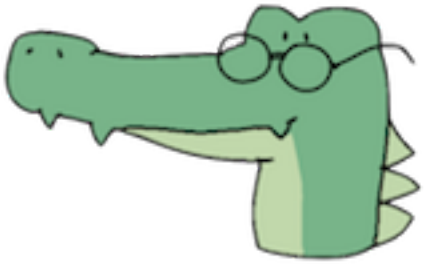


corpkit documentation

Release 2.3.8

Daniel McDonald

January 23, 2017

corpkit is a Python-based tool for doing more sophisticated corpus linguistics. It exists as a graphical interface, a Python API, and a natural language interpreter. The API and interpreter are documented here.

With *corpkit*, 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. Keywording 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.

corpkit leverages Stanford CoreNLP, NLTK and pattern for the linguistic heavy lifting, and pandas and matplotlib for storing, editing and visualising interrogation results. Multiprocessing is available via joblib, and Python 2 and 3 are both supported.

API example

Here's a basic workflow, using a corpus of news articles published between 1987 and 2014, structured like this:

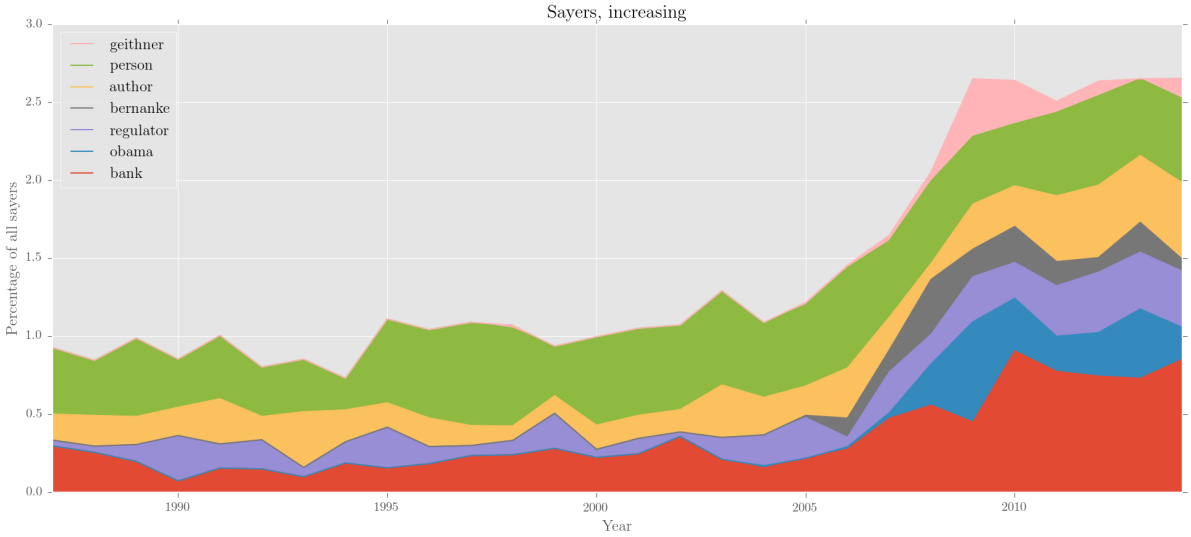
```
./data/NYT:

├---1987
|   ├---NYT-1987-01-01-01.txt
|   ├---NYT-1987-01-02-01.txt
|   ...
|
├---1988
|   ├---NYT-1988-01-01-01.txt
|   ├---NYT-1988-01-02-01.txt
|   ...
...
```

Below, this corpus is made into a *Corpus* object, parsed with *Stanford CoreNLP*, and interrogated for a lexicogrammatical feature. Absolute frequencies are turned into relative frequencies, and results sorted by trajectory. The edited data is then plotted.

```
>>> from corpkit import *
>>> from corpkit.dictionaries import processes

### parse corpus of NYT articles containing annual subcorpora
>>> unparsed = Corpus('data/NYT')
>>> parsed = unparsed.parse()

### query: nominal nsubjs that have verbal process as governor lemma
>>> crit = {F: r'^nsubj$',
...         GL: processes.verbal.lemmata,
...         P: r'^N'}

### interrogate corpus, outputting lemma forms
>>> sayers = parsed.interrogate(crit, show=L)
>>> sayers.quickview(10)

   0: official    (n=4348)
   1: expert      (n=2057)
   2: analyst     (n=1369)
   3: report      (n=1103)
   4: company     (n=1070)
   5: which       (n=1043)
   6: researcher  (n=987)
   7: study       (n=901)
   8: critic      (n=826)
   9: person      (n=802)

### get relative frequency and sort by increasing
>>> rel_say = sayers.edit('%', SELF, sort_by='increase')

### plot via matplotlib, using tex if possible
```

```
>>> rel_say.visualise('Sayers, increasing', kind='area',
...                   y_label='Percentage of all sayers')
```

Output:



Installation

Via pip:

```
$ pip install corpkit
```

via Git:

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

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

Graphical interface

Much of corpkit's command line functionality is also available in the *corpkit GUI*. After installation, it can be started from the command line with:

```
$ python -m corpkit.gui
```

If you're working on a project from within Python, you can open it graphically with:

```
>>> from corpkit import gui
>>> gui()
```

Alternatively, the GUI is available (alongside documentation) as a standalone OSX app here.

Interpreter

corpkit also has its own interpreter, a bit like the Corpus Workbench. You can open it with:

```
$ corpkit
# or, alternatively:
$ python -m corpkit.env
```

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 governor-lemma matching processes:verbal showing pos and lemma
> 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

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.

1.1 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']
```

1.2 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`.

1.3 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.Corpus instance: transcripts; 13 subcorpora>
```

1.4 Pre-processing the data

A *Corpus* object can only be interrogated if tokenisation or parsing has been performed. For this, *corpkit.corpus.Corpus* objects have *tokenise()* and *parse()* methods. Tokenising is faster, simpler, and will work for more languages. As shown below, you can also elect to POS tag and lemmatise the data:

```
> corpus = unparsed.tokenise(postags=True, lemmatisation=True)
# 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:

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

You can run parsing operations from the command line:

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

1.5 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 navigte 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
>>> f.document.sentences[0].parse_string    # get parse tree for first sentence
>>> f.document.sentences.tokens[0].word     # get first word
```

1.6 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
01     4380658  1258606  1092113        643779      614827   277103      68267               35981     16842
02     3185042   922243   800046        471883      450360   209448      51575               26149     10324
03     3157277   917822   795517        471578      446244   209990      51860               26383      9711
04     3261922   948272   820193        486065      462207   216739      53995               27073      9697
05     3164919   921098   796430        473446      447652   210165      52227               26137      9543
06     3187420   928350   797652        480843      447507   209895      52171               25096      8917
07     3080956   900110   771319        466254      433856   202868      50071               24077      8618
08     3356241   972652   833135        502913      469739   218382      52637               25285      9921
09     2908221   840803   725108        434839      405964   191851      47050               21807      8354
10     2868652   815101   708918        421403      393698   185677      43474               20763      8640
```

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.

CHAPTER 2

Interrogating corpora

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

2.1 Introduction

Interrogations can be performed on any *corpkit.corpus.Corpus* 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.interrogate(query)
### interrogate parts of corpus
>>> corpus[2:4].interrogate(query)
>>> corpus.files[:10].interrogate(query)
### if you have a subcorpus called 'abstract':
>>> corpus.subcorpora.abstract.interrogate(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.interrogate({P: '^NNP'}, do_concordancing=True)
>>> propnoun.results

          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.totals

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

>>> propnoun.query

{'case_sensitive': False,
 'corpus': 'gatsby-parsed',
 'dep_type': 'collapsed-ccprocessed-dependencies',
 'do_concordancing': True,
 'exclude': False,
 'excludemode': 'any',
 'files_as_subcorpora': True,
 'gramsize': 1,
 ...}

>>> propnoun.concordance # (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.

2.2 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	Word class
E	NER tag
A	Distance from root
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.interrogate({L: 'be'})
### get words in 'nsubj' position
>>> corpus.interrogate({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.interrogate(goverb, searchmode=ALL)
### get all verbs and any word starting with 'go':
>>> corpus.interrogate(goverb, searchmode=ANY)
```

2.3 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
G	Governor
D	Dependent
H	Coreference head
T	Syntax tree
A1	Token 1 place to left
Z1	Token 1 place to right

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

2.4 Excluding results

You may also wish to exclude particular phenomena from the results. The `exclude` argument takes a `dict` in the same form 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.interrogate({P: r'^n'}, exclude={W: wordlists.closedclass})
### when there's only one search criterion, you can also write:
>>> corpus.interrogate(P, r'^n', exclude={W: wordlists.closedclass})
```

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

2.5 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.interrogate({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.interrogate({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.

2.6 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 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 extent 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.

2.7 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	*prosperous*
T	Tree	*(JJ prosperous)*
p	POS tag	*JJ*
C	Count	*1* (added to total)

2.8 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.interrogate(crit, show=L)
```

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

2.10 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.interrogate(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.interrogate(query, show=L, coref=True)
>>> res.results.columns
```

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

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

2.11 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.interrogate({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.

2.12 N-grams

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

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

2.13 Collocation

Collocations can be shown by making using `window`:

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

2.14 Saving interrogations

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

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

You can also quicksave interrogations:

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

2.15 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())
```

2.16 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.

CHAPTER 3

Concordancing

Concordancing is the task of getting an aligned list of *keywords in context*. Here's a very basic example, using *Industrial Society and Its Future* as a corpus:

```
 >>> tech = corpus.concordance({W: r'techn*'})
 >>> tech.format(n=10, columns=[L, M, R])


0    The continued development of  technology     will worsen the situation
1  vernments but the economic and  technological  basis of the present society
2     They want to make him study  technical      subjects become an executive o
3   program to acquire some petty  technical      skill then come to work on tim
4  rom nature are consequences of  technological  progress
5  n them and modern agricultural  technology     has made it possible for the e
6                      -LRB- Also  technology     exacerbates the effects of cro
7   changes very rapidly owing to  technological  change
8   they enthusiastically support  technological  progress and economic growth
9  e rapid drastic changes in the  technology     and the economy of a society w
```

3.1 Generating a concordance

When using *corpkit*, any interrogation is also optionally a concordance. If you use the `do_concordancing` keyword argument, your interrogation will have a `concordance` attribute containing concordance lines. Like interrogation results, concordances are stored as *Pandas DataFrames*. `maxconc` controls the number of lines produced.

```
>>> withconc = corp.interrogate({L: ['man', 'woman', 'person']},
...                             show=[W,P],
...                             do_concordancing=True,
...                             maxconc=500)

0   T Asian/JJ a/DT disabled/JJ  person/nn    or/cc a/dt woman/nn origin
1   led/JJ person/NN or/CC a/DT  woman/nn     originally/rb had/vbd no/d
2    woman/NN or/CC disabled/JJ  person/nn    but/cc a/dt minority/nn of
3   n/JJ immigrant/JJ abused/JJ  woman/nn     or/cc disabled/jj person/n
4   ing/VBG weak/JJ -LRB-/-LRB-  women/nns    -rrb-/-rrb- defeated/vbn -
```

If you like, you can use `only_format_match=True` to keep the left and right context simple:

```
>>> withconc = corp.interrogate({L: ['man', 'woman', 'person']},
...                             show=[W,P],
...                             only_format_match=True,
...                             do_concordancing=True,
...                             maxconc=500)

0   African an Asian a disabled  person/nn    or a woman originally had
1   sian a disabled person or a  woman/nn     originally had no derogato
2   nt abused woman or disabled  person/nn    but a minority of activist
3   ller Asian immigrant abused  woman/nn     or disabled person but a m
4   n image of being weak -LRB-  women/nns    -rrb- defeated -lrb- ameri
```

If you don't want or need the interrogation data, you can use the *concordance()* method:

```
>>> conc = corpus.concordance(T, r'/JJ.?/ > (NP <<# /man/)')
```

3.2 Displaying concordance lines

How concordance lines will be displayed really depends on your interpreter and environment. For the most part, though, you'll want to use the *format()* method.

```
>>> lines.format(kind='s',
...              n=100,
...              window=50,
...              columns=[L, M, R])
```

`kind='c'/'l'/'s'` allows you to print as CSV, LaTeX, or simple string. `n` controls the number of results shown. `window` controls how much context to show in the left and right columns. `columns` accepts a list of column names to show.

Pandas' set_option can be used to customise some visualisation defaults.

3.3 Working with concordance lines

You can edit concordance lines using the *edit()* method. You can use this method to keep or remove entries or subcorpora matching regular expressions or lists. Keep in mind that because concordance lines are DataFrames, you can use Pandas' dedicated methods for working with text data.

```
### get just uk variants of words with variant spellings
>>> from corpkit.dictionaries import usa_convert
>>> concs = result.concordance.edit(just_entries=usa_convert.keys())
```

Concordance objects can be saved just like any other `corpkit` object:

```
>>> concs.save('adj_modifying_man')
```

You can also easily turn them into CSV data, or into LaTeX:

```
### pandas methods
>>> concs.to_csv()
>>> concs.to_latex()

### corpkit method: csv and latex
>>> concs.format('c', window=20, n=10)
>>> concs.format('l', window=20, n=10)
```

3.4 The *calculate* method

You might have begun to notice that interrogating and concordancing aren't really very different tasks. If we drop the left and right context, and move the data around, we have all the data we get from an interrogation.

For this reason, you can use the *calculate()* method to generate an `corpus.interrogation.Interrogation` object containing a frequency count of the middle column of the concordance as the `results` attribute.

Therefore, one method for ensuring accuracy is to:

1. Run an interrogation, using `do_concordance=True`
2. Remove false positives from the concordance result using *edit()*
3. Use the *calculate()* method to regenerate the overall frequencies

4. Edit, visualise or export the data

If you'd like to randomise the order of your results, you can use `lines.shuffle()`

CHAPTER 4

Editing results

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:

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.

4.1 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))
```

4.2 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')
```

4.3 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.

4.4 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)
```

4.5 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')
```

4.6 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`.

4.7 Calculating trends, P values

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

4.8 Saving results

You can save edited results to disk.

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

4.9 Exporting results

You can generate CSV data very easily using Pandas:

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

4.10 Next step

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

CHAPTER 5

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.

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

5.1 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()
```

5.2 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()
```

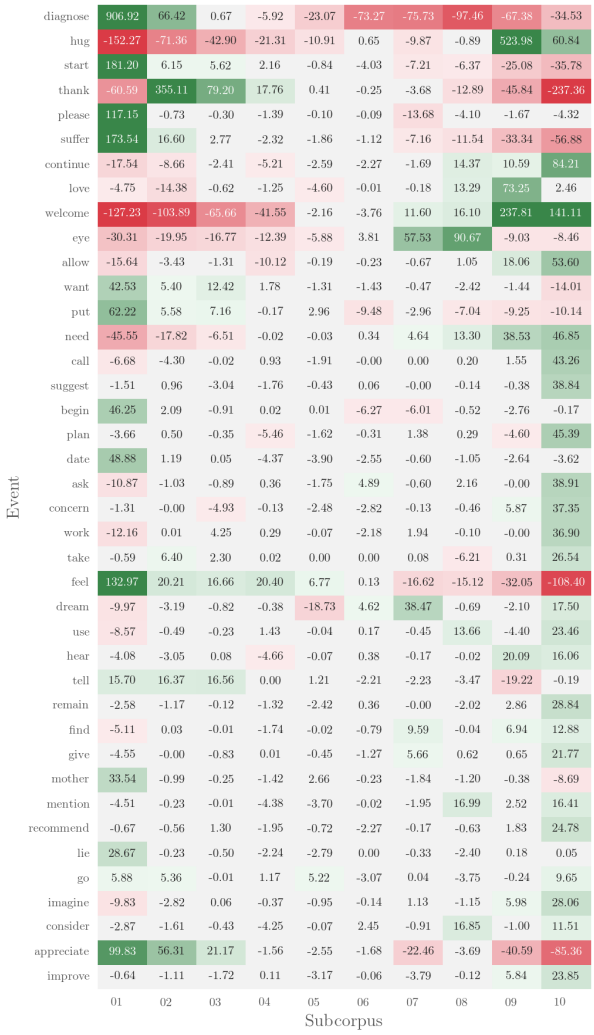
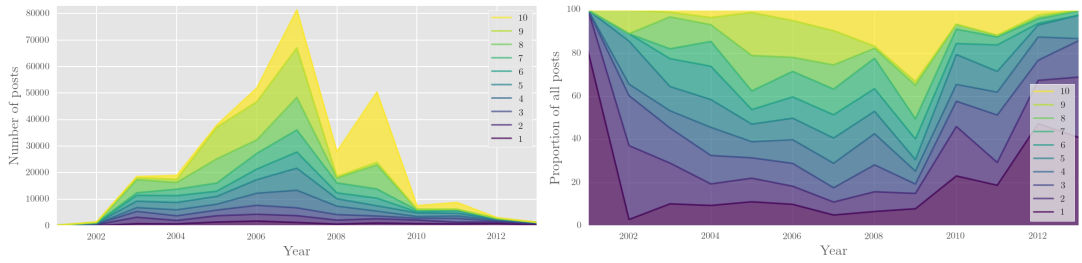


Fig. 5.1: Heatmap example

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.



5.3 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.).

5.4 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`.

5.5 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')
```

5.6 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()
```

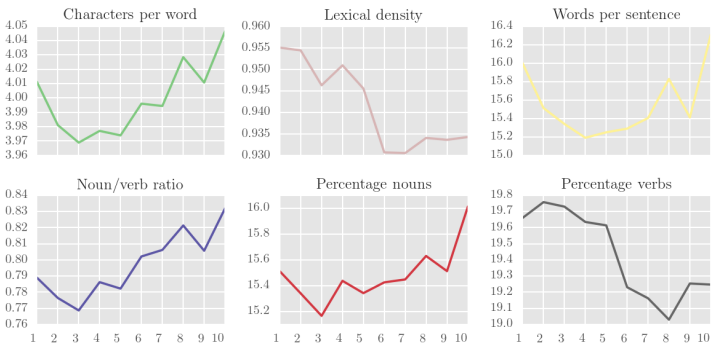


Fig. 5.2: Line charts using subplots and layout specification

5.7 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.

5.8 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.

5.9 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.

5.10 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')
```

5.11 Other options

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

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

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

5.12 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)
```

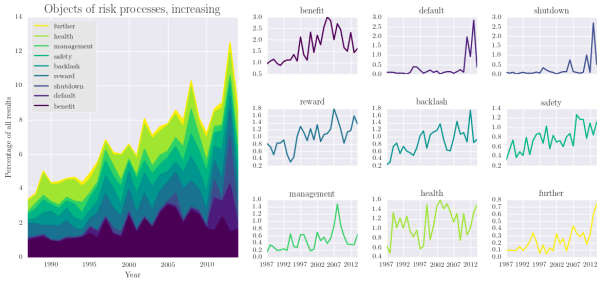


Fig. 5.3: *multiplot* example

CHAPTER 6

Using language models

Warning: Language modelling is currently deprecated, while the tool is updated to use *CONLL* formatted data, rather than *CoreNLP XML*. Sorry!

Language models are probability distributions over sequences of words. They are common in a number of natural language processing tasks. In corpus linguistics, they can be used to judge the similarity between texts.

corpkit's `make_language_model()` method makes it very easy to generate a language model:

```
>>> corpus = Corpus('threads')
# save as models/savename.p
>>> lm = corpus.make_language_model('savename')
```

One simple thing you can do with a language model is pass in a string of text:

```
>>> text = ("We can compare an arbitrary string against the models "\
...         "created for each subcorpus, in order to find out how  "\
...         "similar the text is to the texts in each subcorpus... ")
# get scores for each subcorpus, and the corpus as a whole
>>> lm.score(text)

01       -4.894732
04       -4.986471
02       -5.060964
03       -5.096785
05       -5.106083
07       -5.226934
06       -5.338614
08       -5.829444
09       -5.874777
10       -6.351399
Corpus   -5.285553
```

You can also pass in `corpkit.corpus.Subcorpus` objects, subcorpus names or `corpkit.corpus.File` instances.

6.1 Customising models

Under the hood, *corpkit* interrogates the corpus using some special parameters, then builds a model from the results. This means that you can pass in arbitrary arguments for the `interrogate()` method:

```
>>> lm = corpus.make_language_model('lemma_model',
...                                 show=L,
...                                 just_speakers='MAHSA',
...                                 multiprocess=2)
```

6.2 Compare subcorpora

You can find out which subcorpora are similar using the `score()` method:

```
>>> lm.score('1996')
```

Or get a complete *DataFrame* of values using `score_subcorpora()`:

```
>>> df = lm.score_subcorpora()
```

6.3 Advanced stuff

Note: Coming soon

CHAPTER 7

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.

7.1 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()
```

7.2 Managing multiple corpora

`corpkit` can handle one further level of abstraction for both Corpus and Interrogations. *corpkit.corpus.Corpora* models a collection of *corpkit.corpus.Corpus* objects. To create one, pass in a directory containing corpora, or a list of paths/`Corpus` objects:

```
>>> from corpkit import Corpora
>>> corpora = Corpora('data')
```

Individual corpora can be accessed as attributes, by index, or as keys:

```
>>> corpora.first
>>> corpora[0]
>>> corpora['first']
```

You can use the `interrogate()` method to search them, using the same arguments as you would for *interrogate()*.

Interrogating these objects often returns an *corpkit.interrogation.Interrodict* object, which models a collection of DataFrames.

Editing can be performed with *edit()*. The editor will iterate over each DataFrame in turn, generally returning another `Interrodict`.

Note: There is no `visualise()` method for Interrodict objects.

---

multiindex() can turn an `Interrodict` into a *Pandas MultiIndex*:

```
>>> multiple_res.multiindex()
```

collapse() will collapse one dimension of the `Interrodict`. You can collapse the x axis (`'x'`), the y axis (`'y'`), or the Interrodict keys (`'k'`). In the example below, an `Interrodict` is collapsed along each axis in turn.

```
>>> d = corpora.interrogate({F: 'compound', GL: r'^risk'}, show=L)
>>> d.keys()

    ['CHT', 'WAP', 'WSJ']

>>> d['CHT'].results

    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
    1988      72      24        20      15       7       4      9
    1989     137      61        23      10       5       5      6

>>> d.collapse(axis=Y).results

    ...  health  cancer  credit  security
    CHT    3174    1156     566       697
    WAP    2799     933     582      1127
    WSJ    1812     680    2009       537

>>> d.collapse(axis=X).results

    ...  1987  1988  1989
    CHT   384   328   464
    WAP   389   355   435
    WSJ   428   410   473

>>> d.collapse(axis=K).results

    ...   health  cancer  credit  security
    1987     282     127      65        93
    1988     277     100      70       107
    1989     379     253      83        91
```

topwords() quickly shows the top results from every interrogation in the `Interrodict`.

```
>>> data.topwords(n=5)
```

Output:

```
TBT             %   UST             %   WAP             %   WSJ             %
health      25.70   health      15.25   health      19.64   credit       9.22
security     6.48   cancer      10.85   security     7.91   health       8.31
cancer       6.19   heart        6.31   cancer       6.55   downside     5.46
```

```
flight      4.45   breast      4.29   credit      4.08   inflation   3.37
safety      3.49   security    3.94   safety      3.26   cancer      3.12
```

7.3 Using the GUI

`corpkit` is also designed to work as a GUI. It can be started in `bash` with:

```
$ python -m corpkit.gui
```

The GUI can understand any projects you have defined. If you open it, you can simply select your project via `Open Project` and resume work in a graphical environment.

CHAPTER 8

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   ...
```

8.1 Objects

The most common objects you'll be using are:

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

8.2 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

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

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.

8.3 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
```

CHAPTER 9

Setup

9.1 Dependencies

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

9.2 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()
```

9.3 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!

CHAPTER 10

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
```

10.1 Adding a corpus

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

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

10.2 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
```

10.3 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
```

10.4 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 takeing 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 (I
# 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        _
```

10.5 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.

CHAPTER 11

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.

11.1 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 gramsize 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
```

11.2 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 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
```

11.3 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 `sampled`. 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.

CHAPTER 12

Concordancing

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.

12.1 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 persistant—the window size, number of lines, etc. will remain the same until you shut down the interpreter or provide new values.

12.2 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
```

12.3 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.

12.4 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]$'
> 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
```

12.5 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*.

12.6 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
```

CHAPTER 13

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*.

13.1 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!).

13.2 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.

13.3 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
```

CHAPTER 14

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.

14.1 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.

14.2 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.

14.3 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 15

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

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

CHAPTER 16

Settings and management

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

16.1 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*>.

16.2 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.

16.3 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.

16.4 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.

CHAPTER 17

Corpus classes

Much of *corpkit*'s functionality comes from the ability to work with `Corpus` and `Corpus`-like objects, which have methods for parsing, tokenising, interrogating and concordancing.

17.1 *Corpus*

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

Bases: `object`

A class representing a linguistic text corpus, which contains files, optionally within subcorpus folders.

Methods for concordancing, interrogating, getting general stats, getting behaviour of particular word, etc.

Unparsed, tokenised and parsed corpora use the same class, though some methods are available only to one or the other. Only unparsed corpora can be parsed, and only parsed/tokenised corpora can be interrogated.

subcorpora

A list-like object containing a corpus' subcorpora.

Example

```
>>> corpus.subcorpora
<corpkit.corpus.Datalist instance: 12 items>
```

speakerlist

Lazy-loaded data.

files

A list-like object containing the files in a folder.

Example

```
>>> corpus.subcorpora[0].files
<corpkit.corpus.Datalist instance: 240 items>
```

all_filepaths

Lazy-loaded data.

conll_conform(*errors='raise'*)

This removes sent index column from old corpkit data

all_files

Lazy-loaded data.

tfidf(*search={'w': 'any'}, show=['w'], **kwargs*)

Generate TF-IDF vector representation of corpus using interrogate method. All args and kwargs go to *interrogate()*.

Returns Tuple: the vectoriser and matrix

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

Example

```
>>> corpus.features
    ..  Characters  Tokens  Words  Closed class words  Open class words  Clauses
    01       26873    8513   7308                4809              3704     2212
    02       25844    7933   6920                4313              3620     2270
    03       18376    5683   4877                3067              2616     1640
    04       20066    6354   5366                3587              2767     1775
```

wordclasses
Lazy-loaded data.

postags
Lazy-loaded data.

lexicon
Lazy-loaded data.

configurations(*search*, ***kwargs*)
Get the overall behaviour of tokens or lemmas matching a regular expression. The search below makes DataFrames containing the most common subjects, objects, modifiers (etc.) of 'see':

Parameters **search** (*dict*) – Similar to *search* in the *interrogate()* method.

Valid keys are:

- *W*/*L* match word or lemma
- *F*: match a semantic role (*'participant'*, *'process'* or *'modifier'*. If *F* not specified, each role will be searched for.

Example

```
>>> see = corpus.configurations({L: 'see', F: 'process'}, show=L)
>>> see.has_subject.results.sum()
    i           452
    it          227
    you         162
    we          111
    he           94
```

Returns *corpkit.interrogation.Interrodict*

interrogate(*search='w'*, **args*, ***kwargs*)
Interrogate a corpus of texts for a lexicogrammatical phenomenon.

This method iterates over the files/folders in a corpus, searching the texts, and returning a *corpkit.interrogation.Interrogation* object containing the results. The main options are *search*, where you specify search criteria, and *show*, where you specify what you want to appear in the output.

Example

```
>>> corpus = Corpus('data/conversations-parsed')
### show lemma form of nouns ending in 'ing'
>>> q = {W: r'ing$', P: r'^N'}
>>> data = corpus.interrogate(q, show=L)
>>> data.results
    ..  something  anything  thing  feeling  everything  nothing  morning
    01         14        11     12        1           6        0        1
    02         10        20      4        4           8        3        0
    03         14         5      5        3           1        0        0
    ...                                                              ...
```

Parameters **search** (*dict*) – What part of the lexicogrammar to search, and what criteria to match. The *keys* are the thing to be searched, and values are the criteria. To search parse trees, use the *T* key, and a Tregex query as the value. When searching dependencies, you can use any of:

	Match	Governor	Dependent	Head
Word	*W*	*G*	*D*	*H*
Lemma	*L*	*GL*	*DL*	*HL*
Function	*F*	*GF*	*DF*	*HF*
POS tag	*P*	*GP*	*DP*	*HP*
Word class	*X*	*GX*	*DX*	*HX*
Distance from root	*A*	*GA*	*DA*	*HA*
Index	*I*	*GI*	*DI*	*HI*
Sentence index	*S*	*SI*	*SI*	*SI*

Values should be regular expressions or wordlists to match.

Example

```
>>> corpus.interrogate({T: r'/NN.?/ < /^t/'}) # T- nouns, via trees
>>> corpus.interrogate({W: '^t': P: r'^v'}) # T- verbs, via dependencies
```

Parameters

- **searchmode** (*str – 'any'/'all'*) – Return results matching any/all criteria
- **exclude** (*dict – {L: 'be'}*) – The inverse of *search*, removing results from search
- **excludemode** (*str – 'any'/'all'*) – Exclude results matching any/all criteria
- **query** (*str*, *dict* or *list*) – A search query for the interrogation. This is only used when *search* is a *str*, or when multiprocessing. When *search* If *search* is a str, the search criteria can be passed in as 'query, in order to allow the simpler syntax:

```
>>> corpus.interrogate(GL, '(think|want|feel)')
```

When multiprocessing, the following is possible:

```
>>> q = {'Nouns': r'/NN.?/', 'Verbs': r'/VB.?/'}
### return an :class:`corpkit.interrogation.Interrogation` object with multiindex:
>>> corpus.interrogate(T, q)
### return an :class:`corpkit.interrogation.Interrogation` object without multiindex:
>>> corpus.interrogate(T, q, show=C)
```

- **show** (*str/list* of strings) – What to output. If multiple strings are passed in as a *list*, results will be colon-separated, in the suppled order. Possible values are the same as those for *search*, plus options n-gramming and getting collocates:

Show	Gloss	Example
N	N-gram word	*The women were*
NL	N-gram lemma	*The woman be*
NF	N-gram function	*det nsubj root*
NP	N-gram POS tag	*DT NNS VBN*
NX	N-gram word class	*determiner noun verb*
B	Collocate word	*The_were*
BL	Collocate lemma	*The_be*
BF	Collocate function	*det_root*
BP	Collocate POS tag	*DT_VBN*
BX	Collocate word class	*determiner_verb*

- **lemmatise** (*bool*) – Force lemmatisation on results. **Deprecated: instead, output a lemma form with the 'show' argument**

- **lemmatag** (*'n'/'v'/'a'/'r'/False*) – When using a Tregex/Tgrep query, the tool will attempt to determine the word class of results from the query. Passing in a *str* here will tell the lemmatiser the expected POS of results to lemmatise. It only has an affect if trees are being searched and lemmata are being shown.
- **save** (*str*) – Save result as pickle to *saved_interrogations/<save>* on completion
- **gramsize** (*int*) – Size of n-grams (default 1, i.e. unigrams)
- **multiprocess** (*int/bool* (*bool* determines automatically)) – How many parallel processes to run
- **files_as_subcorpora** (*bool*) – (**Deprecated, use subcorpora=files**). Treat each file as a subcorpus, ignoring actual subcorpora if present
- **conc** (*bool/'only'*) – Generate a concordance while interrogating, store as *.concordance* attribute
- **coref** (*bool*) – Also get coreferents for search matches
- **tgrep** (*bool*) – Use *TGrep* for tree querying. TGrep is less expressive than Tregex, and is slower, but can work without Java. This option may be turned on internally if Java is not found.
- **subcorpora** (*str/list*) – Use a metadata value as subcorpora. Passing a list will create a multiindex. *'file'* and *'folder'/'default'* are also possible values.
- **just_metadata** (*dict*) – One or more metadata fields and criteria to filter sentences by. Only those matching will be kept. Criteria can be a list of words or a regular expression. Passing `{'speaker': 'ENVER'}` will search only sentences annotated with `speaker=ENVER`.
- **skip_metadata** (*dict*) – A field and regex/list to filter sentences by. The inverse of `just_metadata`.
- **discard** (`int`/`float`) – When returning many (i.e. millions) of results, memory can be a problem. Setting a discard value will ignore results occurring infrequently in a subcorpus. An `int` will remove any result occurring `n` times or fewer. A float will remove this proportion of results (i.e. 0.1 will remove 10 per cent)

Returns A *corpkit.interrogation.Interrogation* object, with *.query*, *.results*, *.totals* attributes. If multiprocessing is invoked, result may be multiindexed.

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

Lazy-loaded data.

parse(*corenlppath=False*, *operations=False*, *copula_head=True*, *speaker_segmentation=False*, *memory_mb=False*, *multiprocess=False*, *split_texts=400*, *outname=False*, *metadata=False*, *coref=True*, **args*, ***kwargs*)

Parse an unparsed corpus, saving to disk

Parameters

- **corenlppath** (*str*) – Folder containing corenlp jar files (use if *corpkit* can't find it automatically)
- **operations** (*str*) – Which kinds of annotations to do
- **speaker_segmentation** (*bool*) – Add speaker name to parser output if your corpus is script-like
- **memory_mb** (*int*) – Amount of memory in MB for parser
- **copula_head** (*bool*) – Make copula head in dependency parse
- **split_texts** – Split texts longer than *n* lines for parser memory
- **multiprocess** (*int*) – Split parsing across n cores (for high-performance computers)
- **folderise** (*bool*) – If corpus is just files, move each into own folder
- **output_format** (*str*) – Save parser output as *xml*, *json*, *conll*
- **outname** (*str*) – Specify a name for the parsed corpus
- **metadata** (*bool*) – Use if you have XML tags at the end of lines contaning metadata

Example

```
>>> parsed = corpus.parse(speaker_segmentation=True)
>>> parsed
<corpkit.corpus.Corpus instance: speeches-parsed; 9 subcorpora>
```

Returns The newly created *corpkit.corpus.Corpus*

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

Tokenise a plaintext corpus, saving to disk

Parameters **nltk_data_path** (*str*) – Path to tokeniser if not found automatically

Example

```
>>> tok = corpus.tokenise()
>>> tok
<corpkit.corpus.Corpus instance: speeches-tokenised; 9 subcorpora>
```

Returns The newly created *corpkit.corpus.Corpus*

concordance (**args*, ***kwargs*)

A concordance method for Tregex queries, CoreNLP dependencies, tokenised data or plaintext.

Example

```
>>> wv = ['want', 'need', 'feel', 'desire']
>>> corpus.concordance({L: wv, F: 'root'})
   0   01  1-01.txt.conll                But , so I  feel     like i do that for w
   1   01  1-01.txt.conll                    I  felt     a little like oh , i
   2   01  1-01.txt.conll   he 's a difficult man I  feel     like his work ethic
   3   01  1-01.txt.conll                   So I  felt     like i recognized li
   ...                                                                      ...
```

Arguments are the same as *interrogate()*, plus a few extra parameters:

Parameters

- **only_format_match** (*bool*) – If *True*, left and right window will just be words, regardless of what is in *show*
- **only_unique** (*bool*) – Return only unique lines
- **maxconc** (*int*) – Maximum number of concordance lines

Returns A *corpkit.interrogation.Concordance* instance, with columns showing filename, subcorpus name, speaker name, left context, match and right context.

interroplot (*search*, ***kwargs*)
Interrogate, relativise, then plot, with very little customisability. A demo function.

Example

```
>>> corpus.interroplot(r'/NN.?/ >># NP')
<matplotlib figure>
```

Parameters

- **search** (*dict*) – Search as per *interrogate()*
- **kwargs** (*keyword arguments*) – Extra arguments to pass to `visualise()`

Returns *None* (but show a plot)

save (*savename=False*, ***kwargs*)
Save corpus instance to file. There's not much reason to do this, really.

```
>>> corpus.save(filename)
```

Parameters **savename** (*str*) – Name for the file

Returns *None*

make_language_model (*name*, *search={'w': 'any'}*, *exclude=False*, *show=['w', '+1mw']*, ***kwargs*)
Make a language model for the corpus

Parameters

- **name** (*str*) – a name for the model
- **kwargs** (*keyword arguments*) – keyword arguments for the interrogate() method

Returns a `corpkit.model.MultiModel`

annotate (*conclines*, *annotation*, *dry_run=True*)
Annotate a corpus

Parameters

- **conclines** – a Concordance or DataFrame containing matches to annotate
- **annotation** (`str`/`dict`) – a tag or field and value
- **dry_run** (`bool`) – Show the annotations to be made, but don't do them

Returns `None`

unannotate (*annotation*, *dry_run=True*)
Delete annotation from a corpus

Parameters **annotation** (`str`/`dict`) – a tag or field and value

Returns `None`

17.2 Corpora

class `corpkit.corpus.`**Corpora** (*data=False*, ***kwargs*)
Bases: *corpkit.corpus.Datalist*

Models a collection of Corpus objects. Methods are available for interrogating and plotting the entire collection. This is the highest level of abstraction available.

Parameters **data** (*str/list*) – Corpora to model. A *str* is interpreted as a path containing corpora. A *list* can be a list of corpus paths or *corpkit.corpus.Corpus* objects.)

parse(**kwargs)
Parse multiple corpora

Parameters **kwargs** – Arguments to pass to the *parse()* method.

Returns *corpkit.corpus.Corpora*

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

Example

```
>>> corpus.features
    ..  Characters  Tokens  Words  Closed class words  Open class words  Clauses
    01       26873    8513   7308                4809              3704     2212
    02       25844    7933   6920                4313              3620     2270
    03       18376    5683   4877                3067              2616     1640
    04       20066    6354   5366                3587              2767     1775
```

postags
Lazy-loaded data.

wordclasses
Lazy-loaded data.

lexicon
Lazy-loaded data.

17.3 *Subcorpus*

class corpkit.corpus.**Subcorpus**(*path*, *datatype*, **kwa*)
Bases: *corpkit.corpus.Corpus*

Model a subcorpus, containing files but no subdirectories.

Methods for interrogating, concordancing and configurations are the same as *corpkit.corpus.Corpus*.

17.4 *File*

class corpkit.corpus.**File**(*path*, *dirname=False*, *datatype=False*, **kwa*)
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)
Read file data. If data is pickled, unpickle first

Returns *str*/unpickled data

document
Return a DataFrame representation of a parsed file

trees
Lazy-loaded data.

plain
Lazy-loaded data.

17.5 *Datalist*

class corpkit.corpus.**Datalist**(*data*, ***kwargs*)
Bases: list

interrogate(**args*, ***kwargs*)
Interrogate the corpus using *interrogate()*

concordance(**args*, ***kwargs*)
Concordance the corpus using *concordance()*

configurations(*search*, ***kwargs*)
Get a configuration using *configurations()*

CHAPTER 18

Interrogation classes

Once you have searched a `Corpus` object, you'll want to be able to edit, visualise and store results. Remember that upon importing *corpkit*, any `pandas.DataFrame` or `pandas.Series` object is monkey-patched with `save`, `edit` and `visualise` methods.

18.1 *Interrogation*

class `corpkit.interrogation.`**`Interrogation`**(*results=None*, *totals=None*, *query=None*, *concordance=None*)

Bases: `object`

Stores results of a corpus interrogation, before or after editing. The main attribute, *`results`*, is a Pandas object, which can be edited or plotted.

`results` = **None**
pandas *DataFrame* containing counts for each subcorpus

`totals` = **None**
pandas *Series* containing summed results

`query` = **None**
dict containing values that generated the result

`concordance` = **None**
pandas *DataFrame* containing concordance lines, if concordance lines were requested.

`edit`(**args*, ***kwargs*)
Manipulate results of interrogations.

There are a few overall kinds of edit, most of which can be combined into a single function call. It's useful to keep in mind that many are basic wrappers around *pandas* operations—if you're comfortable with *pandas* syntax, it may be faster at times to use its syntax instead.

Basic mathematical operations

First, you can do basic maths on results, optionally passing in some data to serve as the denominator. Very commonly, you'll want to get relative frequencies:

Example

```
>>> data = corpus.interrogate({W: r'^t'})
>>> rel = data.edit('%', SELF)
>>> rel.results
    ..    to  that   the  then ...   toilet  tolerant  tolerate  ton
    01 18.50 14.65 14.44  6.20 ...     0.00      0.00      0.11 0.00
    02 24.10 14.34 13.73  8.80 ...     0.00      0.00      0.00 0.00
    03 17.31 18.01  9.97  7.62 ...     0.00      0.00      0.00 0.00
```

For the operation, there are a number of possible values, each of which is to be passed in as a *str*:

+, -, /, *, %: self explanatory

k: calculate keywords

a: get distance metric

SELF is a very useful shorthand denominator. When used, all editing is performed on the data. The totals are then extracted from the edited data, and used as denominator. If this is not the desired behaviour, however, a more specific *interrogation.results* or *interrogation.totals* attribute can be used.

In the example above, *SELF* (or *'self'*) is equivalent to:

Example

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

Keeping and skipping data

There are four keyword arguments that can be used to keep or skip rows or columns in the data:

•*just_entries*

•*just_subcorpora*

•*skip_entries*

•*skip_subcorpora*

Each can accept different input types:

•*str*: treated as regular expression to match

•*list*:

–of integers: indices to match

–of strings: entries/subcorpora to match

Example

```
>>> data.edit(just_entries=r'^fr',
...           skip_entries=['free','freedom'],
...           skip_subcorpora=r'[0-9]')
```

Merging data

There are also keyword arguments for merging entries and subcorpora:

•*merge_entries*

•*merge_subcorpora*

These take a *dict*, with the new name as key and the criteria as value. The criteria can be a str (regex) or wordlist.

Example

```
>>> from dictionaries.wordlists import wordlists
>>> mer = {'Articles': ['the', 'an', 'a'], 'Modals': wordlists.modals}
>>> data.edit(merge_entries=mer)
```

Sorting

The *sort_by* keyword argument takes a *str*, which represents the way the result columns should be ordered.

•*increase*: highest to lowest slope value

•*decrease*: lowest to highest slope value

•*turbulent*: most change in y axis values

•*static*: least change in y axis values

•*total/most*: largest number first

•*infreq/least*: smallest number first

•*name*: alphabetically

Example

```
>>> data.edit(sort_by='increase')
```

Editing text

Column labels, corresponding to individual interrogation results, can also be edited with *replace_names*.

Parameters **replace_names** (*str/list of tuples/dict*) – Edit result names, then merge duplicate entries

If *replace_names* is a string, it is treated as a regex to delete from each name. If *replace_names* is a dict, the value is the regex, and the key is the replacement text. Using a list of tuples in the form *(find, replacement)* allows duplicate substitution values.

Example

```
>>> data.edit(replace_names={r'object': r'[di]obj'})
```

Parameters **replace_subcorpus_names** (*str/list of tuples/dict*) – Edit subcorpus names, then merge duplicates. The same as *replace_names*, but on the other axis.

Other options

There are many other miscellaneous options.

Parameters

- **keep_stats** (*bool*) – Keep/drop stats values from dataframe after sorting
- **keep_top** (*int*) – After sorting, remove all but the top *keep_top* results
- **just_totals** (*bool*) – Sum each column and work with sums
- **threshold** (*int/bool*) –

  When using results list as dataframe 2, drop values occurring fewer than n times. If not keywording, you can use:

  'high': *denominator total / 2500*

  'medium': *denominator total / 5000*

  'low': *denominator total / 10000*

  If keywording, there are smaller default thresholds
- **span_subcorpora** (*tuple – (int, int2)*) – If subcorpora are numerically named, span all from *int* to *int2*, inclusive
- **projection** (tuple – *(subcorpus_name, n)*) – multiply results in subcorpus by n
- **remove_above_p** (*bool*) – Delete any result over *p*
- **p** (*float*) – set the p value
- **revert_year** (*bool*) – When doing linear regression on years, turn annual subcorpora into 1, 2 ...

- **print_info** (*bool*) – Print stuff to console showing what's being edited
- **spelling** (*str – 'US'/'UK'*) – Convert/normalise spelling:

Keywording options

If the operation is *k*, you're calculating keywords. In this case, some other keyword arguments have an effect:

Parameters **keyword_measure** – what measure to use to calculate keywords:

ll: log-likelihood 'pd': percentage difference

type keyword_measure: *str*

Parameters

- **selfdrop** (*bool*) – When keywording, try to remove target corpus from reference corpus
- **calc_all** (*bool*) – When keywording, calculate words that appear in either corpus

Returns *corpkit.interrogation.Interrogation*

sort (*way*, ***kwargs*)

visualise (*title=''*, *x_label=None*, *y_label=None*, *style='ggplot'*, *figsize=(8, 4)*, *save=False*, *legend_pos='best'*, *reverse_legend='guess'*, *num_to_plot=7*, *tex='try'*, *colours='Accent'*, *cumulative=False*, *pie_legend=True*, *rot=False*, *partial_pie=False*, *show_totals=False*, *transparent=False*, *output_format='png'*, *interactive=False*, *black_and_white=False*, *show_p_val=False*, *indices=False*, *transpose=False*, ***kwargs*)

Visualise corpus interrogations using *matplotlib*.

Example

```
>>> data.visualise('An example plot', kind='bar', save=True)
<matplotlib figure>
```

Parameters

- **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 ('line'/'bar'/'barh'/'pie'/'area'/'heatmap')*) – The kind of chart to make
- **style** (*str* ('ggplot'/'bmh'/'fivethirtyeight'/'seaborn-talk'/etc)) – Visual theme of plot
- **figsize** (*tuple – (int, int)*) – Size of plot
- **save** (*bool/str*) – If *bool*, save with *title* as name; if *str*, use *str* as name
- **legend_pos** (*str* ('upper right'/'outside right'/etc)) – 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
- **show_totals** (*str* – 'legend'/'plot'/'both') – Print sums in plot where possible

- **transparent** (*bool*) – Transparent .png background
- **output_format** (*str* – 'png'/'pdf') – 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
- **indices** (*bool*) – To use when plotting "distance from root"
- **stacked** (*str*) – When making bar chart, stack bars on top of one another
- **filled** (*str*) – 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 – (int, int)*) – Grid shape to use when *subplots* is True
- **interactive** (*list – [1, 2, 3]*) – Experimental interactive options

Returns matplotlib figure

multiplot(*leftdict={}*, *rightdict={}*, ***kwargs*)

language_model(*name*, **args*, ***kwargs*)

Make a language model from an Interrogation. This is usually done directly on a `corpkit.corpus.Corpus` object with the `make_language_model()` method.

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')
```

Parameters

- **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

Returns None

quickview(*n=25*)

view top n results as painlessly as possible.

Example

```
>>> data.quickview(n=5)
    0: to    (n=2227)
    1: that  (n=2026)
    2: the   (n=1302)
    3: then  (n=857)
    4: think (n=676)
```

Parameters **n** (*int*) – Show top *n* results

Returns *None*

tabview(***kwargs*)

asciiplot(*row_or_col_name*, *axis=0*, *colours=True*, *num_to_plot=100*, *line_length=120*, *min_graph_length=50*, *separator_length=4*, *multivalue=False*, *human_readable='si'*, *graphsymbol='*'*, *float_format='{:, .2f}'*, ***kwargs*)

A very quick ascii chart for result

rel(*denominator='self'*, ***kwargs*)

keyness(*measure='ll'*, *denominator='self'*, ***kwargs*)

multiindex(*indexnames=None*)

Create a *pandas.MultiIndex* object from slash-separated results.

Example

```
>>> data = corpus.interrogate({W: 'st$'}, show=[L, F])
>>> data.results
    ..  just/advmod  almost/advmod  last/amod
01           79             12          6
02          105              6          7
03           86             10          1
>>> data.multiindex().results
Lemma       just almost last first   most
Function  advmod advmod amod  amod advmod
0             79     12    6     2      3
1            105      6    7     1      3
2             86     10    1     3      0
```

Parameters **indexnames** (*list* of strings) – provide custom names for the new index, or leave blank to guess.

Returns *corpkit.interrogation.Interrogation*, with *pandas.MultiIndex* as *results* attribute

topwords(*datatype='n'*, *n=10*, *df=False*, *sort=True*, *precision=2*)

Show top n results in each corpus alongside absolute or relative frequencies.

Parameters

- **datatype** (*str* (n/k/%)) – show abs/rel frequencies, or keyness
- **n** (*int*) – number of result to show
- **df** (*bool*) – return a DataFrame
- **sort** (*bool*) – Sort results, or show as is
- **precision** (*int*) – float precision to show

Example

```
>>> data.topwords(n=5)
1987            %   1988           %   1989           %   1990           %
health      25.70   health     15.25   health     19.64   credit      9.22
security     6.48   cancer     10.85   security    7.91   health      8.31
cancer       6.19   heart       6.31   cancer      6.55   downside    5.46
flight       4.45   breast      4.29   credit      4.08   inflation   3.37
safety       3.49   security    3.94   safety      3.26   cancer      3.12
```

Returns None

perplexity()

Pythonification of the formal definition of perplexity.

input: a sequence of chances (any iterable will do) output: perplexity value.

from https://github.com/zeffii/NLP_class_notes

entropy()

entropy(pos.edit(merge_entries=mergetags, sort_by='total').results.T

shannon()

18.2 *Interrodict*

class corpkit.interrogation.**Interrodict**(*data*)
Bases: collections.OrderedDict

A class for interrogations that do not fit in a single-indexed DataFrame.

Individual interrogations can be looked up via dict keys, indexes or attributes:

Example

```
>>> out_data['WSJ'].results
>>> out_data.WSJ.results
>>> out_data[3].results
```

Methods for saving, editing, etc. are similar to corpkit.corpus.Interrogation. Additional methods are available for collapsing into single (multi-indexed) DataFrames.

This class is now deprecated, in favour of a multiindexed DataFrame.

edit(**args*, ***kwargs*)
Edit each value with *edit()*.

See *edit()* for possible arguments.

Returns A *corpkit.interrogation.Interrodict*

multiindex(*indexnames=False*)
Create a *pandas.MultiIndex* version of results.

Example

```
>>> d = corpora.interrogate({F: 'compound', GL: '^risk'}, show=L)
>>> d.keys()
    ['CHT', 'WAP', 'WSJ']
>>> d['CHT'].results
    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
    1988      72      24        20      15       7       4      9
    1989     137      61        23      10       5       5      6
>>> d.multiindex().results
    ...               health  cancer  credit  security  downside
    Corpus Subcorpus
    CHT    1987           87      25      13        28        20
           1988           72      24      15        20        12
           1989          137      61      10        23        10
    WAP    1987           83      44       8        44        10
           1988           83      27      13        40         6
           1989           95      77      18        25        12
    WSJ    1987           52      27      33         4        21
           1988           39      11      37         9        22
           1989           55      47      43         9        24
```

Returns A *corpkit.interrogation.Interrogation*

save(*savename*, *savedir='saved_interrogations'*, ***kwargs*)
Save an interrogation as pickle to *savedir*.

Parameters

- **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

Example

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

Returns None

collapse(*axis='y'*)

Collapse Interrodict on an axis or along interrogation name.

Parameters **axis** (*str*: x/y/n) – collapse along x, y or name axis

Example

```
>>> d = corpora.interrogate({F: 'compound', GL: r'^risk'}, show=L)

>>> d.keys()
    ['CHT', 'WAP', 'WSJ']

>>> d['CHT'].results
    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
    1988      72      24        20      15       7       4      9
    1989     137      61        23      10       5       5      6

>>> d.collapse().results
    ...  health  cancer  credit  security
    CHT    3174    1156     566       697
    WAP    2799     933     582      1127
    WSJ    1812     680    2009       537

>>> d.collapse(axis='x').results
    ...  1987  1988  1989
    CHT   384   328   464
    WAP   389   355   435
    WSJ   428   410   473

>>> d.collapse(axis='key').results
    ...   health  cancer  credit  security
    1987     282     127      65        93
    1988     277     100      70       107
    1989     379     253      83        91
```

Returns A *corpkit.interrogation.Interrogation*

topwords(*datatype='n'*, *n=10*, *df=False*, *sort=True*, *precision=2*)

Show top n results in each corpus alongside absolute or relative frequencies.

Parameters

- **datatype** (*str* (n/k/%)) – show abs/rel frequencies, or keyness
- **n** (*int*) – number of result to show
- **df** (*bool*) – return a DataFrame
- **sort** (*bool*) – Sort results, or show as is
- **precision** (*int*) – float precision to show

Example

```
>>> data.topwords(n=5)
    TBT             %   UST             %   WAP             %   WSJ             %
    health      25.70   health      15.25   health      19.64   credit       9.22
    security     6.48   cancer      10.85   security     7.91   health       8.31
    cancer       6.19   heart        6.31   cancer       6.55   downside     5.46
    flight       4.45   breast       4.29   credit       4.08   inflation    3.37
    safety       3.49   security     3.94   safety       3.26   cancer       3.12
```

Returns None

visualise(*shape='auto'*, *truncate=8*, ***kwargs*)
Attempt to visualise Interrodict by using subplots

Parameters

- **shape** (*tuple*) – Layout for the subplots (e.g. *(2, 2)*)
- **truncate** (*int*) – Only process the first *n* items in the class:*corpkit.interrogation.Interrodict*
- **kwargs** (*keyword arguments*) – specifications to pass to `plotter()`

copy()

flip(*truncate=30*, *transpose=True*, *repeat=False*, **args*, ***kwargs*)
Change the dimensions of *corpkit.interrogation.Interrodict*, making column names into keys.

Parameters

- **truncate** (*int/'all'*) – Get first *n* columns
- **transpose** (*bool*) – Flip rows and columns:
- **repeat** (*bool*) – Flip twice, to move columns into key position
- **kwargs** – Arguments to pass to the *edit()* method

Returns *corpkit.interrogation.Interrodict*

get_totals()
Helper function to concatenate all totals

18.3 *Concordance*

class corpkit.interrogation.**Concordance**(*data*)
Bases: pandas.core.frame.DataFrame

A class for concordance lines, with methods for saving, formatting and editing.

format(*kind='string'*, *n=100*, *window=35*, *print_it=True*, *columns='all'*, *metadata=True*, ***kwargs*)
Print concordance lines nicely, to string, LaTeX or CSV

Parameters

- **kind** (*str*) – output format: *string/latex/csv*
- **n** (*int/'all'*) – Print first *n* lines only
- **window** (*int*) – how many characters to show to left and right
- **columns** (*list*) – which columns to show

Example

```
>>> lines = corpus.concordance({T: r'/NN.?/ >># NP'}, show=L)
### show 25 characters either side, 4 lines, just text columns
>>> lines.format(window=25, n=4, columns=[L,M,R])
    0                  we 're in  tucson     , then up north to flagst
    1  e 're in tucson , then up  north      to flagstaff , then we we
    2  tucson , then up north to  flagstaff  , then we went through th
    3   through the grand canyon  area       and then phoenix and i sp
```

Returns None

calculate()
Make new Interrogation object from (modified) concordance lines

shuffle(*inplace=False*)

Shuffle concordance lines

Parameters **inplace** (*bool*) – Modify current object, or create a new one

Example

```
>>> lines[:4].shuffle()
    3  01  1-01.txt.conll   through the grand canyon  area       and then phoenix and i sp
    1  01  1-01.txt.conll  e 're in tucson , then up  north      to flagstaff , then we we
    0  01  1-01.txt.conll                  we 're in  tucson     , then up north to flagst
    2  01  1-01.txt.conll  tucson , then up north to  flagstaff  , then we went through th
```

edit(**args*, ***kwargs*)

Delete or keep rows by subcorpus or by middle column text.

```
>>> skipped = conc.edit(skip_entries=r'to_?match')
```

less(***kwargs*)

CHAPTER 19

Functions

corpkit contains a small set of standalone functions.

19.1 *as_regex*

corpkit.other.**as_regex**(*lst*, *boundaries='w'*, *case_sensitive=False*, *inverse=False*, *compile=False*)

Turns a wordlist into an uncompiled regular expression

Parameters

- **lst** (*list*) – A wordlist to convert
- **boundaries** (*str -- 'word''line''space'; tuple -- (leftboundary, rightboundary)*) –
- **case_sensitive** (*bool*) – Make regular expression case sensitive
- **inverse** (*bool*) – Make regular expression inverse matching

Returns regular expression as string

19.2 *load*

corpkit.other.**load**(*savename*, *loaddir='saved_interrogations'*)

Load saved data into memory:

```
>>> loaded = load('interro')
```

will load `./saved_interrogations/interro.p` as loaded

Parameters

- **savename** (*str*) – Filename with or without extension
- **loaddir** (*str*) – Relative path to the directory containg *savename*
- **only_concs** (*bool*) – Set to True if loading concordance lines

Returns loaded data

19.3 *load_all_results*

corpkit.other.**load_all_results**(*data_dir='saved_interrogations'*, ***kwargs*)

Load every saved interrogation in data_dir into a dict:

```
>>> r = load_all_results()
```

Parameters **data_dir** (*str*) – path to saved data

Returns dict with filenames as keys

19.4 new_project

corpkit.other.**new_project**(*name*, *loc='.'*, ***kwargs*)

Make a new project in `loc`.

Parameters

- **name** (*str*) – A name for the project
- **loc** (*str*) – Relative path to directory in which project will be made

Returns None

CHAPTER 20

Wordlists

20.1 Closed class word types

Various wordlists, mostly for subtypes of closed class words

corpkit.dictionaries.wordlists.**wordlists = wordlists(pronouns=[u'all', u'another', u'any', u'anybody', u'an**
wordlists(pronouns, conjunctions, articles, determiners, prepositions, connectors, modals, closedclass, stopwords, titles, whpro)

20.2 Systemic functional process types

Inflected verbforms for systemic process types.

corpkit.dictionaries.process_types.**processes**

20.3 Stopwords

A list of arbitrary stopwords.

corpkit.dictionaries.stopwords.**stopwords**

20.4 Systemic/dependency label conversion

Systemic-functional to dependency role translation.

corpkit.dictionaries.roles.**roles = roles(actor=['agent', 'agent', 'csubj', 'nsubj'], adjunct=['(prep|nmod)(_|:).***
roles(actor, adjunct, any, auxiliary, circumstance, classifier, complement, deictic, epithet, event, existential, finite, goal, modal, modifier, numerative, participant, participant1, participant2, polarity, postmodifier, predicator, premodifier, process, qualifier, subject, textual, thing)

20.5 BNC reference corpus

BNC word frequency list.

corpkit.dictionaries.bnc.**bnc**

20.6 Spelling conversion

A dict with U.S. English spellings as keys, U.K. spellings as values.

corpkit.dictionaries.word_transforms.**usa_convert**

Cite

If you'd like to cite *corpkit*, you can use:

```
McDonald, D. (2015). corpkit: a toolkit for corpus linguistics. Retrieved from
https://www.github.com/interrogator/corpkit. DOI: http://doi.org/10.5281/zenodo.28361
```

N

P

Q

R

S

T

U

V

W