



# **pollux documentation**

*Release 1.0.1*

**Daniel McDonald**

**Jun 13, 2017**



<b>1</b>	<b>Creating projects and building corpora</b>	<b>3</b>
<b>2</b>	<b>Exploring corpora</b>	<b>5</b>
2.1	Querying the corpus . . . . .	5
2.2	Query languages . . . . .	5
2.3	Managing results . . . . .	5
2.4	Concordance . . . . .	5
2.5	Table . . . . .	5
2.6	Pivot . . . . .	5
2.7	Tree . . . . .	5
2.8	Chart . . . . .	5
<b>3</b>	<b>Configuration and settings</b>	<b>7</b>
3.1	Help . . . . .	7
<b>4</b>	<b>Creating projects and building corpora</b>	<b>9</b>
4.1	Creating a new project . . . . .	9
4.2	Adding a corpus . . . . .	10
4.3	Creating a Corpus object . . . . .	10
4.4	Pre-processing the data . . . . .	10
4.5	Manipulating a parsed corpus . . . . .	11
4.6	Counting key features . . . . .	11
<b>5</b>	<b>Interrogating corpora</b>	<b>13</b>
5.1	Introduction . . . . .	13
5.2	Search types . . . . .	15
5.3	Grammatical searching . . . . .	15
5.4	Excluding results . . . . .	16
5.5	What to show . . . . .	16
5.6	Working with trees . . . . .	17
5.7	Tree <i>show</i> values . . . . .	17
5.8	Working with dependencies . . . . .	18
5.9	Working with metadata . . . . .	18
5.10	Working with coreferences . . . . .	18
5.11	Multiprocessing . . . . .	19
5.12	N-grams . . . . .	19
5.13	Collocation . . . . .	19
5.14	Saving interrogations . . . . .	19
5.15	Exporting interrogations . . . . .	20
5.16	Other options . . . . .	20

<b>6</b>	<b>Editing results</b>	<b>21</b>
6.1	Keeping or deleting results and subcorpora	21
6.2	Editing result names	22
6.3	Spelling normalisation	22
6.4	Generating relative frequencies	22
6.5	Keywording	23
6.6	Sorting	23
6.7	Calculating trends, P values	24
6.8	Saving results	24
6.9	Exporting results	24
6.10	Next step	24
<b>7</b>	<b>Visualising results</b>	<b>25</b>
7.1	Basics	25
7.2	Plot type	26
7.3	Plot style	27
7.4	Figure and font size	27
7.5	Title and labels	27
7.6	Subplots	27
7.7	TeX	27
7.8	Legend	28
7.9	Colours	28
7.10	Saving figures	28
7.11	Other options	29
7.12	Multiplotting	29
<b>8</b>	<b>Managing projects</b>	<b>31</b>
8.1	Loading saved data	31
<b>9</b>	<b>Overview</b>	<b>33</b>
9.1	Objects	33
9.2	Commands	34
9.3	Prompt features	35
<b>10</b>	<b>Setup</b>	<b>37</b>
10.1	Dependencies	37
10.2	Accessing	37
10.3	The prompt	37
<b>11</b>	<b>Making projects and corpora</b>	<b>39</b>
11.1	Adding a corpus	39
11.2	Parsing a corpus	39
11.3	Tokenising, POS tagging and lemmatising	39
11.4	Working with metadata	40
11.5	Viewing corpus data	40
<b>12</b>	<b>Interrogating corpora</b>	<b>41</b>
12.1	Search examples	41
12.2	Working with metadata	42
12.3	Sampling a corpus	42
<b>13</b>	<b>Concordancing</b>	<b>45</b>
13.1	Customising appearance	45
13.2	Sorting	45
13.3	Colouring	45
13.4	Editing	46
13.5	Recalculating results from concordance lines	46
13.6	Working with metadata	46

<b>14</b>	<b>Annotating your corpus</b>	<b>47</b>
14.1	Tagging sentences . . . . .	47
14.2	Creating fields and values . . . . .	47
14.3	Removing annotations . . . . .	48
<b>15</b>	<b>Editing results</b>	<b>49</b>
15.1	The edit command . . . . .	49
15.2	Doing basic statistics . . . . .	49
15.3	Sorting results . . . . .	50
<b>16</b>	<b>Plotting</b>	<b>51</b>
<b>17</b>	<b>Settings and management</b>	<b>53</b>
17.1	Managing data . . . . .	53
17.2	Toggles and settings . . . . .	53
17.3	Switching to IPython . . . . .	54
17.4	Running scripts . . . . .	54
<b>18</b>	<b>Corpus classes</b>	<b>55</b>
18.1	<i>Corpus</i> . . . . .	55
18.2	<i>File</i> . . . . .	58
18.3	<i>LoadedCorpus</i> . . . . .	58
18.4	<i>Results</i> . . . . .	58



This site provides documentation for three related projects:

1. *pollux*: a web application for doing text analysis.
2. *corpkit*, a Python backend for pollux
3. *pollux-cl*, a command-line natural language interpreter

With *pollux*, you can create parsed, structured and metadata-annotated corpora, and then search them for complex lexicogrammatical patterns. Search results can be quickly edited, sorted and visualised, saved and loaded within projects, or exported to formats that can be handled by other tools. In fact, you can easily work with any dataset in [CONLL U](#) format, including the freely available, multilingual [Universal Dependencies Treebanks](#).

Concordancing is extended to allow the user to query and display grammatical features alongside tokens. Key-wording can be restricted to certain word classes or positions within the clause. If your corpus contains multiple documents or subcorpora, you can identify keywords in each, compared to the corpus as a whole.

## Installation

Via pip:

```
$ pip install pollux
```

via Git:

```
$ git clone https://www.github.com/interrogator/pollux
$ cd pollux
$ python setup.py install
```

Parsing and interrogation of parse trees will also require *Stanford CoreNLP*. *pollux* can download and install it for you automatically.

## Running the app

After installation, *pollux* can be started from the command line with:

```
# load sample project
$ pollux-quickstart
```

You can parse your own corpus from within the web app, or via the command line:

```
# parse
$ pollux-parse path/to/corpus
$ mkdir ~/corpora
# add to database
$ cp -R path/to/corpus-parsed ~/corpora
$ pollux-build
# open the tool
$ pollux
```

*pollux-cl* is a bit like the [Corpus Workbench](#). You can open it with:

```
$ pollux-cl
# or, alternatively:
$ python -m pollux.cl
```

And then start working with natural language commands:

```
> set junglebook as corpus
> parse junglebook with outname as jb
> set jb as corpus
> search corpus for deps matching "f/nsubj/ <- f/ROOT/"
> calculate result as percentage of self
> plot result as line chart with title as 'Example figure'
```

From the interpreter, you can enter `ipython`, `jupyter notebook` or `gui` to switch between interfaces, preserving the local namespace and data where possible.

Information about the syntax is available at the [Overview](#).

# CHAPTER 1

---

## Creating projects and building corpora

---

Coming soon!



## CHAPTER 2

---

### Exploring corpora

---

Coming soon!

**Querying the corpus**

**Query languages**

**Managing results**

**Concordance**

**Table**

**Pivot**

**Tree**

**Chart**



## CHAPTER 3

---

### Configuration and settings

---

Coming soon!

**Help**



---

## Creating projects and building corpora

---

Doing corpus linguistics involves building and interrogating corpora, and exploring interrogation results. `corpkit` helps with all of these things. This page will explain how to create a new project and build a corpus.

- *Creating a new project*
- *Adding a corpus*
- *Creating a Corpus object*
- *Pre-processing the data*
- *Manipulating a parsed corpus*
- *Counting key features*

### Creating a new project

The simplest way to begin using `corpkit` is to import it and to create a new project. Projects are simply folders containing subfolders where corpora, saved results, images and dictionaries will be stored. The simplest way is to do it is to use the `new_project` command in *bash*, passing in the name you'd like for the project as the only argument:

```
$ new_project psyc
# move there:
$ cd psyc
# now, enter python and begin ...
```

Or, from Python:

```
>>> import corpkit
>>> corpkit.new_project('psyc')
### move there:
>>> import os
>>> os.chdir('psyc')
>>> os.listdir('.')

['data',
```

```
'dictionaries',  
'exported',  
'images',  
'logs',  
'saved_concordances',  
'saved_interrogations']
```

## Adding a corpus

Now that we have a project, we need to add some plain-text data to the *data* folder. At the very least, this is simply a text file. Better than this is a folder containing a number of text files. Best, however, is a folder containing subfolders, with each subfolder containing one or more text files. These subfolders represent subcorpora.

You can add your corpus to the *data* folder from the command line, or using Finder/Explorer if you prefer.

```
$ cp -R /Users/me/Documents/transcripts ./data
```

Or, in *Python*, using *shutil*:

```
>>> import shutil  
>>> shutil.copytree('/Users/me/Documents/transcripts', './data')
```

If you've been using *bash* so far, this is the moment when you'd enter *Python* and `import corpkit`.

## Creating a Corpus object

Once we have a corpus of text files, we need to turn it into a *Corpus* object.

```
>>> from corpkit import Corpus  
### you can leave out the 'data' if it's in there  
>>> unparsed = Corpus('data/transcripts')  
>>> unparsed  
<corpkit.corpus.Corpora instance: transcripts; 13 subcorpora>
```

## Pre-processing the data

A *Corpus* object can only be interrogated if tokenisation or parsing has been performed. For this, *corpkit.corpus.Corpora* objects have *tokenise()* and *parse()* methods. Tokenising is faster, simpler, and will work for more languages.

```
> corpus = unparsed.tokenise()  
# switch either to false to disable--but lemmatisation requires pos
```

Parsing relies on Stanford CoreNLP's parser, and therefore, you must have the parser and Java installed. *corpkit* will look around in your *PATH* for the parser, but you can also pass in its location manually with (e.g.) `corenlppath='users/you/corenlp'`. If it can't be found, you'll be asked if you want to download and install it automatically. Parsing has sensible defaults, and can be run with:

```
>>> corpus = unparsed.parse()
```

---

**Note:** Remember that parsing is a computationally intensive task, and can take a long time!

---

*corpkit* can also work with speaker IDs. If lines in your file contain capitalised alphanumeric names, followed by a colon (as per the example below), these IDs can be stripped out and turned into metadata features in the parsed dataset.

```
JOHN: Why did they change the signs above all the bins?
SPEAKER23: I know why. But I'm not telling.
```

To use this option, use the `speaker_segmentation` keyword argument:

```
>>> corpus = unparsed.parse(speaker_segmentation=True)
```

Tokenising or parsing creates a corpus that is structurally identical to the original, but with annotations in *CONLL-U* formatted files in place of the original `.txt` files. When parsing, there are also methods for multiprocessing, memory allocation and so on:

<code>parse()</code> argument	Type	Purpose
<code>corenlppath</code>	<i>str</i>	Path to CoreNLP
<code>operations</code>	<i>str</i>	<a href="#">List of annotations</a>
<code>copula_head</code>	<i>bool</i>	Make copula head of dependency parse
<code>speaker_segmentation</code>	<i>bool</i>	Do speaker segmentation
<code>memory_mb</code>	<i>int</i>	Amount of memory to allocate
<code>multiprocess</code>	<i>int/bool</i>	Process in <i>n</i> parallel jobs
<code>outname</code>	<i>str</i>	Custom name for parsed corpus

You can run parsing operations from the command line:

```
$ parse mycorpus --multiprocess 4 --outname MyData
```

## Manipulating a parsed corpus

Once you have a parsed corpus, you're ready to analyse it. `corpkit.corpus.Corpus` objects can be navigated in a number of ways. *CoreNLP XML* is used to navigate the internal structure of *CONLL-U* files within the corpus.

```
>>> corpus[:3] # access first three subcorpora
>>> corpus.subcorpora.chapter1 # access subcorpus called chapter1
>>> f = corpus[5][20] # access 21st file in 6th subcorpus
```

## Counting key features

Before constructing your own queries, you may want to use some predefined attributes for counting key features in the corpus.

```
>>> corpus.features()
```

Output:

```
S Characters Tokens Words Closed class Open class Clauses Sentences Unmod.
↳declarative Passives Mental processes Relational processes Mod. declarative
↳Interrogative Verbal processes Imperative Open interrogative Closed interrogative
01 4380658 1258606 1092113 643779 614827 277103 68267
↳35981 16842 11570 11082 3691 5012
↳ 2962 615 787 813
02 3185042 922243 800046 471883 450360 209448 51575
↳26149 10324 8952 8407 3103 3407
↳ 2578 540 547 461
03 3157277 917822 795517 471578 446244 209990 51860
↳26383 9711 9163 8590 3438 3392
↳ 2572 583 556 452
04 3261922 948272 820193 486065 462207 216739 53995
↳27073 9697 9553 9037 3770 3702
↳ 2665 652 669 530
05 3164919 921098 796430 473446 447652 210165 52227
↳26137 9543 8958 8663 3622 3523
↳ 2738 633 571 467
```

06	3187420	928350	797652	480843	447507	209895	52171		
↪25096	8917		9011		8820		3913	3637	┌
↪	2722		686	553		480			┌
07	3080956	900110	771319	466254	433856	202868	50071		
↪24077	8618		8616		8547		3623	3343	┌
↪	2676		615	515		434			┌
08	3356241	972652	833135	502913	469739	218382	52637		
↪25285	9921		9230		9562		3963	3497	┌
↪	2831		692	603		442			┌
09	2908221	840803	725108	434839	405964	191851	47050		
↪21807	8354		8413		8720		3876	3147	┌
↪	2582		675	554		455			┌
10	2868652	815101	708918	421403	393698	185677	43474		
↪20763	8640		8067		8947		4333	3181	┌
↪	2727		584	596		424			┌

This can take a while, as it counts a number of complex features. Once it's done, however, it saves automatically, so you don't need to do it again. There are also `postags`, `wordclasses` and `lexicon` attributes, which behave similarly:

```
>>> corpus.postags ()
>>> corpus.wordclasses ()
>>> corpus.lexicon ()
```

These results can be useful when generating relative frequencies later on. Right now, however, you're probably interested in searching the corpus yourself, however. Hit *Next* to learn about that.

---

## Interrogating corpora

---

Once you've built a corpus, you can search it for linguistic phenomena. This is done with the `interrogate()` method.

- *Introduction*
- *Search types*
- *Grammatical searching*
- *Excluding results*
- *What to show*
- *Working with trees*
- *Tree show values*
- *Working with dependencies*
- *Working with metadata*
- *Working with coreferences*
- *Multiprocessing*
- *N-grams*
- *Collocation*
- *Saving interrogations*
- *Exporting interrogations*
- *Other options*

### Introduction

Interrogations can be performed on any `corpkit.corpus.Corpora` object, but also, on `corpkit.corpus.Subcorpus` objects, `corpkit.corpus.File` objects and `corpkit.corpus.Datalist` objects (slices

of Corpus objects). You can search plaintext corpora, tokenised corpora or fully parsed corpora using the same method. We'll focus on parsed corpora in this guide.

```
>>> from corpkit import *
### words matching 'woman', 'women', 'man', 'men'
>>> query = {W: r'/(^wo)m.n/'}
### interrogate corpus
>>> corpus.search(query)
### interrogate parts of corpus
>>> corpus[2:4].search(query)
>>> corpus.files[:10].search(query)
### if you have a subcorpus called 'abstract':
>>> corpus.subcorpora.abstract.search(query)
```

Corpus interrogations will output a `corpkit.interrogation.Interrogation` object, which stores a DataFrame of results, a Series of totals, a dict of values used in the query, and, optionally, a set of concordance lines. Let's search for proper nouns in *The Great Gatsby* and see what we get:

```
>>> corp = Corpus('gatsby-parsed')
### turn on concordancing:
>>> propnoun = corp.search('p', '^NNP')
>>> propnoun.table()

      gatsby  tom  daisy  mr.  wilson  jordan  new  baker  york  miss
chapter1    12   32   29    4     0     2   10   21    6   19
chapter2     1   30    6    8    26     0    6    0    6    0
chapter3    28    0    1    8     0    22    5    6    5    1
chapter4    38   10   15   25    1     9    5    8    4    7
chapter5    36    3   26    4     0     0    1    1    1    1
chapter6    37   21   19   11    0     1    4    0    3    4
chapter7    63   87   60    9    27    35    9    2    5    1
chapter8    21    3   19    1    19     1    0    1    0    0
chapter9    27    5    9   14     4     3    4    1    4    1

>>> propnoun.table().sum()

chapter1    232
chapter2    252
chapter3    171
chapter4    428
chapter5    128
chapter6    219
chapter7    438
chapter8    139
chapter9    208
dtype: int64

>>> propnoun.conc() # (sample)

54 chapter1          They had spent a year in  france          for no particular reason and_
   ↪then d
55 chapter1  n't believe it I had no sight into  daisy          's heart but i felt that tom_
   ↪would
56 chapter1  into Daisy 's heart but I felt that  tom          would drift on forever seeking_
   ↪a li
57 chapter1          This was a permanent move said  daisy          over the telephone but i did n
   ↪'t be
58 chapter1  windy evening I drove over to East  egg          to see two old friends whom i_
   ↪scarc
59 chapter1  warm windy evening I drove over to  east          egg to see two old friends whom_
   ↪i s
60 chapter1  d a cheerful red and white Georgian  colonial          mansion overlooking the bay
61 chapter1  pen to the warm windy afternoon and  tom          buchanan in riding clothes was_
   ↪stan
62 chapter1  to the warm windy afternoon and Tom  buchanan          in riding clothes was standing_
   ↪with
```

Cool, eh? We'll focus on what to do with these attributes later. Right now, we need to learn how to generate them.

## Search types

Parsed corpora contain many different kinds of things we might like to search. There are word forms, lemma forms, POS tags, word classes, indices, and constituency and (three different) dependency grammar annotations. For this reason, the search query is a `dict` object passed to the `interrogate()` method, whose keys specify what to search, and whose values specify a query. The simplest ones are given in the table below.

**Note:** Single capital letter variables in code examples represent lowercase strings (`W = 'w'`). These variables are made available by doing `from corpkit import *`. They are used here for readability.

Search	Gloss
W	Word
L	Lemma
F	Function
P	POS tag
X	XPOS
E	NER tag
I	Index in sentence
S	Sentence index
R	Coref representative

Because it comes first, and because it's always needed, you can pass it in like an argument, rather than a keyword argument.

```
### get variants of the verb 'be'
>>> corpus.search({L: 'be'})
### get words in 'nsubj' position
>>> corpus.search({F: 'nsubj'})
```

Multiple key/value pairs can be supplied. By default, all must match for the result to be counted, though this can be changed with `searchmode=ANY` or `searchmode=ALL`:

```
>>> goverb = {P: r'^v', L: r'^go'}
### get all variants of 'go' as verb
>>> corpus.search(goverb, searchmode=ALL)
### get all verbs and any word starting with 'go':
>>> corpus.search(goverb, searchmode=ANY)
```

## Grammatical searching

In the examples above, we match attributes of tokens. The great thing about parsed data, is that we can search for relationships between words. So, other possible search keys are:

Search	Gloss
D	Dependency (depgerep)
T	Syntax tree

```
>>> q = {G: r'^b'}
### return any token with governor word starting with 'b'
>>> corpus.search(q)
```

*Governor*, *Dependent* and *Left/Right* can be combined with the earlier table, allowing a large array of search types:

	Match	Governor	Dependent	Coref head	Left/right
Word	W	G	D	H	A1/Z1
Lemma	L	GL	DL	HL	A1L/Z1L
Function	F	GF	DF	HF	A1F/Z1F
POS tag	P	GP	DP	HP	A1P/Z1P
Word class	X	GX	DX	HX	A1X/Z1X
Distance from root	A	GA	DA	HA	A1A/Z1A
Index	I	GI	DI	HI	A1I/Z1I
Sentence index	S	GS	DS	HS	A1S/Z1S

Syntax tree searching can't be combined with other options. We'll return to them in a minute, however.

## Excluding results

You may also wish to exclude particular phenomena from the results. The `exclude` argument takes a dict in the same form as a `search`. By default, if any key/value pair in the `exclude` argument matches, it will be excluded. This is controlled by `excludemode=ANY` or `excludemode=ALL`.

```
>>> from corpkit.dictionaries import wordlists
### get any noun, but exclude closed class words
>>> corpus.pos(r'^n')
```

In many cases, rather than using `exclude`, you could also remove results later, during editing.

## What to show

Up till now, all searches have simply returned words. The final major argument of the `interrogate` method is `show`, which dictates what is returned from a search. Words are the default value. You can use any of the search values as a `show` value. `show` can be either a single string or a list of strings. If a list is provided, each value is returned with forward slashes as delimiters.

```
>>> example = corpus.search({W: r'fr?iends?'}, show=[W, L, P])
>>> list(example.results)

['friend/friend/nn', 'friends/friend/nns', 'fiend/fiend/nn', 'fiends/fiend/nns', ... ]
```

Unigrams are generated by default. To get n-grams, pass in an n value as `gramsize`:

```
>>> example = corpus.search({W: r'wom[ae]n'}, show=N, gramsize=2)
>>> list(example.results)

['a/woman', 'the/woman', 'the/women', 'women/are', ... ]
```

So, this leaves us with a huge array of possible things to show, all of which can be combined if need be:

	Match	Governor	Dependent	Coref Head	1L position	1R position
Word	W	G	D	H	A1	Z1
Lemma	L	GL	DL	HL	A1L	Z1L
Function	F	GF	DF	HF	A1F	Z1F
POS tag	P	GP	DP	HP	A1P	Z1P
Word class	X	GX	DX	HX	A1X	Z1X
Distance from root	A	GA	DA	HA	A1A	Z1R
Index	I	GI	DI	HI	A1I	Z1I
Sentence index	S	GS	DS	HS	A1S	Z1S

One further extra `show` value is `'c'` (count), which simply counts occurrences of a phenomenon. Rather than returning a DataFrame of results, it will result in a single Series. It cannot be combined with other values.

## Working with trees

If you have elected to search trees, by default, searching will be done with Java, using Tregex. If you don't have Java, or if you pass in `tgrep=True`, searching will be done with the more limited Tgrep2 syntax. Here, we'll concentrate on Tregex.

Tregex is a language for searching syntax trees like this one:

To write a Tregex query, you specify *words and/or tags* you want to match, in combination with *operators* that link them together. First, let's understand the Tregex syntax.

To match any adjective, you can simply write:

```
JJ
```

with *JJ* representing adjective as per the [Penn Treebank tagset](#). If you want to get NPs containing adjectives, you might use:

```
NP < JJ
```

where `<` means *with a child/immediately below*. These operators can be reversed: If we wanted to show the adjectives within NPs only, we could use:

```
JJ > NP
```

It's good to remember that **the output will always be the left-most part of your query**.

If you only want to match Subject NPs, you can use bracketting, and the `$` operator, which means *sister/directly to the left/right of*:

```
JJ > (NP $ VP)
```

In this way, you build more complex queries, which can extend all the way from a sentence's *root* to particular tokens. The query below, for example, finds adjectives modifying *book*:

```
JJ > (NP <<# /book/)
```

Notice that here, we have a different kind of operator. The `<<` operator means that the node on the right does not need to be a child, but can be a descendant. the `#` means *head*—that is, in SFL, it matches the *Thing* in a Nominal Group.

If we wanted to also match *magazine* or *newspaper*, there are a few different approaches. One way would be to use `|` as an operator meaning *or*:

```
JJ > (NP ( <<# /book/ | <<# /magazine/ | <<# /newspaper/))
```

This can be cumbersome, however. Instead, we could use a regular expression:

```
JJ > (NP <<# /^ (book|newspaper|magazine) s*$/)
```

Though it is beyond the scope of this guide to teach Regular Expressions, it is important to note that Regular Expressions are extremely powerful ways of searching text, and are invaluable for any linguist interested in digital datasets.

Detailed documentation for Tregex usage (with more complex queries and operators) can be found [here](#).

## Tree show values

Though you can use the same Tregex query for tree searches, the output changes depending on what you select as the `show` value. For the following sentence:

```
These are prosperous times.
```

you could write a query:

```
r'JJ < __'
```

Which would return:

Show	Gloss	Output
W	Word	<i>prosperous</i>
T	Tree	<i>(JJ prosperous)</i>
p	POS tag	<i>JJ</i>
C	Count	<i>1 (added to total)</i>

## Working with dependencies

When working with dependencies, you can use any of the long list of search and *show* values. It's possible to construct very elaborate queries:

```
>>> from corpkit.dictionaries import process_types, roles
### nominal nsubj with verbal process as governor
>>> crit = {F: r'^nsubj$',
...         GL: processes.verbal.lemmata,
...         GF: roles.event,
...         P: r'^N'}
### interrogate corpus, outputting the nsubj lemma
>>> sayers = parsed.search(crit, show=L)
```

## Working with metadata

If you've used speaker segmentation and/or metadata addition when building your corpus, you can tell the *interrogate()* method to use these values as subcorpora, or restrict searches to particular values. The code below will limit searches to sentences spoken by Jason and Martin, or exclude them from the search:

```
>>> corpus.search(query, just_metadata={'speaker': ['JASON', 'MARTIN']})
>>> corpus.search(query, skip_metadata={'speaker': ['JASON', 'MARTIN']})
```

If you wanted to compare Jason and Martin's contributions in the corpus as a whole, you could treat them as subcorpora:

```
>>> corpus.search(query, subcorpora='speaker',
...               just_metadata={'speaker': ['JASON', 'MARTIN']})
```

The method above, however, will make an interrogation with two subcorpora, 'JASON' AND MARTIN. You can pass a list in as the *subcorpora* keyword argument to generate a multiindex:

```
>>> corpus.search(query, subcorpora=['folder', 'speaker'],
...               just_metadata={'speaker': ['JASON', 'MARTIN']})
```

## Working with coreferences

One major challenge in corpus linguistics is the fact that pronouns stand in for other words. Parsing provides coreference resolution, which maps pronouns to the things they denote. You can enable this kind of parsing by specifying the *dcoref* annotator:

```
>>> corpus = Corpus('example.txt')
>>> ops = 'tokenize,ssplit,pos,lemma,parse,ner,dcoref'
>>> parsed = corpus.search(operations=ops)
### print a plaintext representation of the parsed corpus
>>> print(parsed.plain)
```

```
0. Clinton supported the independence of Kosovo
1. He authorized the use of force.
```

If you have done this, you can use `coref=True` while interrogating to allow coreferent forms to be counted alongside query matches. For example, if you wanted to find all the processes Clinton is engaged in, you could do:

```
>>> from corpkit.dictionaries import roles
>>> query = {W: 'clinton', GF: roles.process}
>>> res = parsed.search(query, show=L, coref=True)
>>> res.results.columns
```

This matches both *Clinton* and *he*, and thus gives us:

```
['support', 'authorize']
```

## Multiprocessing

Interrogating the corpus can be slow. To speed it up, you can pass an integer as the `multiprocess` keyword argument, which tells the `interrogate()` method how many processes to create.

```
>>> corpus.search({T: r'__ > MD'}, multiprocess=4)
```

**Note:** Too many parallel processes may slow your computer down. If you pass in `multiprocessing=True`, the number of processes will equal the number of cores on your machine. This is usually a fairly sensible number.

## N-grams

N-gramming can be generated by making `gramsize > 1`:

```
>>> corpus.search({W: 'father'}, show='L', gramsize=3)
```

## Collocation

Collocations can be shown by making using `window`:

```
>>> corpus.search({W: 'father'}, show='L', window=6)
```

## Saving interrogations

```
>>> interro.save('savename')
```

Interrogation savenames will be prefaced with the name of the corpus interrogated.

You can also quicksave interrogations:

```
>>> corpus.search(T, r'/NN.?/', save='savename')
```

## Exporting interrogations

If you want to quickly export a result to CSV, LaTeX, etc., you can use Pandas' DataFrame methods:

```
>>> print(nouns.results.to_csv())
>>> print(nouns.results.to_latex())
```

## Other options

*interrogate()* takes a number of other arguments, each of which is documented in the API documentation.

If you're done interrogating, you can head to the page on [Editing results](#) to learn how to transform raw frequency counts into something more meaningful. Or, hit *Next* to learn about concordancing.

Corpus interrogation is the task of getting frequency counts for a lexicogrammatical phenomenon in a corpus. Simple absolute frequencies, however, are of limited use. The `edit()` method allows us to do complex things with our results, including:

- *Keeping or deleting results and subcorpora*
- *Editing result names*
- *Spelling normalisation*
- *Generating relative frequencies*
- *Keywording*
- *Sorting*
- *Calculating trends, P values*
- *Saving results*
- *Exporting results*
- *Next step*

Each of these will be covered in the sections below. Keep in mind that because results are stored as DataFrames, you can also use Pandas/Numpy/Scipy to manipulate your data in ways not covered here.

### Keeping or deleting results and subcorpora

One of the simplest kinds of editing is removing or keeping results or subcorpora. This is done using keyword arguments: `skip_subcorpora`, `just_subcorpora`, `skip_entries`, `just_entries`. The value for each can be:

1. A string (treated as a regular expression to match)
2. A list (a list of words to match)
3. An integer (treated as an index to match)

```
>>> criteria = r'ing$'
>>> result.edit(just_entries=criteria)
```

```
>>> criteria = ['everything', 'nothing', 'anything']
>>> result.edit(skip_entries=criteria)
```

```
>>> result.edit(just_subcorpora=['Chapter_10', 'Chapter_11'])
```

You can also span subcorpora, using a tuple of (`first_subcorpus`, `second_subcorpus`). This works for numerical and non-numerical subcorpus names:

```
>>> just_span = result.edit(span_subcorpora=(3, 10))
```

## Editing result names

You can use the `replace_names` keyword argument to edit the text of each result. If you pass in a string, it is treated as a regular expression to delete from every result:

```
>>> ingdel = result.edit(replace_names=r'ing$')
```

You can also pass in a dict with the structure of `{newname: criteria}`:

```
>>> rep = {'-ing words': r'ing$', '-ed words': r'ed$'}
>>> replaced = result.edit(replace_names=rep)
```

If you wanted to see how commonly words start with a particular letter, you could do something creative:

```
>>> from string import lowercase
>>> crit = {k.upper() + ' words': r'(?i)^%s.*' % k for k in lowercase}
>>> firstletter = result.edit(replace_names=crit, sort_by='total')
```

## Spelling normalisation

When results are single words, you can normalise to UK/US spelling:

```
>>> spelled = result.edit(spelling='UK')
```

You can also perform this step when interrogating a corpus.

## Generating relative frequencies

Because subcorpora often vary in size, it is very common to want to create relative frequency versions of results. The best way to do this is to pass in an operation and a denominator. The operation is simply a string denoting a mathematical operation: '+', '-', '\*', '/', '%'. The last two of these can be used to get relative frequencies and percentage.

Denominator is what the result will be divided by. Quite often, you can use the string `'self'`. This means, after all other editing (deleting entries, subcorpora, etc.), use the totals of the result being edited as the denominator. When doing no other editing operations, the two lines below are equivalent:

```
>>> rel = result.edit('%', 'self')
>>> rel = result.edit('%', result.totals)
```

The best denominator, however, may not simply be the totals for the results being edited. You may instead want to relativise by the total number of words:

```
>>> rel = result.edit('%', corpus.features.Words)
```

Or by some other result you have generated:

```
>>> words_with_oo = corpus.interrogate(W, 'oo')
>>> rel = result.edit('%', words_with_oo.totals)
```

There is a more complex kind of relative frequency making, where a `.results` attribute is used as the denominator. In the example below, we calculate the percentage of the time each verb occurs as the *root* of the parse.

```
>>> verbs = corpus.interrogate(P, r'^vb', show=L)
>>> roots = corpus.interrogate(F, 'root', show=L)
>>> relv = verbs.edit('%', roots.results)
```

## Keywording

`corpkit` treats keywording as an editing task, rather than an interrogation task. This makes it easy to get key nouns, or key Agents, or key grammatical features. To do keywording, use the `K` operation:

```
>>> from corpkit import *
### * imports predefined global variables like K and SELF
>>> keywords = result.edit(K, SELF)
```

This finds out which words are key in each subcorpus, compared to the corpus as a whole. You can compare subcorpora directly as well. Below, we compare the `plays` subcorpus to the `novels` subcorpus.

. code-block:: python

```
>>> from corpkit import *
>>> keywords = result.edit(K, result.ix['novels'], just_subcorpora='plays')
```

You could also pass in word frequency counts from some other source. A wordlist of the *British National Corpus* is included:

```
>>> keywords = result.edit(K, 'bnc')
```

The default keywording metric is *log-likelihood*. If you'd like to use *percentage difference*, you can do:

```
>>> keywords = result.edit(K, 'bnc', keyword_measure='pd')
```

## Sorting

You can sort results using the `sort_by` keyword. Possible values are:

- *'name'* (alphabetical)
- *'total'* (most common first)
- *'infreq'* (inverse total)
- *'increase'* (most increasing)
- *'decrease'* (most decreasing)
- *'turbulent'* (by most change)
- *'static'* (by least change)
- *'p'* (by p value)
- *'slope'* (by slope)
- *'intercept'* (by intercept)

- ‘*r*’ (by correlation coefficient)
- ‘*stderr*’ (by standard error of the estimate)
- ‘<*subcorpus*>’ by total in <subcorpus>

```
>>> inc = result.edit(sort_by='increase', keep_stats=False)
```

Many of these rely on Scipy’s `linregress` function. If you want to keep the generated statistics, use `keep_stats=True`.

## Calculating trends, P values

`keep_stats=True` will cause slopes, p values and `stderr` to be calculated for each result.

## Saving results

You can save edited results to disk.

```
>>> edited.save('savename')
```

## Exporting results

You can generate CSV data very easily using Pandas:

```
>>> result.results.to_csv()
```

## Next step

Once you’ve edited data, it’s ready to visualise. Hit next to learn how to use the `visualise()` method.

---

## Visualising results

---

One thing missing in a lot of corpus linguistic tools is the ability to produce high-quality visualisations of corpus data. `corpkit` uses the `corpkit.interrogation.Interrogation.visualise` method to do this.

- *Basics*
- *Plot type*
- *Plot style*
- *Figure and font size*
- *Title and labels*
- *Subplots*
- *TeX*
- *Legend*
- *Colours*
- *Saving figures*
- *Other options*
- *Multiplotting*

---

**Note:** Most of the keyword arguments from Pandas' `plot` method are available. See their documentation for more information.

---

### Basics

`visualise()` is a method of all `corpkit.interrogation.Interrogation` objects. If you use `from corpkit import *`, it is also monkey-patched to Pandas objects.

---

**Note:** If you're using a *Jupyter Notebook*, make sure you use `%matplotlib inline` or `%matplotlib`

---

notebook to set the appropriate backend.

---

A common workflow is to interrogate a corpus, relative results, and visualise:

```
>>> from corpkit import *
>>> corpus = Corpus('data/P-parsed', load_saved=True)
>>> counts = corpus.interrogate({T: r'MD < __'})
>>> reldat = counts.edit('%', SELF)
>>> reldat.visualise('Modals', kind='line', num_to_plot=ALL).show()
### the visualise method can also attach to the df:
>>> reldat.results.visualise(...).show()
```

The current behaviour of `visualise()` is to return the `pyplot` module. This allows you to edit figures further before showing them. Therefore, there are two ways to show the figure:

```
>>> data.visualise().show()
```

```
>>> plt = data.visualise()
>>> plt.show()
```

## Plot type

The `visualise` method allows line, bar, horizontal bar (`barh`), area, and pie charts. Those with *seaborn* can also use 'heatmap' ([docs](#)). Just pass in the type as a string with the `kind` keyword argument. Arguments such as `robust=True` can then be used.

```
>>> data.visualise(kind='heatmap', robust=True, figsize=(4,12),
...               x_label='Subcorpus', y_label='Event').show()
```



Fig. 7.1: Heatmap example

Stacked area/line plots can be made with `stacked=True`. You can also use `filled=True` to attempt to make all values sum to 100. Cumulative plotting can be done with `cumulative=True`. Below is an area plot beside an area plot where `filled=True`. Both use the `vidiris` colour scheme.



## Plot style

You can select from a number of styles, such as `ggplot`, `fivethirtyeight`, `bmh`, and `classic`. If you have *seaborn* installed (and you should), then you can also select from *seaborn* styles (`seaborn-paper`, `seaborn-dark`, etc.).

## Figure and font size

You can pass in a tuple of (`width`, `height`) to control the size of the figure. You can also pass an integer as `fontsize`.

## Title and labels

You can label your plot with `title`, `x_label` and `y_label`:

```
>>> data.visualise('Modals', x_label='Subcorpus', y_label='Relative frequency')
```

## Subplots

`subplots=True` makes a separate plot for every entry in the data. If using it, you'll probably also want to use `layout=(rows, columns)` to specify how you'd like the plots arranged.

```
>>> data.visualise(subplots=True, layout=(2,3)).show()
```

## TeX

If you have LaTeX installed, you can use `tex=True` to render text with LaTeX. By default, `visualise()` tries to use LaTeX if it can.



Fig. 7.2: Line charts using subplots and layout specification

## Legend

You can turn the legend off with `legend=False`. Legend placement can be controlled with `legend_pos`, which can be:

Margin	Figure		Margin
outside upper left	upper left	upper right	outside upper right
outside center left	center left	center right	outside center right
outside lower left	lower left	lower right	outside lower right

The default value, `'best'`, tries to find the best place automatically (without leaving the figure boundaries).

If you pass in `draggable=True`, you should be able to drag the legend around the figure.

## Colours

You can use the `colours` keyword argument to pass in:

1. A colour name recognised by *matplotlib*
2. A hex colour string
3. A colourmap object

There is an extra argument, `black_and_white`, which can be set to `True` to make greyscale plots. Unlike `colours`, it also updates line styles.

## Saving figures

To save a figure to a project's *images* directory, you can use the `save` argument. `output_format='png'` / `'pdf'` can be used to change the file format.

```
>>> data.visualise(save='name', output_format='png')
```

## Other options

There are a number of further keyword arguments for customising figures:

Argument	Type	Action
<i>grid</i>	<i>bool</i>	Show grid in background
<i>rot</i>	<i>int</i>	Rotate x axis labels n degrees
<i>shadow</i>	<i>bool</i>	Shadows for some parts of plot
<i>ncol</i>	<i>int</i>	n columns for legend entries
<i>explode</i>	<i>list</i>	Explode these entries in pie
<i>partial_pie</i>	<i>bool</i>	Allow plotting of pie slices
<i>legend_frame</i>	<i>bool</i>	Show frame around legend
<i>legend_alpha</i>	<i>float</i>	Opacity of legend
<i>reverse_legend</i>	<i>bool</i>	Reverse legend entry order
<i>transpose</i>	<i>bool</i>	Flip axes of DataFrame
<i>logx/logy</i>	<i>bool</i>	Log scales
<i>show_p_val</i>	<i>bool</i>	Try to show p value in legend
<i>interactive</i>	<i>bool</i>	Experimental <a href="#">mpld3</a> use

A number of these and other options for customising figures are also described in the `corpkit.interrogation.Interrogation.visualise` method documentation.

## Multiplotting

The `corpkit.interrogation.Interrogation` also comes with a `corpkit.interrogation.Interrogation.multiplot` method, which can be used to show two different kinds of chart within the same figure.

The first two arguments for the function are two *dict* objects, which configure the larger and smaller plots.

For the second dictionary, you may pass in a *data* argument, which is an `corpkit.interrogation.Interrogation` or similar, and will be used as separate data for the subplots. This is useful, for example, if you want your main plot to show absolute frequencies, and your subplots to show relative frequencies.

There is also *layout*, which you can use to choose an overall grid design. You can also pass in a list of tuples if you like, to use your own layout. Below is a complete example, focussing on objects in risk processes:

```
>>> from corpkit import *
>>> from corpkit.dictionaries import *
### parse a collection of text files
>>> corpora = Corus('data/news')
### make dependency parse query: get get 'object' of risk process
>>> query = {F: roles.participant2, GL: r'\brisk', GF: roles.process}
### interrogate corpus, return lemma form, no coreference
>>> result = corpus.interrogate(query, show=[L], coref=False)
### generate relative frequencies, skip closed class, and sort
>>> inc = result.edit('%', SELF,
>>>                  sort_by='increase',
>>>                  skip_entries=wordlists.closedclass)
### visualise as area and line charts combined
>>> inc.multiplot({'title': 'Objects of risk processes, increasing',
>>>               'kind': 'area',
>>>               'x_label': 'Year',
>>>               'y_label': 'Percentage of all results'},
>>>               {'kind': 'line'}, layout=5)
```



Fig. 7.3: *multiplot* example

---

## Managing projects

---

`corpkit` has a few other bits and pieces designed to make life easier when doing corpus linguistic work. This includes methods for loading saved data, for working with multiple corpora at the same time, and for switching between command line and graphical interfaces. Those things are covered here.

- *Loading saved data*

### Loading saved data

When you're starting a new session, you probably don't want to start totally from scratch. It's handy to be able to load your previous work. You can load data in a few ways.

First, you can use `corpkit.load()`, using the name of the filename you'd like to load. By default, `corpkit` looks in the `saved_interrogations` directory, but you can pass in an absolute path instead if you like.

```
>>> import corpkit
>>> nouns = corpkit.load('nouns')
```

Second, you can use `corpkit.loader()`, which provides a list of items to load, and asks the user for input:

```
>>> nouns = corpkit.loader()
```

Third, when instantiating a `Corpus` object, you can add `load_saved=True` keyword argument to load any saved data belonging to this corpus as an attribute.

```
>>> corpus = Corpus('data/psyc-parsed', load_saved=True)
```

A final alternative approach stores all interrogations within an `corpkit.interrogation.Interrodict` object object:

```
>>> r = corpkit.load_all_results()
```



---

 Overview
 

---

*corpkit* comes with a dedicated interpreter, which receives commands in a natural language syntax like these:

```
> set mydata as corpus
> search corpus for pos matching 'JJ.*'
> call result 'adjectives'
> edit adjectives by skipping subcorpora matching 'books'
> plot edited as line chart with title as 'Adjectives'
```

It's a little less powerful than the full Python API, but it is easier to use, especially if you don't know Python. You can also switch instantly from the interpreter to the full API, so you only need the API for the really tricky stuff.

The syntax of the interpreter is based around *objects*, which you do things to, and *commands*, which are actions performed upon the objects. The example below uses the *search* command on a *corpus* object, which produces new objects, called *result*, *concordance*, *totals* and *query*. As you can see, very complex searches can be performed using an English-like syntax:

```
> search corpus for lemma matching '^t' and pos matching 'VB' \
...     excluding words matching 'try' \
...     showing word and dependent-word \
...     with preserve_case
> result
```

This shows us results for each subcorpus:

.	I/think	I/thought	and/turned	me/told	and/took	I/told	...
chapter1	5	3	2	2	1	3	...
chapter2	7	2	5	3	0	2	...
chapter3	5	5	4	4	1	0	...
chapter4	3	7	1	0	3	1	...
chapter5	7	7	2	1	4	2	...
chapter6	2	0	0	2	1	0	...
chapter7	6	2	6	1	1	3	...
chapter8	3	1	2	2	1	1	...
chapter9	5	7	1	4	6	3	...

## Objects

The most common objects you'll be using are:

Object	Contains
<i>corpus</i>	Dataset selected for parsing or searching
<i>result</i>	Search output
<i>edited</i>	Results after sorting, editing or calculating
<i>concordance</i>	Concordance lines from search
<i>features</i>	General linguistic features of corpus
<i>wordclasses</i>	Distribution of word classes in corpus
<i>postags</i>	Distribution of POS tags in corpus
<i>lexicon</i>	Distribution of lexis in the corpus
<i>figure</i>	Plotted data
<i>query</i>	Values used to perform search or edit
<i>previous</i>	Object created before last
<i>sampled</i>	A sampled corpus
<i>wordlists</i>	A list of wordlists for searching, editing

When you start the interpreter, these are all empty. You'll need to run commands in order to fill them with data. You can also create your own object names using the `call` command.

## Commands

You do things to the objects via commands. Each command has its own syntax, designed to be as similar to natural language as possible. Below is a table of common commands, an explanation of their purpose, and an example of their syntax

Com- mand	Purpose	Syntax
<i>new</i>	Make a new project	<i>new project &lt;name&gt;</i>
<i>set</i>	Set current corpus	<i>set &lt;corpusname&gt;</i>
<i>parse</i>	Parse corpus	<i>parse corpus with [options]*</i>
<i>search</i>	Search a corpus for linguistic feature, generate concordance	<i>search corpus for [feature matching pattern]* showing [feature]* with [options]*</i>
<i>edit</i>	Edit results or edited results	<i>edit result by [skipping subcorpora/entries matching pattern]* with [options]*</i>
<i>calculate</i>	Calculate relative frequencies, keyness, etc.	<i>calculate result/edited as operation of denominator</i>
<i>sort</i>	Sort results or concordance	<i>sort result/concordance by value</i>
<i>plot</i>	Visualise result or edited result	<i>plot result/edited as line chart with [options]*</i>
<i>show</i>	Show any object	<i>show object</i>
<i>annotate</i>	Add annotations to corpus based on search results	<i>annotate all with field as &lt;fieldname&gt; and value as m</i>
<i>unannotate</i>	Delete annotation fields from corpus	<i>unannotate &lt;fieldname&gt; field</i>
<i>sample</i>	Get a random sample of subcorpora or files from a corpus	<i>sample 5 subcorpora of corpus</i>
<i>call</i>	Name an object (i.e. make a variable)	<i>call object '&lt;name&gt;'</i>
<i>export</i>	Export result, edited result or concordance to string/file	<i>export result to string/csv/latex/file &lt;filename&gt;</i>
<i>save</i>	Save data to disk	<i>save object to &lt;filename&gt;</i>
<i>load</i>	Load data from disk	<i>load object as result</i>
<i>store</i>	Store something in memory	<i>store object as &lt;name&gt;</i>
<i>fetch</i>	Fetch something from memory	<i>fetch &lt;name&gt; as object</i>
<i>help</i>	Get help on an object or command	<i>help command/object</i>
<i>history</i>	See previously entered commands	<i>history</i>
<i>ipython</i>	Enter IPython with objects available	<i>ipython</i>
<i>py</i>	Execute Python code	<i>py 'print("hello world")'</i>
<i>!</i>	Run a line of bash shell	<i>!ls -al data</i>

In square brackets with asterisks are recursive parts of the syntax, which often also accept *not* operators. *<text>* denotes places where you can choose an identifier, filename, etc.

In the pages that follow, the syntax is provided for the most common commands. You can also type the name of the command with no arguments into the interpreter, in order to show usage examples.

## Prompt features

- You can use *history*, *clear*, *ls* and *cd* commands as you would in the shell
- You can execute arbitrary bash commands by beginning the line with an exclamation point (e.g. `!rm data/*`)
- You can use semicolons to put multiple commands on a line (currently needs a space **before and after** the semicolon)
- There is no piping or output redirection (yet), but you can use the *export* and *save* commands to export results
- You can use backslashes to continue writing on the next line
- You can write scripts and pass them to the *corpkit* interpreter

The below is therefore a possible (but terrible) way to write code in *corpkit*:

```
> !du -h data ; set mycorp ; search corpus for words \  
... matching any \  
... excluding wordlists.closedclass \  
... showing lemma and pos ; concordance
```



- *Dependencies*
- *Accessing*
- *The prompt*

## Dependencies

To use the interpreter, you'll need *corpkit* installed. To use all features of the interpreter, you will also need *readline* and *IPython*.

## Accessing

With *corpkit* installed, you can start the interpreter in a couple of ways:

```
$ corpkit
# or
$ python -m corpkit.env
```

You can start it from a Python session, too:

```
>>> from corpkit import env
>>> env()
```

## The prompt

When using the interpreter, the prompt (the text to the left of where you type your command) displays the directory you are in (with an asterisk if it does not appear to be a *corpkit* project) and the currently active corpus, if any:

```
corpkit@junglebook:no-corpus>
```

When you see it, *corpkit* is ready to accept commands!

---

## Making projects and corpora

---

The first two things you need to do when using *corpkit* are to create a project, and to create (and optionally parse) a corpus. These steps can all be accomplished quickly using shell commands. They can also be done using the interpreter, however.

Once you're in *corpkit*, the command below will create a new project called *iran-news*, and move you into it.

```
> new project named iran-news
```

### Adding a corpus

Adding a corpus simply copies it to the project's data directory. The syntax is simple:

```
> add '../..my_corpus'
```

### Parsing a corpus

To parse a text file, folder of text files, or folder of folder of text files, you first `set` the corpus, and then use the `parse` command:

```
> set my_corpus as corpus
> parse corpus
```

### Tokenising, POS tagging and lemmatising

If you don't want/need full parses, or if you aren't working with English, you might want to use the `tokenise` method.

```
> set abstracts as corpus
> tokenise corpus
```

POS tagging and lemmatisation are switched on by default, but you could also disable them:

```
> tokenise corpus with postag as false and lemmatise as false
```

## Working with metadata

Parsing/tokenising can be made way cooler when your data has some metadata in it. The metadata will be transferred over to the parsed version of the corpus, and then you can search or filter by metadata features, use metadata values as symbolic subcorpora, or display metadata alongside concordances.

Metadata should take the form of an XML tag at the end of a line, which could be a sentence or a paragraph:

```
I hope everyone is hanging in with this blasted heat. As we all know being hot, sticky,
stressed and irritated can bring on a mood swing super fast. So please make sure your
all taking your meds and try to stay out of the heat. <metadata username="Emz45"
totalposts="5063" currentposts="4051" date="2011-07-13" postnum="0" threadlength="1">
```

Then, parse with metadata:

```
> parse corpus with metadata
```

The parser output will look something like:

```
# sent_id 1
# parse=(ROOT (S (NP (PRP I)) (VP (VBP hope) (SBAR (S (NP (NN everyone)) (VP (VBZ is) (VP_
↪ (VBG hanging) (PP (IN in) (IN with) (NP (DT this) (VBN blasted) (NN heat)))))))) (. .)))
# speaker=Emz45
# totalposts=5063
# threadlength=1
# currentposts=4051
# stage=10
# date=2011-07-13
# year=2011
# postnum=0
1 1 I I PRP O 2 nsubj 0 1
1 2 hope hope VBP O 0 ROOT 1,5,11 _
1 3 everyone everyone NN O 5 nsubj 0 _
1 4 is be VBZ O 5 aux 0 _
1 5 hanging hang VBG O 2 ccomp 3,4,10 _
1 6 in in IN O 10 case 0 _
1 7 with with IN O 10 case 0 _
1 8 this this DT O 10 det 0 2
1 9 blasted blast VBN O 10 amod 0 2
1 10 heat heat NN O 5 nmod:with 6,7,8,9 2*
1 11 . . . O 2 punct 0 _
```

## Viewing corpus data

You can interactively work with the parser output.

```
> get file <n> of corpus
```

Or, if your corpus has subcorpora:

```
> get subcorpus <n> of corpus
> get file <n> of sampled
```

This view can be surprisingly powerful: sorting by lemma, POS or dependency function can show you some recurring lexicogrammatical patterns in a file without the need for searching.

The next page will show you how to search the corpus you've built, and to work with metadata if you've added it.

---

## Interrogating corpora

---

The most powerful thing about *corpkit* is its ability to search parsed corpora for very complex constituency, dependency or token level features.

Before we begin, make sure you've set the corpus as the thing to search:

```
> set nyt-parsed as corpus
# you could also try just typing `set` ...
```

---

**Note:** By default, when using the interpreter, searching also produces concordance lines. If you don't need them, you can use `toggle conc` to switch them off, or on again. This can dramatically speed up processing time.

---

### Search examples

```
> search corpus ### interactive search helper
> search corpus for words matching "."*
> search corpus for words matching "[A-M]" showing lemma and word with case_sensitive
> search corpus for cql matching '[pos="DT"] [pos="NN"]' showing pos and word with coref
> search corpus for function matching roles.process showing dependent-lemma
> search corpus for governor-lemma matching processes.verbal showing governor-lemma, lemma
> search corpus for words matching any and not words matching wordlists.closedclass
> search corpus for trees matching '/NN.?/ >># NP'
> search corpus for pos matching NNP showing ngram-word and pos with gramsizes as 3
> etc.
```

Under the surface, what you are doing is selecting a *Corpus* object to search, and then generating arguments for the `interrogate()` method. These arguments, in order, are:

1. *search* criteria
2. *exclude* criteria
3. *show* values
4. Keyword arguments

Here is a syntax example that might help you see how the command gets parsed. Note that there are two ways of setting *exclude* criteria.

```
> search corpus \                               # select object
... for words matching 'ing$' and \           # search criterion
... not lemma matching 'being' and \         # exclude criterion
... pos matching 'NN' \                       # seach criterion
... excluding words matching wordlists.closedclass \ # exclude criterion
... showing lemma and pos and function \     # show values
... with preserve_case and \                 # boolean keyword arg
... not no_punct and \                       # bool keyword arg
... excludemode as 'all'                     # keyword arg
```

## Working with metadata

By default, *corpkit* treats folders within your corpus as subcorpora. If you want to treat files, rather than folders, as subcorpora, you can do:

```
> search corpus for words matching 'ing$' with subcorpora as files
```

If you have metadata in your corpus, you can use the metadata value as the subcorpora:

```
> search corpus for words matching 'ing$' with subcorpora as speaker
```

If you don't want to keep specifying the subcorpus structure every time you search a corpus, you have a couple of choices. First, you can set the default subcorpus value with `for` for the session with `set subcorpora`. This applies the filter globally, to whatever corpus you search:

```
# use speaker metadata as subcorpora
> set subcorpora as speaker
# ignore folders, use files as subcorpora
> set subcorpora as files
```

You can also define metadata filters, which skip sentences matching a metadata feature, or which keep only sentences matching a metadata feature:

```
# if you have a `year` metadata field, skip this decade
> set skip year as '^201'
# if you want only this decade:
> set keep year as '^201'
```

If you want to set subcorpora and filters for a corpus, rather than globally, you can do this by passing in the values when you select the corpus:

```
> set mydata-parsed as corpus with year as subcorpora and \
... just year as '^201' and skip speaker as 'chomsky'
# forget filters for this corpus:
> set mydata-parsed
```

## Sampling a corpus

Sometimes, your corpus is too big to search quickly. If this is the case, you can use the `sample` command to create a randomise sample of the corpus data:

```
> sample 3 subcorpora of corpus
> sample 100 files of corpus
```

If you pass in a float, it will try to get a proportional amount of data: `sample 0.33 subcorpora of corpus` will return a third of the subcorpora in the corpus.

A sampled corpus becomes an object called `sampld`. You can then refer to it when searching:

```
> search sampled for words matching '^[abcde]'
```

Global metadata filters and subcorpus declarations will be observed when searching this corpus as well.



By default, every search also produces concordance lines. You can view them by typing `concordance`. This opens an interactive display, which can be scrolled and searched—hit `h` to get help on possible commands.

### Customising appearance

The first thing you might want to do is adjust how concordance lines are displayed:

```
# hide subcorpus name, speaker name
> show concordance with columns as lmr
# enlarge window
> show concordance with columns as lmr and window as 60
# limit number of results to 100
> show concordance with columns as lmr and window as 60 and n as 100
```

The values you enter here are persistent—the window size, number of lines, etc. will remain the same until you shut down the interpreter or provide new values.

### Sorting

Sorting can be by column, or by word.

```
# middle column, first word
> sort concordance by M1
# left column, last word
> sort concordance by L1
# right column, third word
> sort concordance by R3
# by index (original order)
> sort concordance by index
```

### Colouring

One nice feature is that concordances can be coloured. This can be done through either indexing or regular expression matches. Note that `background` can be added to colour the background instead of the foreground, and

`dim/bright` can be used to adjust text brightness. This means that you can code lines for multiple phenomena. Background highlighting could mark the argument structure, foreground highlighting could mark the mood type, and bright and dim could be used to mark exemplars or false positives.

---

**Note:** If you're using Python 2, you may find that colouring concordance lines causes some interference with *readline*, making it difficult to select or search previous commands. This is a limitation of *readline* in Python 2. Use Python 3 instead!

---

```
# colour by index
> mark 10 blue
> mark -10 background red
> mark 10-15 cyan
> mark 15- magenta
# reset all
> mark - reset
```

```
# regular expression methods: specify column(s) to search
> mark m '^PRP.*' yellow
> mark r 'be(ing)' background green
> mark lm 'JJR$' dim
# reset via regex
> mark m '.*' reset
```

You can then sort by colour with *sort concordance by scheme*. If you export the concordances to a file (*export concordance as csv to conc.csv*), colour information will be added in additional columns.

## Editing

To edit concordance lines, you can use the same syntax as you would use to edit results:

```
> edit concordance by skipping subcorpora matching '[123]$\n'
> edit concordance by keeping entries matching 'PRP'
```

Perhaps faster is the use of *del* and *keep*. For these, specify the column and the criteria using the same methods as you would for colouring:

```
> del m matching 'this'
> keep l matching '^I\s'
> del 10-20
```

## Recalculating results from concordance lines

If you've deleted some concordance lines, you can update the `result` object to reflect these changes with *calculate result from concordance*.

## Working with metadata

You can use `show_conc_metadata` when interrogating or concordancing to collect and display metadata alongside concordance results:

```
> search corpus for words matching any with show_conc_metadata
> concordance
```

---

## Annotating your corpus

---

Another thing you might like to do is add metadata or annotations to your corpus. This can be done by simply editing corpus files, which are stored in a human-readable format. You can also automate annotation, however.

To do annotation, you first run a `search` command and generate a concordance. After deleting any false positives from the concordance, you can use the `annotate` command to annotate each sentence for which a concordance line exists.

`annotate`` works a lot like the ``mark`, `keep`, and `del` commands to begin with, but has some special syntax at the end, which controls whether you annotate using *tags*, or *fields and values*.

### Tagging sentences

The first way of annotating is to add a **tag** to one or more sentences:

```
> search corpus for pos matching NNP and word matching 'daisy'
> annotate m matching '^daisy$' with tag 'has_daisy'
```

You can use *all* to annotate every single concordance line:

```
> search corpus for governor-function matching nsubjpass \
... showing governor-lemma and lemma
> annotate all with tag 'passive'
```

If you try to run this code, you actually get a *dry run*, showing you what would be modified in your corpus. Once you're happy with it, you can do `toggle annotation` to turn file writing on, and then run the previous line again (use the up arrow to get it!).

### Creating fields and values

More complex than adding tags is adding **fields** and **values**. This creates a new metadata category with multiple possible realisations. Below, we tag an sentence sentences based on their containing certain kinds of processes

```
> search corpus for function matching roles.process showing lemma
> mark m matching processes.verbal red
# annotate by colour
> annotate red with field as process \
```

```
...    and value as 'verbal'  
# annotate without colouring first  
> annotate m matching processes.mental with field as process \  
...    and value as 'mental'
```

You can also use `m` as the value, which passes in the text from the middle column of the concordance.

```
> search corpus for pos matching NNP showing word  
> annotate m matching [gatsby, daisy, tom] \  
...    with field as character and value as m
```

The moment these values have been added to your text, you can do really powerful things with them. You can, for example, use them as subcorpora, or use them as filters for the sentences being processed.

```
> set subcorpora as process  
> set skip character as 'gatsby'  
> set skip passive tag
```

Now, the subcorpora will be the different processes (*verbal*, *mental* and *none*), and any sentence annotated as containing the `gatsby` character, or the passive tag, will be ignored.

## Removing annotations

To remove a tag or a field across the dataset, the commands are very simple. Note that again, you need to toggle annotation to actually alter any files.

```
> unannotate character field  
> unannotate typo tag  
> unannotate all tags
```

---

## Editing results

---

Once you have generated a *result* object via the *search* command, you can edit the result in a number of ways. You can delete, merge or otherwise alter entries or subcorpora; you can do statistics, and you can sort results.

Editing, calculating and sorting each create a new object, called *edited*. This means that if you make a mistake, you still have access to the original *result* object, without needing to run the search again.

### The edit command

When using the *edit* command, the main things you'll want to do is skip, keep, span or merge results or subcorpora.

```
> edit result by keeping subcorpora matching '[01234]'  
> edit result by skipping entries matching wordlists.closedclass  
# merge has a slightly different syntax, because you need  
# to specify the name to merge under  
> edit result by merging entries matching 'be|have' as 'aux'
```

---

**Note:** The syntax above works for concordance lines too, if you change *result* to *concordance*. Merging is not possible.

---

### Doing basic statistics

The *calculate* command allows you to turn the absolute frequencies into relative frequencies, keyness scores, etc.

```
> calculate result as percentage of self  
> calculate edited as percentage of features.clauses  
> calculate result as keyness of self
```

If you want to run more complicated operations on the results, you might like to use the *ipython* command to enter an IPython session, and then manipulate the Pandas objects directly.

## Sorting results

The *sort* command allows you to change the search result order.

Possible values are *total*, *name*, *infreq*, *increase*, *decrease*, *static*, *turbulent*.

```
> sort result by total
# requires scipy
> sort edited by increase
```

# CHAPTER 16

---

## Plotting

---

You can plot results and edited results using the `plot` method, which interfaces with `matplotlib`.

```
> plot edited as bar chart with title as 'Example plot' and x_label as 'Subcorpus'  
> plot edited as area chart with stacked and colours as Paired  
> plot edited with style as seaborn-talk # defaults to line chart
```

There are many possible arguments for customising the figure. The table below shows some of them.

```
> plot edited as bar chart with rot as 45 and logy and \  
...     legend_alpha as 0.8 and show_p_val and not grid
```

Argument	Type	Action
<code>grid</code>	<code>bool</code>	Show grid in background
<code>rot</code>	<code>int</code>	Rotate x axis labels n degrees
<code>shadow</code>	<code>bool</code>	Shadows for some parts of plot
<code>ncol</code>	<code>int</code>	n columns for legend entries
<code>explode</code>	<code>list</code>	Explode these entries in pie
<code>partial_pie</code>	<code>bool</code>	Allow plotting of pie slices
<code>legend_frame</code>	<code>bool</code>	Show frame around legend
<code>legend_alpha</code>	<code>float</code>	Opacity of legend
<code>reverse_legend</code>	<code>bool</code>	Reverse legend entry order
<code>transpose</code>	<code>bool</code>	Flip axes of DataFrame
<code>logx/logy</code>	<code>bool</code>	Log scales
<code>show_p_val</code>	<code>bool</code>	Try to show p value in legend

---

**Note:** If you want to set a boolean value, you can just say `value` or `and not value`. If you like, however, you could write it more fully as `with value as true/false` as well.

---



---

## Settings and management

---

The interpreter can do a number of other useful things. They are outlined here.

### Managing data

You should be able to store most of the objects you create in memory using the `store` command:

```
> store result as 'good_result'  
> show store  
> fetch 'good_result' as result
```

A more permanent solution is to use `save` and `load`:

```
> save result as 'good_result'  
> ls saved_interrogations  
> load 'good_result' as result
```

An alternative approach is to create variables using the `call` command:

```
> search corpus for words matching any  
> call result anyword  
> calculate anyword as percentage of self
```

A variable can also be a simple string, which you can then add into searches:

```
> call '/NN.*/ >># NP' headnoun  
> search corpus for trees matching headnoun
```

To forget a variable, just do `remove <name>`.

### Toggles and settings

- Using `toggle interactive`, You can switch between interactive mode, where results and concordances are shown in a way that you can manipulate directly, and non-interactive mode, where results and concordances are simply printed to the console.
- Using `toggle conc`, you can tell *corpkit* not to produce concordances. This can be much faster, especially when there are a lot of results.

- `toggle comma` will display thousands separators in results
- `toggle annotation` is used to switch from dry-run to actual modification of corpus files when annotating
- You can set the number of decimals displayed when viewing results with `set decimal to <n>`
- `set max_rows to <n>` and `set max_cols to <n>` limit the amount of data loaded into results lists. This can speed up interactive viewing.

## Switching to IPython

When the interpreter constrains you, you can switch to IPython with `ipython`. Your objects are available there under the same name. When you're done there, do `quit` to return to the *corpkit* interpreter.

## Running scripts

You can also write and run scripts. If you make a file, `participants.cki`, containing:

```
#!/usr/bin/env corpkit

set mydata-parsed as corpus
search corpus for function matching roles.participant showing lemma
export result as csv to part.csv
```

You can run it from the terminal with:

```
corpkit participants.cki
# or, directly, if there's a shebang and chmod +x:
./participants.cki
```

which will leave you with a CSV file at `exported/part.csv`. This approach can be handy if you need to pipe `stdout` or `stderr`, or if you want to call *corpkit* within a shell script.

Finally, just like Python, you can use the `-c` flag to pass code in on the command line:

```
corpkit -c "set 2 ; search corpus for features ; export result as csv to feat.csv"
```

---

**Note:** When running a script, interactivity will automatically be switched off, and concordancing disabled if the script does not appear to need it.

---

## Corpus

A corpus is an unparsed or parsed collection of files that can be searched, or brought into memory for higher performance operations.

**class** `corpkit.corpus.Corpus` (*path\_or\_data*, *\*\*kwargs*)

Bases: `collections.abc.MutableSequence`

Model a parsed or unparsed corpus with arbitrary depth of subfolders

**insert** (*i*, *v*)

**files** = `None`

**filepaths** = `None`

**store\_as\_hdf** (*\*\*kwargs*)

Store a corpus in an HDF5 file for faster loading

**subcorpora** = `None`

**load** (*multiprocess=False*, *load\_trees=True*, *\*\*kwargs*)

Load corpus into memory (i.e. create one large `pd.DataFrame`)

### Keyword Arguments

- **multiprocess** (*int*) – how many threads to use
- **load\_trees** (*bool*) – Parse constituency trees if present
- **add\_gov** (*bool*) – pre-load each token's governor
- **cols** (*list*) – list of columns to be loaded (can improve performance)
- **just** (*dict*) – restrict load to lines with feature key matching regex value (case insensitive)
- **skip** (*dict*) – the inverse of *just*

**Returns** `corpkit.corpus.LoadedCorpus`

**search** (*target*, *query*, *\*\*kwargs*)

Search a corpus for some linguistic or metadata feature

**Parameters**

- **target** (*str*) – The name of the column or feature to search
  - ‘w’: words
  - ‘l’: lemmas
  - ‘x’: XPOS
  - ‘p’: POS
  - ‘f’: dependency function
  - ‘year’, *speaker*, etc: arbitrary metadata categories
  - ‘t’: Constitutency trees via TGrep2 syntax
  - ‘d’: Dependency graphs via depgrep
- **query** (*str/list*) – regular expression, Tgrep2/depgrep string to match, or list of strings to match against

**Keyword Arguments**

- **inverse** (*bool*) – get non-matches
- **multiprocess** (*int*) – number of parallel threads to start
- **no\_store** (*bool*) – do not store reference corpus in Results object
- **just\_index** (*bool*) – return only pointers to matches, not actual data
- **cols** (*list*) – list of columns to be loaded (can improve performance)
- **just** (*dict*) – restrict load to lines with feature key matching regex value (case insensitive)
- **skip** (*dict*) – the inverse of *just*

**Returns** search result**Return type** *corpkit.interrogation.Results***trees** (*query*, *\*\*kwargs*)Equivalent to `.search('t', query)`**deps** (*query*, *\*\*kwargs*)Equivalent to `.search('d', query)`**cql** (*query*, *\*\*kwargs*)Equivalent to `.search('c', query)`**words** (*query*, *\*\*kwargs*)Equivalent to `.search('w', query)`**lemmas** (*query*, *\*\*kwargs*)Equivalent to `.search('l', query)`**pos** (*query*, *\*\*kwargs*)Equivalent to `.search('x', query)`**functions** (*query*, *\*\*kwargs*)Equivalent to `.search('f', query)`**parse** (*parser='corenlp'*, *lang='english'*, *multiprocess=False*, *\*\*kwargs*)

Parse a plaintext corpus

**Keyword Arguments**

- **parser** (*str*) – name of the parser (only ‘corenlp’ accepted so far)
- **lang** (*str*) – language for parser (*english*, *arabic*, *chinese*, *german*, *french* or *spanish*)

- **multiprocess** (*int*) – number of parallel threads to start
- **memory\_mb** (*int*) – megabytes of memory to use per thread (default 2024)

**Returns** parsed corpus

**Return type** `corpkit.corpus.Corpus`

**interrogate** (*search*, *\*\*kwargs*)

**fsi** (*ix*)

Get a slice of a corpus as a DataFrame

**Parameters** **ix** (*iterable*) – if len(ix) == 1, filename to get if len(ix) == 2, get sent from filename if len(ix) == 3, get token from sent from filename

**Returns** pd.DataFrame

**features** (*subcorpora=False*)

Generate and show basic stats from the corpus, including number of sentences, clauses, process types, etc.

Example:

```
>>> corpus.features
SB Characters Tokens Words Closed class words Open class words
01      26873   8513  7308                4809             3704
02      25844   7933  6920                4313             3620
03      18376   5683  4877                3067             2616
04      20066   6354  5366                3587             2767
```

**wordclasses = None**

**postags = None**

**lexicon = None**

**sample** (*n*, *level='f'*)

Get a sample of the corpus

**Parameters**

- **n** (*int/float*) – amount of data in the the sample. If an *int*, get *n* files. if a *float*, get *float* \* 100 as a percentage of the corpus
- **level** (*str*) – sample subcorpora (*s*) or files (*f*)

**Returns** a Corpus object

**delete\_metadata** ()

Delete metadata for corpus. May be needed if corpus is changed

**metadata = None**

**tokenise** (*postag=True*, *lemmatise=True*, *\*args*, *\*\*kwargs*)

Tokenise a plaintext corpus, saving to disk

**Returns** The newly created `corpkit.corpus.Corpus`

**annotate** (*interro*, *annotation*, *dry\_run=True*)

Annotate a corpus

**Parameters**

- **interro** (`corpkit.Interrogation`) – Search matches
- **annotation** (*str/dict*) – a tag or field: value dict. If a dict, the key is the name of the annotation field, and the value is, well, the value. If the value string matches one of the column names seen when concordancing, the content of that string will be used. If the value is a list, the middle column will be formatted, as per the *show* arguments for `Interrogation.table()` and `Interrogation.conc()`.

- **dry\_run** (*bool*) – Show the annotations to be made, but don't do them

**unannotate** (*annotation, dry\_run=True*)

Delete annotation from a corpus

#### Parameters

- **annotation** (*str/dict*) – just as in *corpus.annotate()*.
- **dry\_run** (*bool*) – Show the changes to be made, but don't do them

## File

Corpora are comprised of files, which can be turned into pandas DataFrames and manipulated.

**class** `corpkit.corpus.File` (*path, \*\*kwargs*)

Bases: `corpkit.corpus.Corpus`

Models a corpus file for reading, interrogating, concordancing.

Methods for interrogating, concordancing and configurations are the same as `corpkit.corpus.Corpus`, plus methods for accessing the file contents directly as a *str*, or as a Pandas DataFrame.

**read** (*\*\*kwargs*)

Get contents of file as string

**document** = None

**trees** = None

**plain** = None

## LoadedCorpus

The *load* method of Corpus objects returns a MultiIndexed DataFrame, with three levels: filename, sentence number, and token number. This object can be searched very quickly, because all data is in memory.

**class** `corpkit.corpus.LoadedCorpus` (*data, path=False*)

Bases: `corpkit.interrogation.Results`

Store a corpus in memory as a DataFrame.

This class has all the same methods as a Results object. The only real difference is that slicing it will do some reindexing to speed up searches.

## Results

Searching a corpus returns an object that can be searched again, turned into tables or concordance lines, or exported to other formats.

**class** `corpkit.interrogation.Results` (*matches, reference=False, path=False, qstring=False*)

Bases: `pandas.core.frame.DataFrame`

Search results, a record of matching tokens in a Corpus

**keyness** (*\*args, \*\*kwargs*)

Calculate keyness for each subcorpus

**Returns** DataFrame

**visualise** (*\*\*kwargs*)

Visualise corpus interrogations.

**Keyword Arguments**

- **title** (*str*) – A title for the plot
- **x\_label** (*str*) – A label for the x axis
- **y\_label** (*str*) – A label for the y axis
- **kind** (*str*) – The kind of chart to make
- **style** (*str*) – Visual theme of plot
- **figsize** (*tuple of dimensions*) – Size of plot
- **save** (*bool/str*) – If *bool*, save with *title* as name; if *str*, use *str* as name
- **legend\_pos** (*str*) – Where to place legend
- **reverse\_legend** (*bool*) – Reverse the order of the legend
- **num\_to\_plot** (*int/‘all’*) – How many columns to plot
- **tex** (*bool*) – Use TeX to draw plot text
- **colours** (*str*) – Colourmap for lines/bars/slices
- **cumulative** (*bool*) – Plot values cumulatively
- **pie\_legend** (*bool*) – Show a legend for pie chart
- **partial\_pie** (*bool*) – Allow plotting of pie slices only
- **(str (show\_totals) – legend/plot)**: Print sums in plot where possible
- **transparent** (*bool*) – Transparent .png background
- **output\_format** (*str*) – File format for saved image
- **black\_and\_white** (*bool*) – Create black and white line styles
- **show\_p\_val** (*bool*) – Attempt to print p values in legend if contained in *df*
- **stacked** (*bool*) – When making bar chart, stack bars on top of one another
- **filled** (*bool*) – For area and bar charts, make every column sum to 100
- **legend** (*bool*) – Show a legend
- **rot** (*int*) – Rotate x axis ticks by *rot* degrees
- **subplots** (*bool*) – Plot each column separately
- **layout** (*tuple*) – Grid shape to use when *subplots* is True
- **interactive** – Experimental interactive options

**Returns** matplotlib figure

**multiplot** (*main\_params={}*, *sub\_params={}*, *\*\*kwargs*)

Plot a figure and subplots together

**Keyword Arguments**

- **main\_params** (*dict*) – arguments for Results.visualise(), used to draw the large figure
- **sub\_params** (*dict*) – arguments for Results.visualise(), used to draw the sub figures. if a key is *data*, use its value as secondary data to plot.
- **layout** (*int/float*) – a number between 1 and 16, corresponding to number of subplots. some numbers have an alternative layout accessible with floats (e.g. 3.5).
- **kwargs** (*dict*) – arguments to pass to both figures

**tabview** (*decimals=3*, *\*\*kwargs*)

**format** (*\*args, \*\*kwargs*)

**calculate** (*\*\*kwargs*)

**table** (*subcorpora='file', \*args, \*\*kwargs*)

Create a spreadsheet-like table, showing one or more features by one or more others

**Parameters**

- **subcorpora** (*str/list*) – which metadata or word feature(s) to put on the y axis
- **show** (*str/list*) – word or metadata features to put on the x axis
- **relative** (*bool/DataFrame*) – calculate relative frequencies using self or passed data
- **keyness** (*bool/DataFrame*) – calculate keyness frequencies using self or passed data

**Returns** *pd.DataFrame*

**conc** (*\*args, \*\*kwargs*)

Generate a concordance

**Parameters**

- **show** (*list of strs*) – how to display concordance matches
- **n** (*int*) – number to show
- **shuffle** (*bool*) – randomise order

**Returns** generated concordance lines

**Return type** *pd.DataFrame*

**sort** (*\*\*kwargs*)

**search** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search()*

**deps** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('d', query)*

**trees** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('t', query)*

**pos** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('p', query)*

**xpos** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('x', query)*

**lemmas** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('l', query)*

**words** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('w', query)*

**functions** (*\*args, \*\*kwargs*)

Equivalent to *corpus.search('w', query)*

**collapse** (*feature, values, name=False*)

Merge result on entries or metadata

**Returns** Results (subset)

**just** (*dct, mode='any'*)

Reduce a DataFrame by string matching

**skip** (*dct*)

Reduce a DataFrame by inverse string matching

**top** (*n=50, feature='w'*)

Get the top n most common results by column

**Parameters**

- **n** (*int*) – number of most common results to show
- **feature** (*str*) – which feature to count

**Returns** Results (subset)

**save** (*savename, savedir='saved\_interrogations', \*\*kwargs*)

Save an interrogation as pickle to *savedir*.

Example:

```
>>> o = corpus.interrogate(W, 'any')
### create ./saved_interrogations/savename.p
>>> o.save('savename')
```

*savename* (*str*): A name for the saved file *savedir* (*str*): Relative path to directory in which to save file *print\_info* (*bool*): Show/hide stdout

**store\_as\_hdf** (*\*\*kwargs*)

Store a result within an HDF5 file.



**A**

annotate() (corpkit.corpus.Corpora method), 57

**C**

calculate() (corpkit.interrogation.Results method), 60  
collapse() (corpkit.interrogation.Results method), 60  
conc() (corpkit.interrogation.Results method), 60  
Corpus (class in corpkit.corpus), 55  
cql() (corpkit.corpus.Corpora method), 56

**D**

delete\_metadata() (corpkit.corpus.Corpora method), 57  
deps() (corpkit.corpus.Corpora method), 56  
deps() (corpkit.interrogation.Results method), 60  
document (corpkit.corpus.File attribute), 58

**F**

features() (corpkit.corpus.Corpora method), 57  
File (class in corpkit.corpus), 58  
filepaths (corpkit.corpus.Corpora attribute), 55  
files (corpkit.corpus.Corpora attribute), 55  
format() (corpkit.interrogation.Results method), 59  
fsi() (corpkit.corpus.Corpora method), 57  
functions() (corpkit.corpus.Corpora method), 56  
functions() (corpkit.interrogation.Results method), 60

**I**

insert() (corpkit.corpus.Corpora method), 55  
interrogate() (corpkit.corpus.Corpora method), 57

**J**

just() (corpkit.interrogation.Results method), 60

**K**

keyness() (corpkit.interrogation.Results method), 58

**L**

lemmas() (corpkit.corpus.Corpora method), 56  
lemmas() (corpkit.interrogation.Results method), 60  
lexicon (corpkit.corpus.Corpora attribute), 57  
load() (corpkit.corpus.Corpora method), 55  
LoadedCorpus (class in corpkit.corpus), 58

**M**

metadata (corpkit.corpus.Corpora attribute), 57  
multiplot() (corpkit.interrogation.Results method), 59

**P**

parse() (corpkit.corpus.Corpora method), 56  
plain (corpkit.corpus.File attribute), 58  
pos() (corpkit.corpus.Corpora method), 56  
pos() (corpkit.interrogation.Results method), 60  
postags (corpkit.corpus.Corpora attribute), 57

**R**

read() (corpkit.corpus.File method), 58  
Results (class in corpkit.interrogation), 58

**S**

sample() (corpkit.corpus.Corpora method), 57  
save() (corpkit.interrogation.Results method), 61  
search() (corpkit.corpus.Corpora method), 55  
search() (corpkit.interrogation.Results method), 60  
skip() (corpkit.interrogation.Results method), 60  
sort() (corpkit.interrogation.Results method), 60  
store\_as\_hdf() (corpkit.corpus.Corpora method), 55  
store\_as\_hdf() (corpkit.interrogation.Results method), 61  
subcorpora (corpkit.corpus.Corpora attribute), 55

**T**

table() (corpkit.interrogation.Results method), 60  
tabview() (corpkit.interrogation.Results method), 59  
tokenise() (corpkit.corpus.Corpora method), 57  
top() (corpkit.interrogation.Results method), 60  
trees (corpkit.corpus.File attribute), 58  
trees() (corpkit.corpus.Corpora method), 56  
trees() (corpkit.interrogation.Results method), 60

**U**

unannotate() (corpkit.corpus.Corpora method), 58

**V**

visualise() (corpkit.interrogation.Results method), 58

## **W**

wordclasses (corpkit.corpus.Corpora attribute), 57

words() (corpkit.corpus.Corpora method), 56

words() (corpkit.interrogation.Results method), 60

## **X**

xpos() (corpkit.interrogation.Results method), 60