameters as keyword arguments. The classes accept any keyword arguments, the dsl then takes all arguments passed to the constructor and serializes them as top-level keys in the resulting dictionary (and thus the resulting json being sent to elasticsearch). This means that there is a clear one-to-one mapping between the raw query and its equivalent in the DSL:

```
from elasticsearch_dsl.query import MultiMatch, Match

# {"multi_match": {"query": "python django", "fields": ["title", "body"]}}
MultiMatch(query='python django', fields=['title', 'body'])

# {"match": {"title": {"query": "web framework", "type": "phrase"}}}
Match(title={"query": "web framework", "type": "phrase"})
```

---

**Note:** In some cases this approach is not possible due to python's restriction on identifiers - for example if your field is called `@timestamp`. In that case you have to fall back to unpacking a dictionary: `Range(** {'@timestamp': {'lt': 'now'}})`

---

You can use the `Q` shortcut to construct the instance using a name with parameters or the raw `dict`:

```
from elasticsearch_dsl import Q

Q("multi_match", query='python django', fields=['title', 'body'])
Q({"multi_match": {"query": "python django", "fields": ["title", "body"]}})
```

To add the query to the `Search` object, use the `.query()` method:

```
q = Q("multi_match", query='python django', fields=['title', 'body'])
s = s.query(q)
```

The method also accepts all the parameters as the `Q` shortcut:

```
s = s.query("multi_match", query='python django', fields=['title', 'body'])
```

If you already have a query object, or a `dict` representing one, you can just override the query used in the `Search` object:

```
s.query = Q('bool', must=[Q('match', title='python'), Q('match', body='best')])
```

### Dotted fields

Sometimes you want to refer to a field within another field, either as a multi-field (`title.keyword`) or in a structured `json` document like `address.city`. To make it easier, the `Q` shortcut (as well as the `query`, `filter`, and `exclude` methods on `Search` class) allows you to use `__` (double underscore) in place of a dot in a keyword argument:

```
s = Search()
s = s.filter('term', category__keyword='Python')
s = s.query('match', address__city='prague')
```

Alternatively you can always fall back to python's kwarg unpacking if you prefer:

```
s = Search()
s = s.filter('term', **{'category.keyword': 'Python'})
s = s.query('match', **{'address.city': 'prague'})
```

### Query combination

Query objects can be combined using logical operators:

```
Q("match", title='python') | Q("match", title='django')
# {"bool": {"should": [...]}}

Q("match", title='python') & Q("match", title='django')
# {"bool": {"must": [...]}}

~Q("match", title="python")
# {"bool": {"must_not": [...]}}
```

When you call the `.query()` method multiple times, the `&` operator will be used internally:

```
s = s.query().query()
print(s.to_dict())
# {"query": {"bool": {...}}}
```

If you want to have precise control over the query form, use the `Q` shortcut to directly construct the combined query:

```
q = Q('bool',
    must=[Q('match', title='python')],
    should=[Q(...), Q(...)],
    minimum_should_match=1
)
s = Search().query(q)
```

### Filters

If you want to add a query in a filter context you can use the `filter()` method to make things easier:

```
s = Search()
s = s.filter('terms', tags=['search', 'python'])
```

Behind the scenes this will produce a `Bool` query and place the specified `terms` query into its `filter` branch, making it equivalent to:

```
s = Search()
s = s.query('bool', filter=[Q('terms', tags=['search', 'python'])])
```

If you want to use the post_filter element for faceted navigation, use the `.post_filter()` method.

You can also `exclude()` items from your query like this:

```
s = Search()
s = s.exclude('terms', tags=['search', 'python'])
```

which is shorthand for: `s = s.query('bool', filter=[~Q('terms', tags=['search', 'python'])])`

### Aggregations

To define an aggregation, you can use the `A` shortcut:

```
from elasticsearch_dsl import A

A('terms', field='tags')
# {"terms": {"field": "tags"}}
```

To nest aggregations, you can use the `.bucket()`, `.metric()` and `.pipeline()` methods:

```
a = A('terms', field='category')
# {'terms': {'field': 'category'}}

a.metric('clicks_per_category', 'sum', field='clicks')\
    .bucket('tags_per_category', 'terms', field='tags')
# {
#   'terms': {'field': 'category'},
#   'aggs': {
#     'clicks_per_category': {'sum': {'field': 'clicks'}},
#     'tags_per_category': {'terms': {'field': 'tags'}}
#   }
# }
```

To add aggregations to the `Search` object, use the `.aggs` property, which acts as a top-level aggregation:

```
s = Search()
a = A('terms', field='category')
s.aggs.bucket('category_terms', a)
# {
#   'aggs': {
#     'category_terms': {
#       'terms': {
#         'field': 'category'
#       }
#     }
#   }
# }
```

or

```
s = Search()
s.aggs.bucket('articles_per_day', 'date_histogram', field='publish_date', interval=
↪'day')\
    .metric('clicks_per_day', 'sum', field='clicks')\
    .pipeline('moving_click_average', 'moving_avg', buckets_path='clicks_per_day')\
    .bucket('tags_per_day', 'terms', field='tags')

s.to_dict()
# {
#   "aggs": {
#     "articles_per_day": {
#       "date_histogram": { "interval": "day", "field": "publish_date" },
#       "aggs": {
#         "clicks_per_day": { "sum": { "field": "clicks" } },
#         "moving_click_average": { "moving_avg": { "buckets_path": "clicks_per_day" }
↪ },
#         "tags_per_day": { "terms": { "field": "tags" } }
#       }
#     }
#   }
# }
```

You can access an existing bucket by its name:

```
s = Search()

s.aggs.bucket('per_category', 'terms', field='category')
s.aggs['per_category'].metric('clicks_per_category', 'sum', field='clicks')
s.aggs['per_category'].bucket('tags_per_category', 'terms', field='tags')
```

**Note:** When chaining multiple aggregations, there is a difference between what `.bucket()` and `.metric()` methods return - `.bucket()` returns the newly defined bucket while `.metric()` returns its parent bucket to allow further chaining.

As opposed to other methods on the `Search` objects, defining aggregations is done in-place (does not return a copy).

### Sorting

To specify sorting order, use the `.sort()` method:

```
s = Search().sort(
    'category',
    '-title',
    {"lines" : {"order" : "asc", "mode" : "avg"}}
)
```

It accepts positional arguments which can be either strings or dictionaries. String value is a field name, optionally prefixed by the − sign to specify a descending order.

To reset the sorting, just call the method with no arguments:

```
s = s.sort()
```

### Pagination

To specify the from/size parameters, use the Python slicing API:

```
s = s[10:20]
# {"from": 10, "size": 10}
```

If you want to access all the documents matched by your query you can use the `scan` method which uses the scan/scroll elasticsearch API:

```
for hit in s.scan():
    print(hit.title)
```

Note that in this case the results won't be sorted.

### Highlighting

To set common attributes for highlighting use the `highlight_options` method:

```
s = s.highlight_options(order='score')
```

Enabling highlighting for individual fields is done using the `highlight` method:

```
s = s.highlight('title')
# or, including parameters:
s = s.highlight('title', fragment_size=50)
```

The fragments in the response will then be available on each `Result` object as `.meta.highlight.FIELD` which will contain the list of fragments:

```
response = s.execute()
for hit in response:
    for fragment in hit.meta.highlight.title:
        print(fragment)
```

### Suggestions

To specify a suggest request on your `Search` object use the `suggest` method:

```
# check for correct spelling
s = s.suggest('my_suggestion', 'pyhton', term={'field': 'title'})
```

The first argument is the name of the suggestions (name under which it will be returned), second is the actual text you wish the suggester to work on and the keyword arguments will be added to the suggest's json as-is which means that it should be one of `term`, `phrase` or `completion` to indicate which type of suggester should be used.

### Extra properties and parameters

To set extra properties of the search request, use the `.extra()` method. This can be used to define keys in the body that cannot be defined via a specific API method like `explain` or `search_after`:

```
s = s.extra(explain=True)
```

To set query parameters, use the `.params()` method:

```
s = s.params(routing="42")
```

If you need to limit the fields being returned by elasticsearch, use the `source()` method:

```
# only return the selected fields
s = s.source(['title', 'body'])
# don't return any fields, just the metadata
s = s.source(False)
# explicitly include/exclude fields
s = s.source(includes=["title"], excludes=["user.*"])
# reset the field selection
s = s.source(None)
```

#### Serialization and Deserialization

The search object can be serialized into a dictionary by using the `.to_dict()` method.

You can also create a `Search` object from a `dict` using the `from_dict` class method. This will create a new `Search` object and populate it using the data from the dict:

```
s = Search.from_dict({"query": {"match": {"title": "python"}}})
```

If you wish to modify an existing `Search` object, overriding it's properties, instead use the `update_from_dict` method that alters an instance **in-place**:

```
s = Search(index='i')
s.update_from_dict({"query": {"match": {"title": "python"}}, "size": 42})
```

### 9.2.2 Response

You can execute your search by calling the `.execute()` method that will return a `Response` object. The `Response` object allows you access to any key from the response dictionary via attribute access. It also provides some convenient helpers:

```
response = s.execute()

print(response.success())
# True

print(response.took)
# 12

print(response.hits.total.relation)
# eq
print(response.hits.total.value)
# 142

print(response.suggest.my_suggestions)
```

If you want to inspect the contents of the `response` objects, just use its `to_dict` method to get access to the raw data for pretty printing.

#### Hits

To access to the hits returned by the search, access the `hits` property or just iterate over the `Response` object:

```
response = s.execute()
print('Total %d hits found.' % response.hits.total)
for h in response:
    print(h.title, h.body)
```

#### Result

The individual hits is wrapped in a convenience class that allows attribute access to the keys in the returned dictionary. All the metadata for the results are accessible via `meta` (without the leading _):

```
response = s.execute()
h = response.hits[0]
print('/%s/%s/%s returned with score %f' % (
    h.meta.index, h.meta.doc_type, h.meta.id, h.meta.score))
```

**Note:** If your document has a field called `meta` you have to access it using the get item syntax: `hit['meta']`.

#### Aggregations

Aggregations are available through the `aggregations` property:

```
for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

### 9.2.3 MultiSearch

If you need to execute multiple searches at the same time you can use the `MultiSearch` class which will use the `_msearch` API:

```
from elasticsearch_dsl import MultiSearch, Search

ms = MultiSearch(index='blogs')

ms = ms.add(Search().filter('term', tags='python'))
ms = ms.add(Search().filter('term', tags='elasticsearch'))

responses = ms.execute()

for response in responses:
    print("Results for query %r." % response.search.query)
    for hit in response:
        print(hit.title)
```

## 9.3 Persistence

You can use the dsl library to define your mappings and a basic persistent layer for your application.

For more comprehensive examples have a look at the examples directory in the repository.

### 9.3.1 Document

If you want to create a model-like wrapper around your documents, use the `Document` class. It can also be used to create all the necessary mappings and settings in elasticsearch (see *Document life cycle* for details).

```
from datetime import datetime
from elasticsearch_dsl import Document, Date, Nested, Boolean, \
    analyzer, InnerDoc, Completion, Keyword, Text

html_strip = analyzer('html_strip',
    tokenizer="standard",
    filter=["standard", "lowercase", "stop", "snowball"],
    char_filter=["html_strip"]
)

class Comment(InnerDoc):
    author = Text(fields={'raw': Keyword()})
    content = Text(analyzer='snowball')
    created_at = Date()

    def age(self):
        return datetime.now() - self.created_at

class Post(Document):
    title = Text()
    title_suggest = Completion()
    created_at = Date()
    published = Boolean()
    category = Text(
        analyzer=html_strip,
        fields={'raw': Keyword()}
    )

    comments = Nested(Comment)

    class Index:
        name = 'blog'

    def add_comment(self, author, content):
        self.comments.append(
          Comment(author=author, content=content, created_at=datetime.now()))

    def save(self, ** kwargs):
        self.created_at = datetime.now()
        return super().save(** kwargs)
```

#### Data types

The `Document` instances should be using native python types like `datetime`. In case of `Object` or `Nested` fields an instance of the `InnerDoc` subclass should be used just like in the `add_comment` method in the above example where we are creating an instance of the `Comment` class.

There are some specific types that were created as part of this library to make working with specific field types easier, for example the `Range` object used in any of the range fields:

```
from elasticsearch_dsl import Document, DateRange, Keyword, Range

class RoomBooking(Document):
    room = Keyword()
    dates = DateRange()


rb = RoomBooking(
  room='Conference Room II',
  dates=Range(
    gte=datetime(2018, 11, 17, 9, 0, 0),
    lt=datetime(2018, 11, 17, 10, 0, 0)
  )
)

# Range supports the in operator correctly:
datetime(2018, 11, 17, 9, 30, 0) in rb.dates # True

# you can also get the limits and whether they are inclusive or exclusive:
rb.dates.lower # datetime(2018, 11, 17, 9, 0, 0), True
rb.dates.upper # datetime(2018, 11, 17, 10, 0, 0), False

# empty range is unbounded
Range().lower # None, False
```

### Note on dates

`elasticsearch-dsl` will always respect the timezone information (or lack thereof) on the `datetime` objects passed in or stored in Elasticsearch. Elasticsearch itself interprets all datetimes with no timezone information as `UTC`. If you wish to reflect this in your python code, you can specify `default_timezone` when instantiating a `Date` field:

```
class Post(Document):
    created_at = Date(default_timezone='UTC')
```

In that case any `datetime` object passed in (or parsed from elasticsearch) will be treated as if it were in `UTC` timezone.

### Document life cycle

Before you first use the `Post` document type, you need to create the mappings in Elasticsearch. For that you can either use the *Index* object or create the mappings directly by calling the `init` class method:

```
# create the mappings in Elasticsearch
Post.init()
```

This code will typically be run in the setup for your application during a code deploy, similar to running database migrations.

To create a new `Post` document just instantiate the class and pass in any fields you wish to set, you can then use standard attribute setting to change/add more fields. Note that you are not limited to the fields defined explicitly:

```
# instantiate the document
first = Post(title='My First Blog Post, yay!', published=True)
```

(continues on next page)

(continued from previous page)

```
# assign some field values, can be values or lists of values
first.category = ['everything', 'nothing']
# every document has an id in meta
first.meta.id = 47


# save the document into the cluster
first.save()
```

All the metadata fields (`id`, `routing`, `index` etc) can be accessed (and set) via a `meta` attribute or directly using the underscored variant:

```
post = Post(meta={'id': 42})

# prints 42
print(post.meta.id)

# override default index
post.meta.index = 'my-blog'
```

**Note:** Having all metadata accessible through `meta` means that this name is reserved and you shouldn't have a field called `meta` on your document. If you, however, need it you can still access the data using the get item (as opposed to attribute) syntax: `post['meta']`.

To retrieve an existing document use the `get` class method:

```
# retrieve the document
first = Post.get(id=42)
# now we can call methods, change fields, ...
first.add_comment('me', 'This is nice!')
# and save the changes into the cluster again
first.save()
```

The Update API can also be used via the `update` method. By default any keyword arguments, beyond the parameters of the API, will be considered fields with new values. Those fields will be updated on the local copy of the document and then sent over as partial document to be updated:

```
# retrieve the document
first = Post.get(id=42)
# you can update just individual fields which will call the update API
# and also update the document in place
first.update(published=True, published_by='me')
```

In case you wish to use a `painless` script to perform the update you can pass in the script string as `script` or the `id` of a stored script via `script_id`. All additional keyword arguments to the `update` method will then be passed in as parameters of the script. The document will not be updated in place.

```
# retrieve the document
first = Post.get(id=42)
# we execute a script in elasticsearch with additional kwargs being passed
# as params into the script
first.update(script='ctx._source.category.add(params.new_category)',
             new_category='testing')
```

If the document is not found in elasticsearch an exception (`elasticsearch.NotFoundError`) will be raised. If you wish to return `None` instead just pass in `ignore=404` to suppress the exception:

```
p = Post.get(id='not-in-es', ignore=404)
p is None
```

When you wish to retrive multiple documents at the same time by their `id` you can use the `mget` method:

```
posts = Post.mget([42, 47, 256])
```

`mget` will, by default, raise a `NotFoundError` if any of the documents wasn't found and `RequestError` if any of the document had resulted in error. You can control this behavior by setting parameters:

**`raise_on_error`** If `True` (default) then any error will cause an exception to be raised. Otherwise all documents containing errors will be treated as missing.

**`missing`** Can have three possible values: `'none'` (default), `'raise'` and `'skip'`. If a document is missing or errored it will either be replaced with `None`, an exception will be raised or the document will be skipped in the output list entirely.

The index associated with the `Document` is accessible via the `_index` class property which gives you access to the *Index* class.

The `_index` attribute is also home to the `load_mappings` method which will update the mapping on the `Index` from elasticsearch. This is very useful if you use dynamic mappings and want the class to be aware of those fields (for example if you wish the `Date` fields to be properly (de)serialized):

```
Post._index.load_mappings()
```

To delete a document just call its `delete` method:

```
first = Post.get(id=42)
first.delete()
```

### Analysis

To specify `analyzer` values for `Text` fields you can just use the name of the analyzer (as a string) and either rely on the analyzer being defined (like built-in analyzers) or define the analyzer yourself manually.

Alternatively you can create your own analyzer and have the persistence layer handle its creation, from our example earlier:

```
from elasticsearch_dsl import analyzer, tokenizer

my_analyzer = analyzer('my_analyzer',
    tokenizer=tokenizer('trigram', 'nGram', min_gram=3, max_gram=3),
    filter=['lowercase']
)
```

Each analysis object needs to have a name (`my_analyzer` and `trigram` in our example) and tokenizers, token filters and char filters also need to specify type (`nGram` in our example).

Once you have an instance of a custom `analyzer` you can also call the analyze API on it by using the `simulate` method:

```
response = my_analyzer.simulate('Hello World!')
```

(continues on next page)

(continued from previous page)

```
# ['hel', 'ell', 'llo', 'lo ', 'o w', ' wo', 'wor', 'orl', 'rld', 'ld!']
tokens = [t.token for t in response.tokens]
```

**Note:** When creating a mapping which relies on a custom analyzer the index must either not exist or be closed. To create multiple `Document`-defined mappings you can use the *Index* object.

### Search

To search for this document type, use the `search` class method:

```
# by calling .search we get back a standard Search object
s = Post.search()
# the search is already limited to the index and doc_type of our document
s = s.filter('term', published=True).query('match', title='first')


results = s.execute()

# when you execute the search the results are wrapped in your document class (Post)
for post in results:
    print(post.meta.score, post.title)
```

Alternatively you can just take a `Search` object and restrict it to return our document type, wrapped in correct class:

```
s = Search()
s = s.doc_type(Post)
```

You can also combine document classes with standard doc types (just strings), which will be treated as before. You can also pass in multiple `Document` subclasses and each document in the response will be wrapped in it's class.

If you want to run suggestions, just use the `suggest` method on the `Search` object:

```
s = Post.search()
s = s.suggest('title_suggestions', 'pyth', completion={'field': 'title_suggest'})

response = s.execute()

for result in response.suggest.title_suggestions:
    print('Suggestions for %s:' % result.text)
    for option in result.options:
        print('  %s (%r)' % (option.text, option.payload))
```

### `class Meta` options

In the `Meta` class inside your document definition you can define various metadata for your document:

**`mapping`** optional instance of `Mapping` class to use as base for the mappings created from the fields on the document class itself.

Any attributes on the `Meta` class that are instance of `MetaField` will be used to control the mapping of the meta fields (`_all`, `dynamic` etc). Just name the parameter (without the leading underscore) as the field you wish to map and pass any parameters to the `MetaField` class:

```
class Post(Document):
    title = Text()

    class Meta:
        all = MetaField(enabled=False)
        dynamic = MetaField('strict')
```

#### class Index options

This section of the `Document` definition can contain any information about the index, its name, settings and other attributes:

**name** name of the index to use, if it contains a wildcard (`*`) then it cannot be used for any write operations and an `index` kwarg will have to be passed explicitly when calling methods like `.save()`.

**using** default connection alias to use, defaults to `'default'`

**settings** dictionary containing any settings for the `Index` object like `number_of_shards`.

**analyzers** additional list of analyzers that should be defined on an index (see *Analysis* for details).

**aliases** dictionary with any aliases definitions

### Document Inheritance

You can use standard Python inheritance to extend models, this can be useful in a few scenarios. For example if you want to have a `BaseDocument` defining some common fields that several different `Document` classes should share:

```
class User(InnerDoc):
    username = Text(fields={'keyword': Keyword()})
    email = Text()

class BaseDocument(Document):
    created_by = Object(User)
    created_date = Date()
    last_updated = Date()

    def save(**kwargs):
        if not self.created_date:
            self.created_date = datetime.now()
        self.last_updated = datetime.now()
        return super(BaseDocument, self).save(**kwargs)

class BlogPost(BaseDocument):
    class Index:
        name = 'blog'
```

Another use case would be using the join type to have multiple different entities in a single index. You can see an example of this approach. Note that in this case, if the subclasses don't define their own *Index* classes, the mappings are merged and shared between all the subclasses.

## 9.3.2 Index

In typical scenario using `class Index` on a `Document` class is sufficient to perform any action. In a few cases though it can be useful to manipulate an `Index` object directly.

`Index` is a class responsible for holding all the metadata related to an index in elasticsearch - mappings and settings. It is most useful when defining your mappings since it allows for easy creation of multiple mappings at the same time. This is especially useful when setting up your elasticsearch objects in a migration:

```
from elasticsearch_dsl import Index, Document, Text, analyzer

blogs = Index('blogs')

# define custom settings
blogs.settings(
    number_of_shards=1,
    number_of_replicas=0
)

# define aliases
blogs.aliases(
    old_blogs={}
)

# register a document with the index
blogs.document(Post)

# can also be used as class decorator when defining the Document
@blogs.document
class Post(Document):
    title = Text()

# You can attach custom analyzers to the index

html_strip = analyzer('html_strip',
    tokenizer="standard",
    filter=["standard", "lowercase", "stop", "snowball"],
    char_filter=["html_strip"]
)

blogs.analyzer(html_strip)

# delete the index, ignore if it doesn't exist
blogs.delete(ignore=404)

# create the index in elasticsearch
blogs.create()
```

You can also set up a template for your indices and use the `clone` method to create specific copies:

```
blogs = Index('blogs', using='production')
blogs.settings(number_of_shards=2)
blogs.document(Post)

# create a copy of the index with different name
company_blogs = blogs.clone('company-blogs')

# create a different copy on different cluster
dev_blogs = blogs.clone('blogs', using='dev')
# and change its settings
dev_blogs.setting(number_of_shards=1)
```

#### IndexTemplate

`elasticsearch-dsl` also exposes an option to manage index templates in elasticsearch using the `IndexTemplate` class which has very similar API to `Index`.

Once an index template is saved in elasticsearch it's contents will be automatically applied to new indices (existing indices are completely unaffected by templates) that match the template pattern (any index starting with `blogs-` in our example), even if the index is created automatically upon indexing a document into that index.

Potential workflow for a set of time based indices governed by a single template:

```
from datetime import datetime

from elasticsearch_dsl import Document, Date, Text


class Log(Document):
    content = Text()
    timestamp = Date()

    class Index:
        name = "logs-*"
        settings = {
          "number_of_shards": 2
        }

    def save(self, **kwargs):
        # assign now if no timestamp given
        if not self.timestamp:
            self.timestamp = datetime.now()

        # override the index to go to the proper timeslot
        kwargs['index'] = self.timestamp.strftime('logs-%Y%m%d')
        return super().save(**kwargs)

# once, as part of application setup, during deploy/migrations:
logs = Log._index.as_template('logs', order=0)
logs.save()

# to perform search across all logs:
search = Log.search()
```

## 9.4 Faceted Search

The library comes with a simple abstraction aimed at helping you develop faceted navigation for your data.

**Note:** This API is experimental and will be subject to change. Any feedback is welcome.

### 9.4.1 Configuration

You can provide several configuration options (as class attributes) when declaring a `FacetedSearch` subclass:

**`index`** the name of the index (as string) to search through, defaults to `'_all'`.

**doc_types** list of `Document` subclasses or strings to be used, defaults to `['_all']`.

**fields** list of fields on the document type to search through. The list will be passes to `MultiMatch` query so can contain boost values (`'title^5'`), defaults to `['*']`.

**facets** dictionary of facets to display/filter on. The key is the name displayed and values should be instances of any `Facet` subclass, for example: `{'tags': TermsFacet(field='tags')}`

**sort** tuple or list of fields on which the results should be sorted. The format of the individual fields are to be the same as those passed to *sort()*.

#### Facets

There are several different facets available:

**TermsFacet** provides an option to split documents into groups based on a value of a field, for example `TermsFacet(field='category')`

**DateHistogramFacet** split documents into time intervals, example: `DateHistogramFacet(field="published_date", interval="day")`

**HistogramFacet** similar to `DateHistogramFacet` but for numerical values: `HistogramFacet(field="rating", interval=2)`

**RangeFacet** allows you to define your own ranges for a numerical fields: `RangeFacet(field="comment_count", ranges=[("few", (None, 2)), ("lots", (2, None))])`

**NestedFacet** is just a simple facet that wraps another to provide access to nested documents: `NestedFacet('variants', TermsFacet(field='variants.color'))`

By default facet results will only calculate document count, if you wish for a different metric you can pass in any single value metric aggregation as the `metric` kwarg (`TermsFacet(field='tags', metric=A('max', field=timestamp))`). When specifying `metric` the results will be, by default, sorted in descending order by that metric. To change it to ascending specify `metric_sort="asc"` and to just sort by document count use `metric_sort=False`.

#### Advanced

If you require any custom behavior or modifications simply override one or more of the methods responsible for the class' functions:

**search(self)** is responsible for constructing the `Search` object used. Override this if you want to customize the search object (for example by adding a global filter for published articles only).

**query(self, search)** adds the query postion of the search (if search input specified), by default using `MultiField` query. Override this if you want to modify the query type used.

**highlight(self, search)** defines the highlighting on the `Search` object and returns a new one. Default behavior is to highlight on all fields specified for search.

### 9.4.2 Usage

The custom subclass can be instantiated empty to provide an empty search (matching everything) or with `query` and `filters`.

**query** is used to pass in the text of the query to be performed. If `None` is passed in (default) a `MatchAll` query will be used. For example `'python web'`

**filters** is a dictionary containing all the facet filters that you wish to apply. Use the name of the facet (from `.facets` attribute) as the key and one of the possible values as value. For example `{'tags': 'python'}`.

#### Response

the response returned from the `FacetedSearch` object (by calling `.execute()`) is a subclass of the standard `Response` class that adds a property called `facets` which contains a dictionary with lists of buckets - each represented by a tuple of key, document count and a flag indicating whether this value has been filtered on.

### 9.4.3 Example

```
from datetime import date

from elasticsearch_dsl import FacetedSearch, TermsFacet, DateHistogramFacet

class BlogSearch(FacetedSearch):
    doc_types = [Article, ]
    # fields that should be searched
    fields = ['tags', 'title', 'body']

    facets = {
        # use bucket aggregations to define facets
        'tags': TermsFacet(field='tags'),
        'publishing_frequency': DateHistogramFacet(field='published_from', interval=
↪'month')
    }

    def search(self):
        # override methods to add custom pieces
        s = super().search()
        return s.filter('range', publish_from={'lte': 'now/h'})

bs = BlogSearch('python web', {'publishing_frequency': date(2015, 6)})
response = bs.execute()

# access hits and other attributes as usual
total = response.hits.total
print('total hits', total.relation, total.value)
for hit in response:
    print(hit.meta.score, hit.title)

for (tag, count, selected) in response.facets.tags:
    print(tag, ' (SELECTED):' if selected else ':', count)

for (month, count, selected) in response.facets.publishing_frequency:
    print(month.strftime('%B %Y'), ' (SELECTED):' if selected else ':', count)
```

## 9.5 Update By Query

### 9.5.1 The `Update By Query` object

The `Update By Query` object enables the use of the _update_by_query endpoint to perform an update on documents that match a search query.

The object is implemented as a modification of the `Search` object, containing a subset of its query methods, as well as a script method, which is used to make updates.

The `Update By Query` object implements the following `Search` query types:

- queries
- filters
- excludes

For more information on queries, see the *Search DSL* chapter.

Like the `Search` object, the API is designed to be chainable. This means that the `Update By Query` object is immutable: all changes to the object will result in a shallow copy being created which contains the changes. This means you can safely pass the `Update By Query` object to foreign code without fear of it modifying your objects as long as it sticks to the `Update By Query` object APIs.

You can define your client in a number of ways, but the preferred method is to use a global configuration. For more information on defining a client, see the *Configuration* chapter.

Once your client is defined, you can instantiate a copy of the `Update By Query` object as seen below:

```
from elasticsearch_dsl import UpdateByQuery

ubq = UpdateByQuery().using(client)
# or
ubq = UpdateByQuery(using=client)
```

**Note:** All methods return a *copy* of the object, making it safe to pass to outside code.

The API is chainable, allowing you to combine multiple method calls in one statement:

```
ubq = UpdateByQuery().using(client).query("match", title="python")
```

To send the request to Elasticsearch:

```
response = ubq.execute()
```

It should be noted, that there are limits to the chaining using the script method: calling script multiple times will overwrite the previous value. That is, only a single script can be sent with a call. An attempt to use two scripts will result in only the second script being stored.

Given the below example:

```
ubq = UpdateByQuery().using(client).script(source="ctx._source.likes++").
↪script(source="ctx._source.likes+=2")
```

This means that the stored script by this client will be `'source': 'ctx._source.likes+=2'` and the previous call will not be stored.

For debugging purposes you can serialize the `Update By Query` object to a `dict` explicitly:

```
print(ubq.to_dict())
```

Also, to use variables in script see below example:

```
ubq.script(
  source="ctx._source.messages.removeIf(x -> x.somefield == params.some_var)",
  params={
    'some_var': 'some_string_val'
  }
)
```

#### Serialization and Deserialization

The search object can be serialized into a dictionary by using the `.to_dict()` method.

You can also create a `Update By Query` object from a `dict` using the `from_dict` class method. This will create a new `Update By Query` object and populate it using the data from the dict:

```
ubq = UpdateByQuery.from_dict({"query": {"match": {"title": "python"}}})
```

If you wish to modify an existing `Update By Query` object, overriding it's properties, instead use the `update_from_dict` method that alters an instance **in-place**:

```
ubq = UpdateByQuery(index='i')
ubq.update_from_dict({"query": {"match": {"title": "python"}}, "size": 42})
```

#### Extra properties and parameters

To set extra properties of the search request, use the `.extra()` method. This can be used to define keys in the body that cannot be defined via a specific API method like `explain`:

```
ubq = ubq.extra(explain=True)
```

To set query parameters, use the `.params()` method:

```
ubq = ubq.params(routing="42")
```

### 9.5.2 Response

You can execute your search by calling the `.execute()` method that will return a `Response` object. The `Response` object allows you access to any key from the response dictionary via attribute access. It also provides some convenient helpers:

```
response = ubq.execute()

print(response.success())
# True

print(response.took)
# 12
```

If you want to inspect the contents of the `response` objects, just use its `to_dict` method to get access to the raw data for pretty printing.

## 9.6 API Documentation

Below please find the documentation for the public classes and functions of `elasticsearch_dsl`.

### 9.6.1 Search

**class** elasticsearch_dsl.**Search**(***kwargs*)

Search request to elasticsearch.

**Parameters**

- **using** – *Elasticsearch* instance to use
- **index** – limit the search to index
- **doc_type** – only query this type.

All the parameters supplied (or omitted) at creation type can be later overridden by methods (*using*, *index* and *doc_type* respectively).

**count**()

Return the number of hits matching the query and filters. Note that only the actual number is returned.

**delete**() *executes the query by delegating to delete_by_query()*

**execute**(*ignore_cache=False*)

Execute the search and return an instance of `Response` wrapping all the data.

**Parameters** **ignore_cache** – if set to `True`, consecutive calls will hit ES, while cached result will be ignored. Defaults to *False*

**classmethod from_dict**(*d*)

Construct a new *Search* instance from a raw dict containing the search body. Useful when migrating from raw dictionaries.

Example:

```
s = Search.from_dict({
    "query": {
        "bool": {
            "must": [...]
        }
    },
    "aggs": {...}
})
s = s.filter('term', published=True)
```

**highlight**(**fields*, ***kwargs*)

Request highlighting of some fields. All keyword arguments passed in will be used as parameters for all the fields in the `fields` parameter. Example:

```
Search().highlight('title', 'body', fragment_size=50)
```

will produce the equivalent of:

```
{
    "highlight": {
        "fields": {
            "body": {"fragment_size": 50},
```

(continues on next page)

(continued from previous page)

```
            "title": {"fragment_size": 50}
        }
    }
}
```

If you want to have different options for different fields you can call `highlight` twice:

```
Search().highlight('title', fragment_size=50).highlight('body', fragment_
↪size=100)
```

which will produce:

```
{
    "highlight": {
        "fields": {
            "body": {"fragment_size": 100},
            "title": {"fragment_size": 50}
        }
    }
}
```

**highlight_options**(***kwargs*)
Update the global highlighting options used for this request. For example:

```
s = Search()
s = s.highlight_options(order='score')
```

**response_class**(*cls*)
Override the default wrapper used for the response.

**scan**()
Turn the search into a scan search and return a generator that will iterate over all the documents matching the query.

Use `params` method to specify any additional arguments you with to pass to the underlying `scan` helper from `elasticsearch-py` - https://elasticsearch-py.readthedocs.io/en/master/helpers.html#elasticsearch.helpers.scan

**script_fields**(***kwargs*)
Define script fields to be calculated on hits. See https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-script-fields.html for more details.

Example:

```
s = Search()
s = s.script_fields(times_two="doc['field'].value * 2")
s = s.script_fields(
    times_three={
        'script': {
            'inline': "doc['field'].value * params.n",
            'params': {'n': 3}
        }
    }
)
```

**sort**(**keys*)
Add sorting information to the search request. If called without arguments it will remove all sort requirements. Otherwise it will replace them. Acceptable arguments are:

```
'some.field'
'-some.other.field'
{'different.field': {'any': 'dict'}}
```

so for example:

```
s = Search().sort(
    'category',
    '-title',
    {"price" : {"order" : "asc", "mode" : "avg"}}
)
```

will sort by `category`, `title` (in descending order) and `price` in ascending order using the `avg` mode.

The API returns a copy of the Search object and can thus be chained.

**source** (*fields=None*, ***kwargs*)

Selectively control how the _source field is returned.

**Parameters** **fields** – wildcard string, array of wildcards, or dictionary of includes and excludes

If `fields` is None, the entire document will be returned for each hit. If fields is a dictionary with keys of 'includes' and/or 'excludes' the fields will be either included or excluded appropriately.

Calling this multiple times with the same named parameter will override the previous values with the new ones.

Example:

```
s = Search()
s = s.source(includes=['obj1.*'], excludes=["*.description"])

s = Search()
s = s.source(includes=['obj1.*']).source(excludes=["*.description"])
```

**suggest** (*name*, *text*, ***kwargs*)

Add a suggestions request to the search.

**Parameters**

- **name** – name of the suggestion
- **text** – text to suggest on

All keyword arguments will be added to the suggestions body. For example:

```
s = Search()
s = s.suggest('suggestion-1', 'Elasticsearch', term={'field': 'body'})
```

**to_dict** (*count=False*, ***kwargs*)

Serialize the search into the dictionary that will be sent over as the request's body.

**Parameters** **count** – a flag to specify if we are interested in a body for count - no aggregations, no pagination bounds etc.

All additional keyword arguments will be included into the dictionary.

**update_from_dict** (*d*)

Apply options from a serialized body to the current instance. Modifies the object in-place. Used mostly by `from_dict`.

**class** elasticsearch_dsl.**MultiSearch**(***kwargs*)
Combine multiple *Search* objects into a single request.

**add**(*search*)
Adds a new *Search* object to the request:

```
ms = MultiSearch(index='my-index')
ms = ms.add(Search(doc_type=Category).filter('term', category='python'))
ms = ms.add(Search(doc_type=Blog))
```

**execute**(*ignore_cache=False*, *raise_on_error=True*)
Execute the multi search request and return a list of search results.

### 9.6.2 Document

**class** elasticsearch_dsl.**Document**(*meta=None*, ***kwargs*)
Model-like class for persisting documents in elasticsearch.

**delete**(*using=None*, *index=None*, ***kwargs*)
Delete the instance in elasticsearch.

**Parameters**

- **index** – elasticsearch index to use, if the Document is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'

Any additional keyword arguments will be passed to Elasticsearch.delete unchanged.

**classmethod get**(*id*, *using=None*, *index=None*, ***kwargs*)
Retrieve a single document from elasticsearch using its id.

**Parameters**

- **id** – id of the document to be retrieved
- **index** – elasticsearch index to use, if the Document is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'

Any additional keyword arguments will be passed to Elasticsearch.get unchanged.

**classmethod init**(*index=None*, *using=None*)
Create the index and populate the mappings in elasticsearch.

**classmethod mget**(*docs*, *using=None*, *index=None*, *raise_on_error=True*, *missing='none'*, ***kwargs*)
Retrieve multiple document by their ids. Returns a list of instances in the same order as requested.

**Parameters**

- **docs** – list of ids of the documents to be retrieved or a list of document specifications as per https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-multi-get.html
- **index** – elasticsearch index to use, if the Document is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'

- **missing** – what to do when one of the documents requested is not found. Valid options are `'none'` (use `None`), `'raise'` (raise `NotFoundError`) or `'skip'` (ignore the missing document).

Any additional keyword arguments will be passed to `Elasticsearch.mget` unchanged.

**save**(*using=None*, *index=None*, *validate=True*, *skip_empty=True*, ***kwargs*)
Save the document into elasticsearch. If the document doesn't exist it is created, it is overwritten otherwise. Returns `True` if this operations resulted in new document being created.

**Parameters**

- **index** – elasticsearch index to use, if the `Document` is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to `'default'`
- **validate** – set to `False` to skip validating the document
- **skip_empty** – if set to `False` will cause empty values (`None`, `[]`, `{}`) to be left on the document. Those values will be stripped out otherwise as they make no difference in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.index` unchanged.

:return operation result created/updated

**classmethod search**(*using=None*, *index=None*)
Create an *Search* instance that will search over this `Document`.

**to_dict**(*include_meta=False*, *skip_empty=True*)
Serialize the instance into a dictionary so that it can be saved in elasticsearch.

**Parameters**

- **include_meta** – if set to `True` will include all the metadata (`_index`, `_id` etc). Otherwise just the document's data is serialized. This is useful when passing multiple instances into `elasticsearch.helpers.bulk`.
- **skip_empty** – if set to `False` will cause empty values (`None`, `[]`, `{}`) to be left on the document. Those values will be stripped out otherwise as they make no difference in elasticsearch.

**update**(*using=None*, *index=None*, *detect_noop=True*, *doc_as_upsert=False*, *refresh=False*, *retry_on_conflict=None*, *script=None*, *script_id=None*, *scripted_upsert=False*, *upsert=None*, ***fields*)
Partial update of the document, specify fields you wish to update and both the instance and the document in elasticsearch will be updated:

```
doc = MyDocument(title='Document Title!')
doc.save()
doc.update(title='New Document Title!')
```

**Parameters**

- **index** – elasticsearch index to use, if the `Document` is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to `'default'`
- **detect_noop** – Set to `False` to disable noop detection.
- **refresh** – Control when the changes made by this request are visible to search. Set to `True` for immediate effect.

- **retry_on_conflict** – In between the get and indexing phases of the update, it is possible that another process might have already updated the same document. By default, the update will fail with a version conflict exception. The retry_on_conflict parameter controls how many times to retry the update before finally throwing an exception.
- **doc_as_upsert** – Instead of sending a partial doc plus an upsert doc, setting doc_as_upsert to true will use the contents of doc as the upsert value

:return operation result noop/updated

### 9.6.3 Index

**class** elasticsearch_dsl.**Index**(*name*, *using='default'*)

**Parameters**

- **name** – name of the index
- **using** – connection alias to use, defaults to `'default'`

**aliases**(***kwargs*)

Add aliases to the index definition:

```
i = Index('blog-v2')
i.aliases(blog={}, published={'filter': Q('term', published=True)})
```

**analyze**(*using=None*, ***kwargs*)

Perform the analysis process on a text and return the tokens breakdown of the text.

Any additional keyword arguments will be passed to `Elasticsearch.indices.analyze` unchanged.

**analyzer**(**args*, ***kwargs*)

Explicitly add an analyzer to an index. Note that all custom analyzers defined in mappings will also be created. This is useful for search analyzers.

Example:

```
from elasticsearch_dsl import analyzer, tokenizer

my_analyzer = analyzer('my_analyzer',
    tokenizer=tokenizer('trigram', 'nGram', min_gram=3, max_gram=3),
    filter=['lowercase']
)

i = Index('blog')
i.analyzer(my_analyzer)
```

**clear_cache**(*using=None*, ***kwargs*)

Clear all caches or specific cached associated with the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.clear_cache` unchanged.

**clone**(*name=None*, *using=None*)

Create a copy of the instance with another name or connection alias. Useful for creating multiple indices with shared configuration:

```
i = Index('base-index')
i.settings(number_of_shards=1)
i.create()

i2 = i.clone('other-index')
i2.create()
```

**Parameters**

- **name** – name of the index
- **using** – connection alias to use, defaults to `'default'`

**close**(*using=None*, ***kwargs*)
Closes the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.close` unchanged.

**create**(*using=None*, ***kwargs*)
Creates the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.create` unchanged.

**delete**(*using=None*, ***kwargs*)
Deletes the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.delete` unchanged.

**delete_alias**(*using=None*, ***kwargs*)
Delete specific alias.

Any additional keyword arguments will be passed to `Elasticsearch.indices.delete_alias` unchanged.

**document**(*document*)
Associate a *Document* subclass with an index. This means that, when this index is created, it will contain the mappings for the `Document`. If the `Document` class doesn't have a default index yet (by defining `class Index`), this instance will be used. Can be used as a decorator:

```
i = Index('blog')

@i.document
class Post(Document):
    title = Text()

# create the index, including Post mappings
i.create()

# .search() will now return a Search object that will return
# properly deserialized Post instances
s = i.search()
```

**exists**(*using=None*, ***kwargs*)
Returns `True` if the index already exists in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists` unchanged.

**exists_alias**(*using=None*, ***kwargs*)
Return a boolean indicating whether given alias exists for this index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists_alias` unchanged.

**exists_type**(*using=None*, ***kwargs*)
Check if a type/types exists in the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists_type` unchanged.

**flush**(*using=None*, ***kwargs*)
Performs a flush operation on the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.flush` unchanged.

**flush_synced**(*using=None*, ***kwargs*)
Perform a normal flush, then add a generated unique marker (sync_id) to all shards.

Any additional keyword arguments will be passed to `Elasticsearch.indices.flush_synced` unchanged.

**forcemerge**(*using=None*, ***kwargs*)
The force merge API allows to force merging of the index through an API. The merge relates to the number of segments a Lucene index holds within each shard. The force merge operation allows to reduce the number of segments by merging them.

This call will block until the merge is complete. If the http connection is lost, the request will continue in the background, and any new requests will block until the previous force merge is complete.

Any additional keyword arguments will be passed to `Elasticsearch.indices.forcemerge` unchanged.

**get**(*using=None*, ***kwargs*)
The get index API allows to retrieve information about the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get` unchanged.

**get_alias**(*using=None*, ***kwargs*)
Retrieve a specified alias.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_alias` unchanged.

**get_field_mapping**(*using=None*, ***kwargs*)
Retrieve mapping definition of a specific field.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_field_mapping` unchanged.

**get_mapping**(*using=None*, ***kwargs*)
Retrieve specific mapping definition for a specific type.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_mapping` unchanged.

**get_settings**(*using=None*, ***kwargs*)
Retrieve settings for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_settings` unchanged.

**get_upgrade**(*using=None*, ***kwargs*)
Monitor how much of the index is upgraded.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_upgrade` unchanged.

**mapping**(*mapping*)
Associate a mapping (an instance of `Mapping`) with this index. This means that, when this index is created, it will contain the mappings for the document type defined by those mappings.

**open**(*using=None*, ***kwargs*)
Opens the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.open` unchanged.

**put_alias**(*using=None*, ***kwargs*)
Create an alias for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_alias` unchanged.

**put_mapping**(*using=None*, ***kwargs*)
Register specific mapping definition for a specific type.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_mapping` unchanged.

**put_settings**(*using=None*, ***kwargs*)
Change specific index level settings in real time.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_settings` unchanged.

**recovery**(*using=None*, ***kwargs*)
The indices recovery API provides insight into on-going shard recoveries for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.recovery` unchanged.

**refresh**(*using=None*, ***kwargs*)
Performs a refresh operation on the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.refresh` unchanged.

**save**(*using=None*)
Sync the index definition with elasticsearch, creating the index if it doesn't exist and updating its settings and mappings if it does.

Note some settings and mapping changes cannot be done on an open index (or at all on an existing index) and for those this method will fail with the underlying exception.

**search**(*using=None*)
Return a *Search* object searching over the index (or all the indices belonging to this template) and its `Document`s.

**segments**(*using=None*, ***kwargs*)
Provide low level segments information that a Lucene index (shard level) is built with.

Any additional keyword arguments will be passed to `Elasticsearch.indices.segments` unchanged.

**settings**(***kwargs*)
Add settings to the index:

```
i = Index('i')
i.settings(number_of_shards=1, number_of_replicas=0)
```

Multiple calls to `settings` will merge the keys, later overriding the earlier.

**shard_stores** (*using=None*, ***kwargs*)
Provides store information for shard copies of the index. Store information reports on which nodes shard copies exist, the shard copy version, indicating how recent they are, and any exceptions encountered while opening the shard index or from earlier engine failure.

Any additional keyword arguments will be passed to `Elasticsearch.indices.shard_stores` unchanged.

**shrink** (*using=None*, ***kwargs*)
The shrink index API allows you to shrink an existing index into a new index with fewer primary shards. The number of primary shards in the target index must be a factor of the shards in the source index. For example an index with 8 primary shards can be shrunk into 4, 2 or 1 primary shards or an index with 15 primary shards can be shrunk into 5, 3 or 1. If the number of shards in the index is a prime number it can only be shrunk into a single primary shard. Before shrinking, a (primary or replica) copy of every shard in the index must be present on the same node.

Any additional keyword arguments will be passed to `Elasticsearch.indices.shrink` unchanged.

**stats** (*using=None*, ***kwargs*)
Retrieve statistics on different operations happening on the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.stats` unchanged.

**updateByQuery** (*using=None*)
Return a *UpdateByQuery* object searching over the index (or all the indices belonging to this template) and updating Documents that match the search criteria.

For more information, see here: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-update-by-query.html

**upgrade** (*using=None*, ***kwargs*)
Upgrade the index to the latest format.

Any additional keyword arguments will be passed to `Elasticsearch.indices.upgrade` unchanged.

**validate_query** (*using=None*, ***kwargs*)
Validate a potentially expensive query without executing it.

Any additional keyword arguments will be passed to `Elasticsearch.indices.validate_query` unchanged.

### 9.6.4 Faceted Search

**class** elasticsearch_dsl.**FacetedSearch** (*query=None*, *filters={}*, *sort=()*)
Abstraction for creating faceted navigation searches that takes care of composing the queries, aggregations and filters as needed as well as presenting the results in an easy-to-consume fashion:

```
class BlogSearch(FacetedSearch):
    index = 'blogs'
    doc_types = [Blog, Post]
    fields = ['title^5', 'category', 'description', 'body']
```

(continues on next page)

(continued from previous page)

```
    facets = {
        'type': TermsFacet(field='_type'),
        'category': TermsFacet(field='category'),
        'weekly_posts': DateHistogramFacet(field='published_from', interval='week
↪')
    }

    def search(self):
        ' Override search to add your own filters '
        s = super(BlogSearch, self).search()
        return s.filter('term', published=True)

# when using:
blog_search = BlogSearch("web framework", filters={"category": "python"})

# supports pagination
blog_search[10:20]

response = blog_search.execute()

# easy access to aggregation results:
for category, hit_count, is_selected in response.facets.category:
    print(
        "Category %s has %d hits%s." % (
            category,
            hit_count,
            ' and is chosen' if is_selected else ''
        )
    )
```

**Parameters**

- **query** – the text to search for
- **filters** – facet values to filter
- **sort** – sort information to be passed to *Search*

**add_filter**(*name*, *filter_values*)
Add a filter for a facet.

**aggregate**(*search*)
Add aggregations representing the facets selected, including potential filters.

**build_search**()
Construct the `Search` object.

**execute**()
Execute the search and return the response.

**filter**(*search*)
Add a `post_filter` to the search request narrowing the results based on the facet filters.

**highlight**(*search*)
Add highlighting for all the fields

**query**(*search*, *query*)
Add query part to `search`.

Override this if you wish to customize the query used.

**search**()
Returns the base Search object to which the facets are added.

You can customize the query by overriding this method and returning a modified search object.

**sort**(*search*)
Add sorting information to the request.

### 9.6.5 Update By Query

**class** elasticsearch_dsl.**UpdateByQuery**(***kwargs*)
Update by query request to elasticsearch.

**Parameters**

- **using** – *Elasticsearch* instance to use
- **index** – limit the search to index
- **doc_type** – only query this type.

All the parameters supplied (or omitted) at creation type can be later overriden by methods (*using*, *index* and *doc_type* respectively).

**execute**()
Execute the search and return an instance of `Response` wrapping all the data.

**classmethod from_dict**(*d*)
Construct a new *UpdateByQuery* instance from a raw dict containing the search body. Useful when migrating from raw dictionaries.

Example:

```
ubq = UpdateByQuery.from_dict({
    "query": {
        "bool": {
            "must": [...]
        }
    },
    "script": {...}
})
ubq = ubq.filter('term', published=True)
```

**response_class**(*cls*)
Override the default wrapper used for the response.

**script**(***kwargs*)
Define update action to take: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting-using.html for more details.

Note: the API only accepts a single script, so calling the script multiple times will overwrite.

Example:

```
ubq = Search()
ubq = ubq.script(source="ctx._source.likes++"")
ubq = ubq.script(source="ctx._source.likes += params.f"",
                 lang="expression",
                 params={'f': 3})
```

**to_dict**(***kwargs*)
Serialize the search into the dictionary that will be sent over as the request'ubq body.

All additional keyword arguments will be included into the dictionary.

**update_from_dict**(*d*)
Apply options from a serialized body to the current instance. Modifies the object in-place. Used mostly by `from_dict`.

### 9.6.6 Mappings

If you wish to create mappings manually you can use the `Mapping` class, for more advanced use cases, however, we recommend you use the *Document* abstraction in combination with *Index* (or *IndexTemplate*) to define index-level settings and properties. The mapping definition follows a similar pattern to the query dsl:

```
from elasticsearch_dsl import Keyword, Mapping, Nested, Text

# name your type
m = Mapping('my-type')

# add fields
m.field('title', 'text')

# you can use multi-fields easily
m.field('category', 'text', fields={'raw': Keyword()})

# you can also create a field manually
comment = Nested(
                properties={
                    'author': Text(),
                    'created_at': Date()
                })

# and attach it to the mapping
m.field('comments', comment)

# you can also define mappings for the meta fields
m.meta('_all', enabled=False)

# save the mapping into index 'my-index'
m.save('my-index')
```

**Note:** By default all fields (with the exception of `Nested`) will expect single values. You can always override this expectation during the field creation/definition by passing in `multi=True` into the constructor (`m.field('tags', Keyword(multi=True))`). Then the value of the field, even if the field hasn't been set, will be an empty list enabling you to write `doc.tags.append('search')`.

Especially if you are using dynamic mappings it might be useful to update the mapping based on an existing type in Elasticsearch, or create the mapping directly from an existing type:

```
# get the mapping from our production cluster
m = Mapping.from_es('my-index', 'my-type', using='prod')

# update based on data in QA cluster
m.update_from_es('my-index', using='qa')
```

(continues on next page)

(continued from previous page)

```
# update the mapping on production
m.save('my-index', using='prod')
```

Common field options:

**multi** If set to `True` the field's value will be set to `[]` at first access.

**required** Indicates if a field requires a value for the document to be valid.

## 9.7 Contribution Guide

If you have a bugfix or new feature that you would like to contribute to elasticsearch-dsl-py, please find or open an issue about it first. Talk about what you would like to do. It may be that somebody is already working on it, or that there are particular issues that you should know about before implementing the change.

We enjoy working with contributors to get their code accepted. There are many approaches to fixing a problem and it is important to find the best approach before writing too much code.

The process for contributing to any of the Elasticsearch repositories is similar.

1. Please make sure you have signed the Contributor License Agreement. We are not asking you to assign copyright to us, but to give us the right to distribute your code without restriction. We ask this of all contributors in order to assure our users of the origin and continuing existence of the code. You only need to sign the CLA once.
2. Run the test suite to ensure your changes do not break existing code:

   ```
   $ python setup.py test
   ```

3. Rebase your changes. Update your local repository with the most recent code from the main elasticsearch-dsl-py repository, and rebase your branch on top of the latest master branch. We prefer your changes to be squashed into a single commit.
4. Submit a pull request. Push your local changes to your forked copy of the repository and submit a pull request. In the pull request, describe what your changes do and mention the number of the issue where discussion has taken place, eg "Closes #123. Please consider adding or modifying tests related to your changes.

Then sit back and wait. There will probably be discussion about the pull request and, if any changes are needed, we would love to work with you to get your pull request merged into elasticsearch-dsl-py.

## 9.8 Changelog

### 9.8.1 7.1.0 (2019-10-23)

- Optimistic concurrent control for Document.delete
- Removing deprecated `DocType`
- Proper count caching for ES 7.x
- Support for `multiplexer` token filter
- Don't substitute for __ in `FacetedSearch`

### 9.8.2 7.0.0 (2019-04-26)

- Compatibility with Elasticsearch 7.x
- `Document.save()` now returns `"created"` or `"updated"`
- Dropped support for Python 2.6, 3.2, and 3.3
- When using `fields` the values are no longer merged into the body of the document and have to be accessed via `.meta.fields` only

### 9.8.3 6.4.0 (2019-04-26)

- `Index.document` now correctly sets the `Document`'s `_index` only when using default index (`#1091`)
- `Document` inheritance allows overriding `Object` and `Nested` field metadata like `dynamic`
- adding `auto_date_histogram` aggregation
- Do not change data in place when (de)serializing

### 9.8.4 6.3.1 (2018-12-05)

- `Analyzer.simulate` now supports built-in analyzers
- proper (de)serialization of the `Range` wrapper
- Added `search_analyzer` to `Completion` field

### 9.8.5 6.3.0 (2018-11-21)

- Fixed logic around defining a different `doc_type` name.
- Added `retry_on_conflict` parameter to `Document.update`.
- fields defined on an index are now used to (de)serialize the data even when not defined on a `Document`
- Allow `Index.analyzer` to construct the analyzer
- Detect conflict in analyzer definitions when calling `Index.analyzer`
- Detect conflicting mappings when creating an index
- Add `simulate` method to `analyzer` object to test the analyzer using the `_analyze` API.
- Add `script` and `script_id` options to `Document.update`
- `Facet` can now use other metric than `doc_count`
- `Range` objects to help with storing and working with `_range` fields
- Improved behavior of `Index.save` where it does a better job when index already exists
- Composite aggregations now correctly support multiple `sources` aggs
- `UpdateByQuery` implementated by @emarcey

### 9.8.6 6.2.1 (2018-07-03)

- allow users to redefine `doc_type` in `Index` (`#929`)
- include `DocType` in `elasticsearch_dsl` module directly (`#930`)

### 9.8.7 6.2.0 (2018-07-03)

**Backwards incompatible change** - `DocType` refactoring.

In `6.2.0` we refactored the `DocType` class and renamed it to `Document`. The primary motivation for this was the support for types being dropped from elasticsearch itself in `7.x` - we needed to somehow link the `Index` and `Document` classes. To do this we split the options that were previously defined in the `class Meta` between it and newly introduced `class Index`. The split is that all options that were tied to mappings (like setting `dynamic = MetaField('strict')`) remain in `class Meta` and all options for index definition (like `settings`, `name`, or `aliases`) got moved to the new `class Index`.

You can see some examples of the new functionality in the `examples` directory. Documentation has been updated to reflect the new API.

`DocType` is now just an alias for `Document` which will be removed in `7.x`. It does, however, work in the new way which is not fully backwards compatible.

- `Percolator` field now expects `Query` objects as values
- you can no longer access meta fields on a `Document` instance by specifying `._id` or similar. Instead all access needs to happen via the `.meta` attribute.
- Implemented `NestedFacet` for `FacetedSearch`. This brought a need to slightly change the semantics of `Facet.get_values` which now expects the whole data dict for the aggregation, not just the `buckets`. This is a backwards incompatible change for custom aggregations that redefine that method.
- `Document.update` now supports `refresh` kwarg
- `DslBase._clone` now produces a shallow copy, this means that modifying an existing query can have effects on existing `Search` objects.
- Empty `Search` no longer defaults to `match_all` query and instead leaves the `query` key empty. This is backwards incompatible when using `suggest`.

### 9.8.8 6.1.0 (2018-01-09)

- Removed `String` field.
- Fixed issue with `Object`/`Nested` deserialization

### 9.8.9 6.0.1 (2018-01-02)

Fixing wheel package for Python 2.7 (#803)

### 9.8.10 6.0.0 (2018-01-01)

Backwards incompatible release compatible with elasticsearch 6.0, changes include:

- use `doc` as default `DocType` name, this change includes: * `DocType._doc_type.matches` method is now used to determine which `DocType` should be used for a hit instead of just checking `_type`

- `Nested` and `Object` field refactoring using newly introduced `InnerDoc` class. To define a `Nested`/`Object` field just define the `InnerDoc` subclass and then use it when defining the field:

```
class Comment(InnerDoc):
    body = Text()
    created_at = Date()

class Blog(DocType):
    comments = Nested(Comment)
```

- methods on `connections` singleton are now exposed on the `connections` module directly.
- field values are now only deserialized when coming from elasticsearch (via `from_es` method) and not when assigning values in python (either by direct assignment or in `__init__`).

### 9.8.11 5.4.0 (2017-12-06)

- fix `ip_range` aggregation and rename the class to `IPRange`. `Iprange` is kept for bw compatibility
- fix bug in loading an aggregation with meta data from dict
- add support for `normalizer` parameter of `Keyword` fields
- `IndexTemplate` can now be specified using the same API as `Index`
- `Boolean` field now accepts `"false"` as `False`

### 9.8.12 5.3.0 (2017-05-18)

- fix constant score query definition
- `DateHistogramFacet` now works with `datetime` objects
- respect `__` in field names when creating queries from dict

### 9.8.13 5.2.0 (2017-03-26)

- make sure all response structers are pickleable (for caching)
- adding `exclude` to `Search`
- fix metric aggregation deserialization
- expose all index-level APIs on `Index` class
- adding `delete` to `Search` which calls `delete_by_query` API

### 9.8.14 5.1.0 (2017-01-08)

- Renamed `Result` and `ResultMeta` to `Hit` and `HitMeta` respectively
- `Response` now stores `Search` which it gets as first arg to `__init__`
- aggregation results are now wrapped in classes and properly deserialized
- `Date` fields now allow for numerical timestamps in the java format (in millis)
- Added API documentation

- replaced generated classes with manually created

### 9.8.15 5.0.0 (2016-11-04)

Version compatible with elasticsearch 5.0.

Breaking changes:

- `String` field type has been deprecated in favor of `Text` and `Keyword`
- `fields` method has been removed in favor of `source` filtering

### 9.8.16 2.2.0 (2016-11-04)

- accessing missing string fields no longer returnd `''` but returns `None` instead.
- fix issues with bool's `|` and `&` operators and `minimum_should_match`

### 9.8.17 2.1.0 (2016-06-29)

- `inner_hits` are now also wrapped in `Response`
- `+` operator is deprecated, `.query()` now uses `&` to combine queries
- added `mget` method to `DocType`
- fixed validation for "empty" values like `''` and `[]`

### 9.8.18 2.0.0 (2016-02-18)

Compatibility with Elasticsearch 2.x:

- Filters have been removed and additional queries have been added. Instead of `F` objects you can now use `Q`.
- `Search.filter` is now just a shortcut to add queries in filter context
- support for pipeline aggregations added

Backwards incompatible changes:

- list of analysis objects and classes was removed, any string used as tokenizer, char or token filter or analyzer will be treated as a builtin
- internal method `Field.to_python` has been renamed to `deserialize` and an optional serialization mechanic for fields has been added.
- Custom response class is now set by `response_class` method instead of a kwarg to `Search.execute`

Other changes:

- `FacetedSearch` now supports pagination via slicing

### 9.8.19 0.0.10 (2016-01-24)

- `Search` can now be iterated over to get back hits
- `Search` now caches responses from Elasticsearch
- `DateHistogramFacet` now defaults to returning empty intervals
- `Search` no longer accepts positional parameters
- Experimental `MultiSearch` API
- added option to talk to `_suggest` endpoint (`execute_suggest`)

### 9.8.20 0.0.9 (2015-10-26)

- `FacetedSearch` now uses its own `Facet` class instead of built in aggregations

### 9.8.21 0.0.8 (2015-08-28)

- `0.0.5` and `0.0.6` was released with broken .tar.gz on pypi, just a build fix

### 9.8.22 0.0.5 (2015-08-27)

- added support for (index/search)_analyzer via #143, thanks @wkiser!
- even keys accessed via `['field']` on `AttrDict` will be wrapped in `Attr[Dict|List]` for consistency
- Added a convenient option to specify a custom `doc_class` to wrap inner/Nested documents
- `blank` option has been removed
- `AttributeError` is no longer raised when accessing an empty field.
- added `required` flag to fields and validation hooks to fields and (sub)documents
- removed `get` method from `AttrDict`. Use `getattr(d, key, default)` instead.
- added `FacetedSearch` for easy declarative faceted navigation

### 9.8.23 0.0.4 (2015-04-24)

- Metadata fields (such as id, parent, index, version etc) must be stored (and retrieved) using the `meta` attribute (#58) on both `Result` and `DocType` objects or using their underscored variants (`_id`, `_parent` etc)
- query on Search can now be directly assigned
- `suggest` method added to `Search`
- `Search.doc_type` now accepts `DocType` subclasses directly
- `Properties.property` method renamed to `field` for consistency
- Date field now raises `ValidationException` on incorrect data

### 9.8.24 0.0.3 (2015-01-23)

Added persistence layer (`Mapping` and `DocType`), various fixes and improvements.

### 9.8.25 0.0.2 (2014-08-27)

Fix for python 2

### 9.8.26 0.0.1 (2014-08-27)

Initial release.

# Python Module Index

## e

# Index

## M



## O



## P



## Q



## R



## S



## T



## U



## V