# python-textops Documentation

***Release 0.1.7***

**Eric Lapouyade**

November 27, 2015

Contents

python-textops provides many text operations at string level, list level or whole text level.
These operations can be chained with a 'dotted' or 'piped' notation.
Chained operations are stored into a single lazy object, they will be executed only when an input text will be provided.

CHAPTER 1

# Getting started

## 1.1 Install

To install:

```
pip install python-textops
```

## 1.2 Quickstart

The usual way to use textops is something like below. IMPORTANT : Note that textops library redefines the python **bitwise OR** operator `|` in order to use it as a 'pipe' like in a Unix shell:

```
from textops import *

result = "an input text" | my().chained().operations()

or

for result_item in "an input text" | my().chained().operations():
   do_something(result_item)

or

myops = my().chained().operations()
# and later in the code, use them :
result = myops("an input text")
or
result = "an input text" | myops
```

An "input text" can be :

- a simple string,
- a multi-line string (one string having newlines),
- a list of strings,
- a strings generator,
- a list of lists (useful when you cut lines into columns),
- a list of dicts (useful when you parse a line).

So one can do:

```
>>> 'line1line2line3' | grep('2').tolist()
['line1line2line3']
>>> 'line1\nline2\nline3' | grep('2').tolist()
['line2']
>>> ['line1','line2','line3'] | grep('2').tolist()
['line2']
>>> [['line','1'],['line','2'],['line','3']] | grep('2').tolist()
[['line', '2']]
>>> [{'line':1},{'line':'2'},{'line':3}] | grep('2').tolist()
[{'line': '2'}]
```

**Note :** As many operations return a generator, they can be used directly in for-loops, but in this documentation we added `.tolist()` to show the result as a list.

Textops library also redefines >> operator that works like the | except that it converts generators results into lists:

```
>>> 'a\nb' | grep('a')
<generator object extend_type_gen at ...>
>>> 'a\nb' | grep('a').tolist()
['a']
>>> 'a\nb' >> grep('a')
['a']
>>> for line in 'a\nb' | grep('a'):
...     print line
a
>>> 'abc' | length()
3
>>> 'abc' >> length()
3
```

**Note :** You should use the pipe | when you are expecting a huge result or when using for-loops, otherwise, the >> operator is easier to handle as you are not keeping generators.

Here is an example of chained operations to find the first line with an error and put it in uppercase:

```
>>> from textops import *
>>> myops = grepi('error').first().upper()
```

**Note :** str standard methods (like 'upper') can be used directly in chained dotted notation.

You can use unix shell 'pipe' symbol into python code to chain operations:

```
>>> from textops import *
>>> myops = grepi('error') | first() | strop.upper()
```

The main interest for the piped notation is the possibility to avoid importing all operations, that is to import only textops module:

```
>>> import textops as op
>>> myops = op.grepi('error') | op.first() | op.strop.upper()
```

**Note :** str methods must be prefixed with `strop.` in piped notations.

Chained operations are not executed (lazy object) until an input text has been provided. You can use chained operations like a function, or use the pipe symbol to "stream" input text:

```
>>> myops = grepi('error').first().upper()
>>> print myops('this is an error\nthis is a warning')
THIS IS AN ERROR
```

```
>>> print 'this is an error\nthis is a warning' | myops
THIS IS AN ERROR
```

**Note :** python generators are used as far as possible to be able to manage huge data set like big files. Prefer to use the dotted notation, it is more optimized.

To execute operations at once, specify the input text on the same line:

```
>>> print grepi('error').first().upper()('this is an error\nthis is a warning')
THIS IS AN ERROR
```

A more readable way is to use ONE pipe symbol, then use dotted notation for other operations : this is the **recommended way to use textops**. Because of the first pipe, there is no need to use special textops Extended types, you can use standard strings or lists as an input text:

```
>>> print 'this is an error\nthis is a warning' | grepi('error').first().upper()
THIS IS AN ERROR
```

You could use the pipe everywhere (internally a little less optimized, but looks like shell):

```
>>> print 'this is an error\nthis is a warning' | grepi('error') | first() | strop.upper()
THIS IS AN ERROR
```

To execute an operation directly from strings, lists or dicts *with the dotted notation*, you must use textops Extended types : `StrExt`, `ListExt` or `DictExt`:

```
>>> s = StrExt('this is an error\nthis is a warning')
>>> print s.grepi('error').first().upper()
THIS IS AN ERROR
```

**Note :** As soon as you are using textops Extended type, textops cannot use gnerators internally anymore : all data must fit into memory (it is usually the case, so it is not a real problem).

You can use the operations result in a 'for' loop:

```
>>> open('/tmp/errors.log','w').write('error 1\nwarning 1\nwarning 2\nerror 2')
>>> for line in '/tmp/errors.log' | cat().grepi('warning').head(1).upper():
...   print line
WARNING 1
```

A shortcut is possible : the input text can be put as the first parameter of the first operation. nevertheless, in this case, despite the input text is provided, chained operations won't be executed until used in a for-loop, converted into a string/list or forced by special attributes:

```
>>> open('/tmp/errors.log','w').write('error 1\nwarning 1\nwarning 2\nerror 2')

# Here, operations are excuted because 'print' converts into string :
# it triggers execution.
>>> print cat('/tmp/errors.log').grepi('warning').head(1).upper()
WARNING 1

# Here, operations are excuted because for-loops or list casting triggers execution.
>>> for line in cat('/tmp/errors.log').grepi('warning').head(1).upper():
...   print line
WARNING 1

# Here, operations are NOT executed because there is no for-loops nor string/list cast :
# operations are considered as a lazy object, that is the reason why
# only the object representation is returned (chained operations in dotted notation)
>>> logs = cat('/tmp/errors.log')
```

```
>>> logs
cat('/tmp/errors.log')
>>> print type(logs)
<class 'textops.ops.listops.cat'>

# To force execution, use special attribute .s .l or .g :
>>> open('/tmp/errors.log','w').write('error 1\nwarning 1')
>>> logs = cat('/tmp/errors.log').s
>>> print type(logs)
<class 'textops.base.StrExt'>
>>> print logs
error 1
warning 1

>>> logs = cat('/tmp/errors.log').l
>>> print type(logs)
<class 'textops.base.ListExt'>
>>> print logs
['error 1', 'warning 1']

>>> logs = cat('/tmp/errors.log').g
>>> print type(logs)
<type 'generator'>
>>> print list(logs)
['error 1', 'warning 1']
```

**Note :**

- `.s` : execute operations and get a string
- `.l` : execute operations and get a list of strings
- `.g` : execute operations and get a generator of strings

your input text can be a list:

```
>>> print ['this is an error','this is a warning'] | grepi('error').first().upper()
THIS IS AN ERROR
```

textops works also on list of lists (you can optionally grep on a specific column):

```
>>> l = ListExt([['this is an','error'],['this is a','warning']])
>>> print l.grepi('error',1).first().upper()
['THIS IS AN', 'ERROR']
```

... or a list of dicts (you can optionally grep on a specific key):

```
>>> l = ListExt([{ 'msg':'this is an', 'level':'error'},
... {'msg':'this is a','level':'warning'}])
>>> print l.grepi('error','level').first()
{'msg': 'this is an', 'level': 'error'}
```

textops provides DictExt class that has got the attribute access functionnality:

```
>>> d = DictExt({ 'a' : { 'b' : 'this is an error\nthis is a warning'}})
>>> print d.a.b.grepi('error').first().upper()
THIS IS AN ERROR
```

If attributes are reserved or contains space, one can use normal form:

```
>>> d = DictExt({ 'this' : { 'is' : { 'a' : {'very deep' : { 'dict' : 'yes it is'}}}}})
>>> print d.this['is'].a['very deep'].dict
yes it is
```

You can use dotted notation for setting information in dict BUT only on one level at a time:

```
>>> d = DictExt()
>>> d.a = DictExt()
>>> d.a.b = 'this is my logging data'
>>> print d
{'a': {'b': 'this is my logging data'}}
```

You saw `cat`, `grep`, `first`, `head` and `upper`, but there are many more operations available.

Read The Fabulous Manual !

## 1.3 Run tests

Many doctests as been developped, you can run them this way:

```
cd tests
python ./runtests.py
```

## 1.4 Build documentation

An already compiled and up-to-date documentation should be available *here<http://python-textops.readthedocs.org>*. Nevertheless, one can build the documentation :

For HTML:

```
cd docs
make html
cd _build/html
firefox ./index.html
```

For PDF, you may have to install some linux packages:

```
sudo apt-get install texlive-latex-recommended texlive-latex-extra
sudo apt-get install texlive-latex-base preview-latex-style lacheck tipa

cd docs
make latexpdf
cd _build/latex
evince python-textops.pdf   (evince is a PDF reader)
```

- *genindex*
- *modindex*
- *search*

CHAPTER 2

# strops

This module gathers text operations to be run on a string

## 2.1 cut

**class** `textops.`**`cut`**(*sep=None*, *col=None*, *default=''*)

Extract columns from a string or a list of strings

This works like the unix shell command 'cut'. It uses `str.split()` function.

- if the input is a simple string, cut() will return a list of strings representing the splitted input string.
- if the input is a list of strings or a string with newlines, cut() will return a list of list of strings : each line of the input will splitted and put in a list.
- if only one column is extracted, one level of list is removed.

**Parameters**

- **sep** (*str*) – a string as a column separator, default is None : this means 'any kind of spaces'
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** A string, a list of strings or a list of list of strings

### Examples

```
>>> s='col1 col2 col3'
>>> s | cut()
['col1', 'col2', 'col3']
>>> s | cut(col=1)
```

```
'col2'
>>> s | cut(col='1,2,10',default='N/A')
['col2', 'col3', 'N/A']
>>> s='col1.1 col1.2 col1.3\ncol2.1 col2.2 col2.3'
>>> s | cut()
[['col1.1', 'col1.2', 'col1.3'], ['col2.1', 'col2.2', 'col2.3']]
>>> s | cut(col=1)
['col1.2', 'col2.2']
>>> s | cut(col='0,1')
[['col1.1', 'col1.2'], ['col2.1', 'col2.2']]
>>> s | cut(col=[1,2])
[['col1.2', 'col1.3'], ['col2.2', 'col2.3']]
>>> s='col1.1 | col1.2 |  col1.3\ncol2.1 | col2.2 | col2.3'
>>> s | cut()
[['col1.1', '|', 'col1.2', '|', 'col1.3'], ['col2.1', '|', 'col2.2', '|', 'col2.3']]
>>> s | cut(sep=' | ')
[['col1.1', 'col1.2', ' col1.3'], ['col2.1', 'col2.2', 'col2.3']]
```

## 2.2 cutca

**class** textops.**cutca**(*sep*, *col=None*, *default=''*)

Extract columns from a string or a list of strings through pattern capture

This works like `textops.cutre` except it needs a pattern having parenthesis to capture column. It uses `re.match()` for capture, this means the pattern must start at line beginning.

- if the input is a simple string, cutca() will return a list of strings representing the splitted input string.
- if the input is a list of strings or a string with newlines, cut() will return a list of list of strings : each line of the input will splitted and put in a list.
- if only one column is extracted, one level of list is removed.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having capture parenthesis
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** a list of strings or a list of list of strings

### Examples

```
>>> s='-col1- =col2= _col3_'
>>> s | cutca(r'[^-]*-([^-]*)-[^=]*=([^=]*)=[^_]*_([^_]*)_')
['col1', 'col2', 'col3']
>>> s=['-col1- =col2= _col3_','-col11- =col22= _col33_']
>>> s | cutca(r'[^-]*-([^-]*)-[^=]*=([^=]*)=[^_]*_([^_]*)_','0,2,4','not present')
[['col1', 'col3', 'not present'], ['col11', 'col33', 'not present']]
```

## 2.3 cutm

**class** `textops.`**`cutm`**(*sep*, *col=None*, *default=''*)

Extract exactly one column by using `re.match()`

This is a shortcut for `cutca('mypattern',col=0)`

- if the input is a simple string, `textops.cutm` will return a strings representing the captured substring.
- if the input is a list of strings or a string with newlines, `textops.cutm` will return a list of captured substring.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having capture parenthesis
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** a list of strings or a list of list of strings

### Examples

```
>>> s='-col1- =col2= _col3_'
>>> s | cutm(r'[^-]*-([^-]*)-')
'col1'
>>> s=['-col1- =col2= _col3_','-col11- =col22= _col33_']
>>> s | cutm(r'[^-]*-([^-]*)-')
['col1', 'col11']
>>> s | cutm(r'[^-]*-(badpattern)-',default='-')
['-', '-']
```

## 2.4 cutmi

**class** `textops.`**`cutmi`**(*sep*, *col=None*, *default=''*)

Extract exactly one column by using `re.match()` (case insensitive)

This works like `textops.cutm` except it is caseinsensitive.

- if the input is a simple string, `textops.cutmi` will return a strings representing the captured substring.
- if the input is a list of strings or a string with newlines, `textops.cutmi` will return a list of captured substring.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having capture parenthesis
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** a list of strings or a list of list of strings

#### Examples

```
>>> s='-col1- =col2= _col3_'
>>> s | cutm(r'.*(COL\d+)',default='no found')
'no found'
>>> s='-col1- =col2= _col3_'
>>> s | cutmi(r'.*(COL\d+)',default='no found')
'col3'
```

## 2.5 cutdct

**class** `textops.`**`cutdct`**(*sep=None*, *col=None*, *default=''*)

Extract columns from a string or a list of strings through pattern capture

This works like `textops.cutca` except it needs a pattern having *named* parenthesis to capture column.

- if the input is a simple string, cutca() will return a list of strings representing the splitted input string.
- if the input is a list of strings or a string with newlines, cut() will return a list of list of strings : each line of the input will splitted and put in a list.
- if only one column is extracted, one level of list is removed.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having *named* capture parenthesis

- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** A string, a list of strings or a list of list of strings

#### Examples

```
>>> s='item="col1" count="col2" price="col3"'
>>> s | cutdct(r'item="(?P<item>[^"]*)" count="(?P<i_count>[^"]*)" price="(?P<i_price>[^"]*)
{'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'}
>>> s='item="col1" count="col2" price="col3"\nitem="col11" count="col22" price="col33"'
>>> s | cutdct(r'item="(?P<item>[^"]*)" count="(?P<i_count>[^"]*)" price="(?P<i_price>[^"]*)
[{'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'},...
{'item': 'col11', 'i_price': 'col33', 'i_count': 'col22'}]
```

## 2.6 cutkv

**class** `textops.`**`cutkv`**(*sep=None*, *col=None*, *default=''*, *key_name = 'key'*)

Extract columns from a string or a list of strings through pattern capture

This works like `textops.cutdct` except it return a dict where the key is the one captured with the name given in parameter 'key_name', and where the value is the full dict of captured values. The interest is to merge informations into a bigger dict : see `merge_dicts()`

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having *named* capture parenthesis
- **key_name** (*str*) – specify the named capture to use as the key for the returned dict Default value is 'key'

---

**Note:** `key_name=` must be specified (not a positionnal parameter)

---

**Returns** A dict or a list of dict

#### Examples

```
>>> s='item="col1" count="col2" price="col3"'
>>> pattern=r'item="(?P<item>[^"]*)" count="(?P<i_count>[^"]*)" price="(?P<i_price>[^"]*)"'
>>> s | cutkv(pattern,key_name='item')
{'col1': {'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'}}
>>> s='item="col1" count="col2" price="col3"\nitem="col11" count="col22" price="col33"'
>>> s | cutkv(pattern,key_name='item')
```

```
[{'col1': {'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'}},...
{'col11': {'item': 'col11', 'i_price': 'col33', 'i_count': 'col22'}}]
```

## 2.7 cutre

**class** textops.**cutre**(*sep=None*, *col=None*, *default=''*)

Extract columns from a string or a list of strings with re.split()

This works like the unix shell command 'cut'. It uses `re.split()` function.

- if the input is a simple string, cutre() will return a list of strings representing the splitted input string.
- if the input is a list of strings or a string with newlines, cut() will return a list of list of strings : each line of the input will splitted and put in a list.
- if only one column is extracted, one level of list is removed.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object as a column separator
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** A string, a list of strings or a list of list of strings

### Examples

```
>>> s='col1.1 | col1.2 | col1.3\ncol2.1 | col2.2 | col2.3'
>>> print s
col1.1 | col1.2 | col1.3
col2.1 | col2.2 | col2.3
>>> s | cutre(r'\s+')
[['col1.1', '|', 'col1.2', '|', 'col1.3'], ['col2.1', '|', 'col2.2', '|', 'col2.3']]
>>> s | cutre(r'[\s|]+')
[['col1.1', 'col1.2', 'col1.3'], ['col2.1', 'col2.2', 'col2.3']]
>>> s | cutre(r'[\s|]+','0,2,4','-')
[['col1.1', 'col1.3', '-'], ['col2.1', 'col2.3', '-']]
>>> mysep = re.compile(r'[\s|]+')
>>> s | cutre(mysep)
[['col1.1', 'col1.2', 'col1.3'], ['col2.1', 'col2.2', 'col2.3']]
```

## 2.8 cuts

**class** textops.**cuts**(*sep*, *col=None*, *default=''*)
Extract exactly one column by using re.search()

This works like textops.cutm except it searches the first occurence of the pattern in the string.

- if the input is a simple string, textops.cuts will return a strings representing the captured substring.
- if the input is a list of strings or a string with newlines, textops.cuts will return a list of captured substring.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having capture parenthesis
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** a list of strings or a list of list of strings

### Examples

```
>>> s='-col1- =col2= _col3_'
>>> s | cuts(r'_([^_]*)_')
'col3'
>>> s=['-col1- =col2= _col3_','-col11- =col22= _col33_']
>>> s | cuts(r'_([^_]*)_')
['col3', 'col33']
```

## 2.9 cutsi

**class** textops.**cutsi**(*sep*, *col=None*, *default=''*)
Extract exactly one column by using re.search()

This works like textops.cutm except it searches the first occurence of the pattern in the string.

- if the input is a simple string, textops.cutsi will return a strings representing the captured substring.
- if the input is a list of strings or a string with newlines, textops.cutsi will return a list of captured substring.

**Parameters**

- **sep** (*str or re.RegexObject*) – a regular expression string or object having capture parenthesis
- **col** (*int or list of int or str*) – specify one or many columns you want to get back, You can specify :
  - an int as a single column number (starting with 0)
  - a list of int as the list of colmun
  - a string containing a comma separated list of int
  - None (default value) for all columns
- **default** (*str*) – A string to display when requesting a column that does not exist

**Returns** a list of strings or a list of list of strings

#### Examples

```
>>> s='-col1- =col2= _col3_'
>>> s | cuts(r'_(COL[^_]*)_')
''
>>> s='-col1- =col2= _col3_'
>>> s | cutsi(r'_(COL[^_]*)_')
'col3'
```

## 2.10 echo

**class** textops.**echo**()

identity operation

it returns the same text, except that is uses textops Extended classes (StrExt, ListExt ...). This could be usefull in some cases to access str methods (upper, replace, ...) just after a pipe.

**Returns** length of the string

**Return type** int

#### Examples

```
>>> s='this is a string'
>>> type(s)
<type 'str'>
>>> t=s | echo()
>>> type(t)
<class 'textops.base.StrExt'>
>>> s.upper()
'THIS IS A STRING'
>>> s | upper()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'upper' is not defined
>>> s | echo().upper()
'THIS IS A STRING'
>>> s | strop.upper()
'THIS IS A STRING'
```

## 2.11 length

**class** textops.**length**()

Returns the length of a string, list or generator

**Returns** length of the string

**Return type** int

#### Examples

```
>>> s='this is a string'
>>> s | length()
16
>>> s=StrExt(s)
>>> s.length()
16
>>> ['a','b','c'] | length()
3
>>> def mygenerator():yield 3; yield 2
>>> mygenerator() | length()
2
```

## 2.12 matches

**class** textops.**matches**(*pattern*)

Tests whether a pattern is present or not

Uses re.match() to match a pattern against the string.

**Parameters** **pattern** (*str*) – a regular expression string

**Returns** The pattern found

**Return type** re.RegexObject

---

**Note:** Be careful : the pattern is tested from the beginning of the string, the pattern is NOT searched somewhere in the middle of the string.

---

#### Examples

```
>>> state=StrExt('good')
>>> print 'OK' if state.matches(r'good|not_present|charging') else 'CRITICAL'
OK
>>> state=StrExt('looks like all is good')
>>> print 'OK' if state.matches(r'good|not_present|charging') else 'CRITICAL'
CRITICAL
>>> print 'OK' if state.matches(r'.*(good|not_present|charging)') else 'CRITICAL'
OK
>>> state=StrExt('Error')
>>> print 'OK' if state.matches(r'good|not_present|charging') else 'CRITICAL'
CRITICAL
```

## 2.13 searches

**class** textops.**searches**(*pattern*)

Search a pattern

Uses re.search() to find a pattern in the string.

**Parameters** **pattern** (*str*) – a regular expression string

**Returns** The pattern found

**Return type** re.RegexObject

#### Examples

```
>>> state=StrExt('good')
>>> print 'OK' if state.searches(r'good|not_present|charging') else 'CRITICAL'
OK
>>> state=StrExt('looks like all is good')
>>> print 'OK' if state.searches(r'good|not_present|charging') else 'CRITICAL'
OK
>>> print 'OK' if state.searches(r'.*(good|not_present|charging)') else 'CRITICAL'
OK
>>> state=StrExt('Error')
>>> print 'OK' if state.searches(r'good|not_present|charging') else 'CRITICAL'
CRITICAL
```

## 2.14 splitln

**class** textops.**splitln**()

Transforms a string with newlines into a list of lines

It uses python str.splitlines() : newline separator can be \n or \r or both. They are removed during the process.

**Returns** The splitted text

**Return type** list

#### Example

```
>>> s='this is\na multi-line\nstring'
>>> s | splitln()
['this is', 'a multi-line', 'string']
```

- *genindex*
- *modindex*
- *search*

CHAPTER 3

# listops

This module gathers list/line operations

## 3.1 after

**class** textops.**after**(*pattern*, *get_begin=False*, *key=None*)
Extract lines after a patterns

Works like textops.before except that it will yields all lines from the input AFTER the given pattern has been found.

**Parameters**

- **pattern** (*str or regex or list*) – start yielding lines after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines after the specified pattern

### Examples

```
>>> ['a','b','c','d','e','f'] | after('c').tolist()
['d', 'e', 'f']
>>> ['a','b','c','d','e','f'] | after('c',True).tolist()
['c', 'd', 'e', 'f']
>>> input_text = [{'k':1},{'k':2},{'k':3},{'k':4},{'k':5},{'k':6}]
>>> input_text | after('3',key='k').tolist()
[{'k': 4}, {'k': 5}, {'k': 6}]
>>> input_text >> after('3',key='k')
[{'k': 4}, {'k': 5}, {'k': 6}]
```

## 3.2 afteri

**class** textops.**afteri**(*pattern*, *get_begin=False*, *key=None*)
Extract lines after a patterns (case insensitive)

Works like textops.after except that the pattern is case insensitive.

**Parameters**

- **pattern** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines before the specified pattern

#### Examples

```
>>> ['a','b','c','d','e','f'] | after('C').tolist()
[]
>>> ['a','b','c','d','e','f'] | afteri('C').tolist()
['d', 'e', 'f']
>>> ['a','b','c','d','e','f'] | afteri('C',True).tolist()
['c', 'd', 'e', 'f']
>>> ['a','b','c','d','e','f'] >> afteri('C',True)
['c', 'd', 'e', 'f']
```

## 3.3 before

**class** `textops.`**`before`**(*pattern*, *get_end=False*, *key=None*)

Extract lines before a patterns

Works like `textops.between` except that it requires only the ending pattern : it will yields all lines from the input text beginning until the specified pattern has been reached.

**Parameters**

- **pattern** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_end** (*bool*) – if True : include the line matching the end pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines before the specified pattern

#### Examples

```
>>> ['a','b','c','d','e','f'] | before('c').tolist()
['a', 'b']
>>> ['a','b','c','d','e','f'] | before('c',True).tolist()
['a', 'b', 'c']
>>> input_text = [{'k':1},{'k':2},{'k':3},{'k':4},{'k':5},{'k':6}]
>>> input_text | before('3',key='k').tolist()
[{'k': 1}, {'k': 2}]
>>> input_text >> before('3',key='k')
[{'k': 1}, {'k': 2}]
```

## 3.4 beforei

**class** textops.**beforei**(*pattern*, *get_end=False*, *key=None*)
Extract lines before a patterns (case insensitive)

Works like textops.before except that the pattern is case insensitive.

**Parameters**

- **pattern** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_end** (*bool*) – if True : include the line matching the pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines before the specified pattern

#### Examples

```
>>> ['a','b','c','d','e','f'] | before('C').tolist()
['a', 'b', 'c', 'd', 'e', 'f']
>>> ['a','b','c','d','e','f'] | beforei('C').tolist()
['a', 'b']
>>> ['a','b','c','d','e','f'] | beforei('C',True).tolist()
['a', 'b', 'c']
>>> ['a','b','c','d','e','f'] >> beforei('C',True)
['a', 'b', 'c']
```

## 3.5 between

**class** textops.**between**(*begin*, *end*, *get_begin=False*, *get_end=False*, *key=None*)
Extract lines between two patterns

It will search for the starting pattern then yield lines until it reaches the ending pattern. Pattern can be a string or a Regex object, it can be also a list of strings or Regexs, in this case, all patterns in the list must be matched in the same order, this may be useful to better select some part of the text in some cases.

between works for any kind of list of strings, but also for list of lists and list of dicts. In these cases, one can test only one column or one key but return the whole list/dict.

**Parameters**

- **begin** (*str or regex or list*) – the pattern(s) to reach before yielding lines from the input
- **end** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the begin pattern (Default : False)
- **get_end** (*bool*) – if True : include the line matching the end pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines between two patterns

**Examples**

```
>>> 'a\nb\nc\nd\ne\nf' | between('b','e').tostr()
'c\nd'
>>> 'a\nb\nc\nd\ne\nf' | between('b','e',True,True).tostr()
'b\nc\nd\ne'
>>> ['a','b','c','d','e','f'] | between('b','e').tolist()
['c', 'd']
>>> ['a','b','c','d','e','f'] >> between('b','e')
['c', 'd']
>>> ['a','b','c','d','e','f'] | between('b','e',True,True).tolist()
['b', 'c', 'd', 'e']
>>> input_text = [('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
>>> input_text | between('b','e').tolist()
[('c', 3), ('d', 4)]
>>> input_text = [{'a':1},{'b':2},{'c':3},{'d':4},{'e':5},{'f':6}]
>>> input_text | between('b','e').tolist()
[{'c': 3}, {'d': 4}]
>>> input_text = [{'k':1},{'k':2},{'k':3},{'k':4},{'k':5},{'k':6}]
>>> input_text | between('2','5',key='k').tolist()
[{'k': 3}, {'k': 4}]
>>> input_text = [{'k':1},{'k':2},{'k':3},{'k':4},{'k':5},{'k':6}]
>>> input_text | between('2','5',key='v').tolist()
[]
>>> input_text = [('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
>>> input_text | between('b','e',key=0).tolist()
[('c', 3), ('d', 4)]
>>> input_text = [('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
>>> input_text | between('b','e',key=1).tolist()
[]
>>> s='''Chapter 1
... ------------
... some infos
...
... Chapter 2
... ---------
... infos I want
...
... Chaper 3
... --------
... some other infos'''
>>> print s | between('---',r'^\s*$').tostr()
some infos
>>> print s | between(['Chapter 2','---'],r'^\s*$').tostr()
infos I want
```

## 3.6 betweeni

**class** `textops.`**`betweeni`**(*begin*, *end*, *get_begin=False*, *get_end=False*, *key=None*)

Extract lines between two patterns (case insensitive)

Works like `textops.between` except patterns are case insensitive

**Parameters**

- **begin** (*str or regex or list*) – the pattern(s) to reach before yielding lines from the input

- **end** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the begin pattern (Default : False)
- **get_end** (*bool*) – if True : include the line matching the end pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines between two patterns

#### Examples

```
>>> ['a','b','c','d','e','f'] | between('B','E').tolist()
[]
>>> ['a','b','c','d','e','f'] | betweeni('B','E').tolist()
['c', 'd']
>>> ['a','b','c','d','e','f'] >> betweeni('B','E')
['c', 'd']
```

## 3.7 betweenb

**class** `textops.`**`betweenb`**(*begin*, *end*, *get_begin=False*, *get_end=False*, *key=None*)

Extract lines between two patterns (includes boundaries)

Works like `textops.between` except it return boundaries by default that is get_begin = get_end = True.

**Parameters**

- **begin** (*str or regex or list*) – the pattern(s) to reach before yielding lines from the input
- **end** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the begin pattern (Default : False)
- **get_end** (*bool*) – if True : include the line matching the end pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines between two patterns

#### Examples

```
>>> ['a','b','c','d','e','f'] | betweenb('b','e').tolist()
['b', 'c', 'd', 'e']
>>> ['a','b','c','d','e','f'] >> betweenb('b','e')
['b', 'c', 'd', 'e']
```

## 3.8 betweenbi

**class** `textops.`**`betweenbi`**(*begin*, *end*, *get_begin=False*, *get_end=False*, *key=None*)

Extract lines between two patterns (includes boundaries and case insensitive)

Works like `textops.between` except patterns are case insensitive and it yields boundaries too. That is get_begin = get_end = True.

**Parameters**

- **begin** (*str or regex or list*) – the pattern(s) to reach before yielding lines from the input
- **end** (*str or regex or list*) – no more lines are yield after reaching this pattern(s)
- **get_begin** (*bool*) – if True : include the line matching the begin pattern (Default : False)
- **get_end** (*bool*) – if True : include the line matching the end pattern (Default : False)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str or list or dict* – lines between two patterns

#### Examples

```
>>> ['a','b','c','d','e','f'] | betweenb('B','E').tolist()
[]
>>> ['a','b','c','d','e','f'] | betweenbi('B','E').tolist()
['b', 'c', 'd', 'e']
>>> ['a','b','c','d','e','f'] >> betweenbi('B','E')
['b', 'c', 'd', 'e']
```

## 3.9 cat

**class** `textops.`**`cat`**(*context={}*)

Return the content of the file with the path given in the input text

If a context dict is specified, the path is formatted with that context (str.format)

**Parameters** **context** (*dict*) – The context to format the file path (Optionnal)

**Yields** *str* – the file content lines

#### Examples

```
>>> open('/tmp/testfile.txt','w').write('here is the file content')
>>> '/tmp/testfile.txt' | cat()
<generator object extend_type_gen at ...>
>>> '/tmp/testfile.txt' | cat().tostr()
'here is the file content'
>>> '/tmp/testfile.txt' >> cat()
['here is the file content']
>>> '/tmp/testfile.txt' | cat().upper().tostr()
'HERE IS THE FILE CONTENT'
>>> context = {'path':'/tmp/'}
>>> '{path}testfile.txt' | cat(context)
<generator object extend_type_gen at ...>
>>> '{path}testfile.txt' | cat(context).tostr()
'here is the file content'
>>> cat('/tmp/testfile.txt').s
```

```
'here is the file content'
>>> cat('/tmp/testfile.txt').upper().s
'HERE IS THE FILE CONTENT'
>>> cat('/tmp/testfile.txt').l
['here is the file content']
>>> cat('/tmp/testfile.txt').g
<generator object extend_type_gen at ...>
>>> for line in cat('/tmp/testfile.txt'):
...     print line
...
here is the file content
>>> for bits in cat('/tmp/testfile.txt').grep('content').cut():
...     print bits
...
['here', 'is', 'the', 'file', 'content']
>>> open('/tmp/testfile.txt','w').write('here is the file content\nanother line')
>>> '/tmp/testfile.txt' | cat().tostr()
'here is the file content\nanother line'
>>> '/tmp/testfile.txt' | cat().tolist()
['here is the file content', 'another line']
>>> cat('/tmp/testfile.txt').s
'here is the file content\nanother line'
>>> cat('/tmp/testfile.txt').l
['here is the file content', 'another line']
>>> context = {'path': '/tmp/'}
>>> cat('/{path}/testfile.txt',context).l
['here is the file content', 'another line']
>>> for bits in cat('/tmp/testfile.txt').grep('content').cut():
...     print bits
...
['here', 'is', 'the', 'file', 'content']
```

## 3.10 doreduce

**class** `textops.`**`doreduce`**(*reduce_fn*, *initializer=None*)

Reduce the input text

Uses python reduce() function.

**Parameters**

- **reduce_fn** (*callable*) – a function or a callable to reduce every line.
- **initializer** (*object*) – initial accumulative value (Default : None)

**Returns** reduced value

**Return type** any

### Examples

```
>>> import re
>>> 'a1\nb2\nc3\nd4' | doreduce(lambda x,y:x+re.sub(r'\d','',y),'')
'abcd'
>>> 'a1\nb2\nc3\nd4' >> doreduce(lambda x,y:x+re.sub(r'\d','',y),'')
'abcd'
```

## 3.11 doslice

**class** textops.**doslice**(*begin=0*, *end=sys.maxsize*, *step=1*)
Get lines/items from begin line to end line with some step

**Parameters**

- **begin** (*int*) – first line number to get. must be None or an integer: 0 <= x <= maxint
- **end** (*int*) – end line number (get lines up to end - 1). must be None or an integer: 0 <= x <= maxint
- **step** (*int*) – get every step line (Default : 1)

**Returns** A slice of the original text

**Return type** generator

### Examples

```
>>> s='a\nb\nc\nd\ne\nf'
>>> s | doslice(1,4).tolist()
['b', 'c', 'd']
>>> s >> doslice(1,4)
['b', 'c', 'd']
>>> s >> doslice(2)
['c', 'd', 'e', 'f']
>>> s >> doslice(0,4,2)
['a', 'c']
>>> s >> doslice(None,None,2)
['a', 'c', 'e']
```

## 3.12 first

**class** textops.**first**()
Return the first line/item from the input text

**Returns** the first line/item from the input text

**Return type** StrExt, ListExt or DictExt

### Examples

```
>>> 'a\nb\nc' | first()
'a'
>>> ['a','b','c'] | first()
'a'
>>> [('a',1),('b',2),('c',3)] | first()
['a', 1]
>>> [['key1','val1','help1'],['key2','val2','help2']] | first()
['key1', 'val1', 'help1']
>>> [{'key':'a','val':1},{'key':'b','val':2},{'key':'c','val':3}] | first()
{'key': 'a', 'val': 1}
```

## 3.13 formatdicts

**class** textops.**formatdicts**(*format_str='{key} : {val}\n'*, *join_str = ''*, *defvalue='-'*)
Formats list of dicts

Useful to convert list of dicts into a simple string. It converts the list of dicts into a list of strings by using the format_str, then it joins all the strings with join_str to get a unique simple string.

**Parameters**

- **format_str** (*str*) – format string, default is '{key} : {val}n'
- **join_str** (*str*) – string to join all strings into one unique string, default is ''
- **defvalue** (*str*) – The replacement string or function for unexisting keys when formating.

**Returns** formatted input

**Return type** str

#### Examples

```
>>> input = [{'key':'a','val':1},{'key':'b','val':2},{'key':'c'}]
>>> input | formatdicts()
'a : 1\nb : 2\nc : -\n'
>>> input | formatdicts('{key} -> {val}\n',defvalue='N/A')
'a -> 1\nb -> 2\nc -> N/A\n'
>>> input = [{'name':'Eric','age':47,'level':'guru'},
... {'name':'Guido','age':59,'level':'god'}]
>>> print input | formatdicts('{name}({age}) : {level}\n')
Eric(47) : guru
Guido(59) : god
>>> print input | formatdicts('{name}', ', ')
Eric, Guido
```

## 3.14 formatitems

**class** textops.**formatitems**(*format_str='{0} : {1}\n'*, *join_str = ''*)
Formats list of 2-sized tuples

Useful to convert list of 2-sized tuples into a simple string It converts the list of tuple into a list of strings by using the format_str, then it joins all the strings with join_str to get a unique simple string.

**Parameters**

- **format_str** (*str*) – format string, default is '{0} : {1}n'
- **join_str** (*str*) – string to join all strings into one unique string, default is ''

**Returns** formatted input

**Return type** str

#### Examples

```
>>> [('key1','val1'),('key2','val2')] | formatitems('{0} -> {1}\n')
'key1 -> val1\nkey2 -> val2\n'
>>> [('key1','val1'),('key2','val2')] | formatitems('{0}:{1}',', ')
'key1:val1, key2:val2'
```

## 3.15 formatlists

**class** textops.**formatlists**(*format_str='{0} : {1}\n'*, *join_str = ''*)
Formats list of lists

Useful to convert list of lists into a simple string It converts the list of lists into a list of strings by using the `format_str`, then it joins all the strings with `join_str` to get a unique simple string.

**Parameters**

- **format_str** (*str*) – format string, default is '{0} : {1}n'
- **join_str** (*str*) – string to join all strings into one unique string, default is ''

**Returns** formatted input

**Return type** str

#### Examples

```
>>> [['key1','val1','help1'],['key2','val2','help2']] | formatlists('{2} : {0} -> {1}\n')
'help1 : key1 -> val1\nhelp2 : key2 -> val2\n'
>>> [['key1','val1','help1'],['key2','val2','help2']] | formatlists('{0}:{1} ({2})',', ')
'key1:val1 (help1), key2:val2 (help2)'
```

## 3.16 greaterequal

**class** textops.**greaterequal**(*value*, *key=None*)
Extract lines with value strictly less than specified string

It works like `textops.greaterthan` except the test is "greater than or equal to"

**Parameters**

- **value** (*str*) – string to test with
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values greater than or equal to the specified value

#### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | greaterequal('2015-09-14 ccc').tolist()
['2015-09-14 ccc', '2015-11-05 ddd']
>>> logs >> greaterequal('2015-09-14 ccc')
['2015-09-14 ccc', '2015-11-05 ddd']
```

## 3.17 greaterthan

**class** `textops.`**`greaterthan`**(*value*, *key=None*)

Extract lines with value strictly less than specified string

It works like `textops.lessthan` except the test is "greater than"

**Parameters**

- **value** (*str*) – string to test with
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values greater than the specified value

#### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | greaterthan('2015-09-14 ccc').tolist()
['2015-11-05 ddd']
>>> logs >> greaterthan('2015-09-14 ccc')
['2015-11-05 ddd']
```

## 3.18 grep

**class** `textops.`**`grep`**(*pattern*, *key=None*)

Select lines having a specified pattern

This works like the shell command 'egrep' : it will filter the input text and retain only lines matching the pattern.

It works for any kind of list of strings, but also for list of lists and list of dicts. In these cases, one can test only one column or one key but return the whole list/dict. before testing, the object to be tested is converted into a string with str() so the grep will work for any kind of object.

**Parameters**

- **pattern** (*str*) – a regular expression string (case sensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str, list or dict* – the filtered input text

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grep('error')
<generator object extend_type_gen at ...>
>>> input | grep('error').tolist()
['error1', 'error2']
>>> input >> grep('error')
['error1', 'error2']
>>> input | grep('ERROR').tolist()
[]
>>> input | grep('error|warning').tolist()
['error1', 'error2', 'warning1', 'warning2']
>>> input | cutca(r'(\D+)(\d+)')
[('error', '1'), ('error', '2'), ('warning', '1'),
('info', '1'), ('warning', '2'), ('info', '2')]
>>> input | cutca(r'(\D+)(\d+)').grep('1',1).tolist()
[('error', '1'), ('warning', '1'), ('info', '1')]
>>> input | cutdct(r'(?P<level>\D+)(?P<nb>\d+)')
[{'nb': '1', 'level': 'error'}, {'nb': '2', 'level': 'error'},
{'nb': '1', 'level': 'warning'}, {'nb': '1', 'level': 'info'},
{'nb': '2', 'level': 'warning'}, {'nb': '2', 'level': 'info'}]
>>> input | cutdct(r'(?P<level>\D+)(?P<nb>\d+)').grep('1','nb').tolist()
[{'nb': '1', 'level': 'error'}, {'nb': '1', 'level': 'warning'},
{'nb': '1', 'level': 'info'}]
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grep('way').tolist()
[{'way to grep': 2}]
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grep('3').tolist()
[{'list of dicts': 3}]
```

## 3.19 grepi

**class** `textops.`**`grepi`**(*pattern*, *key=None*)

grep case insensitive

This works like `textops.grep`, except it is case insensitive.

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str, list or dict* – the filtered input text

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepi('ERROR').tolist()
['error1', 'error2']
>>> input >> grepi('ERROR')
['error1', 'error2']
```

## 3.20 grepv

**class** textops.**grepv**(*pattern*, *key=None*)
grep with inverted matching

This works like textops.grep, except it returns lines that does NOT match the specified pattern.

**Parameters**

- **pattern** (*str*) – a regular expression string
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str, list or dict* – the filtered input text

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepv('error').tolist()
['warning1', 'info1', 'warning2', 'info2']
>>> input >> grepv('error')
['warning1', 'info1', 'warning2', 'info2']
>>> input | grepv('ERROR').tolist()
['error1', 'error2', 'warning1', 'info1', 'warning2', 'info2']
```

## 3.21 grepvi

**class** textops.**grepvi**(*pattern*, *key=None*)
grep case insensitive with inverted matching

This works like textops.grepv, except it is case insensitive.

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Yields** *str, list or dict* – the filtered input text

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepvi('ERROR').tolist()
['warning1', 'info1', 'warning2', 'info2']
```

```
>>> input >> grepvi('ERROR')
['warning1', 'info1', 'warning2', 'info2']
```

## 3.22 grepc

**class** textops.**grepc**(*pattern*, *key=None*)
Count lines having a specified pattern

This works like textops.grep except that instead of filtering the input text, it counts lines matching the pattern.

**Parameters**

- **pattern** (*str*) – a regular expression string (case sensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** the matched lines count

**Return type** int

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepc('error')
2
>>> input | grepc('ERROR')
0
>>> input | grepc('error|warning')
4
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grepc('3')
1
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grepc('2','way to grep')
1
```

## 3.23 grepci

**class** textops.**grepci**(*pattern*, *key=None*)
Count lines having a specified pattern (case insensitive)

This works like textops.grepc except that the pattern is case insensitive

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** the matched lines count

**Return type** int

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepci('ERROR')
2
```

## 3.24 grepcv

**class** textops.**grepcv**(*pattern*, *key=None*)
Count lines NOT having a specified pattern

This works like textops.grepc except that it counts line that does NOT match the pattern.

**Parameters**

- **pattern** (*str*) – a regular expression string (case sensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** the NOT matched lines count

**Return type** int

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepcv('error')
4
>>> input | grepcv('ERROR')
6
```

## 3.25 grepcvi

**class** textops.**grepcvi**(*pattern*, *key=None*)
Count lines NOT having a specified pattern (case insensitive)

This works like textops.grepcv except that the pattern is case insensitive

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** the NOT matched lines count

**Return type** int

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepcvi('ERROR')
4
```

## 3.26 haspattern

**class** textops.**haspattern**(*pattern*, *key=None*)
Tests if the input text matches the specified pattern

This reads the input text line by line (or item by item for lists and generators), cast into a string before testing. like textops.grepc it accepts testing on a specific column for a list of lists or testing on a specific key for list of dicts. It stops reading the input text as soon as the pattern is found : it is useful for big input text.

**Parameters**

- **pattern** (*str*) – a regular expression string (case sensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** True if the pattern is found.

**Return type** bool

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | haspattern('error')
True
>>> input | haspattern('ERROR')
False
```

## 3.27 haspatterni

**class** textops.**haspatterni**(*pattern*, *key=None*)
Tests if the input text matches the specified pattern

Works like textops.haspattern except that it is case insensitive.

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive)
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** True if the pattern is found.

**Return type** bool

#### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | haspatterni('ERROR')
True
```

## 3.28 head

**class** textops.**head**(*lines*)

Return first lines from the input text

**Parameters** **lines** (*int*) – The number of lines/items to return.

**Yields** *str, lists or dicts* – the first 'lines' lines from the input text

### Examples

```
>>> 'a\nb\nc' | head(2).tostr()
'a\nb'
>>> for l in 'a\nb\nc' | head(2):
...   print l
a
b
>>> ['a','b','c'] | head(2).tolist()
['a', 'b']
>>> ['a','b','c'] >> head(2)
['a', 'b']
>>> [('a',1),('b',2),('c',3)] | head(2).tolist()
[('a', 1), ('b', 2)]
>>> [{'key':'a','val':1},{'key':'b','val':2},{'key':'c','val':3}] | head(2).tolist()
[{'val': 1, 'key': 'a'}, {'val': 2, 'key': 'b'}]
```

## 3.29 iffn

**class** textops.**iffn**(*filter_fn=None*)

Filters the input text with a specified function

It works like the python filter() fonction.

**Parameters**

- **filter_fn** (*callable*) – a function to be called against each line and returning a boolean.
- **means** (*True*) – yield the line.

**Yields** *any* – lines filtered by the filter_fn function

### Examples

```
>>> import re
>>> 'line1\nline2\nline3\nline4' | iffn(lambda l:int(re.sub(r'\D','',l)) % 2).tolist()
['line1', 'line3']
>>> 'line1\nline2\nline3\nline4' >> iffn(lambda l:int(re.sub(r'\D','',l)) % 2)
['line1', 'line3']
```

## 3.30 inrange

**class** `textops.`**`inrange`**(*begin*, *end*, *get_begin=True*, *get_end=False*, *key=None*)

Extract lines between a range of strings

For each input line, it tests whether it is greater or equal than `begin` argument and strictly less than `end` argument. At the opposite of textops.between, there no need to match begin or end string.

`inrange` works for any kind of list of strings, but also for list of lists and list of dicts. In these cases, one can test only one column or one key but return the whole list/dict.

Each strings that will be tested is converted with the same type of the first argument.

**Parameters**

- **begin** (*str*) – range begin string
- **end** (*str*) – range end string
- **get_begin** (*bool*) – if True : include lines having the same value as the range begin, Default : True
- **get_end** (*bool*) – if True : include lines having the same value as the range end, Default : False
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values inside the specified range

### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | inrange('2015-08-12','2015-11-05').tolist()
['2015-08-23 bbbb', '2015-09-14 ccc']
>>> logs >> inrange('2015-08-12','2015-11-05')
['2015-08-23 bbbb', '2015-09-14 ccc']

>>> logs = '''aaaa 2015-08-11
... bbbb 2015-08-23
... cccc 2015-09-14
... dddd 2015-11-05'''
>>> logs >> inrange('2015-08-12','2015-11-05')
[]
>>> logs >> inrange('2015-08-12','2015-11-05',key=lambda l:l.cut(col=1))
['bbbb 2015-08-23', 'cccc 2015-09-14']
```

```
>>> logs = [ ('aaaa','2015-08-11'),
... ('bbbb','2015-08-23'),
... ('ccc','2015-09-14'),
... ('ddd','2015-11-05') ]
>>> logs | inrange('2015-08-12','2015-11-05',key=1).tolist()
[('bbbb', '2015-08-23'), ('ccc', '2015-09-14')]

>>> logs = [ {'data':'aaaa','date':'2015-08-11'},
... {'data':'bbbb','date':'2015-08-23'},
... {'data':'ccc','date':'2015-09-14'},
... {'data':'ddd','date':'2015-11-05'} ]
>>> logs | inrange('2015-08-12','2015-11-05',key='date').tolist()
[{'date': '2015-08-23', 'data': 'bbbb'}, {'date': '2015-09-14', 'data': 'ccc'}]

>>> ints = '1\n2\n01\n02\n11\n12\n22\n20'
>>> ints | inrange(1,3).tolist()
['1', '2', '01', '02']
>>> ints | inrange('1','3').tolist()
['1', '2', '11', '12', '22', '20']
>>> ints | inrange('1','3',get_begin=False).tolist()
['2', '11', '12', '22', '20']
```

## 3.31 last

**class** textops.**last**()

Return the last line/item from the input text

**Returns** the last line/item from the input text

**Return type** StrExt, ListExt or DictExt

#### Examples

```
>>> 'a\nb\nc' | last()
'c'
>>> ['a','b','c'] | last()
'c'
>>> [('a',1),('b',2),('c',3)] | last()
['c', 3]
>>> [['key1','val1','help1'],['key2','val2','help2']] | last()
['key2', 'val2', 'help2']
>>> [{'key':'a','val':1},{'key':'b','val':2},{'key':'c','val':3}] | last()
{'key': 'c', 'val': 3}
```

## 3.32 lessequal

**class** textops.**lessequal**(*value*, *key=None*)

Extract lines with value strictly less than specified string

It works like textops.lessthan except the test is “less or equal”

**Parameters**

- **value** (*str*) – string to test with
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values less than or equal to the specified value

#### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | lessequal('2015-09-14').tolist()
['2015-08-11 aaaa', '2015-08-23 bbbb']
>>> logs >> lessequal('2015-09-14')
['2015-08-11 aaaa', '2015-08-23 bbbb']
>>> logs | lessequal('2015-09-14 ccc').tolist()
['2015-08-11 aaaa', '2015-08-23 bbbb', '2015-09-14 ccc']
```

## 3.33 lessthan

**class** `textops.`**`lessthan`**(*value*, *key=None*)

Extract lines with value strictly less than specified string

It works for any kind of list of strings, but also for list of lists and list of dicts. In these cases, one can test only one column or one key but return the whole list/dict.

Each strings that will be tested is temporarily converted with the same type as the first argument given to `lessthan` (see examples).

**Parameters**

- **value** (*str*) – string to test with
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values strictly less than the specified reference value

#### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | lessthan('2015-09-14').tolist()
['2015-08-11 aaaa', '2015-08-23 bbbb']
>>> logs = [ ('aaaa','2015-08-11'),
... ('bbbb','2015-08-23'),
... ('ccc','2015-09-14'),
... ('ddd','2015-11-05') ]
>>> logs | lessthan('2015-11-05',key=1).tolist()
[('aaaa', '2015-08-11'), ('bbbb', '2015-08-23'), ('ccc', '2015-09-14')]
>>> logs = [ {'data':'aaaa','date':'2015-08-11'},
... {'data':'bbbb','date':'2015-08-23'},
... {'data':'ccc','date':'2015-09-14'},
... {'data':'ddd','date':'2015-11-05'} ]
>>> logs | lessthan('2015-09-14',key='date').tolist()
[{'date': '2015-08-11', 'data': 'aaaa'}, {'date': '2015-08-23', 'data': 'bbbb'}]
>>> ints = '1\n2\n01\n02\n11\n12\n22\n20'
>>> ints | lessthan(3).tolist()
['1', '2', '01', '02']
>>> ints | lessthan('3').tolist()
['1', '2', '01', '02', '11', '12', '22', '20']
```

## 3.34 merge_dicts

**class** textops.**merge_dicts**

Merge a list of dicts into one single dict

**Returns** merged dicts

**Return type** dict

#### Examples

```
>>> pattern=r'item="(?P<item>[^"]*)" count="(?P<i_count>[^"]*)" price="(?P<i_price>[^"]*)"'
>>> s='item="col1" count="col2" price="col3"\nitem="col11" count="col22" price="col33"'
>>> s | cutkv(pattern,key_name='item')
[{'col1': {'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'}},...
{'col11': {'item': 'col11', 'i_price': 'col33', 'i_count': 'col22'}}]
>>> s | cutkv(pattern,key_name='item').merge_dicts()
{'col11': {'item': 'col11', 'i_price': 'col33', 'i_count': 'col22'},...
'col1': {'item': 'col1', 'i_price': 'col3', 'i_count': 'col2'}}
```

## 3.35 mapfn

**class** textops.**mapfn**(*map_fn*)

Apply a specified function on every line

It works like the python map() function.

**Parameters** **map_fn** (*callable*) – a function or a callable to apply on every line

**Yields** *any* – lines modified by the map_fn function

#### Examples

```
>>> ['a','b','c'] | mapfn(lambda l:l*2).tolist()
['aa', 'bb', 'cc']
>>> ['a','b','c'] >> mapfn(lambda l:l*2)
['aa', 'bb', 'cc']
```

## 3.36 mapif

**class** textops.**mapif**(*map_fn*, *filter_fn=None*)

Filters and maps the input text with 2 specified functions

Filters input text AND apply a map function on every filtered lines.

**Parameters**

- **map_fn** (*callable*) – a function or a callable to apply on every line to be yield
- **filter_fn** (*callable*) – a function to be called against each line and returning a boolean.
- **means** (*True*) – yield the line.

**Yields** *any* – lines filtered by the filter_fn function and modified by map_fn function

#### Examples

```
>>> import re
>>> 'a1\nb2\nc3\nd4' | mapif(lambda l:l*2,lambda l:int(re.sub(r'\D','',l)) % 2).tolist()
['a1a1', 'c3c3']
>>> 'a1\nb2\nc3\nd4' >> mapif(lambda l:l*2,lambda l:int(re.sub(r'\D','',l)) % 2)
['a1a1', 'c3c3']
```

## 3.37 mrun

**class** textops.**mrun**(*context={}*)

Run multiple commands from the input text and return execution output

This works like `textops.run` except that each line of the input text will be used as a command.
The input text must be a list of strings (list, generator, or newline separated), not a list of lists.
Commands will be executed inside a shell.
If a context dict is specified, commands are formatted with that context (str.format)

**Parameters** **context** (*dict*) – The context to format the command to run

**Yields** *str* – the execution output

**Examples**

```
>>> cmds = 'mkdir -p /tmp/textops_tests_run\n'
>>> cmds+= 'cd /tmp/textops_tests_run;touch f1 f2 f3\n'
>>> cmds+= 'ls /tmp/textops_tests_run'
>>> print cmds | mrun().tostr()
f1
f2
f3
>>> cmds=['mkdir -p /tmp/textops_tests_run',
... 'cd /tmp/textops_tests_run; touch f1 f2 f3']
>>> cmds.append('ls /tmp/textops_tests_run')
>>> print cmds | mrun().tostr()
f1
f2
f3
>>> print cmds >> mrun()
['f1', 'f2', 'f3']
>>> cmds = ['ls {path}', 'echo "Cool !"']
>>> print cmds | mrun({'path':'/tmp/textops_tests_run'}).tostr()
f1
f2
f3
Cool !
```

## 3.38 outrange

**class** `textops.`**`outrange`**(*begin*, *end*, *get_begin=False*, *get_end=False*, *key=None*)

Extract lines NOT between a range of strings

Works like `textops.inrange` except it yields lines that are NOT in the range

**Parameters**

- **begin** (*str*) – range begin string
- **end** (*str*) – range end string
- **get_begin** (*bool*) – if True : include lines having the same value as the range begin, Default : False
- **get_end** (*bool*) – if True : include lines having the same value as the range end, Default : False
- **key** (*int or str or callable*) – Specify what should really be compared:
  - None : the whole current line,
  - an int : test only the specified column (for list or lists),
  - a string : test only the dict value for the specified key (for list of dicts),
  - a callable : it will receive the line being tested and return the string to really compare.

  Note : `key` argument *MUST BE PASSED BY NAME*

**Yields** *str or list or dict* – lines having values outside the specified range

#### Examples

```
>>> logs = '''2015-08-11 aaaa
... 2015-08-23 bbbb
... 2015-09-14 ccc
... 2015-11-05 ddd'''
>>> logs | outrange('2015-08-12','2015-11-05').tolist()
['2015-08-11 aaaa', '2015-11-05 ddd']
>>> logs | outrange('2015-08-23 bbbb','2015-09-14 ccc').tolist()
['2015-08-11 aaaa', '2015-11-05 ddd']
>>> logs | outrange('2015-08-23 bbbb','2015-09-14 ccc', get_begin=True).tolist()
['2015-08-11 aaaa', '2015-08-23 bbbb', '2015-11-05 ddd']
```

## 3.39 renderdicts

**class** textops.**renderdicts**(*format_str='{key} : {val}'*, *defvalue='-'*)

Formats list of dicts

It works like renderdicts except it does NOT do the final join.

**Parameters**

- **format_str** (*str*) – format string, default is '{key} : {val}n'
- **defvalue** (*str*) – The replacement string or function for unexisting keys when formating.

**Returns** list of formatted string

**Return type** generator of strings

#### Examples

```
>>> input = [{'key':'a','val':1},{'key':'b','val':2},{'key':'c'}]
>>> input >> renderdicts()
['a : 1', 'b : 2', 'c : -']
>>> input >> renderdicts('{key} -> {val}',defvalue='N/A')
['a -> 1', 'b -> 2', 'c -> N/A']
>>> input = [{'name':'Eric','age':47,'level':'guru'},
... {'name':'Guido','age':59,'level':'god'}]
>>> input >> renderdicts('{name}({age}) : {level}')
['Eric(47) : guru', 'Guido(59) : god']
>>> input >> renderdicts('{name}')
['Eric', 'Guido']
```

## 3.40 renderitems

**class** textops.**renderitems**(*format_str='{0} : {1}'*)

Renders list of 2-sized tuples

It works like formatitems except it does NOT do the final join.

**Parameters** **format_str** (*str*) – format string, default is '{0} : {1}'

**Returns** list of formatted string

**Return type** generator of strings

#### Examples

```
>>> [('key1','val1'),('key2','val2')] >> renderitems('{0} -> {1}')
['key1 -> val1', 'key2 -> val2']
>>> [('key1','val1'),('key2','val2')] >> renderitems('{0}:{1}')
['key1:val1', 'key2:val2']
```

## 3.41 renderlists

**class** `textops.`**`renderlists`**(*format_str='{0} : {1}'*)
Formats list of lists

It works like `formatlists` except it does NOT do the final join.

**Parameters** **format_str** (*str*) – format string, default is '{0} : {1}'

**Returns** list of formatted string

**Return type** generator of strings

#### Examples

```
>>> [['key1','val1','help1'],['key2','val2','help2']] >> renderlists('{2} : {0} -> {1}')
['help1 : key1 -> val1', 'help2 : key2 -> val2']
>>> [['key1','val1','help1'],['key2','val2','help2']] >> renderlists('{0}:{1} ({2})')
['key1:val1 (help1)', 'key2:val2 (help2)']
```

## 3.42 resplitblock

**class** `textops.`**`resplitblock`**(*pattern*, *include_separator=0*, *skip_first=False*)
split a text into blocks using `re.finditer()`

This works like `textops.splitblock` except that is uses `re` : it is faster and gives the possibility to search multiple lines patterns. BUT, the whole input text must fit into memory. List of strings are also converted into a single string with newlines during the process.

**Parameters**

- **pattern** (*str*) – The pattern to find
- **include_separator** (*int*) – Tells whether blocks must include searched pattern
  - 0 or SPLIT_SEP_NONE : no,
  - 1 or SPLIT_SEP_BEGIN : yes, at block beginning,
  - 2 or SPLIT_SEP_END : yes, at block ending

  Default: 0
- **skip_first** (*bool*) – If True, the result will not contain the block before the first pattern found. Default : False.

**Returns** splitted input text

**Return type** generator

### Examples

```
>>> s='''
... this
... is
... section 1
... =================
... this
... is
... section 2
... =================
... this
... is
... section 3
... '''
>>> s >> resplitblock(r'^======+$')
['\nthis\nis\nsection 1\n', '\nthis\nis\nsection 2\n', '\nthis\nis\nsection 3\n']
>>> s >> resplitblock(r'^======+$',skip_first=True)
['\nthis\nis\nsection 2\n', '\nthis\nis\nsection 3\n']

>>> s='''Section: 1
... info 1.1
... info 1.2
... Section: 2
... info 2.1
... info 2.2
... Section: 3
... info 3.1
... info 3.2'''
>>> s >> resplitblock(r'^Section:',SPLIT_SEP_BEGIN)
['', 'Section: 1\ninfo 1.1\ninfo 1.2\n', 'Section: 2\ninfo 2.1\ninfo 2.2\n',
'Section: 3\ninfo 3.1\ninfo 3.2']
>>> s >> resplitblock(r'^Section:',SPLIT_SEP_BEGIN,True)
['Section: 1\ninfo 1.1\ninfo 1.2\n', 'Section: 2\ninfo 2.1\ninfo 2.2\n',
'Section: 3\ninfo 3.1\ninfo 3.2']

>>> s='''info 1.1
... Last info 1.2
... info 2.1
... Last info 2.2
... info 3.1
... Last info 3.2'''
>>> s >> resplitblock(r'^Last info[^\n\r]*[\n\r]?',SPLIT_SEP_END)
['info 1.1\nLast info 1.2\n', 'info 2.1\nLast info 2.2\n', 'info 3.1\nLast info 3.2']

>>> s='''
... =========
... Section 1
... =========
... info 1.1
... info 1.2
... =========
... Section 2
... =========
```

```
... info 2.1
... info 2.2
... '''
>>> s >> resplitblock('^===+\n[^\n]+\n===+\n')
['\n', 'info 1.1\ninfo 1.2\n', 'info 2.1\ninfo 2.2\n']
>>> s >> resplitblock('^===+\n[^\n]+\n===+\n',SPLIT_SEP_BEGIN)
['\n', '=========\nSection 1\n=========\ninfo 1.1\ninfo 1.2\n',
'=========\nSection 2\n=========\ninfo 2.1\ninfo 2.2\n']
>>> s >> resplitblock('^===+\n[^\n]+\n===+\n',SPLIT_SEP_BEGIN, True)
['=========\nSection 1\n=========\ninfo 1.1\ninfo 1.2\n',
'=========\nSection 2\n=========\ninfo 2.1\ninfo 2.2\n']
```

## 3.43 run

**class** textops.**run**(*context={}*)

Run the command from the input text and return execution output

This text operation use subprocess.Popen to call the command.
If the command is a string, it will be executed within a shell.
If the command is a list (the command and its arguments), the command is executed without a shell.
If a context dict is specified, the command is formatted with that context (str.format)

**Parameters** **context** (*dict*) – The context to format the command to run

**Yields** *str* – the execution output

### Examples

```
>>> cmd = 'mkdir -p /tmp/textops_tests_run;\
... cd /tmp/textops_tests_run; touch f1 f2 f3; ls'
>>> print cmd | run().tostr()
f1
f2
f3
>>> print cmd >> run()
['f1', 'f2', 'f3']
>>> print ['ls', '/tmp/textops_tests_run'] | run().tostr()
f1
f2
f3
>>> print ['ls', '{path}'] | run({'path':'/tmp/textops_tests_run'}).tostr()
f1
f2
f3
```

## 3.44 sed

**class** textops.**sed**(*pat*, *repl*)

Replace pattern on-the-fly

Works like the shell command 'sed'. It uses `re.sub()` to replace the pattern, this means that you can include back-reference into the replacement string.

**Parameters**

- **pat** (*str*) – a string (case sensitive) or a regular expression for the pattern to search
- **repl** (*str*) – the replace string.

**Yields** *str* – the replaced lines from the input text

#### Examples

```
>>> 'Hello Eric\nHello Guido' | sed('Hello','Bonjour').tostr()
'Bonjour Eric\nBonjour Guido'
>>> [ 'Hello Eric','Hello Guido'] | sed('Hello','Bonjour').tolist()
['Bonjour Eric', 'Bonjour Guido']
>>> [ 'Hello Eric','Hello Guido'] >> sed('Hello','Bonjour')
['Bonjour Eric', 'Bonjour Guido']
>>> [ 'Hello Eric','Hello Guido'] | sed(r'$',' !').tolist()
['Hello Eric !', 'Hello Guido !']
>>> import re
>>> [ 'Hello Eric','Hello Guido'] | sed(re.compile('hello',re.I),'Good bye').tolist()
['Good bye Eric', 'Good bye Guido']
>>> [ 'Hello Eric','Hello Guido'] | sed('hello','Good bye').tolist()
['Hello Eric', 'Hello Guido']
```

## 3.45 sedi

**class** `textops.`**`sedi`**(*pat*, *repl*)

Replace pattern on-the-fly (case insensitive)

Works like `textops.sed` except that the string as the search pattern is case insensitive.

**Parameters**

- **pat** (*str*) – a string (case insensitive) or a regular expression for the pattern to search
- **repl** (*str*) – the replace string.

**Yields** *str* – the replaced lines from the input text

#### Examples

```
>>> [ 'Hello Eric','Hello Guido'] | sedi('hello','Good bye').tolist()
['Good bye Eric', 'Good bye Guido']
>>> [ 'Hello Eric','Hello Guido'] >> sedi('hello','Good bye')
['Good bye Eric', 'Good bye Guido']
```

## 3.46 skip

**class** `textops.`**`skip`**(*lines*)

Skip n lines

It will return the input text except the n first lines

**Parameters** **lines** (*int*) – The number of lines/items to skip.

**Yields** *str, lists or dicts* – skip 'lines' lines from the input text

#### Examples

```
>>> 'a\nb\nc' | skip(1).tostr()
'b\nc'
>>> for l in 'a\nb\nc' | skip(1):
...   print l
b
c
>>> ['a','b','c'] | skip(1).tolist()
['b', 'c']
>>> ['a','b','c'] >> skip(1)
['b', 'c']
>>> [('a',1),('b',2),('c',3)] | skip(1).tolist()
[('b', 2), ('c', 3)]
>>> [{'key':'a','val':1},{'key':'b','val':2},{'key':'c','val':3}] | skip(1).tolist()
[{'val': 2, 'key': 'b'}, {'val': 3, 'key': 'c'}]
```

## 3.47 span

**class** `textops.`**`span`**(*nbcols*, *fill_str=''*)

Ensure that a list of lists has exactly the specified number of column

This is useful in for-loop with multiple assignment

**Parameters**

- **nbcols** (*int*) – number columns to return
- **fill_str** (*str*) – the value to return for not exsiting columns

**Returns** A list with exactly `nbcols` columns

**Return type** list

#### Examples

```
>>> s='a\nb c\nd e f g h\ni j k\n\n'
>>> s | cut()
[['a'], ['b', 'c'], ['d', 'e', 'f', 'g', 'h'], ['i', 'j', 'k'], []]
>>> s | cut().span(3,'-').tolist()
[['a', '-', '-'], ['b', 'c', '-'], ['d', 'e', 'f'], ['i', 'j', 'k'], ['-', '-', '-']]
>>> s >> cut().span(3,'-')
[['a', '-', '-'], ['b', 'c', '-'], ['d', 'e', 'f'], ['i', 'j', 'k'], ['-', '-', '-']]
>>> for x,y,z in s | cut().span(3,'-'):
...    print x,y,z
a - -
b c -
d e f
i j k
- - -
```

## 3.48 splitblock

**class** textops.**splitblock**(*pattern*, *include_separator=0*, *skip_first=False*)

split a text into blocks

This operation split a text that has several blocks seperated by a same pattern. The separator pattern must fit into one line, by this way, this operation is not limited with the input text size, nevertheless one block must fit in memory (ie : input text can include an unlimited number of blocks that must fit into memory one-by-one)

**Parameters**

- **pattern** (*str*) – The pattern to find
- **include_separator** (*int*) – Tells whether blocks must include searched pattern
  - 0 or SPLIT_SEP_NONE : no,
  - 1 or SPLIT_SEP_BEGIN : yes, at block beginning,
  - 2 or SPLIT_SEP_END : yes, at block ending

  Default: 0
- **skip_first** (*bool*) – If True, the result will not contain the block before the first pattern found. Default : False.

**Returns** splitted input text

**Return type** generator

### Examples

```
>>> s='''
... this
... is
... section 1
... ==================
... this
... is
... section 2
... ==================
... this
... is
... section 3
... '''
>>> s >> splitblock(r'^======+$')
[['', 'this', 'is', 'section 1'], ['this', 'is', 'section 2'], ['this', 'is', 'section 3']]
>>> s >> splitblock(r'^======+$',skip_first=True)
[['this', 'is', 'section 2'], ['this', 'is', 'section 3']]

>>> s='''Section: 1
... info 1.1
... info 1.2
... Section: 2
... info 2.1
... info 2.2
... Section: 3
... info 3.1
... info 3.2'''
```

```
>>> s >> splitblock(r'^Section:',SPLIT_SEP_BEGIN)
[[], ['Section: 1', 'info 1.1', 'info 1.2'], ['Section: 2', 'info 2.1', 'info 2.2'],
['Section: 3', 'info 3.1', 'info 3.2']]
>>> s >> splitblock(r'^Section:',SPLIT_SEP_BEGIN,True)
[['Section: 1', 'info 1.1', 'info 1.2'], ['Section: 2', 'info 2.1', 'info 2.2'],
['Section: 3', 'info 3.1', 'info 3.2']]

>>> s='''info 1.1
... Last info 1.2
... info 2.1
... Last info 2.2
... info 3.1
... Last info 3.2'''
>>> s >> splitblock(r'^Last info',SPLIT_SEP_END)
[['info 1.1', 'Last info 1.2'], ['info 2.1', 'Last info 2.2'],
['info 3.1', 'Last info 3.2']]
```

## 3.49 subslice

**class** textops.**subslice**(*begin=0*, *end=sys.maxsize*, *step=1*)

Get a slice of columns for list of lists

**Parameters**

- **begin** (*int*) – first columns number to get. must be None or an integer: 0 <= x <= maxint
- **end** (*int*) – end columns number (get columns up to end - 1). must be None or an integer: 0 <= x <= maxint
- **step** (*int*) – get every `step` columns (Default : 1)

**Returns** A slice of the original text

**Return type** generator

### Examples

```
>>> s='a\nb c\nd e f g h\ni j k\n\n'
>>> s | cut().span(3,'-').tolist()
[['a', '-', '-'], ['b', 'c', '-'], ['d', 'e', 'f'], ['i', 'j', 'k'], ['-', '-', '-']]
>>> s | cut().span(3,'-').subslice(1,3).tolist()
[['-', '-'], ['c', '-'], ['e', 'f'], ['j', 'k'], ['-', '-']]
>>> s >> cut().span(3,'-').subslice(1,3)
[['-', '-'], ['c', '-'], ['e', 'f'], ['j', 'k'], ['-', '-']]
```

## 3.50 subitem

**class** textops.**subitem**(*n*)

Get a specified column for list of lists

**Parameters** **n** (*int*) – column number to get.

**Returns** A list

**Return type** generator

#### Examples

```
>>> s='a\nb c\nd e f g h\ni j k\n\n'
>>> s | cut().span(3,'-').tolist()
[['a', '-', '-'], ['b', 'c', '-'], ['d', 'e', 'f'], ['i', 'j', 'k'], ['-', '-', '-']]
>>> s | cut().span(3,'-').subitem(1).tolist()
['-', 'c', 'e', 'j', '-']
>>> s >> cut().span(3,'-').subitem(1)
['-', 'c', 'e', 'j', '-']
>>> s >> cut().span(3,'-').subitem(-1)
['-', '-', 'f', 'k', '-']
```

## 3.51 subitems

**class** textops.**subitem**(*ntab*)

Get a specified column for list of lists

**Parameters** **n** (*int*) – column number to get.

**Returns** A list

**Return type** generator

#### Examples

```
>>> s='a\nb c\nd e f g h\ni j k\n\n'
>>> s | cut().span(3,'-').tolist()
[['a', '-', '-'], ['b', 'c', '-'], ['d', 'e', 'f'], ['i', 'j', 'k'], ['-', '-', '-']]
>>> s | cut().span(3,'-').subitem(1).tolist()
['-', 'c', 'e', 'j', '-']
>>> s >> cut().span(3,'-').subitem(1)
['-', 'c', 'e', 'j', '-']
>>> s >> cut().span(3,'-').subitem(-1)
['-', '-', 'f', 'k', '-']
```

## 3.52 tail

**class** textops.**tail**(*lines*)

Return last lines from the input text

**Parameters** **lines** (*int*) – The number of lines/items to return.

**Yields** *str, lists or dicts* – the last 'lines' lines from the input text

#### Examples

```
>>> 'a\nb\nc' | tail(2).tostr()
'b\nc'
>>> for l in 'a\nb\nc' | tail(2):
...   print l
b
c
>>> ['a','b','c'] | tail(2).tolist()
['b', 'c']
>>> ['a','b','c'] >> tail(2)
['b', 'c']
>>> [('a',1),('b',2),('c',3)] | tail(2).tolist()
[('b', 2), ('c', 3)]
>>> [{'key':'a','val':1},{'key':'b','val':2},{'key':'c','val':3}] | tail(2).tolist()
[{'val': 2, 'key': 'b'}, {'val': 3, 'key': 'c'}]
```

## 3.53 uniq

**class** textops.**uniq**

Remove all line repetitions

If a line is many times in the same text (even if there are some different lines between), only the first will be taken. Works also with list of lists or dicts.

**Returns** Unified text line by line.

**Return type** generator

#### Examples

```
>>> s='f\na\nb\na\nc\nc\ne\na\nc\nf'
>>> s >> uniq()
['f', 'a', 'b', 'c', 'e']
>>> for line in s | uniq():
...     print line
f
a
b
c
e
>>> l = [ [1,2], [3,4], [1,2] ]
>>> l >> uniq()
[[1, 2], [3, 4]]
>>> d = [ {'a':1}, {'b':2}, {'a':1} ]
>>> d >> uniq()
[{'a': 1}, {'b': 2}]
```

- *genindex*
- *modindex*
- *search*

CHAPTER 4

# warpops

This module gathers text operations that are wrapped from standard python functions

## 4.1 alltrue

**class** textops.**alltrue**

Return True if all elements of the input are true

**Returns** True if all elements of the input are true

**Return type** bool

#### Examples

```
>>> '\n\n' | alltrue()
False
>>> 'a\n\n' | alltrue()
False
>>> 'a\nb\n' | alltrue()
True
>>> 'a\nb\nc' | alltrue()
True
>>> ['',''] >> alltrue()
False
>>> ['1','2'] >> alltrue()
True
>>> [True,False] >> alltrue()
False
>>> [True,True] >> alltrue()
True
```

## 4.2 anytrue

**class** textops.**anytrue**

Return True if any element of the input is true

**Returns** True if any element of the input is true

**Return type** bool

#### Examples

```
>>> '\n\n' | anytrue()
False
>>> 'a\n\n' | anytrue()
True
>>> 'a\nb\n' | anytrue()
True
>>> 'a\nb\nc' | anytrue()
True
>>> [0,0] >> anytrue()
False
>>> [0,1] >> anytrue()
True
>>> [1,2] >> anytrue()
True
>>> [False,False] >> anytrue()
False
>>> [True,False] >> anytrue()
True
>>> [True,True] >> anytrue()
True
```

## 4.3 dosort

**class** `textops.`**`dosort`**(*[cmp[, key[, reverse]]]*)

Sort input text

Return a new sorted list from the input text. The sorting is done on a by-line/list item basis.

**Parameters**

- **cmp** (*callable*) – specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument, ex: `cmp=lambda x,y: cmp(x.lower(), y.lower())`. The default value is None.
- **key** (*callable*) – specifies a function of one argument that is used to extract a comparison key from each list element, ex: `key=str.lower`. The default value is None (compare the elements directly).
- **reverse** (*bool*) – If set to True, then the list elements are sorted as if each comparison were reversed.

**Returns** The sorted input text

**Return type** generator

#### Examples

```
>>> 'a\nd\nc\nb' | dosort().tolist()
['a', 'b', 'c', 'd']
>>> 'a\nd\nc\nb' >> dosort()
['a', 'b', 'c', 'd']
```

```
>>> 'a\nd\nc\nb' >> dosort(reverse=True)
['d', 'c', 'b', 'a']
>>> 'a\nB\nc' >> dosort()
['B', 'a', 'c']
>>> 'a\nB\nc' >> dosort(cmp=lambda x,y:cmp(x.lower(),y.lower()))
['a', 'B', 'c']
>>> [('a',3),('c',1),('b',2)] >> dosort()
[('a', 3), ('b', 2), ('c', 1)]
>>> [('a',3),('c',1),('b',2)] >> dosort(key=lambda x:x[1])
[('c', 1), ('b', 2), ('a', 3)]
>>> [{'k':3},{'k':1},{'k':2}] >> dosort(key=lambda x:x['k'])
[{'k': 1}, {'k': 2}, {'k': 3}]
```

## 4.4 getmax

**class** textops.**getmax**([*key*])
get the max value

Return the largest item/line from the input.

**Parameters** **key** (*callable*) – specifies a function of one argument that is used to extract a comparison key from each list element, ex: `key=str.lower`.

**Returns** The largest item/line

**Return type** object

### Examples

```
>>> 'E\nc\na' | getmax()
'c'
>>> 'E\nc\na' >> getmax()
'c'
>>> 'E\nc\na' >> getmax(key=str.lower)
'E'
```

## 4.5 getmin

**class** textops.**getmin**([*key*])
get the min value

Return the smallest item/line from the input.

**Parameters** **key** (*callable*) – specifies a function of one argument that is used to extract a comparison key from each list element, ex: `key=str.lower`.

**Returns** The smallest item/line

**Return type** object

#### Examples

```
>>> 'c\nE\na' | getmin()
'E'
>>> 'c\nE\na' >> getmin()
'E'
>>> 'c\nE\na' >> getmin(key=str.lower)
'a'
```

## 4.6 linenbr

**class** textops.**linenbr**(*start=0*)
Enumerate input text lines

add a column to the input text with the line number within.

**Parameters** **start** (*int*) – starting number (default : 0)

**Returns** input text with line numbering

**Return type** generator

#### Examples

```
>>> 'a\nb\nc' >> linenbr()
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> 'a\nb\nc' | linenbr(1).tolist()
[(1, 'a'), (2, 'b'), (3, 'c')]
```

## 4.7 resub

**class** textops.**resub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)
Substitute a regular expression within a string or a list of strings

It uses `re.sub()` to replace the input text.

**Parameters**

- **pattern** (*str*) – Split string by the occurrences of pattern
- **repl** (*str*) – Replacement string.
- **count** (*int*) – the maximum number of pattern occurrences to be replaced
- **flags** (*int*) – regular expression flags (re.I etc...). Only available in Python 2.7+

**Returns** The replaced text

**Return type** str or list

#### Examples

```
>>> 'Words, words, words.' | resub('[Ww]ords','Mots')
'Mots, Mots, Mots.'
>>> ['Words1 words2', 'words', 'words.' ] >> resub('[Ww]ords','Mots',1)
['Mots1 words2', 'Mots', 'Mots.']
```

- *genindex*
- *modindex*
- *search*

CHAPTER 5

# parse

This module gathers parsers to handle whole input text

## 5.1 find_first_pattern

**class** textops.**find_first_pattern**(*patterns*)
Fast multiple pattern search, returns on first match

It works like textops.find_patterns except that it stops searching on first match.

**Parameters** **patterns** (*list*) – a list of patterns.

**Returns** matched value if only one capture group otherwise the full groupdict

**Return type** str or dict

#### Examples

```
>>> s = '''creation: 2015-10-14
... update: 2015-11-16
... access: 2015-11-17'''
>>> s | find_first_pattern([r'^update:\s*(.*)', r'^access:\s*(.*)', r'^creation:\s*(.*)'])
'2015-11-16'
>>> s | find_first_pattern([r'^UPDATE:\s*(.*)'])
NoAttr
>>> s | find_first_pattern([r'^update:\s*(?P<year>.*)-(?P<month>.*)-(?P<day>.*)'])
{'year': '2015', 'day': '16', 'month': '11'}
```

## 5.2 find_first_patterni

**class** textops.**find_first_patterni**(*patterns*)
Fast multiple pattern search, returns on first match

It works like textops.find_first_pattern except that patterns are case insensitives.

**Parameters** **patterns** (*list*) – a list of patterns.

**Returns** matched value if only one capture group otherwise the full groupdict

**Return type** str or dict

#### Examples

```
>>> s = '''creation: 2015-10-14
... update: 2015-11-16
... access: 2015-11-17'''
>>> s | find_first_patterni([r'^UPDATE:\s*(.*)'])
'2015-11-16'
```

## 5.3 find_pattern

**class** textops.**find_pattern**(*pattern*)

Fast pattern search

This operation can be use to find a pattern very fast : it uses `re.search()` on the whole input text at once. The input text is not read line by line, this means it must fit into memory. It returns the first captured group (named or not named group).

**Parameters** **pattern** (*str*) – a regular expression string (case sensitive).

**Returns** the first captured group or NoAttr if not found

**Return type** str

#### Examples

```
>>> s = '''This is data text
... Version: 1.2.3
... Format: json'''
>>> s | find_pattern(r'^Version:\s*(.*)')
'1.2.3'
>>> s | find_pattern(r'^Format:\s*(?P<format>.*)')
'json'
>>> s | find_pattern(r'^version:\s*(.*)') # 'version' : no match because case sensitive
NoAttr
```

## 5.4 find_patterni

**class** textops.**find_patterns**(*pattern*)

Fast multiple pattern search

It works like `textops.find_pattern` except that one can specify a list or a dictionary of patterns. Patterns must contains capture groups. It returns a list or a dictionary of results depending on the patterns argument type. Each result will be the re.MatchObject groupdict if there are more than one capture named group in the pattern otherwise directly the value corresponding to the unique captured group. It is recommended to use *named* capture group, if not, the groups will be automatically named 'groupN' with N the capture group order in the pattern.

**Parameters** **patterns** (*list or dict*) – a list or a dictionary of patterns.

**Returns** patterns search result

**Return type** dict

#### Examples

```
>>> s = '''This is data text
... Version: 1.2.3
... Format: json'''
>>> r = s | find_patterns({
... 'version':r'^Version:\s*(?P<major>\d+)\.(?P<minor>\d+)\.(?P<build>\d+)',
... 'format':r'^Format:\s*(?P<format>.*)',
... })
>>> r
{'version': {'major': '1', 'build': '3', 'minor': '2'}, 'format': 'json'}
>>> r.version.major
'1'
>>> s | find_patterns({
... 'version':r'^Version:\s*(\d+)\.(\d+)\.(\d+)',
... 'format':r'^Format:\s*(.*)',
... })
{'version': {'group1': '2', 'group0': '1', 'group2': '3'}, 'format': 'json'}
>>> s | find_patterns({'version':r'^version:\s*(.*)'}) # lowercase 'version' : no match
{}
>>> s = '''creation: 2015-10-14
... update: 2015-11-16
... access: 2015-11-17'''
>>> s | find_patterns([r'^update:\s*(.*)', r'^access:\s*(.*)', r'^creation:\s*(.*)'])
['2015-11-16', '2015-11-17', '2015-10-14']
>>> s | find_patterns([r'^update:\s*(?P<year>.*)-(?P<month>.*)-(?P<day>.*)',
... r'^access:\s*(.*)', r'^creation:\s*(.*)'])
[{'month': '11', 'day': '16', 'year': '2015'}, '2015-11-17', '2015-10-14']
```

## 5.5 find_patterns

**class** textops.**find_patterns**(*patterns*)

Fast multiple pattern search

It works like `textops.find_pattern` except that one can specify a list or a dictionary of patterns. Patterns must contains capture groups. It returns a list or a dictionary of results depending on the patterns argument type. Each result will be the re.MatchObject groupdict if there are more than one capture named group in the pattern otherwise directly the value corresponding to the unique captured group. It is recommended to use *named* capture group, if not, the groups will be automatically named 'groupN' with N the capture group order in the pattern.

**Parameters** **patterns** (*list or dict*) – a list or a dictionary of patterns.

**Returns** patterns search result

**Return type** dict

#### Examples

```
>>> s = '''This is data text
... Version: 1.2.3
... Format: json'''
>>> r = s | find_patterns({
... 'version':r'^Version:\s*(?P<major>\d+)\.(?P<minor>\d+)\.(?P<build>\d+)',
... 'format':r'^Format:\s*(?P<format>.*)',
```

```
... })
>>> r
{'version': {'major': '1', 'build': '3', 'minor': '2'}, 'format': 'json'}
>>> r.version.major
'1'
>>> s | find_patterns({
... 'version':r'^Version:\s*(\d+)\.(\d+)\.(\d+)',
... 'format':r'^Format:\s*(.*)',
... })
{'version': {'group1': '2', 'group0': '1', 'group2': '3'}, 'format': 'json'}
>>> s | find_patterns({'version':r'^version:\s*(.*)'}) # lowercase 'version' : no match
{}
>>> s = '''creation: 2015-10-14
... update: 2015-11-16
... access: 2015-11-17'''
>>> s | find_patterns([r'^update:\s*(.*)', r'^access:\s*(.*)', r'^creation:\s*(.*)'])
['2015-11-16', '2015-11-17', '2015-10-14']
>>> s | find_patterns([r'^update:\s*(?P<year>.*)-(?P<month>.*)-(?P<day>.*)',
... r'^access:\s*(.*)', r'^creation:\s*(.*)'])
[{'month': '11', 'day': '16', 'year': '2015'}, '2015-11-17', '2015-10-14']
```

## 5.6 find_patternsi

**class** textops.**find_patternsi**(*patterns*)

Fast multiple pattern search (case insensitive)

It works like `textops.find_patterns` except that patterns are case insensitive.

**Parameters** **patterns** (*dict*) – a dictionary of patterns.

**Returns** patterns search result

**Return type** dict

#### Examples

```
>>> s = '''This is data text
... Version: 1.2.3
... Format: json'''
>>> s | find_patternsi({'version':r'^version:\s*(.*)'})      # case insensitive
{'version': '1.2.3'}
```

## 5.7 index_normalize

textops.**index_normalize**(*index_val*)

Normalize dictionary calculated key

When parsing, keys within a dictionary may come from the input text. To ensure there is no space or other special caracters, one should use this function. This is useful because DictExt dictionaries can be access with a dotted notation that only supports `A-Za-z0-9_` chars.

**Parameters** **index_val** (*str*) – The candidate string to a dictionary key.

**Returns** A normalized string with only `A-Za-z0-9_` chars

**Return type** str

#### Examples

```
>>> index_normalize('this my key')
'this_my_key'
>>> index_normalize('this -my- %key%')
'this_my_key'
```

## 5.8 mgrep

**class** textops.**mgrep**(*patterns_dict*, *key=None*)

Multiple grep

This works like `textops.grep` except that it can do several greps in a single command. By this way, you can select many patterns in a big file.

**Parameters**

- **patterns_dict** (*dict*) – a dictionary where all patterns to search are in values.
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** A dictionary where the keys are the same as for `patterns_dict`, the values will contain the `textops.grep` result for each corresponding patterns.

**Return type** dict

#### Examples

```
>>> logs = '''
... error 1
... warning 1
... warning 2
... info 1
... error 2
... info 2
... '''
>>> t = logs | mgrep({
... 'errors' : r'^err',
... 'warnings' : r'^warn',
... 'infos' : r'^info',
... })
>>> print t
{'infos': ['info 1', 'info 2'],
'errors': ['error 1', 'error 2'],
'warnings': ['warning 1', 'warning 2']}

>>> s = '''
... Disk states
... -----------
... name: c1t0d0s0
... state: good
... fs: /
... name: c1t0d0s4
```

```
... state: failed
... fs: /home
...
... '''
>>> t = s | mgrep({
... 'disks' : r'^name:',
... 'states' : r'^state:',
... 'fss' : r'^fs:',
... })
>>> print t
{'states': ['state: good', 'state: failed'],
'disks': ['name: c1t0d0s0', 'name: c1t0d0s4'],
'fss': ['fs: /', 'fs: /home']}
>>> dict(zip(t.disks.cutre(': *',1),zip(t.states.cutre(': *',1),t.fss.cutre(': *',1))))
{'c1t0d0s0': ('good', '/'), 'c1t0d0s4': ('failed', '/home')}
```

## 5.9 mgrepi

**class** textops.**mgrepi**(*patterns_dict*, *key=None*)
same as mgrep but case insensitive

This works like textops.mgrep, except it is case insensitive.

**Parameters**

- **patterns_dict** (*dict*) – a dictionary where all patterns to search are in values.
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** A dictionary where the keys are the same as for patterns_dict, the values will contain the textops.grepi result for each corresponding patterns.

**Return type** dict

#### Examples

```
>>> 'error 1' | mgrep({'errors':'ERROR'})
{}
>>> 'error 1' | mgrepi({'errors':'ERROR'})
{'errors': ['error 1']}
```

## 5.10 mgrepv

**class** textops.**mgrepv**(*patterns_dict*, *key=None*)
Same as mgrep but exclusive

This works like textops.mgrep, except it searches lines that DOES NOT match patterns.

**Parameters**

- **patterns_dict** (*dict*) – a dictionary where all patterns to exclude are in values().
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** A dictionary where the keys are the same as for patterns_dict, the values will contain the textops.grepv result for each corresponding patterns.

**Return type** dict

#### Examples

```
>>> logs = '''error 1
... warning 1
... warning 2
... error 2
... '''
>>> t = logs | mgrepv({
... 'not_errors' : r'^err',
... 'not_warnings' : r'^warn',
... })
>>> print t
{'not_warnings': ['error 1', 'error 2'], 'not_errors': ['warning 1', 'warning 2']}
```

## 5.11 mgrepvi

**class** textops.**mgrepvi**(*patterns_dict*, *key=None*)

Same as mgrepv but case insensitive

This works like `textops.mgrepv`, except it is case insensitive.

**Parameters**

- **patterns_dict** (*dict*) – a dictionary where all patterns to exclude are in values().
- **key** (*int or str*) – test only one column or one key (optional)

**Returns** A dictionary where the keys are the same as for `patterns_dict`, the values will contain the `textops.grepvi` result for each corresponding patterns.

**Return type** dict

#### Examples

```
>>> logs = '''error 1
... WARNING 1
... warning 2
... ERROR 2
... '''
>>> t = logs | mgrepv({
... 'not_errors' : r'^err',
... 'not_warnings' : r'^warn',
... })
>>> print t
{'not_warnings': ['error 1', 'WARNING 1', 'ERROR 2'],
'not_errors': ['WARNING 1', 'warning 2', 'ERROR 2']}
>>> t = logs | mgrepvi({
... 'not_errors' : r'^err',
... 'not_warnings' : r'^warn',
... })
>>> print t
{'not_warnings': ['error 1', 'ERROR 2'], 'not_errors': ['WARNING 1', 'warning 2']}
```

## 5.12 parse_indented

**class** textops.**parse_indented**(*sep=':'*)

Parse key:value indented text

It looks for key:value patterns, store found values in a dictionary. Each time a new indent is found, a sub-dictionary is created. The keys are normalized (only keep A-Za-z0-9_), the values are stripped.

**Parameters** **sep** (*str*) – key:value separator (Default : ':')

**Returns** structured keys:values

**Return type** dict

#### Examples

```
>>> s = '''
... a:val1
... b:
...     c:val3
...     d:
...         e ... : val5
...         f ... :val6
...     g:val7
... f: val8'''
>>> s | parse_indented()
{'a': 'val1', 'b': {'c': 'val3', 'd': {'e': 'val5', 'f': 'val6'}, 'g': 'val7'}, 'f': 'val8'}
>>> s = '''
... a --> val1
... b --> val2'''
>>> s | parse_indented(r'-->')
{'a': 'val1', 'b': 'val2'}
```

## 5.13 parseg

**class** textops.**parseg**(*pattern*)

Find all occurrences of one pattern, return MatchObject groupdict

**Parameters** **pattern** (*str*) – a regular expression string (case sensitive)

**Returns** A list of dictionaries (MatchObject groupdict)

**Return type** list

#### Examples

```
>>> s = '''name: Lapouyade
... first name: Eric
... country: France'''
>>> s | parseg(r'(?P<key>.*):\s*(?P<val>.*)')
[{'key': 'name', 'val': 'Lapouyade'},
{'key': 'first name', 'val': 'Eric'},
{'key': 'country', 'val': 'France'}]
```

## 5.14 parsegi

**class** textops.**parsegi**(*pattern*)
Same as parseg but case insensitive

**Parameters** **pattern** (*str*) – a regular expression string (case insensitive)

**Returns** A list of dictionaries (MatchObject groupdict)

**Return type** list

#### Examples

```
>>> s = '''Error: System will reboot
... Notice: textops rocks
... Warning: Python must be used without moderation'''
>>> s | parsegi(r'(?P<level>error|warning):\s*(?P<msg>.*)')
[{'msg': 'System will reboot', 'level': 'Error'},
{'msg': 'Python must be used without moderation', 'level': 'Warning'}]
```

## 5.15 parsek

**class** textops.**parsek**(*pattern*, *key_name = 'key'*, *key_update = None*)
Find all occurrences of one pattern, return one Key

One have to give a pattern with named capturing parenthesis, the function will return a list of value corresponding to the specified key. It works a little like textops.parseg except that it returns from the groupdict, a value for a specified key ('key' be default)

**Parameters**

- **pattern** (*str*) – a regular expression string.
- **key_name** (*str*) – The key to get ('key' by default)
- **key_update** (*callable*) – function to convert the found value

**Returns** A list of values corrsponding to *MatchObject groupdict[key]*

**Return type** list

#### Examples

```
>>> s = '''Error: System will reboot
... Notice: textops rocks
... Warning: Python must be used without moderation'''
>>> s | parsek(r'(?P<level>Error|Warning):\s*(?P<msg>.*)','msg')
['System will reboot', 'Python must be used without moderation']
```

## 5.16 parseki

**class** textops.**parseki**(*pattern*, *key_name = 'key'*, *key_update = None*)
Same as parsek but case insensitive

It works like `textops.parsek` except the pattern is case insensitive.

**Parameters**

- **pattern** (*str*) – a regular expression string.
- **key_name** (*str*) – The key to get ('key' by default)
- **key_update** (*callable*) – function to convert the found value

**Returns** A list of values corrsponding to *MatchObject groupdict[key]*

**Return type** list

#### Examples

```
>>> s = '''Error: System will reboot
... Notice: textops rocks
... Warning: Python must be used without moderation'''
>>> s | parsek(r'(?P<level>error|warning):\s*(?P<msg>.*)','msg')
[]
>>> s | parseki(r'(?P<level>error|warning):\s*(?P<msg>.*)','msg')
['System will reboot', 'Python must be used without moderation']
```

## 5.17 parsekv

**class** `textops.`**`parsekv`**(*pattern*, *key_name = 'key'*, *key_update = None*)

Find all occurrences of one pattern, returns a dict of groupdicts

It works a little like `textops.parseg` except that it returns a dict of dicts : values are MatchObject groupdicts, keys are a value in the groupdict at a specified key (By default : 'key'). Note that calculated keys are normalized (spaces are replaced by underscores)

**Parameters**

- **pattern** (*str*) – a regular expression string.
- **key_name** (*str*) – The key name to optain the value that will be the key of the groupdict ('key' by default)
- **key_update** (*callable*) – function to convert/normalize the calculated key

**Returns** A dict of MatchObject groupdicts

**Return type** dict

#### Examples

```
>>> s = '''name: Lapouyade
... first name: Eric
... country: France'''
>>> s | parsekv(r'(?P<key>.*):\s*(?P<val>.*)')
{'country': {'val': 'France', 'key': 'country'},
'first_name': {'val': 'Eric', 'key': 'first name'},
'name': {'val': 'Lapouyade', 'key': 'name'}}
>>> s | parsekv(r'(?P<item>.*):\s*(?P<val>.*)','item',str.upper)
{'FIRST NAME': {'item': 'first name', 'val': 'Eric'},
```

```
'NAME': {'item': 'name', 'val': 'Lapouyade'},
'COUNTRY': {'item': 'country', 'val': 'France'}}
```

## 5.18 parsekvi

**class** textops.**parsekvi**(*pattern*, *key_name = 'key'*, *key_update = None*)
Find all occurrences of one pattern (case insensitive), returns a dict of groupdicts

It works a little like textops.parsekv except that the pattern is case insensitive.

**Parameters**

- **pattern** (*str*) – a regular expression string (case insensitive).
- **key_name** (*str*) – The key name to optain the value that will be the key of the groupdict ('key' by default)
- **key_update** (*callable*) – function to convert/normalize the calculated key

**Returns** A dict of MatchObject groupdicts

**Return type** dict

#### Examples

```
>>> s = '''name: Lapouyade
... first name: Eric
... country: France'''
>>> s | parsekvi(r'(?P<key>NAME):\s*(?P<val>.*)')
{'name': {'val': 'Lapouyade', 'key': 'name'}}
```

## 5.19 state_pattern

**class** textops.**state_pattern**(*states_patterns_desc*, *reflags=0*, *autostrip=True*)
States and patterns parser

This is a *state machine* parser : The main advantage is that it reads line-by-line the whole input text only once to collect all data you want into a multi-level dictionary. It uses patterns to select rules to be applied. It uses states to ensure only a set of rules are used against specific document sections.

**Parameters**

- **states_patterns_desc** (*tupple*) – descrption of states and patterns : see below for explaination
- **reflags** – re flags, ie re.I or re.M or re.I | re.M (Default : no flag)
- **autostrip** – before being stored, groupdict keys and values are stripped (Default : True)

**Returns** parsed data from text

**Return type** dict

**The states_patterns_desc :**

It looks like this:

```
((<if state1>,<goto state1>,<pattern1>,<out data path1>,<out filter1>),
...
(<if stateN>,<goto stateN>,<patternN>,<out data pathN>,<out filterN>))
```

**<if state>** is a string telling on what state(s) the pattern must be searched, one can specify several states with comma separated string or a tupple. if `<if state>` is empty, the pattern will be searched for all lines. Note : at the beginning, the state is 'top'

**<goto state>** is a string corresponding to the new state if the pattern matches. use an empty string to not change the current state. One can use any string, usually, it corresponds to a specific section name of the document to parse where specific rules has to be used.

**<pattern>** is a string or a re.regex to match a line of text. one should use named groups for selecting data, ex: `(?P<key1>pattern)`

**<out data path>** is a string with a dot separator or a tuple telling where to place the groupdict from pattern maching process, The syntax is:

```
'{contextkey1}.{contextkey2}. ... .{contextkeyN}'
or
('{contextkey1}','{contextkey2}', ... ,'{contextkeyN}')
or
'key1.key2.keyN'
or
'key1.key2.keyN[]'
or
'{contextkey1}.{contextkey2}. ... .keyN[]'
```

The contextdict is used to format strings with `{contextkeyN}` syntax. instead of `{contextkeyN}`, one can use a simple string to put data in a fixed path. Once the path fully formatted, let's say to `key1.key2.keyN`, the parser will store the value into the result dictionnary at : `{'key1':{'key2':{'keyN' : thevalue }}}` One can use the string `[]` at the end of the path : the groupdict will be appended in a list ie : `{'key1':{'key2':{'keyN' : [thevalue,...] }}}`

**<out filter>** is used to build the value to store,

it could be :

- None : no filter is applied, the re.MatchObject.groupdict() is stored
- a string : used as a format string with context dict, the formatted string is stored
- a callable : to calculate the value to be stored, the context dict is given as param.

**How the parser works :**

You have a document where the syntax may change from one section to an another : You have just to give a name to these kind of sections : it will be your state names. The parser reads line by line the input text : For each line, it will look for the *first* matching rule from `states_patterns_desc` table, then will apply the rule. One rule has got 2 parts : the matching parameters, and the action parameters.

**Matching parameters:** To match, a rule requires the parser to be at the specified state `<if state>` AND the line to be parsed must match the pattern `<pattern>`. When the parser

is at the first line, it has the default state `top`. The pattern follow the standard python `re` module syntax. It is important to note that you must capture text you want to collect with the named group capture syntax, that is `(?P<mydata>mypattern)`. By this way, the parser will store text corresponding to `mypattern` to a contextdict at the key `mydata`.

**Action parameters:** Once the rule matches, the action is to store `<out filter>` into the final dictionary at a specified `<out data path>`.

**Context dict :**

The context dict is used within `<out filter>` and `<out data path>`, it is a dictionary that is *PERSISTENT* during the whole parsing process : It is empty at the parsing beginning and will accumulate all captured pattern. For exemple, if a first rule pattern contains `(?P<key1>.*),(?P<key2>.*)` and matches the document line `val1,val2`, the context dict will be `{ 'key1' : 'val1', 'key2' : 'val2' }`. Then if a second rule pattern contains `(?P<key2>.*):(?P<key3>.*)` and matches the document line `val4:val5` then the context dict will be *UPDATED* to `{ 'key1' : 'val1', 'key2' : 'val4', 'key3' : 'val5' }`. As you can see, the choice of the key names are *VERY IMPORTANT* in order to avoid collision across all the rules.

### Examples

```
>>> s = '''
... first name: Eric
... last name: Lapouyade'''
>>> s | state_pattern( (('',None,'(?P<key>.*):(?P<val>.*)','{key}','{val}'),) )
{'first_name': 'Eric', 'last_name': 'Lapouyade'}
>>> s | state_pattern( (('',None,'(?P<key>.*):(?P<val>.*)','{key}',None),) )
{'first_name': {'val': 'Eric', 'key': 'first name'},
'last_name': {'val': 'Lapouyade', 'key': 'last name'}}
>>> s | state_pattern((('',None,'(?P<key>.*):(?P<val>.*)','my.path.{key}','{val}'),))
{'my': {'path': {'first_name': 'Eric', 'last_name': 'Lapouyade'}}}

>>> s = '''Eric
... Guido'''
>>> s | state_pattern( (('',None,'(?P<val>.*)','my.path.info[]','{val}'),) )
{'my': {'path': {'info': ['Eric', 'Guido']}}}

>>> s = '''
... Section 1
... ---------
...   email = ericdupo@gmail.com
...
... Section 2
... ---------
...   first name: Eric
...   last name: Dupont'''
>>> s | state_pattern( (
... ('','section1','^Section 1',None,None),
... ('','section2','^Section 2',None,None),
... ('section1', '', '(?P<key>.*)=(?P<val>.*)', 'section1.{key}', '{val}'),
... ('section2', '', '(?P<key>.*):(?P<val>.*)', 'section2.{key}', '{val}')) )
{'section2': {'first_name': 'Eric', 'last_name': 'Dupont'},
'section1': {'email': 'ericdupo@gmail.com'}}

>>> s = '''
... Disk states
```

```
... -----------
... name: c1t0d0s0
... state: good
... fs: /
... name: c1t0d0s4
... state: failed
... fs: /home
...
... '''
>>> s | state_pattern( (
... ('top','disk',r'^Disk states',None,None),
... ('disk','top', r'^\s*$',None,None),
... ('disk', '', r'^name:(?P<diskname>.*)',None, None),
... ('disk', '', r'(?P<key>.*):(?P<val>.*)', 'disks.{diskname}.{key}', '{val}')) )
{'disks': {'c1t0d0s0': {'state': 'good', 'fs': '/'},
'c1t0d0s4': {'state': 'failed', 'fs': '/home'}}}
```

- *genindex*
- *modindex*
- *search*

CHAPTER 6

# cast

This modules provides casting features, that is to force the output type

## 6.1 pretty

**class** textops.**pretty**()
Pretty format the input text

**Returns** Converted result as a pretty string ( uses pprint.PrettyPrinter.pformat() )

**Return type** str

Examples:

```
>>> s = '''
... a:val1
... b:
...     c:val3
...     d:
...         e ... : val5
...         f ... :val6
...     g:val7
... f: val8'''
>>> print s | parse_indented()
{'a': 'val1', 'b': {'c': 'val3', 'd': {'e': 'val5', 'f': 'val6'}, 'g': 'val7'}, 'f': 'val8'}
>>> print s | parse_indented().pretty()
{   'a': 'val1',
    'b': {   'c': 'val3', 'd': {   'e': 'val5', 'f': 'val6'}, 'g': 'val7'},
    'f': 'val8'}
```

## 6.2 todatetime

**class** textops.**todatetime**()
Convert the result to a datetime python object

**Returns** converted result as a datetime python object

**Return type** datetime

#### Examples

```
>>> '2015-10-28' | todatetime()
datetime.datetime(2015, 10, 28, 0, 0)
>>> '2015-10-28 22:33:00' | todatetime()
datetime.datetime(2015, 10, 28, 22, 33)
>>> '2015-10-28 22:33:44' | todatetime()
datetime.datetime(2015, 10, 28, 22, 33, 44)
>>> '2014-07-08T09:02:21.377' | todatetime()
datetime.datetime(2014, 7, 8, 9, 2, 21, 377000)
>>> '28-10-2015' | todatetime()
datetime.datetime(2015, 10, 28, 0, 0)
>>> '10-28-2015' | todatetime()
datetime.datetime(2015, 10, 28, 0, 0)
>>> '10-11-2015' | todatetime()
datetime.datetime(2015, 10, 11, 0, 0)
```

## 6.3 todict

**class** textops.**todict**()
Converts list or 2 items-tuples into dict

**Returns** Converted result as a dict

**Return type** dict

Examples:

```
>>> [ ('a',1), ('b',2), ('c',3) ] | echo().todict()
{'a': 1, 'c': 3, 'b': 2}
```

## 6.4 tofloat

**class** textops.**tofloat**()
Convert the result to a float

**Returns** converted result as an int or list of int

**Return type** str or list

#### Examples

```
>>> '53' | tofloat()
53.0
>>> 'not a float' | tofloat()
0.0
>>> '3.14' | tofloat()
3.14
>>> '3e3' | tofloat()
3000.0
>>> ['53','not an int','3.14'] | tofloat()
[53.0, 0.0, 3.14]
```

## 6.5 toint

**class** textops.**toint**()
Convert the result to an integer

**Returns** converted result as an int or list of int

**Return type** str or list

### Examples

```
>>> '53' | toint()
53
>>> 'not an int' | toint()
0
>>> '3.14' | toint()
3
>>> '3e3' | toint()
3000
>>> ['53','not an int','3.14'] | toint()
[53, 0, 3]
```

## 6.6 tolist

**class** textops.**tolist**(*return_if_none=None*)
Convert the result to a list

If the input text is a string, it is put in a list.
If the input text is a list : nothing is done.
If the input text is a generator : it is converted into a list

**Parameters** **return_if_none** (*str*) – the object to return if the input text is None (Default : None)

**Returns** converted result as a string

**Return type** str

### Examples

```
>>> 'hello' | tolist()
['hello']
>>> ['hello','world'] | tolist()
['hello', 'world']
>>> type(None|tolist())
<type 'NoneType'>
>>> def g(): yield 'hello'
...
>>> g()|tolist()
['hello']
```

## 6.7 toliste

**class** textops.**toliste**(*return_if_none=''*)
Convert the result to a list

If the input text is a string, it is put in a list.
If the input text is a list : nothing is done.
If the input text is a generator : it is converted into a list

**Parameters** **return_if_none** (*str*) – the object to return if the input text is None (Default : empty list)

**Returns** converted result as a string

**Return type** str

### Examples

```
>>> 'hello' | toliste()
['hello']
>>> type(None|toliste())
<class 'textops.base.ListExt'>
>>> None|toliste()
[]
```

## 6.8 toslug

**class** textops.**toslug**()
Convert a string to a slug

**Returns** a slug

**Return type** str

### Examples

```
>>> 'this is my article' | toslug()
'this-is-my-article'
>>> 'this%%% is### my___article' | toslug()
'this-is-my-article'
```

## 6.9 tostr

**class** textops.**tostr**(*join_str='\n'*, *return_if_none=None*)
Convert the result to a string

If the input text is a list or a generator, it will join all the lines with a newline.

If the input text is None, None is NOT converted to a string : None is returned

**Parameters**

- **join_str** (*str*) – the join string to apply on list or generator (Default : newline)
- **return_if_none** (*str*) – the object to return if the input text is None (Default : None)

**Returns** converted result as a string

**Return type** str or None

### Examples

```
>>> 'line1\nline2' | tostr()
'line1\nline2'
>>> ['line1','line2'] | tostr()
'line1\nline2'
>>> ['line1','line2'] | tostr('---')
'line1---line2'
>>> def g(): yield 'hello';yield 'world'
...
>>> g()|tostr()
'hello\nworld'
>>> type(None | tostr())
<type 'NoneType'>
>>> None | tostr(return_if_none='N/A')
'N/A'
```

## 6.10 tostre

**class** `textops.`**`tostre`**(*join_str='\n'*, *return_if_none=''*)
Convert the result to a string

If the input text is a list or a generator, it will join all the lines with a newline.
If the input text is None, None is converted to an empty string.

**Parameters**

- **join_str** (*str*) – the join string to apply on list or generator (Default : newline)
- **return_if_none** (*str*) – the object to return if the input text is None (Default : empty string)

**Returns** converted result as a string

**Return type** str

### Examples

```
>>> ['line1','line2'] | tostre()
'line1\nline2'
>>> type(None | tostre())
<class 'textops.base.StrExt'>
>>> None | tostre()
''
```

- *genindex*
- *modindex*
- *search*

CHAPTER 7

# base

This module defines base classes for python-textops

## 7.1 activate_debug

textops.**activate_debug**()
Activate debug logging on console

This function is useful when playing with python-textops through a python console. It is not recommended to use this function in a real application : use standard logging functions instead.

## 7.2 add_textop

textops.**add_textop**(*class_or_func*)
Decorator to declare custom function or custom class as a new textops op

the custom function/class will receive the whole raw input text at once.

### Examples

```
>>> @add_textop
... def repeat(text, n, *args,**kwargs):
...     return text * n
>>> 'hello' | repeat(3)
'hellohellohello'

>>> @add_textop
... class cool(TextOp):
...     @classmethod
...     def op(cls, text, *args,**kwargs):
...         return text + ' is cool.'
>>> 'textops' | cool()
'textops is cool.'
```

## 7.3 add_textop_iter

textops.**add_textop_iter**(*func*)

Decorator to declare custom *ITER* function as a new textops op

An *ITER* function is a function that will receive the input text as a *LIST* of lines. One have to iterate over this list and generate a result (it can be a list, a generator, a dict, a string, an int ...)

#### Examples

```
>>> @add_textop_iter
... def odd(lines, *args,**kwargs):
...     for i,line in enumerate(lines):
...         if not i % 2:
...             yield line
>>> s = '''line 1
... line 2
... line 3'''
>>> s >> odd()
['line 1', 'line 3']
>>> s | odd().tolist()
['line 1', 'line 3']

>>> @add_textop_iter
... def sumsize(lines, *args,**kwargs):
...     sum = 0
...     for line in lines:
...         sum += int(re.search(r'\d+',line).group(0))
...     return sum
>>> '''1492 file1
... 1789 file2
... 2015 file3''' | sumsize()
5296
```

## 7.4 dformat

textops.**dformat**(*format_str*, *dct*, *defvalue='-'*)

Formats a dictionary, manages unkown keys

It works like `string.Formatter.vformat()` except that it accepts only a dict for values and a defvalue for not matching keys. Defvalue can be a callable that will receive the requested key as argument and return a string

**Parameters**

- **format_string** (*str*) – Same format string as for `str.format()`
- **defvalue** (*str or callable*) – the default value to display when the data is not in the dict

Examples:

```
>>> d = {'count': '32591', 'soft': 'textops'}
>>> dformat('{soft} : {count} dowloads',d)
'textops : 32591 dowloads'
>>> dformat('{software} : {count} dowloads',d,'N/A')
```

```
'N/A : 32591 dowloads'
>>> dformat('{software} : {count} dowloads',d,lambda k:'unknown_tag_%s' % k)
'unknown_tag_software : 32591 dowloads'
```

## 7.5 DictExt

**class** `textops.`**`DictExt`**(**args*, ***kwargs*)

Extend dict class with new features

New features are :

- •Access to textops operations with attribute notation
- •All dict values (dict, list, str, unicode) are extended on-the-fly when accessed
- •Access to dict values with attribute notation
- •Add a key:value in the dict with attribute notation (one level at a time)
- •Returns NoAttr object when a key is not in the Dict
- •add modification on-the-fly `amend()` and rendering to string `render()`

---

**Note:** `NoAttr` is a special object that returns always `NoAttr` when accessing to any attribute. it behaves like `False` for testing, `[]` in foor-loops. The goal is to be able to use very long expression with dotted notation without being afraid to get an exception.

---

#### Examples

```
>>> {'a':1,'b':2}.items().grep('a')
Traceback (most recent call last):
    ...
AttributeError: 'list' object has no attribute 'grep'

>>> DictExt({'a':1,'b':2}).items().grep('a')
[['a', 1]]

>>> d = DictExt({ 'this' : { 'is' : { 'a' : {'very deep' : { 'dict' : 'yes it is'}}}}})
>>> print d.this['is'].a['very deep'].dict
yes it is
>>> d.not_a_valid_key
NoAttr
>>> d['not_a_valid_key']
NoAttr
>>> d.not_a_valid_key.and_i.can.put.things.after.without.exception
NoAttr
>>> for obj in d.not_a_valid_key.objects:
...     do_things(obj)
... else:
...     print 'no object'
no object

>>> d = DictExt()
>>> d.a = DictExt()
>>> d.a.b = 'this is my logging data'
```

```
>>> print d
{'a': {'b': 'this is my logging data'}}

>>> d = { 'mykey' : 'myval' }
>>> d['mykey']
'myval'
>>> type(d['mykey'])
<type 'str'>
>>> d = DictExt(d)
>>> d['mykey']
'myval'
>>> type(d['mykey'])
<class 'textops.base.StrExt'>
```

**amend** (**args*, ***kwargs*)
Modify on-the-fly a dictionary

The method will generate a new extended dictionary and update it with given params

Examples:

```
>>> s = '''soft:textops
... count:32591'''
>>> s | parse_indented()
{'count': '32591', 'soft': 'textops'}
>>> s | parse_indented().amend(date='2015-11-19')
{'count': '32591', 'date': '2015-11-19', 'soft': 'textops'}
```

**as_list**
Convert to ListExt object

**render** (*format_string*, *defvalue='-'*)
Render a DictExt as a string

It uses the fonction `dformat()` to format the dictionary

**Parameters**

- **format_string** (*str*) – Same format string as for `str.format()`
- **defvalue** (*str or callable*) – the default value to display when the data is not in the dict

Examples:

```
>>> d = DictExt({'count': '32591', 'date': '2015-11-19', 'soft': 'textops'})
>>> d.render('On {date}, "{soft}" has been downloaded {count} times')
'On 2015-11-19, "textops" has been downloaded 32591 times'
>>> d.render('On {date}, "{not_in_dict}" has been downloaded {count} times','?')
'On 2015-11-19, "?" has been downloaded 32591 times'
```

## 7.6 ListExt

**class** `textops.`**`ListExt`**
Extend list class to gain access to textops as attributes

In addition, all list items (dict, list, str, unicode) are extended on-the-fly when accessed

#### Examples

```
>>> ['normal','list'].grep('t')
Traceback (most recent call last):
    ...
AttributeError: 'list' object has no attribute 'grep'

>>> ListExt(['extended','list']).grep('t')
['extended', 'list']
```

**as_list**
Convert to ListExt object

## 7.7 StrExt

**class** textops.**StrExt**
Extend str class to gain access to textops as attributes

#### Examples

```
>>> 'normal string'.cut()
Traceback (most recent call last):
    ...
AttributeError: 'str' object has no attribute 'cut'

>>> StrExt('extended string').cut()
['extended', 'string']
```

**as_list**
Convert to ListExt object

## 7.8 TextOp

**class** textops.**TextOp**(**args*, ***kwargs*)
Base class for text operations

All operations must be derived from this class. Subclasses must redefine an `op()` method that will be called when the operations will be triggered by an input text.

**f**
Execute operations, returns a float.

#### Examples

```
>>> echo('1789').f
1789.0
>>> echo('3.14').f
3.14
>>> echo('Tea for 2').f
0.0
```

**g**

Execute operations, return a generator when possible or a list otherwise

This is to be used ONLY when the input text has be set as the first argument of the first operation.

#### Examples

```
>>> echo('hello')
echo('hello')
>>> echo('hello').g
['hello']
>>> def mygen(): yield 'hello'
>>> cut(mygen(),'l')
cut(<generator object mygen at ...>,'l')
>>> cut(mygen(),'l').g
<generator object extend_type_gen at ...>
>>> def mygen(): yield None
>>> type(echo(None).g)
<type 'NoneType'>
```

**ge**

Execute operations, return a generator when possible or a list otherwise, ( [] if the result is None ).

This works like `g` except it returns an empty list if the execution result is None.

#### Examples

```
>>> echo(None).ge
[]
```

**i**

Execute operations, returns an int.

#### Examples

```
>>> echo('1789').i
1789
>>> echo('3.14').i
3
>>> echo('Tea for 2').i
0
```

**j**

Execute operations, return a string (join = '')

This works like `s` except that joins will be done with an empty string

#### Examples

```
>>> echo(['hello','world']).j
'helloworld'
>>> type(echo(None).j)
<type 'NoneType'>
```

**je**

Execute operations, returns a string ( '' if the result is None, join='').

This works like `j` except it returns an empty string if the execution result is None.

#### Examples

```
>>> echo(None).je
''
```

**l**

Execute operations, return a list

This is to be used ONLY when the input text has be set as the first argument of the first operation.

#### Examples

```
>>> echo('hello')
echo('hello')
>>> echo('hello').l
['hello']
>>> type(echo(None).g)
<type 'NoneType'>
```

**le**

Execute operations, returns a list ( [] if the result is None ).

This works like `l` except it returns an empty list if the execution result is None.

#### Examples

```
>>> echo(None).le
[]
```

**classmethod op**(*text*, **args*, ***kwargs*)

This method must be overriden in derived classes

**pp**

Execute operations, return Prettyprint version of the result

Examples:

```
>>> s = '''
... a:val1
... b:
...     c:val3
...     d:
...         e ... : val5
...         f ... :val6
```

```
...     g:val7
... f: val8'''
>>> print parse_indented(s).r
{'a': 'val1', 'b': {'c': 'val3', 'd': {'e': 'val5', 'f': 'val6'}, 'g': 'val7'}, 'f': 'va
>>> print parse_indented(s).pp
{   'a': 'val1',
    'b': {   'c': 'val3', 'd': {   'e': 'val5', 'f': 'val6'}, 'g': 'val7'},
    'f': 'val8'}
```

**r**

Execute operations, do not convert.

#### Examples

```
>>> echo('1789').length().l
[4]
>>> echo('1789').length().s
'4'
>>> echo('1789').length().r
4
```

**s**

Execute operations, return a string (join = newline)

This is to be used ONLY when the input text has be set as the first argument of the first operation.
If the result is a list or a generator, it is converted into a string by joinning items with a newline.

#### Examples

```
>>> echo('hello')
echo('hello')
>>> echo('hello').s
'hello'
>>> echo(['hello','world']).s
'hello\nworld'
>>> type(echo(None).s)
<type 'NoneType'>
```

**se**

Execute operations, returns a string ( '' if the result is None ).

This works like `s` except it returns an empty string if the execution result is None.

#### Examples

```
>>> echo(None).se
''
```

## 7.9 UnicodeExt

**class** `textops.`**`UnicodeExt`**

Extend Unicode class to gain access to textops as attributes

#### Examples

```
>>> u'normal unicode'.cut()
Traceback (most recent call last):
    ...
AttributeError: 'unicode' object has no attribute 'cut'

>>> UnicodeExt('extended unicode').cut()
[u'extended', u'unicode']
```

**`as_list`**

Convert to ListExt object

- *genindex*
- *modindex*
- *search*

CHAPTER 8

# Indices and tables

- *genindex*
- *modindex*
- *search*

## t