Cape API Client Library Documentation

Release 0.1

Bloomsbury AI

Mar 23, 2018

Contents:

Cape is an API from Bloomsbury AI that makes it easy to build software that answers questions about the contents of documents. For example, you can use cape to:

- Build a super-powered ctrl+f that finds the answer to questions like 'Who is the CFO?', rather than just all the occurrences of a keyword.
- Build an expressive query tool for textual data - extract features with a single question.
- Build a virtual office assistant that answers routine questions through Slack.
- Build an add-on to your private search that mimics Google's 'direct answers'.

To install the Cape client library simply run:

```
pip3 install cape-client
```

CHAPTER 1

Examples

1.1 Admin Authentication

There are two primary mechanisms for authentication within Cape: admin authentication, which provides full access to your account and *user authentication*, which only provides access to the answer endpoint.

There are two different ways to authenticate as an administrator, either through the *cape.client.CapeClient.login()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
```

Or you can authenticate using an admin token when creating the CapeClient object. This admin token can be retrieved through the Cape UI:

```
from cape.client import CapeClient

cc = CapeClient(admin_token='youradmintoken')
```

1.2 Answering Questions

1.2.1 Authentication

Requests to the answer endpoint require a "user token", this enables developers to provide access to their Cape AI by embedding their user token within their application.

The user token for your AI can either be retrieved through the Cape UI or through a call to the API:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
print(cc.get_user_token())
```

This will output a user token specific to your account, for example:

```
08aerv08ajkdp
```

Please note that while it is safe to distribute your user token as part of your application you should not include your login credentials, as this provides full administration access to your account.

1.2.2 Answering A Question

With the user token retrieved in the previous step we can now make calls to the answer endpoint. In its simplest use-case this just requires us to pass the question and the user token to the *cape.client.CapeClient.answer()* method:

```
from cape.client import CapeClient

USER_TOKEN = '08aerv08ajkdp'

cc = CapeClient()
answers = cc.answer('How easy is this API to use?', USER_TOKEN)
print(answers)
```

This will output the following answer list:

```
[
    {
        'answerText': "Hopefully it's pretty easy",
        'answerContext: "Welcome to the Cape API 0.1. Hopefully it's pretty easy to␣
↪use.",
        'confidence': 0.75,
        'sourceType': 'document',
        'sourceId': '358e1b77c9bcc353946dfe107d6b32ff',
        'answerTextStartOffset': 30,
        'answerTextEndOffset': 56,
        'answerContextStartOffset': 0,
        'answerContextEndOffset': 64
    }
]
```

By default *cape.client.CapeClient.answer()* will only fetch the answer with the highest confidence value, for details on fetching multiple results see the *multiple answers* section.

Each answer in the list contains the following properties:

Property	Description
answerText	The proposed answer to the question
answerContext	The context surrounding the proposed answer to the question
confidence	How confident the AI is that this is the correct answer
sourceType	Whether this result came from a 'document'or a 'saved_reply'
sourceId	The ID of the document or saved reply this answer was found in (depending on sourceType)
answerTextStartOffset	The starting position of this answer in the document (if sourceType is 'document')
answerTextEndOffset	The end position of this answer in the document (if sourceType is 'document')
answerContextStartOffset	The starting position of this answer context in the document (if sourceType is 'document')
answerContextEndOffset	The end position of this answer context in the document (if sourceType is 'document')

1.2.3 Multiple Answers

In some cases, such as when searching through a document or extracting information from multiple documents, it may be desirable to retrieve more than one answer. This can be done via the *number_of_items* and *offset* parameters. For example to retrieve the first 5 answers:

```
from cape.client import CapeClient

USER_TOKEN = '08aerv08ajkdp'

cc = CapeClient()
answers = cc.answer('When were people born?',
                    USER_TOKEN,
                    number_of_items=5)
print(answers)
```

Which will produce output like:

```
[
    {
        'answerText': "Sam was born in 1974",
        'answerContext: "did very good work. Sam was born in 1974 on the sunny island␣
↪of",
        'confidence': 0.75,
        'sourceType': 'document',
        'sourceId': 'employee_info.txt',
        'answerTextStartOffset': 80,
        'answerTextEndOffset': 100,
        'answerContextStartOffset':40,
        'answerContextEndOffset':123
    },
    {
        'answerText': "James was born in 1982",
        'answerContext: "James was born in 1982 on the sunny island of",
        'confidence': 0.64,
        'sourceType': 'document',
        'sourceId': 'employee_info.txt',
        'answerTextStartOffset': 0,
        'answerTextEndOffset': 22,
        'answerContextStartOffset':0,
        'answerContextEndOffset':45
    },
    {
        'answerText': "Alice was born in 1973",
        'answerContext: "did very good work. Alice was born in 1973 on the sunny␣
↪island of",
        'confidence': 0.61,
        'sourceType': 'document',
        'sourceId': 'employee_info.txt',
        'answerTextStartOffset': 220,
        'answerTextEndOffset': 242,
        'answerContextStartOffset':200,
        'answerContextEndOffset':265
    },
    {
        'answerText': "Bob was born in 1965",
        'answerContext: "did very good work. Bob was born in 1965 on the sunny island␣
↪of",
        'confidence': 0.59,
        'sourceType': 'document',
        'sourceId': 'employee_info.txt',
        'answerTextStartOffset': 180,
        'answerTextEndOffset': 200,
        'answerContextStartOffset':140,
```

```
        'answerContextEndOffset':223
    },
    {
        'answerText': "Jill was born in 1986",
        'answerContext: "did very good work. Jill was born in 1986 on the sunny␣
↪island of",
        'confidence': 0.57,
        'sourceType': 'document',
        'sourceId': 'employee_info.txt',
        'answerTextStartOffset': 480,
        'answerTextEndOffset': 501,
        'answerContextStartOffset':440,
        'answerContextEndOffset':524
    }
]
```

If we then wished to retrieve the next 5 answers we could run:

```
answers = cc.answer('When were people born?',
                    USER_TOKEN,
                    number_of_items=5,
                    offset=5)
```

Which will return a further 5 answers starting with the 6th one. This allows us to retrieve answers in batches, only fetching more when the user needs them.

1.2.4 Searching Specific Documents

If we wish to search within a specific document (e.g. the document the user is currently viewing in our application) or in a set of documents we can specify the *document_ids* when requesting an answer. For example:

```
from cape.client import CapeClient

USER_TOKEN = '08aerv08ajkdp'

cc = CapeClient()
answers = cc.answer('When was James born?',
                    USER_TOKEN,
                    document_ids = ['employee_info_2016.txt',
                                    'employee_info_2017.txt',
                                    'employee_info_2018.txt'])
print(answers)
```

If we're explicitly searching through a document we may also wish to disable saved reply responses, this can be done with the *source_type* parameter:

```
answers = cc.answer('When was James born?',
                    USER_TOKEN,
                    document_ids = ['employee_info_2016.txt',
                                    'employee_info_2017.txt',
                                    'employee_info_2018.txt'],
                    source_type = 'document')
```

1.3 Managing Documents

Documents can be uploaded, updated and deleted using the client API. This functionality is only available to users with *administrative access*.

1.3.1 Creating Documents

There are two ways to create a new document, we can either provide the text contents of a document via the *text* parameter of the `cape.client.CapeClient.add_document()` method or we can upload a file via the *file_path* parameter.

To create a document using the *text* parameter:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
doc_id = cc.add_document("Document title",
                         "Hello and welcome to my document!")
print(doc_id)
```

If we don't supply a *document_id* when calling `cape.client.CapeClient.add_document()` an ID will be automatically generated for us. Automatically generated IDs are created by taking the SHA256 hash of the document contents. So for this document the following ID will be produced:

```
356477322741dbf8d8f0375ecdc6ae03357641829ae7ccf10283af36c5508a9d
```

Alternatively we can upload a file:

```
from cape.client import CapeClient

# Create an example file
fh = open('/tmp/example_file.txt', 'w')
fh.write("Hello! This is an example file!")
fh.close()

cc = CapeClient()
cc.login('username', 'password')
doc_id = cc.add_document("Document title",
                         file_path="/tmp/example_file.txt",
                         document_id='my_document_id')
print(doc_id)
```

Because we supplied a *document_id* in this example the document ID we get returned will be what we requested:

```
my_document_id
```

As large file uploads could take a long time we may wish to provide the user with updates on the progress of our upload. To do this we can provide a callback function via the *monitor_callback* parameter which will provide us with frequent updates about the upload's progress:

```
from cape.client import CapeClient

def upload_cb(monitor):
    print("%d/%d" % (monitor.bytes_read, monitor.len))

# Create a large example file
fh = open('/tmp/large_example.txt', 'w')
```

```
fh.write("Hello! This is a large example file! " * 100000)
fh.close()

cc = CapeClient()
cc.login('username', 'password')
doc_id = cc.add_document("Document title",
                         file_path="/tmp/large_example.txt",
                         monitor_callback=upload_cb)
```

This will then print a series of status updates showing the progress of our file upload:

```
...
2523136/3700494
2531328/3700494
2539520/3700494
2547712/3700494
2555904/3700494
2564096/3700494
2572288/3700494
2580480/3700494
...
```

1.3.2 Updating Documents

To update a document we simply upload a new document with the same *document_id* and set the *replace* parameter to True. Without explicitly informing the server that we wish to replace the document it will report an error to avoid accidental replacement of documents. For example:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')

# Create the original document
doc_id = cc.add_document("My document",
                         "This is a good document.")

# Replace it with an improved version
cc.add_document("My document",
                "This is a great document.",
                document_id=doc_id,
                replace=True)
```

1.3.3 Deleting Documents

To delete a document simply call the *cape.client.CapeClient.delete_document()* method with the ID of the document you wish to remove:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')

cc.delete_document('my_bad_document')
```

1.3.4 Retrieving Documents

The *cape.client.CapeClient.get_documents()* method can be used to retrieve all previously uploaded documents:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')

documents = cc.get_documents()
print(documents)
```

This will output:

```
{
    'totalItems': 2,
    'items': [
        {
            'id': 'custom_id_2',
            'title': 'document2.txt',
            'origin': 'document2.txt',
            'text': 'This is another document.',
            'created': 1508169352
        },
        {
           'id': '358e1b77c9bcc353946dfe107d6b32ff',
            'title': 'cape_api.txt',
            'origin': 'cape_api.txt',
            'text': "Welcome to the Cape API 0.1. " \
                    "Hopefully it's pretty easy to use.",
            'created': 1508161723
        }
    ]
}
```

By default this will retrieve 30 documents at a time. The *number_of_items* and *offset* parameters can be used to control the size of the batches and retrieve multiple batches of documents (similar to the mechanism described in the *multiple answers* section). The response also includes the *totalItems* property which tells us the total number of items available (beyond those retrieved in this specific batch).

Each document in the list contains the following properties:

Property	Description
id	The ID of this document
title	The document's title (specified at upload)
origin	Where this document originally came from
text	The contents of the document
type	Whether this document was created by submitting text or from a file upload
created	Timestamp of when this document was first uploaded

1.4 Managing Saved Replies

Saved replies are made up of a canonical question and the response it should produce. In addition to the canonical question a saved reply may have many paraphrased questions associated with it which should produce the same answer

(e.g. "How old are you?" vs "What is your age?"). This functionality is only available to users with *administrative access*.

1.4.1 Creating Saved Replies

To create a new saved reply simply call the *cape.client.CapeClient.add_saved_reply()* method with a question and answer pair:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
reply_id = cc.add_saved_reply('What colour is the sky?', 'Blue')
print(reply_id)
```

This will respond with a dictionary containing the ID of the new reply and the ID of the new answer:

```
{
    'replyId': 'f9f1cf90-c3b1-11e7-91a1-9801a7ae6c69',
    'answerId': 'd2780710-c3c3-11e7-8d29-d15d28ee5381'
}
```

Saved replies must have a unique question. If this question already exists then an error is returned.

1.4.2 Deleting Saved Replies

To delete a saved reply pass its ID to the *cape.client.CapeClient.delete_saved_reply()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.delete_saved_reply("f9f1cf90-c3b1-11e7-91a1-9801a7ae6c69")
```

1.4.3 Retrieving Saved Replies

To retrieve a list of all saved replies use the *cape.client.CapeClient.get_saved_replies()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
replies = cc.get_saved_replies()
print(replies)
```

This will return a list of replies:

```
{
    'totalItems': 2,
    'items': [
        {
            'id': 'd277e000-c3c3-11e7-8d29-d15d28ee5381',
            'canonicalQuestion': 'How old are you?',
            'answers': [
```

```
                {
                    'id': 'd2780710-c3c3-11e7-8d29-d15d28ee5381',
                    'answer': '18'
                }
            ],
            'paraphraseQuestions': [
                {
                    'id': 'd2780711-c3c3-11e7-8d29-d15d28ee5381',
                    'question': 'What is your age?'
                },
                {
                    'id': 'd2780712-c3c3-11e7-8d29-d15d28ee5381',
                    'question': 'How many years old are you?'
                }
            ],
            'created': 1508161734,
            'modified': 1508161734
        },
        {
            'id': 'd2780713-c3c3-11e7-8d29-d15d28ee5381',
            'canonicalQuestion': 'What colour is the sky?',
            'answers': [
                {
                    'id': 'd2780714-c3c3-11e7-8d29-d15d28ee5381',
                    'answer': 'Blue'
                }
            ],
            'paraphraseQuestions': [],
            'created': 1508161323,
            'modified': 1508161323
        }
    ]
}
```

By default this will retrieve 30 saved replies at a time. The *number_of_items* and *offset* parameters can be used to control the size of the batches and retrieve multiple batches of saved replies (similar to the mechanism described in the *multiple answers* section). The response also includes the *totalItems* property which tells us the total number of items available (beyond those retrieved in this specific batch).

Each saved reply in the list contains the following properties:

Property	Description
id	The reply ID
canonicalQuestion	The question to which the saved reply corresponds
answers	A list of saved answers, one of which will be selected at random as the response to the question.
paraphraseQuestions	A list of questions which paraphase the canonical question
modified	Timestamp indicating when this saved reply was last modified
created	Timestamp indicating when this saved reply was created

It's also possible to search saved replies, for example to retrieve only saved replies containing the word 'blue':

```
from cape.client import CapeClient

cc = CapeClient()
```

```
cc.login('username', 'password')
replies = cc.get_saved_replies(search_term='blue')
```

1.4.4 Editing Saved Replies

There are three different parts of a saved reply that can be edited, the canonical question, the paraphrase questions and the answers.

Adding Paraphrase Questions

Paraphrase questions are alternative phrasings of the canonical question which should produce the same answer. For example "What is your age?" can be considered a paraphrase of "How old are you?". These can be added with the *cape.client.CapeClient.add_paraphrase_question()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
question_id = cc.add_paraphrase_question("f9f1cf90-c3b1-11e7-91a1-9801a7ae6c69",
↪'What is your age?')
print(question_id)
```

This will respond with the ID of the newly created question:

```
21e9689e-c3b2-11e7-8a22-9801a7ae6c69
```

Editing Paraphrase Questions

To edit a paraphrase question call *cape.client.CapeClient.edit_paraphrase_question()* with the ID of the question to edit and the new question text to modify it with:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.edit_paraphrase_question("21e9689e-c3b2-11e7-8a22-9801a7ae6c69", 'How many years␣
↪old are you?')
```

Deleting Paraphrase Questions

To delete a paraphrase question simply call *cape.client.CapeClient.delete_paraphrase_question()* with the ID of question to be deleted:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.delete_paraphrase_question("21e9689e-c3b2-11e7-8a22-9801a7ae6c69")
```

Adding Answers

If multiple answers are added to a saved reply then one will be selected at random when responding. Additional answers can be added with the *cape.client.CapeClient.add_answer()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
answer_id = cc.add_answer("68c445cc-c3b2-11e7-8a88-9801a7ae6c69", 'Grey')
print(answer_id)
```

This will respond with the ID of the new answer:

```
703acab4-c3b2-11e7-b8b1-9801a7ae6c69
```

Deleting Answers

To delete an answer call *cape.client.CapeClient.delete_answer()* with the ID of the answer to be deleted:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.delete_answer("703acab4-c3b2-11e7-b8b1-9801a7ae6c69")
```

Because every saved reply must have at least one answer it's not possible to delete the last remaining answer in a saved reply, in this case you may wish to consider deleting the saved reply itself.

Editing Canonical Questions

To edit the canonical question call *cape.client.CapeClient.edit_canonical_question()* with the ID of the saved reply that it belongs to:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.edit_canonical_question("f9f1cf90-c3b1-11e7-91a1-9801a7ae6c69", 'What age are you?
↪')
```

1.5 Managing The Inbox

The inbox provides a list of questions that have been asked by users and the response the system has replied with. This functionality is only available to users with *administrative access*.

1.5.1 Retrieving Inbox Items

To retrieve inbox items call the *cape.client.CapeClient.get_inbox()* method:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
inbox = cc.get_inbox()
print(inbox)
```

This returns a list of inbox items:

```
{
    'totalItems': 2,
    'items': [
        {
            'id': '4124',
            'answered': False,
            'read': False,
            'question': 'Who are you?',
            'questionSource': 'API',
            'created': 1508162032,
            'answers': []
        },
        {
            'id': '4123',
            'answered': True,
            'read': False,
            'question': 'How easy is the API to use?',
            'questionSource': 'API',
            'created': 1508161834,
            'answers': [
                {
                    'answerText': "Hopefully it's pretty easy",
                    'answerContext: "Welcome to the Cape API 0.1. Hopefully it's␣
↪pretty easy to use.",
                    'confidence': 0.75,
                    'sourceType': 'document',
                    'sourceId': '358e1b77c9bcc353946dfe107d6b32ff',
                    'answerTextStartOffset': 30,
                    'answerTextEndOffset': 56,
                    'answerContextStartOffset': 0,
                    'answerContextEndOffset': 64
                }
            ]
        }
    ]
}
```

By default this will retrieve 30 inbox items at a time. The *number_of_items* and *offset* parameters can be used to control the size of the batches and retrieve multiple batches of inbox items (similar to the mechanism described in the *multiple answers* section). The response also includes the *totalItems* property which tells us the total number of items available (beyond those retrieved in this specific batch).

Each inbox item in the list has the following properties:

Property	Description
id	Unique ID for this inbox item
question	The question that a user asked
read	Whether this item has been read
answered	Whether an answer could be found for this question
answers	A list of *answer objects*
created	Timestamp indicating when this question was asked

Inbox items can be searched and filtered, for example to retrieve only inbox items that haven't been read but have been answered and contain the word 'API':

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
inbox = cc.get_inbox(read=False, answered=True, search_term='api')
```

1.5.2 Marking Inbox Items As Read

To mark an inbox item as having been read call the *cape.client.CapeClient.mark_inbox_read()* method with the ID of the inbox item to mark as having been read:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.mark_inbox_read('4123')
```

1.5.3 Archiving Inbox Items

Once an inbox item has been archived it will no longer appear in the list of inbox items returned by *cape.client.CapeClient.get_inbox()*. To archive an item call *cape.client.CapeClient.archive_inbox()* with the ID of the inbox item to archive:

```
from cape.client import CapeClient

cc = CapeClient()
cc.login('username', 'password')
cc.archive_inbox('4123')
```

CHAPTER 2

How To Build A Simple AI-Powered Ctrl-F with Cape

The Cape API is a an easy to use way to add advanced, AI-powered question answering to your application. In this tutorial we show you how you can create an AI-Powered Ctrl-F using the Cape API with only a few lines of code.

Once finished, you'll have something that allows you to copy and paste in a piece of text and ask questions about it.

2.1 Getting Set Up

2.1.1 Requirements

- Python 3.6+
- Python packages are defined within requirements.txt, you can install them with the following command:

```
pip3 install -r requirements.txt
```

- A Cape Account. You can sign up with Google or Facebook, it's easy. Do it now.

2.1.2 Getting The Code

The default branch, `master` is the completed code—once you've installed the Python requirements in `requirements.txt` you should be able to type the command `python app.py` in the root directory and play with the finished tutorial already.

There is another branch, `begin`, which has removed parts of the Python and JavaScript code. Starting with this branch, you should be able to copy and paste in parts as we go along and end up with the finished product :).

2.2 Introduction To The Cape Client

We've written a Python client for Cape API that we'll be using to build our Ctrl+F functionality. Before we get started building,

- Authentication and Tokens.

2.2.1 Authenticating

There are two ways to authenticate - with an **Admin Token** or your **username and password**.

Note: If you've created an account with Facebook or Google you won't have a password, so you'll have to use your admin token. To get your amin token, login here, go to the top right hand corner, click on the cartoon head and shoulders and your admin token should be displayed.

Here's the code to authenticate with the **username and password**:

```
from cape.client import CapeClient
cape_client = CapeClient()
# login with the client
cape_client.login('USERNAME', 'PASSWORD')
```

Alternatively, you can use the **Admin Token**.

```
from cape.client import CapeClient
cape_client = CapeClient(admin_token=ADMIN_TOKEN)
```

2.2.2 The User Token And The Admin Token

There are two types of token used in the Cape API: the **Admin Token** and the **User Token**.

The **Admin Token** gives you full control over your account, and is required for things like adding Documents, adding Saved Replies etc. You can get your admin token from your online dashboard once logged in, and use it as a parameter to your API calls even if you aren't logged in.:

```
# set the default threshold to low - means the AI will
# answer you more of the time, but will get more wrong
cape_client.set_default_threshold('low')
```

The **User Token** is only used when answering questions. You can think of your User token as giving someone 'read' access to your AI. The following code retrieves your user token so we can query our document for answers.:

```
my_user_token = cape_client.get_user_token()
print(my_user_token) # 08aerv08ajkdp
```

2.3 Adding The Document You Want To Search

In our final demo we have area where you can copy and paste in a document you'd like to be able to search and see the results immediately. To get started with this, we'll explore the upload documents functionality of the Cape API.

2.3.1 The Upload Document Method

There are two ways to create a document - literally uploading a document from your filesystem (for the time being restricted to markdown and txt documents), or passing a string as the body of the document in the add document method. During this tutorial we'll be using the latter.

Let's say that someone has copied and pasted the following Wikipedia article on football into our text area:

```
Football is a family of team sports that involve, to varying degrees, kicking a ball␣
↪with the foot to score a goal.
Unqualified, the word football is understood to refer to whichever form of football␣
↪is the most popular in the
regional context in which the word appears. Sports commonly called 'football' in␣
↪certain places include:
association football (known as soccer in some countries); gridiron football␣
↪(specifically American football
or Canadian football); Australian rules football; rugby football (either rugby league␣
↪or rugby union); and Gaelic
football.[1][2] These different variations of football are known as football codes.

Various forms of football can be identified in history, often as popular peasant␣
↪games. Contemporary codes of
football can be traced back to the codification of these games at English public␣
↪schools during the nineteenth
century.[3][4] The expanse of the British Empire allowed these rules of football to␣
↪spread to areas of British
influence outside the directly controlled Empire.[5] By the end of the nineteenth␣
↪century, distinct regional codes
were already developing: Gaelic football, for example, deliberately incorporated the␣
↪rules of local traditional
football games in order to maintain their heritage.[6] In 1888, The Football League␣
↪was founded in England,
becoming the first of many professional football competitions. During the twentieth␣
↪century, several of the
various kinds of football grew to become some of the most popular team sports in the␣
↪world.
```

Once we've got this string, we can add a Document to Cape using the Cape Client and start answering questions straight away.

```
# WIKIPEDIA_TEXT is the string of the doc you want to upload
doc_id = cape_client.upload_document("Football Document", WIKIPEDIA_TEXT)
# you can ask a question to a specific document by referencing the document id
answers = cc.answer(question='What is football?',
                    user_token=my_user_token,
                    document_ids=[doc_id],
                    source_type='document',
                    number_of_items=1)
print(answers) # [{'answerText':'Football is a family of team sports',...},...]
```

2.3.2 Adding The Upload Document Method Into Our App

For our tutorial app, we'll be taking the value of a content editable input and uploading that as our document. For the time being we only have a Python client, so let's create an endpoint that takes in the document and uploads it. Since this is a tutorial, we'll use the Flask framework.

In our tutorial we have an editable content HTML element that contains text about Football in `templates/index.html`:

```
<div class="form-control" id="documentText" contenteditable="True">Football is a␣
↪family of team sports that
    involve, to varying degrees, kicking a ball with the foot to score a goal.␣
↪Unqualified, the word
    football is understood to refer to whichever form of football is the most popular␣
↪in the regional
    context in which the word appears. Sports commonly called 'football' in certain␣
↪places include:
    association football (known as soccer in some countries); gridiron football␣
↪(specifically American
    football or Canadian football); Australian rules football; rugby football (either␣
↪rugby league or rugby
    union); and Gaelic football.[1][2] These different variations of football are␣
↪known as football codes.
</div>
```

And we've already written the following jQuery snippet that will hit an 'add_document' endpoint with a post request with the contents of the element. You can add this to `static/app.js`:

```
$(document).ready(function(){
    $('#documentText').bind('input propertychange', function () {
        $.post('/add_document', {'doc':$(this).val()});
    });
});
```

We can then create an endpoint using a logged-in Cape Client. The file you want to edit here is app.py in the root directory:

```
from flask import Flask, render_template, jsonify, request
from cape.client import CapeClient

_CAPE_CLIENT = CapeClient()
_CAPE_CLIENT.login('USERNAME', 'PASSWORD')
```

```
_LAST_DOC_ID = None
_ANSWER_TOKEN = _CAPE_CLIENT.get_user_token() # to be used later
_LAST_DOC_ID = None

# create add_document endpoint
@app.route('/add_document', methods=['POST'])
def add_document():
    global _LAST_DOC_ID
    doc_text = request.form.get('doc', "") # get the document text from the post␣
↪request
    _LAST_DOC_ID = _CAPE_CLIENT.upload_document(title='ctrl_f_doc',
                                                text=doc_text,
                                                replace=True) # upload the document,
    print(f'uploaded doc with id: {_LAST_DOC_ID}')
    return jsonify({'success': True})
```

2.4 Adding The Search Functionality

On to the exciting bit! Now we'll go over how we can add the search functionality to our website.

2.4.1 The Answer Method & Object

Once you've uploaded your documents, getting a response back is as simple as calling one method - *cape.client.CapeClient.answer()* which returns a ranked list of answers. We've got an example below, which we'll discuss in more detail before jumping in to implementing the tutorial.:

```
answers = cape_client.answer(question='What is football?',
                             user_token=ANSWER_TOKEN,
                             document_ids=[FOOTBALL_DOCUMENT_ID],
                             source_type='document',
                             number_of_items=5)
print(answers)
#  [{'answerText':'Football is a family of team sports',...}, ..., ... ]
```

Now let's go through each of these parameters in detail.

query is the string of the question you want answered.

token is your **User Token** (not your Admin Token!).

document_ids is an optional argument. It's a list of document IDs you want read when trying to find the answer to your question. If you don't know, or don't care, which document your answer comes from you can set this to *None*.

source_type is another optional argument. We don't go into it here, but there are two ways you can answer questions with Cape API - the first is by reading documents, but occasionally the right answer isn't found. Using something called a **Saved Reply** you can manually override our reading AI. Since we aren't interested in this behaviour for this tutorial we are going to explicitly set this parameter to *document* which means 'only get answers by reading documents'.

number_of_items is the number of answers you want returned. Our reading AI will try to find this number of answers in the documents, and will return a sorted list of all those it thinks are good enough.

And what is an **Answer** object? Each **Answer** is a Python dictionary containing lots of useful information. A sample Answer will look something like this:

```
{
     'answerText': 'This is the answer text',
     'answerContext': 'context for This is the answer text',
     'confidence': 0.88,
     'sourceType': 'document',
     'sourceId': '8dce9e4841fc944b120f7c5a31ea4dd73bfe41258206af37d5d43a2c74ab27c9',
     'answerTextStartOffset': 10,
     'answerTextEndOffset': 100,
     'answerContextStartOffset': 0,
     'answerContextEndOffset': 120,
}
```

Again, let's go through these attributes in turn to make sure we understand what's going on.

`answerText` is the raw string that the AI thinks is the answer to your query.

`confidence` is a float between 0 and 1 that represents how confident the AI is with this answer.

`sourceType` tells you what type of object contained the answer. In this tutorial the `sourceType` key will always be 'document'.

`sourceId` is the ID of the document that contained the answer.

`answerTextStartOffset` is the location in the document that corresponds to the first character of `answerText`.

`answerTextEndOffset` is the location in the document that corresponds to the last character of `answerText`.

`answerContextStartOffset` is the location in the document that corresponds to the first character of `answerContext`.

`answerContextEndOffset` is the location in the document that corresponds to the last character of `answerContext`.

2.4.2 Integrating The Answer Method Into Our Ctrl+F

Ok, so now we've introduction the answer method, let's integrate it into our tutorial. First, let's start with the html. In our boilerplate code, we have the following input element:

```
<input type="search" class="form-control mb-3" id="ctrlfField" placeholder="ctrl+f␣
↪search bar"/>
```

For which we have the following jQuery:

```
$('#ctrlfField').bind('input propertychange', function (e) {
    e.preventDefault();
    if (typeof(myTimeout) !== "undefined") {
        clearTimeout(myTimeout);
    }
    myTimeout = setTimeout(function () {
        $.get('/ctrl_f', {'query': $('#ctrlfField').val()}, function (data) {
            var answers = data.answers;
            var answer = {};
            var range = [];
            for (i = 0; i < answers.length; i++) {
                answer = answers[i];
                range = {'start': answer.startTextOffset, 'length': (answer.
↪endTextOffset - answer.startTextOffset)};
                if (i === 0) {
                    $('#documentText').markRanges([range], {element: 'span',␣
↪className: 'success'})
```

```
                } else if (i < 4) {
                    $('#documentText').markRanges([range], {element: 'span',␣
↪className: 'info'})
                } else {
                    $('#documentText').markRanges([range], {element: 'span',␣
↪className: 'danger'})
                }
            }
        });
    }, 1000);
    return false;
});
```

Since this isn't a jQuery or JavaScript tutorial, I won't go into this code very much. The gist is that a get request is sent to our 'ctrl_f' endpoint, and we leverage the excellent mark.js package to achieve the highlighting effect.

I've added a few additional bits of logic to make the user experience better, but that complicate the code a little. First, I've added a timeout to only send the request once the user has stopped typing for one second.Second, I've assigned difference classes to different answers based on the order to indicate how confident the AI is about an answer.

Now let's get on to using the Python Cape Client. First we'll add the endpoint to our Flask server:

```
@app.route('/ctrl_f', methods=['GET'])
def ctrl_f():
    # DO CTRL-F LOGIC HERE
    pass
```

Our method inside the endpoint should do the following: (1) get the text from the search input field, (2) make a request to the Cape API with this text and the document ID and (3) return the results of the request as a json object for our JavaScript to highlight. The following code is an example of how we can get this done with the Cape Client:

```
@app.route('/ctrl_f', methods=['GET'])
def ctrl_f():
    if _LAST_DOC_ID is None:
        return jsonify({'success': False, 'answers': []}) # check that we've uploaded␣
↪a document
    query_text = request.args['query'] # get the query text
    # get the answers from our answer endpoint, making sure to reference the correct␣
↪document
    answers = _CAPE_CLIENT.answer(query_text,
                                  _ANSWER_TOKEN,
                                  document_ids=[_LAST_DOC_ID],
                                  number_of_items=5)
    print(f'answers: {answers}')
    return jsonify({'success': True,'answers': answers})
```

This is pretty much the full functionality required for our Ctrl+F demo. Now we just need to put it all together.

2.5 Putting It All Together

This is what our Python file looks like once we've added our index endpoint:

```
from flask import Flask, render_template, jsonify, request
from cape.client import CapeClient
from settings import USERNAME, PASSWORD
```

```
app = Flask(__name__)

_CAPE_CLIENT = CapeClient()
_CAPE_CLIENT.login(USERNAME, PASSWORD)

_LAST_DOC_ID = None
_ANSWER_TOKEN = _CAPE_CLIENT.get_user_token()


@app.route('/')
def index():
    return render_template('index.html')


@app.route('/add_document', methods=['POST'])
def add_document():
    global _LAST_DOC_ID
    doc_text = request.form.get('doc', "")
    _LAST_DOC_ID = _CAPE_CLIENT.upload_document(title='ctrl_f_doc', text=doc_text,␣
↪replace=True)
    print(f'uploaded doc with id: {_LAST_DOC_ID}')
    return jsonify({'success': True})


@app.route('/ctrl_f', methods=['GET'])
def ctrl_f():
    if _LAST_DOC_ID is None:
        return jsonify({'success': False, 'answers': []})
    query_text = request.args['query']
    answers = _CAPE_CLIENT.answer(query_text,
                                  _ANSWER_TOKEN,
                                  document_ids=[_LAST_DOC_ID],
                                  number_of_items=5)
    print(f'answers: {answers}')
    return jsonify({'success': True,'answers': answers})


if __name__ == '__main__':
    app.run(port='5050')
```

Our html file, *templates/index.html* is also very basic:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.
↪2/html/bootstrap.min.html"
          integrity="sha384-
↪PsH8R72JQ3SOdhVi3uxftmaW6Vc51MKb0q5P2rRUpPvrszuE4W1povHYgTpBfshb" crossorigin=
↪"anonymous">
    <link rel="stylesheet" href="/static/style.html">
    <meta charset="UTF-8">
    <title>Basic AI Powered Ctrl+F Demo</title>
</head>
<body>
<div class="container">
    <div class="col">
        <h1 class="display-1">Cape Ctrl+F Demo</h1>
```

```
        <p class="text-muted lead">This super-powered Ctrl+F demo was built using␣
↪Cape API. View the tutorial <a
                href="#">here.</a></p>
        <div class="form-group">
            <input type="search" class="form-control mb-3" id="ctrlfField"␣
↪placeholder="ctrl+f search bar"/>
            <div class="form-control" id="documentText" contenteditable="True">
↪Football is a family of team sports that
                involve, to varying degrees, kicking a ball with the foot to score a␣
↪goal. Unqualified, the word
                football is understood to refer to whichever form of football is the␣
↪most popular in the regional
                context in which the word appears. Sports commonly called 'football'␣
↪in certain places include:
                association football (known as soccer in some countries); gridiron␣
↪football (specifically American
                football or Canadian football); Australian rules football; rugby␣
↪football (either rugby league or rugby
                union); and Gaelic football.[1][2] These different variations of␣
↪football are known as football codes.
            </div>
        </div>
    </div>
</div>

<script src="https://code.jQuery.com/jQuery-3.2.1.min.js"
        integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js
↪"
        integrity="sha384-
↪vFJXuSJphROIrBnz7yo7oB41mKfc8JzQZiCq4NCceLEaO4IHwicKwpJf9c9IpFgh"
        crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.
↪js"
        integrity="sha384-
↪alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ"
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/mark.js/8.11.0/jQuery.mark.es6.
↪min.js"></script>
<script src="/static/app.js"></script>
</body>
</html>
```

Our JavaScript is only a few lines long:

```
$(document).ready(function () {
    var doc_text_selector = $('#documentText');
    $.post('/add_document', {'doc': doc_text_selector.text()}); // initialise doc
    var myTimeout = null;
    doc_text_selector.bind('input propertychange', function () {
        $.post('/add_document', {'doc': $(this).text()});
    });
    $('#ctrlfField').bind('input propertychange', function (e) {
        e.preventDefault();
        $(this).addClass('loading');
        if (typeof(myTimeout) !== "undefined") {
            clearTimeout(myTimeout);
```

```
        }
        myTimeout = setTimeout(function () {
            $.get('/ctrl_f', {'query': $('#ctrlfField').val()}, function (data) {
                    var answers = data.answers;
                    var answer = {};
                    var range = [];
                    var doc_text = $('#documentText');
                    doc_text.unmark();
                    for (i = 0; i < answers.length; i++) {
                        answer = answers[i];
                        range = {'start': answer.startTextOffset, 'length': (answer.
↪endTextOffset - answer.startTextOffset)};
                        if (i === 0) {
                            doc_text.markRanges([range], {element: 'span', className:
↪'success'})
                        } else if (i < 4) {
                            doc_text.markRanges([range], {element: 'span', className:
↪'info'})
                        } else {
                            doc_text.markRanges([range], {element: 'span', className:
↪'danger'})
                        }
                    }
                    $('#ctrlfField').removeClass('loading');
                }
            );
        }, 1000);
        return false;
    });
});
```

And our stylesheet even shorter:

```
.success {
    background: #86f3a0;
}

.info {
    background: rgba(23, 162, 184, 0.15);
}

.danger {
    background: rgba(23, 162, 184, 0.05);
}

.loading {
    background-color: #ffffff;
    background-image: url("http://loadinggif.com/images/image-selection/3.gif");
    background-size: 25px 25px;
    background-position:right center;
    background-repeat: no-repeat;
}
```

We can now run the whole thing by typing `python3 app.py` in the root of the directory and you are done!

- *API Documentation*
- genindex

- search

CHAPTER 3

cape

3.1 cape.client package

3.1.1 Module contents

Cape API client module.

This module provides a python interface to the Cape API: http://thecape.ai

class cape.client.**CapeClient**(*api_base='https://responder.thecape.ai/api'*, *admin_token=None*)

The CapeClient provides access to all methods of the Cape API.

add_annotation(*question*, *answer*, *document_id*, *start_offset=None*, *end_offset=None*, *metadata=None*)

Create a new annotation for a specified document.

Annotations are made up of a pair consisting of a canonical question, the response it should produce and a location within a specific document that this answer corresponds to.

In addition to the canonical question an annotation may have many paraphrased questions associated with it which should produce the same answer (e.g. "How old are you?" vs "What is your age?").

Parameters

- **question** – The question this annotation relates to.
- **answer** – The answer to reply with when the question is asked.
- **document_id** – The document which this annotation corresponds to.
- **start_offset** – The starting location of the annotation within the specified document.
- **end_offset** – The ending location of the annotation within the specified document.
- **metadata** – A dictionary containing user definable metadata about this annotation.

Returns The IDs of the new annotation and answer.

add_annotation_answer(*annotation_id*, *answer*)
Add a new answer to an existing annotation.

Parameters

- **annotation_id** – The ID of the annotation to add this answer to.
- **answer** – The answer to add to the annotation.

Returns The ID of the answer that was created.

add_annotation_paraphrase_question(*annotation_id*, *question*)
Add a new paraphrase question to an existing annotation.

Parameters

- **annotation_id** – The ID of the annotation to add this question to.
- **question** – The new paraphrase of this annotation's canonical question.

Returns The ID of the new question.

add_answer(*reply_id*, *answer*)
Add a new answer to an existing saved reply.

Parameters

- **reply_id** – The ID of the saved reply to add this answer to.
- **answer** – A new answer to add to the saved reply.

Returns The ID of the newly created answer.

add_document(*title*, *text=None*, *file_path=None*, *document_id=''*, *origin=''*, *replace=False*, *document_type=None*, *monitor_callback=None*)
Create a new document or replace an existing document.

Parameters

- **title** – The title to give the new document.
- **text** – The plain text contents of the document (either text or file_path must be supplied).
- **file_path** – A file to upload (either text or file_path must be supplied).
- **document_id** – The ID to give the new document (Default: An SHA256 hash of the document contents).
- **origin** – Where the document came from.
- **replace** – If true and a document already exists with the same document ID it will be overwritten with the new upload. If false an error is returned when a document ID already exists.
- **document_type** – Whether this document was created by inputting text or uploading a file (if not set this will be automatically determined).
- **monitor_callback** – A method to call with updates on the file upload progress.

Returns The ID of the uploaded document.

add_paraphrase_question(*reply_id*, *question*)
Add a new paraphrase question to an existing saved reply.

Parameters

- **reply_id** – The ID of the saved reply to add this question to.
- **question** – The new paraphrase of this saved reply's canonical question.

Returns The ID of the new question.

add_saved_reply(*question*, *answer*)
Create a new saved reply.

Saved replies are made up of a pair consisting of a canonical question and the response it should produce. In addition to the canonical question a saved reply may have many paraphrased questions associated with it which should produce the same answer (e.g. "How old are you?" vs "What is your age?").

Parameters

- **question** – The question this saved reply relates to.
- **answer** – The answer to reply with when the question is asked.

Returns The IDs of the new saved reply and answer.

answer(*question*, *user_token=None*, *threshold=None*, *document_ids=None*, *source_type='all'*, *speed_or_accuracy='balanced'*, *number_of_items=1*, *offset=0*, *text=None*)
Provide a list of answers to a given question.

Parameters

- **question** – The question to ask.
- **user_token** – A token retrieved from get_user_token (Default: the token for the currently authenticated user).
- **threshold** – The minimum confidence of answers to return ('very-low'/'low'/'medium'/'medium'/'veryhigh').
- **document_ids** – A list of documents to search for answers (Default: all documents).
- **source_type** – Whether to search documents, saved replies or all ('document'/'saved_reply'/'all').
- **speed_or_accuracy** – Prioritise speed or accuracy in answers ('speed'/'accuracy'/'balanced').
- **number_of_items** – The number of answers to return.
- **offset** – The starting point in the list of answers, used in conjunction with number_of_items to retrieve multiple batches of answers.
- **text** – An inline text to be treated as a document with id "Inline Text".

Returns A list of answers.

archive_inbox(*inbox_id*)
Archive an inbox item.

Parameters **inbox_id** – The inbox item to archive.

Returns The ID of the inbox item that was archived.

delete_annotation(*annotation_id*)
Delete an annotation.

Parameters **annotation_id** – The ID of the annotation to delete.

Returns The ID of the annotation that was deleted.

delete_annotation_answer(*answer_id*)
Delete an answer from an annotation.

At least one answer must remain associated with an annotation.

Parameters **answer_id** – The answer to delete

Returns The ID of the answer that was deleted

delete_annotation_paraphrase_question(*question_id*)
Delete an annotation's paraphrase question.

Parameters **question_id** – The ID of the question to delete.

Returns The ID of the question that was deleted.

delete_answer(*answer_id*)
Delete an existing an answer.

Parameters **answer_id** – The ID of the answer to delete.

Returns The ID of the answer that was deleted.

delete_document(*document_id*)
Delete a document.

Parameters **document_id** – The ID of the document to delete.

Returns The ID of the document that was deleted.

delete_paraphrase_question(*question_id*)
Delete a paraphrase question.

Parameters **question_id** – The ID of the paraphrase question to delete.

Returns The ID of the paraphrase question that was deleted.

delete_saved_reply(*reply_id*)
Delete a saved reply.

Parameters **reply_id** – The ID of the saved reply to delete.

Returns The ID of the saved reply that was deleted.

edit_annotation_answer(*answer_id*, *answer*)
Edit an annotation's answer.

Parameters

- **answer_id** – The ID of the answer to edit.
- **answer** – The new text to be used for this answer.

Returns The ID of the answer that was edited.

edit_annotation_canonical_question(*annotation_id*, *question*)
Edit the canonical question of an annotation.

Parameters

- **annotation_id** – The ID of the annotation to edit.
- **question** – The new canonical question for this annotation.

Returns The ID of the annotation that was edited.

edit_annotation_paraphrase_question(*question_id*, *question*)
Modify an existing paraphrase question in an annotation.

Parameters

- **question_id** – The ID of the question to modify.
- **question** – The modified question text.

Returns The ID of the question that was modified.

edit_answer(*answer_id*, *answer*)
Modify an existing answer.

Parameters

- **answer_id** – The ID of the answer to edit.
- **answer** – The modified answer text.

Returns The ID of the answer that was modified.

edit_canonical_question(*reply_id*, *question*)
Modify the canonical question belonging to a saved reply.

Parameters

- **reply_id** – The ID of the saved reply to modify the canonical question of.
- **question** – The modified question text.

Returns The ID of the saved reply that was modified.

edit_paraphrase_question(*question_id*, *question*)
Modify an existing paraphrase question.

Parameters

- **question_id** – The ID of the question to modify.
- **question** – The modified question text.

Returns The ID of the question that was modified.

get_admin_token()
Retrieve the admin token for the currently logged in user.

Returns An admin token.

get_annotations(*search_term=''*, *annotation_ids=None*, *document_ids=None*, *pages=None*, *number_of_items=30*, *offset=0*)
Retrieve a list of annotations.

Parameters

- **search_term** – Filter annotations based on whether they contain the search term.
- **annotation_ids** – A list of annotations to return/search within (Default: all annotations).
- **document_ids** – A list of documents to return annotations from (Default: all documents).
- **pages** – A list of pages to return annotations from (Default: all pages).
- **number_of_items** – The number of annotations to return.
- **offset** – The starting point in the list of annotations, used in conjunction with number_of_tems to retrieve multiple batches of annotations.

Returns A list of annotations.

get_default_threshold()
Retrieve the default threshold used if one isn't explicitly specified when calling answer().

Returns The current default threshold (either 'verylow', 'low', 'medium', 'high' or 'veryhigh').

get_documents(*document_ids=None*, *number_of_items=30*, *offset=0*)
Retrieve this user's documents.

Parameters

- **document_ids** – A list of documents to return.
- **number_of_items** – The number of documents to return.
- **offset** – The starting point in the list of documents, used in conjunction with number_of_items to retrieve multiple batches of documents.

Returns A list of documents in reverse chronological order (newest first).

get_inbox(*read='both'*, *answered='both'*, *search_term=''*, *number_of_items=30*, *offset=0*)
Retrieve the items in the current user's inbox.

Parameters

- **read** – Filter messages based on whether they have been read.
- **answered** – Filter messages based on whether they have been answered.
- **search_term** – Filter messages based on whether they contain the search term.
- **number_of_items** – The number of inbox items to return.
- **offset** – The starting point in the list of inbox items, used in conjunction with number_of_tems to retrieve multiple batches of inbox items.

Returns A list of inbox items in reverse chronological order (newest first).

get_profile()
Retrieve the current user's profile.

Returns A dictionary containing the user's profile.

get_saved_replies(*search_term=''*, *saved_reply_ids=None*, *number_of_items=30*, *offset=0*)
Retrieve a list of saved replies.

Parameters

- **search_term** – Filter saved replies based on whether they contain the search term.
- **saved_reply_ids** – List of saved reply IDs to return.
- **number_of_items** – The number of saved replies to return.
- **offset** – The starting point in the list of saved replies, used in conjunction with number_of_tems to retrieve multiple batches of saved replies.

Returns A list of saved replies in reverse chronological order (newest first).

get_user_token()
Retrieve a user token suitable for making 'answer' requests.

Returns A user token.

logged_in()
Reports whether we're currently logged in.

Returns Whether we're logged in or not.

login(*login*, *password*)
Log in to the Cape API as an AI builder.

Parameters

- **login** – The username to log in with.
- **password** – The password to log in with.

Returns

logout()
Log out and clear the current session cookie.

Returns

mark_inbox_read(*inbox_id*)
Mark an inbox item as having been read.

Parameters **inbox_id** – The inbox item to mark as being read.

Returns The ID of the inbox item that was marked as read.

set_default_threshold(*threshold*)
Set the default threshold used if one isn't explicitly specified when calling answer().

Parameters **threshold** – The new default threshold to set, must be either 'verylow', 'low', 'medium', 'high' or 'veryhigh'.

Returns The new default threshold that's just been set.

set_forward_email(*email*)
Set the email address that emails which couldn't be answered automatically are forwarded to.

Parameters **email** – The new forward email address to set.

Returns The new forward email address that's just been set.

Python Module Index

C

Index