

---

# **python-textops3 Documentation**

***Release 3.2.1***

**Eric Lapouyade**

**Apr 25, 2022**



---

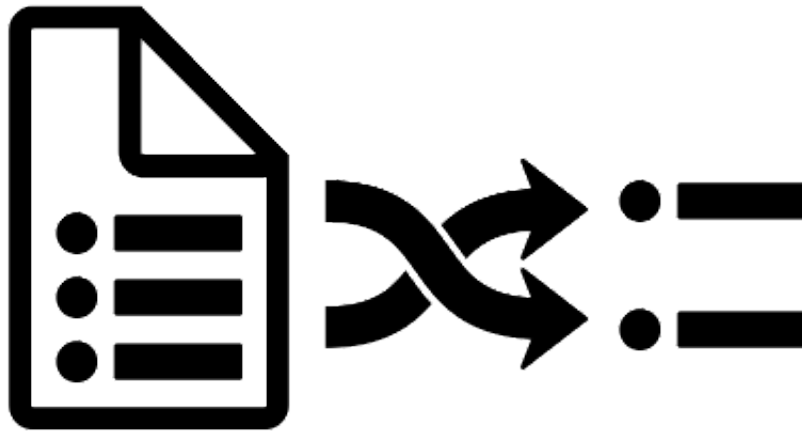
## Contents

---

<b>1</b>	<b>Install</b>	<b>3</b>
<b>2</b>	<b>Quickstart</b>	<b>5</b>
<b>3</b>	<b>Run tests</b>	<b>11</b>
<b>4</b>	<b>Build documentation</b>	<b>13</b>
<b>5</b>	<b>strops</b>	<b>15</b>
<b>6</b>	<b>listops</b>	<b>27</b>
<b>7</b>	<b>fileops</b>	<b>73</b>
<b>8</b>	<b>runops</b>	<b>83</b>
<b>9</b>	<b>wrapops</b>	<b>87</b>
<b>10</b>	<b>parse</b>	<b>89</b>
<b>11</b>	<b>cast</b>	<b>109</b>
<b>12</b>	<b>recode</b>	<b>115</b>
<b>13</b>	<b>base</b>	<b>119</b>
<b>14</b>	<b>Indices and tables</b>	<b>131</b>
	<b>Python Module Index</b>	<b>133</b>
	<b>Index</b>	<b>135</b>



# *python-textops*



`python-textops3` 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.

Here is a simple example to count number of mails received from `spammer@hacker.com` since May 25th:

```
>>> '/var/log/mail.log' | cat().grep('spammer@hacker.com').since('May 25').lcount()  
37
```

`python-textops3` is used into some other projects like `python-nagios-helpers3`



# CHAPTER 1

---

## Install

---

To install:

```
pip install python-textops3
```





## CHAPTER 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()
```

If you do not want to import all textops operations, you can only import textops as `op`:

```
>>> 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 in parenthesis after chained operation as they were a function:

```
>>> 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')
35
>>> 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:

```
# Just creating a test file here :
>>> open('/tmp/errors.log','w').write('error 1\nwarning 1\nwarning 2\nerror 2')
35

# Here, operations are excuted because 'print' converts into string :
# it triggers execution.
>>> print(cat('/tmp/errors.log').gredi('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').gredi('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                                     # the cat() is not executed, you see only its_
↳ python representation :
cat('/tmp/errors.log')
>>> print(type(logs))
<class 'textops.ops.fileops.cat'>

# To force execution, use special attribute .s .l or .g :
>>> open('/tmp/errors.log','w').write('error 1\nwarning 1')
17
>>> logs = cat('/tmp/errors.log').s # '.s' to execute operations and get a string_
↳ (StrExt)
>>> print(type(logs))                                )# you get a textops extended string
<class 'textops.base.StrExt'>
>>> print(logs)
error 1
warning 1

>>> logs = cat('/tmp/errors.log').l # '.l' to execute operations and get a list_
↳ (ListExt)
>>> print(type(logs))
<class 'textops.base.ListExt'>
>>> print(logs)
['error 1', 'warning 1']

>>> logs = cat('/tmp/errors.log').g # '.g' to execute operations and get a generator
>>> print(type(logs))
<class '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 fonctionnality:

```
>>> 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 !



## CHAPTER 3

---

### Run tests

---

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

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





## CHAPTER 4

---

### Build documentation

---

An already compiled and up-to-date documentation should be available [here](#). 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-textops3.pdf    (evince is a PDF reader)
```

- `genindex`
- `modindex`
- `search`



This module gathers text operations to be run on a string

## 5.1 StrOp

```
class textops.StrOp
```

## 5.2 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']]
```

## 5.3 cutca

**class** `textops.cutca` (*sep=None, 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 column
  - 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']]
```

## 5.4 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 column
  - 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_count': 'col2', 'i_price': 'col3'}
>>> 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>[^"]*)"') # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[{'item': 'col1', 'i_count': 'col2', 'i_price': 'col3'},
 {'item': 'col11', 'i_count': 'col22', 'i_price': 'col33'}]
```

## 5.5 cutkv

**class** `textops.cutkv` (*sep=None, col=None, default=""*)

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_count': 'col2', 'i_price': 'col3'}}
>>> s='item="col1" count="col2" price="col3"\nitem="col11" count="col22"
↳ price="col33"'
>>> s | cutkv(pattern,key_name='item')
↳ # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[{'col1': {'item': 'col1', 'i_count': 'col2', 'i_price': 'col3'}},
 {'col11': {'item': 'col11', 'i_count': 'col22', 'i_price': 'col33'}}]
```

## 5.6 cutm

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

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

It returns the matched pattern. Beware : the pattern must match the beginning of the line. One may use capture parenthesis to only return a part of the found pattern.

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

## 5.7 cutmi

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

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

This works like `textops.cutm` except it is case insensitive.

- 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')  #as .* is the longest_
possible, only last column is extracted
'col3'
```

## 5.8 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']]
```

## 5.9 cuts

**class** `textops.cuts` (*sep=None, col=None, default=""*)

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

This works like `textops.cutm` except it searches the first occurrence of the pattern in the string. One may use capture parenthesis to only return a part of the found pattern.

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

## 5.10 cutsa

**class** `textops.cutsa` (*sep=None, col=None, default=""*)

Extract all columns having the specified pattern.

It uses `re.finditer()` to find all occurrences of the pattern.

- if the input is a simple string, `textops.cutfa` will return a list of found strings.
- if the input is a list of strings or a string with newlines, `textops.cutfa` will return a list of list of found string.

### 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 | cutsa(r'col\d+')
['col1', 'col2', 'col3']
>>> s | cutsa(r'col(\d+)')
['1', '2', '3']
>>> s=['-col1- =col2= _col3_', '-col11- =col22= _col33_']
>>> s | cutsa(r'col\d+')
[['col1', 'col2', 'col3'], ['col11', 'col22', 'col33']]
```

## 5.11 cutsai

**class** `textops.cutsai` (*sep=None, col=None, default=""*)

Extract all columns having the specified pattern. (case insensitive)

It works like `textops.cutsa` but is case insensitive if the pattern is given as a string.

### 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 | cutsa(r'COL\d+')
[]
>>> s | cutsai(r'COL\d+')
['col1', 'col2', 'col3']
>>> s | cutsai(r'COL(\d+) ')
['1', '2', '3']
>>> s=['-col1- =col2= _col3_', '-col11- =col22= _col33_']
>>> s | cutsai(r'COL\d+')
[['col1', 'col2', 'col3'], ['col11', 'col22', 'col33']]
```

## 5.12 cutsi

**class** `textops.cutsi` (*sep=None, col=None, default=""*)

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

This works like `textops.cuts` except it is case insensitive.

### 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'
```

## 5.13 echo

**class** textops.**echo**

identity operation

it returns the same text, except that it uses textops Extended classes (StrExt, ListExt ...). This could be useful 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)
<class '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'
```

## 5.14 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
```

## 5.15 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
```

## 5.16 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
```

## 5.17 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']
```

This module gathers list/line operations

## 6.1 after

**class** `textops.after` (*pattern*, *get\_begin=False*)

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}]
```

## 6.2 afteri

**class** textops.**afteri** (*pattern*, *get\_begin=False*)

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']
```

## 6.3 aggregate

**class** textops.**aggregate** (*having*, *same\_key=False*, *join\_str='|'*)

aggregate several lines into one depending on arguments

This is useful when some messages are splitted into several lines and one want to have them on one single line back (to do some grep for instance).

### Parameters

- **having** (*str* or *regex* or *callable*) – The lines having the specified pattern are merged with the previous one. The pattern must include two named groups : (*?<key>pattern1*) that must match the pattern which tells the line must be aggregated or not, and (*?<msg>pattern2*) that specifies what part of the line have to be aggregated (the whole line is not present). One can also specify a callable that accepts as arguments : the buffer and the line being tested and returns a *key, msg* tuple. *key* must contain the string that is the criteria to have the line aggregated or be None if the line is not to be aggregated. *msg* must be the part of the line to be aggregated
- **same\_key** (*bool*) – if False (Default) : lines are aggregated only if key is not None, if True, lines are aggregated only if the key is not None AND with the same value as the previous one. This is useful for log files with timestamp : a message block has usually the same timestamp on all the lines.
- **join\_str** (*str*) – Join string when merging lines (Default: '|')



**Returns** aggregated input text

**Return type** generator

## Examples

```
>>> s='''Message 1
... Message 2 : This is a multiple line message :
... > In this example,
... > lines to be aggregated begin with '> '
... > use '(?P<msg>pattern)' in regex to select what should be aggregated
... > default join string is '|'
... Message 3
... Message 4
... '''
>>> print(s | aggregate(r'(?P<key>> )(?P<msg>.*)').tostr())
Message 1
Message 2 : This is a multiple line message :|In this example,|lines to
↳be aggregated begin with '> '|use '(?P<msg>pattern)' in regex to
↳select what should be aggregated|default join string is '|'
Message 3
Message 4
```

```
>>> s='''2016-02-16 11:39:03 Message 1
... 2016-02-16 11:40:10 Message 2 info 1
... 2016-02-16 11:40:10 info 2
... 2016-02-16 11:41:33 Message 3 info A
... 2016-02-16 11:41:33 info B
... 2016-02-16 11:41:33 info C
... 2016-02-16 11:44:26 Message 4
... '''
>>> print(s | aggregate(r'(?P<key>\d+-\d+-\d+ \d+:\d+:\d+) (?P<msg>.*)').
↳tostr())
Message 1|Message 2 info 1|info 2|Message 3 info A|info B|info C|Message
↳4
>>> print(s | aggregate(r'(?P<key>\d+-\d+-\d+ \d+:\d+:\d+) (?P<msg>.*)',
↳same_key=True).tostr())
2016-02-16 11:39:03 Message 1
2016-02-16 11:40:10 Message 2 info 1|info 2
2016-02-16 11:41:33 Message 3 info A|info B|info C
2016-02-16 11:44:26 Message 4
```

## 6.4 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}]
```

## 6.5 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']
```

## 6.6 between

**class** textops.**between** (*begin*, *end*, *get\_begin=None*, *get\_end=None*, *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
```

(continues on next page)

(continued from previous page)

```

...
... 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

```

## 6.7 betweenb

**class** textops.**betweenb** (*begin*, *end*, *get\_begin=None*, *get\_end=None*, *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']

```

## 6.8 betweenbi

**class** textops.**betweenbi** (*begin*, *end*, *get\_begin=None*, *get\_end=None*, *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']
```

## 6.9 betweeni

**class** textops.**betweeni** (*begin*, *end*, *get\_begin*=None, *get\_end*=None, *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']
```

## 6.10 doformat

**class** textops.doformat (format\_str='{0}n', join\_str=",", context={}, defvalue='-')

Formats list of strings

Useful to convert list of string into a simple string It converts each string of the list with the format\_str ({0} will receive the string to format), then it joins all the strings with join\_str to get a unique simple string. One can specify a context dictionary and a default value : they will be used into the format string : see examples.

### Parameters

- **format\_str** (*str*) – format string, default is '{0}n'
- **join\_str** (*str*) – string to join all strings into one unique string, default is “,”
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str* or *callable*) – default string to display when a key or an index is unreachable.

**Returns** formatted input

**Return type** *str*

### Examples

```
>>> print(['Eric', 'Guido'] | doformat('First name : {0}', '\n'))
First name : Eric
First name : Guido
>>> ['Eric', 'Guido'] | doformat('{0} <{0}@github.com>', ',')
'Eric <Eric@github.com>, Guido <Guido@github.com>'
>>> ctx = {'hostname' : 'pcubuntu'}
>>> print(['eth0', 'eth1'] | doformat('{hostname}/network/{0}', '\n',
↪ context=ctx))
pcubuntu/network/eth0
pcubuntu/network/eth1
>>> print(['eth0', 'eth1'] | doformat('{nodename}/network/{4}', '\n', ctx,
↪ '(unknown)'))
(unknown)/network/(unknown)
(unknown)/network/(unknown)
```

## 6.11 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'
```

## 6.12 dorender

**class** textops.dorender (*format\_str='{0}n', context={}, defvalue='-'*)

Formats list of strings

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

#### Parameters

- **format\_str** (*str*) – format string, default is '{0}n'
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str or callable*) – The replacement string or function for unexisting keys when formatting.

**Yields** *str* – formatted input

#### Examples

```
>>> ['Eric','Guido'] >> dorender('First name : {0}')
['First name : Eric', 'First name : Guido']
>>> ['Eric','Guido'] >> dorender('{0} <{0}@github.com>')
['Eric <Eric@github.com>', 'Guido <Guido@github.com>']
>>> ctx = {'mail_domain' : 'gmail.com'}
>>> ['Eric','Guido'] >> dorender('{0} <{0}@{mail_domain}>',context=ctx)
['Eric <Eric@gmail.com>', 'Guido <Guido@gmail.com>']
>>> ['Eric','Guido'] >> dorender('{0} <{0}@{key_not_in_context}>',
↵context=ctx, defvalue='N/A')
['Eric <Eric@N/A>', 'Guido <Guido@N/A>']
```

## 6.13 doslice

**class** textops.doslice (*begin=0, end=sys.maxint, 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:  $0 \leq x \leq \text{maxint}$
- **end** (*int*) – end line number (get lines up to end - 1). must be None or an integer:  $0 \leq x \leq \text{maxint}$
- **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']
```

## 6.14 dostrip

**class** textops.dostrip

Strip lines

Works like the python `str.strip()` except it is more flexible it that way it works on list of lists or list of dicts.

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

**Examples**

```
>>> ' Hello Eric \n Hello Guido ' | dostrip().tostr()
'Hello Eric\nHello Guido'
```

## 6.15 findhighlight

**class** textops.findhighlight (*pattern*, *line\_prefix*=' ', *line\_suffix*="", *hline\_prefix*='-> ', *hline\_suffix*="", *found\_prefix*='>>>', *found\_suffix*='<<<', *nlines*=0, *blines*=0, *elines*=0, *ellipsis*='...', *findall*=True, *ignorecase*=False, *line\_nbr*=False)

find one or more pattern in a text then highlight it



This returns lines with found pattern with surrounding lines. By default, it surrounds found pattern with >>> <<< and its lines with ->. This can be redefined for console (to add color escape sequence) or for the web (html tags).

### Parameters

- **pattern** (*str* or *regex*) – the pattern to search in the input text
- **line\_prefix** (*str*) – the prefix string to add on every line except the lines where the pattern has been found. You may use {line} to add the line number.
- **line\_suffix** (*str*) – the suffix string to add on every line except the lines where the pattern has been found. You may use {line} to add the line number.
- **hline\_prefix** (*str*) – the prefix string to add on lines where the pattern has been found. You may use {line} to add the line number.
- **hline\_suffix** (*str*) – the suffix string to add on lines where the pattern has been found. You may use {line} to add the line number.
- **found\_prefix** (*str*) – The prefix to be added at the found pattern start position.
- **found\_suffix** (*str*) – The suffix to be added at the found pattern end position.
- **nlines** (*int*) – the number of lines to display before and after the line with the found pattern. If 0 is specified (This is the Default) : all lines are displayed. If you want not the same number of lines before and after, do not specify this parameter and use `blines` and `elines`
- **blines** (*int*) – number of lines to display before
- **elines** (*int*) – number of lines to display after
- **ellipsis** (*str*) – the string to display between groups of lines (Default : ...)
- **findall** (*bool*) – Find all pattern (Default) otherwise only find the first line with the pattern.
- **ignorecase** (*bool*) – if True, the given pattern is case insensitive) (Default : False)
- **line\_nbr** (*bool*) – add line numbers in `line_prefix` and `hline_prefix` (Default : False)

**Yields** *str* – lines with found pattern with surrounding lines

### Examples

```
>>> s='''
... this is
... a big
... listing
... with some ERROR
... and I want to know
... where are
... located
... this error in
... the
... whole
... text'''
>>> print(s | findhighlight('error',line_nbr=True,ignorecase=True) .
↳tostr() ) # doctest: +NORMALIZE WHITESPACE
```

(continues on next page)

(continued from previous page)

```

1
2   this is
3   a big
4   listing
5 -> with some >>>ERROR<<<
6   and I want to know
7   where are
8   located
9 -> this >>>error<<< in
10  the
11  whole
12  text
>>> print(s | findhighlight('error',line_nbr=True,ignorecase=True,
↵nlines=1).tostr() ) # doctest: +NORMALIZE_WHITESPACE
4   listing
5 -> with some >>>ERROR<<<
6   and I want to know
...
8   located
9 -> this >>>error<<< in
10  the

```

## 6.16 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}

```

## 6.17 formatdicts

**class** textops.**formatdicts** (*format\_str='{key} : {val}n', join\_str=", context={},  
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. One can specify a context dictionary and a default value : they will be used into the format string : see examples.

#### 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 ''
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str or callable*) – The replacement string or function for unexisting keys when formatting.

**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')      )#doctest:
↪+NORMALIZE_WHITESPACE
Eric(47) : guru
Guido(59) : god
>>> print(input | formatdicts('{name}', ', '))
Eric, Guido
>>> ctx = {'today':'2015-12-15'}
>>> print(input | formatdicts('[{today}] {name}({age}) : {level}', '\n',
↪context=ctx))
[2015-12-15] Eric(47) : guru
[2015-12-15] Guido(59) : god
>>> del input[0]['name']
>>> print(input | formatdicts('[{today}] {name}({age}) : {level}', '\n',
↪ctx, 'Unknown'))
[2015-12-15] Unknown(47) : guru
[2015-12-15] Guido(59) : god
```

## 6.18 formatitems

```
class textops.formatitems (format_str='{0} : {1}n', join_str="", context={},
                             defvalue='-')
```

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. One can specify a context dictionary and a default value : they will be used into the format string : see examples.

**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 ‘’
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str* or *callable*) – default string to display when a key or an index is unreachable.

**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'
>>> ctx = {'hostname' : 'pcubuntu'}
>>> d = [['Dimm1', '1024'], ['Dimm2', '512']]
>>> print(d | formatlists('{hostname}/{0} : {1} MB', '\n', ctx))
pcubuntu/Dimm1 : 1024 MB
pcubuntu/Dimm2 : 512 MB
>>> print(d | formatlists('{nodename}/{0} : {4} MB', '\n', ctx, '??'))
??/Dimm1 : ?? MB
??/Dimm2 : ?? MB
```

## 6.19 formatlists

**class** textops.**formatlists** (*format\_str*, *join\_str*=“, *context*={}, *defvalue*='-')

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. One can specify a context dictionary and a default value : they will be used into the format string : see examples.

**Parameters**

- **format\_str** (*str*) – format string
- **join\_str** (*str*) – string to join all strings into one unique string, default is ‘’
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str* or *callable*) – default string to display when a key or an index is unreachable.

**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)'
>>> ctx = {'hostname' : 'pcubuntu'}
>>> d = [['Dimm1','1','GB'],['Dimm2','512','MB']]
>>> print(d | formatlists('{hostname}/{0} : {1} {2}','\n',ctx))
pcubuntu/Dimm1 : 1 GB
pcubuntu/Dimm2 : 512 MB
>>> print(d | formatlists('{nodename}/{0} : {1} {4}','\n',ctx,'??'))
??/Dimm1 : 1 ??
??/Dimm2 : 512 ??
```

## 6.20 greaterequal

**class** `textops.greaterequal` (*value*, \**args*, \*\**kwargs*)

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']
```

## 6.21 greaterthan

**class** textops.**greaterthan** (*value*, \**args*, \*\**kwargs*)

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']
```

## 6.22 grep

**class** textops.**grep** (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
*has\_attr=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, Optional)
- **key** (*int* or *str*) – test the pattern only one column or one key (optional)
- **has\_key** (*int* or *str*) – test only if the test\_key is in the inner list or dict (optional)

- **attr** (*str*) – for list of objects, test the pattern on the object *attr* attribute (optional)
- **has\_attr** (*int* or *str*) – For list of objects, test if the attribute *has\_attr* exists (optional)

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

## Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grep('error') #doctest: +ELLIPSIS
<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+)') #doctest: +NORMALIZE_WHITESPACE
[ ['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+)') #doctest: +NORMALIZE_WHITESPACE
[{'level': 'error', 'nb': '1'}, {'level': 'error', 'nb': '2'},
 {'level': 'warning', 'nb': '1'}, {'level': 'info', 'nb': '1'},
 {'level': 'warning', 'nb': '2'}, {'level': 'info', 'nb': '2'}]
>>> input | cutdct(r'(?P<level>\D+)(?P<nb>\d+)').grep('1','nb').tolist()
[{'level': 'error', 'nb': '1'}, {'level': 'warning', 'nb': '1'},
 {'level': 'info', 'nb': '1'}]
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grep('way')
[{'way to grep': 2}]
>>> [{'more simple':1},{'way to grep':2},{'list of dicts':3}] | grep('3')
[{'list of dicts': 3}]
```

## 6.23 grepc

```
class textops.grepc(pattern=None, key=None, has_key=None, attr=None,
                    has_attr=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, Optional)
- **key** (*int* or *str*) – test the pattern only one column or one key (optional)

- **has\_key** (*int* or *str*) – test only if the test\_key is in the inner list or dict (optional)
- **attr** (*str*) – for list of objects, test the pattern on the object *attr* attribute (optional)
- **has\_attr** (*int* or *str*) – For list of objects, test if the attribute *has\_attr* exists (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
```

## 6.24 grepci

**class** `textops.grepci` (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
*has\_attr=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
```



## 6.25 grepcv

**class** `textops.grepcv` (*pattern=None, key=None, has\_key=None, attr=None, has\_attr=None*)

Count lines NOT having a specified pattern

This works like `textops.grep` 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
```

## 6.26 grepcvi

**class** `textops.grepcvi` (*pattern=None, key=None, has\_key=None, attr=None, has\_attr=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
```

## 6.27 grepi

**class** `textops.grepi` (*pattern=None, key=None, has\_key=None, attr=None, has\_attr=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 | grep('ERROR').tolist()
['error1', 'error2']
>>> input >> grep('ERROR')
['error1', 'error2']
```

## 6.28 grepv

**class** `textops.grepv` (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
                          *has\_attr=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']
```

## 6.29 grepvi

**class** `textops.grepvi` (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
                          *has\_attr=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']
```

## 6.30 haspattern

**class** textops.**haspattern** (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
*has\_attr=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.grepv` 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
```

## 6.31 haspatterni

**class** textops.**haspatterni** (*pattern=None*, *key=None*, *has\_key=None*, *attr=None*,  
*has\_attr=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
```

## 6.32 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()
[{'key': 'a', 'val': 1}, {'key': 'b', 'val': 2}]
```

## 6.33 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']
```

## 6.34 inrange

**class** textops.inrange(begin, end, get\_begin=True, get\_end=False, \*args, \*\*kwargs)

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']
```

(continues on next page)

(continued from previous page)

```
>>> 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()
[{'data': 'bbbb', 'date': '2015-08-23'}, {'data': 'ccc', 'date': '2015-
↪09-14'}]
```

```
>>> 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']
```

## 6.35 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)
```

(continues on next page)

(continued from previous page)

```
>>> [['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}
```

## 6.36 lcount

**class** textops.lcount

Count lines

**Returns** number of lines

**Return type** int

### Examples

```
>>> input = 'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | lcount()
6
```

## 6.37 less

**class** textops.less(*lines*)

Return all lines from the input text except the n last lines

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

**Yields** *str, lists or dicts* – all lines except the n last

### Examples

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

## 6.38 lessequal

**class** textops.lessequal(*value*, \**args*, \*\**kwargs*)

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']
```

## 6.39 lessthan

**class** textops.lessthan(*value*, \**args*, \*\**kwargs*)

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'),
... ('bbb', '2015-08-23'),
... ('ccc', '2015-09-14'),
... ('ddd', '2015-11-05') ]
>>> logs | lessthan('2015-11-05',key=1).tolist()
[('aaaa', '2015-08-11'), ('bbb', '2015-08-23'), ('ccc', '2015-09-14')]
>>> logs = [ {'data': 'aaaa', 'date': '2015-08-11'},
... {'data': 'bbb', 'date': '2015-08-23'},
... {'data': 'ccc', 'date': '2015-09-14'},
... {'data': 'ddd', 'date': '2015-11-05'} ]
>>> logs | lessthan('2015-09-14',key='date').tolist()
[{'data': 'aaaa', 'date': '2015-08-11'}, {'data': 'bbb', 'date': '2015-
↪08-23'}]
>>> 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']
```

## 6.40 linetester

**class** textops.linetester(\*args, \*\*kwargs)

Abstract class for by-line testing

## 6.41 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:1*2).tolist()
['aa', 'bb', 'cc']
>>> ['a', 'b', 'c'] >> mapfn(lambda l:1*2)
['aa', 'bb', 'cc']
```

## 6.42 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:1*2, lambda l:int(re.sub(r'\D', '',
↳l)) % 2).tolist()
['a1a1', 'c3c3']
>>> 'a1\nb2\nc3\nd4' >> mapif(lambda l:1*2, lambda l:int(re.sub(r'\D', '',
↳l)) % 2)
['a1a1', 'c3c3']
```

## 6.43 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')
↳ # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
```

(continues on next page)

(continued from previous page)

```

[{'coll': {'item': 'coll', 'i_count': 'col2', 'i_price': 'col3'}},
 {'coll1': {'item': 'coll1', 'i_count': 'col22', 'i_price': 'col33'}}]
>>> s | cutkv(pattern,key_name='item').merge_dicts()
↪ # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
{'coll': {'item': 'coll', 'i_count': 'col2', 'i_price': 'col3'},
 'coll1': {'item': 'coll1', 'i_count': 'col22', 'i_price': 'col33'}}

```

## 6.44 norepeat

**class** textops.**norepeat**

Remove line repetitions that follows

If a line is the same as the previous one, it is not yield. Works also with list of lists or dicts. Input order is preserved and it can be used on huge files.

**Returns** Unrepeated text line by line.

**Return type** generator

### Examples

```

>>> s='f\ na\ nb\ na\ nc\ nc\ ne\ na\ nc\ nf'
>>> s >> norepeat()
['f', 'a', 'b', 'a', 'c', 'e', 'a', 'c', 'f']
>>> l = [ [1,2], [3,4], [1,2], [1,2] ]
>>> l >> norepeat()
[[1, 2], [3, 4], [1, 2]]
>>> d = [ {'a':1}, {'b':2}, {'a':1}, {'a':1} ]
>>> d >> norepeat()
[{'a': 1}, {'b': 2}, {'a': 1}]

```

## 6.45 outrange

**class** textops.**outrange** (*begin, end, get\_begin=False, get\_end=False, \*args, \*\*kwargs*)

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']
```

## 6.46 renderdicts

**class** textops.**renderdicts** (*format\_str='{key} : {val}', context={}, defvalue='-'*)

Formats list of dicts

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

### Parameters

- **format\_str** (*str*) – format string, default is '{key} : {val}n'
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str or callable*) – The replacement string or function for unexisting keys when formatting.

**Yields** *str* – formatted string

## 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}') #doctest:
↳ +NORMALIZE_WHITESPACE
['Eric(47) : guru', 'Guido(59) : god']
>>> input >> renderdicts('{name}')
```

(continues on next page)

(continued from previous page)

```

['Eric', 'Guido']
>>> ctx = {'today': '2015-12-15'}
>>> input >> renderdicts('[{today}] {name}({age}) : {level}', ctx)
↪ #doctest: +NORMALIZE_WHITESPACE
['[2015-12-15] Eric(47) : guru', '[2015-12-15] Guido(59) : god']
>>> input >> renderdicts('[{to_day}] {name}({age}) : {level}', ctx,
↪ 'unknown')
['[unknown] Eric(47) : guru', '[unknown] Guido(59) : god']

```

## 6.47 renderitems

**class** textops.**renderitems** (*format\_str*='{0} : {1}', *context*={}, *defvalue*='-')

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}'
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str* or *callable*) – The replacement string or function for unexisting keys when formatting.

**Yields** *str* – formatted string

### Examples

```

>>> [('key1', 'val1'), ('key2', 'val2')] >> renderitems('{0} -> {1}')
['key1 -> val1', 'key2 -> val2']
>>> [('key1', 'val1'), ('key2', 'val2')] >> renderitems('{0}:{1}')
['key1:val1', 'key2:val2']
>>> ctx = {'today': '2015-12-15'}
>>> [('key1', 'val1'), ('key2', 'val2')] >> renderitems('[{today}] {0}:{1}',
↪ ctx)
['[2015-12-15] key1:val1', '[2015-12-15] key2:val2']
>>> [('key1', 'val1'), ('key2', 'val2')] >> renderitems('[{to_day}] {0}:{1}'
↪ ', ctx, 'unknown')
['[unknown] key1:val1', '[unknown] key2:val2']

```

## 6.48 renderlists

**class** textops.**renderlists** (*format\_str*, *context*={}, *defvalue*='-')

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}'
- **context** (*dict*) – additional context dictionary

- **defvalue** (*str* or *callable*) – The replacement string or function for unexisting keys when formatting.

**Yields** *str* – formatted string

### Examples

```
>>> input = [['key1', 'val1', 'help1'], ['key2', 'val2', 'help2']]
>>> input >> renderlists('{2} : {0} -> {1}')
['help1 : key1 -> val1', 'help2 : key2 -> val2']
>>> input >> renderlists('{0}:{1} ({2})')
['key1:val1 (help1)', 'key2:val2 (help2)']
>>> ctx = {'today':'2015-12-15'}
>>> input >> renderlists('{[today]} {0}:{1} ({2})', ctx)
['[2015-12-15] key1:val1 (help1)', '[2015-12-15] key2:val2 (help2)']
>>> input >> renderlists('{[to_day]} {0}:{1} ({2})', ctx, 'unknown')
['[unknown] key1:val1 (help1)', '[unknown] key2:val2 (help2)']
```

## 6.49 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 it 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
... =====
```

(continues on next page)

(continued from previous page)

```

... 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)      # doctest:␣
↪ +ELLIPSIS, +NORMALIZE_WHITESPACE
['', '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)  # doctest:␣
↪ +ELLIPSIS, +NORMALIZE_WHITESPACE
['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)  #␣
↪ doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
['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

```

(continues on next page)

(continued from previous page)

```

... '''
>>> 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) #
↳doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
['\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) #
↳doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
['=====\nSection 1\n=====\ninfo 1.1\ninfo 1.2\n',
'=====\nSection 2\n=====\ninfo 2.1\ninfo 2.2\n']

```

## 6.50 rmblank

**class** textops.**rmblank** (*pattern=None, key=None, has\_key=None, attr=None, has\_attr=None*)

Remove any kind of blank lines from the input text

A blank line can be an empty line or a line with only spaces and/or tabs.

**Returns** input text without blank lines

**Return type** generator

### Examples

```

>>> input = 'error1\n\n\nerror2\nwarning1\n\t\t\t'
↳\ninfo1\nwarning2\ninfo2'
>>> input | rmblank().tostr()
'error1\nerror2\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input = ['a', '', 'b', ' ', 'c', ' ', '\t\t', 'd']
>>> input | rmblank().tolist()
['a', 'b', 'c', 'd']

```

## 6.51 sed

**class** textops.**sed** (*pats, repls*)

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. if you specify a list of search patterns, all matches will be replaced by the same replace string specified. If you specify a list of string as replacement, pattern N from pattern list will be replaced by string N from replace string list.

### Parameters

- **pats** (*str* or *list*) – a string (case sensitive) or a regular expression or a list of that for the pattern(s) to search
- **repls** (*str* or *list*) – the replace string or a list of strings for a list of search patterns.

**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']
>>> [ ['Hello','Eric'], ['Hello','Guido'] ] | sed('Hello','Good bye') .
↳tolist()
[['Good bye', 'Eric'], ['Good bye', 'Guido']]
>>> [ 'Hello Eric','Hello Guido'] >> sed(['Eric','Guido'],'Mister')
['Hello Mister', 'Hello Mister']
>>> [ 'Hello Eric','Hello Guido'] >> sed(['Eric','Guido'], ['Padawan',
↳'Jedi'])
['Hello Padawan', 'Hello Jedi']
```

## 6.52 sedi

**class** textops.**sedi** (*pats, repls*)

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']
```

## 6.53 since

**class** textops.**since** (*pattern, key=None*)

Extract lines beginning with and after a patterns

Works like *textops.after* except that it includes the first pattern found.

**Parameters**

- **pattern** (*str* or *regex* or *list*) – start yielding lines after reaching this pattern(s)
- **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'] | since('c').tolist()
['c', 'd', 'e', 'f']
```

## 6.54 skess

**class** textops.**skess** (*begin*, *end*)

skip *x* lines at the beginning and *y* at the end from the input text

This will do a *skip* and a *less* in a single operation.

**Parameters**

- **begin** (*int*) – The number of beginning lines/items to remove.
- **end** (*int*) – The number of ending lines/items to remove.

**Yields** *str*, *lists* or *dicts* – all lines except the specified number at begin and end

**Examples**

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

## 6.55 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()
[{'key': 'b', 'val': 2}, {'key': 'c', 'val': 3}]
```

## 6.56 sortdicts

**class** textops.**sortdicts** (*key, reverse=False*)

Sort list of dicts

#### Parameters

- **key** (*int or tuple/list*) – The dict key or list of keys as sorting criteria
- **reverse** (*bool*) – Reverse the sort (Default : False)

**Returns** sorted input

**Return type** list of dicts

### Examples

to come...

## 6.57 sortlists

**class** textops.**sortlists** (*col, reverse=False*)

Sort list of dicts

#### Parameters

- **col** (*int or tuple/list*) – The column number or list of columns as sorting criteria
- **reverse** (*bool*) – Reverse the sort (Default : False)

**Returns** sorted input

**Return type** list of lists

## Examples

to come...

## 6.58 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. If only simple list or strings are given, they are converted into list of list.

### Parameters

- **nbcols** (*int*) – number columns to return
- **fill\_str** (*str*) – the value to return for not existing columns

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

**Return type** list

## Examples

```
>>> s='a\nc\nd e f g h\ni j k\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
- - -
>>> 'a b' | cut()
['a', 'b']
>>> 'a b' | cut().span(3, '-').tolist()
[['a', 'b', '-']]
```

## 6.59 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)      # doctest:␣
↪+ELLIPSIS, +NORMALIZE_WHITESPACE
[['', ['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)  # doctest:␣
↪+ELLIPSIS, +NORMALIZE_WHITESPACE
```

(continues on next page)

(continued from previous page)

```
[['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)      # doctest:␣  
↪ +ELLIPSIS, +NORMALIZE_WHITESPACE  
[['info 1.1', 'Last info 1.2'], ['info 2.1', 'Last info 2.2'],  
['info 3.1', 'Last info 3.2']]
```

## 6.60 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', '-']
```

## 6.61 subitems

**class** textops.subitems(*ntab*)

Get the specified columns for list of lists

**Parameters** *ntab* (*list of int*) – columns numbers to get.

**Returns** A list of lists

**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,'-').subitems([0,2]).tolist()
[['a', '-'], ['b', '-'], ['d', 'f'], ['i', 'k'], ['- ', '-']]
>>> s >> cut().span(3,'-').subitems('0,2')
[['a', '-'], ['b', '-'], ['d', 'f'], ['i', 'k'], ['- ', '-']]
```

## 6.62 subslice

**class** textops.**subslice** (*begin=0, end=sys.maxint, 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:  $0 \leq x \leq \text{maxint}$
- **end** (*int*) – end columns number (get columns up to end - 1). must be None or an integer:  $0 \leq x \leq \text{maxint}$
- **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'], ['- ', '-']]
```

## 6.63 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()
[{'key': 'b', 'val': 2}, {'key': 'c', 'val': 3}]
```

## 6.64 uniq

**class** textops.uniq

Remove all line repetitions at any place

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. Input order is preserved. Do not use this operation on huge data set as an internal list is maintained : if possible, use *textops.norepeat*.

**Returns** Uniqified 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}]
```



## 6.65 until

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

Extract lines before a patterns (case insensitive)

Works like `textops.before` except that `get_end=True` (includes first found pattern)

### Parameters

- **pattern** (*str* or *regex* or *list*) – no more lines are yield after reaching this pattern(s)
- **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'] | until('c').tolist()
['a', 'b', 'c']
```

## 6.66 wcount

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

Count words having a specified pattern

### Parameters

- **pattern** (*str*) – a regular expression string (case sensitive), if *None*, all words are counted.
- **key** (*int* or *str*) – test only one column or one key (optional)

**Returns** the matched words count

**Return type** `int`

### Examples

```
>>> input = 'error1\nerror2 error3\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | grepc('error')
2
>>> input | wcount('error')
3
>>> input | wcount(None)
7
```

## 6.67 wcounti

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

Count words having a specified pattern (case insensitive)

### Parameters

- **pattern** (*str*) – a regular expression string (case insensitive), if None, all words are counted.
- **key** (*int* or *str*) – test only one column or one key (optional)

**Returns** the matched words count

**Return type** `int`

### Examples

```
>>> input = 'error1\nerror2 error3\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | wcount('ERROR')
0
>>> input | wcounti('ERROR')
3
```

## 6.68 wcountv

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

Count words NOT having a specified 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 words count

**Return type** `int`

### Examples

```
>>> input = 'error1\nerror2 error3\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | wcountv('error')
4
```

## 6.69 wcountvi

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

Count words NOT having a specified pattern (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 words count

**Return type** `int`

## Examples

```
>>> input = 'error1\nerror2 error3\nwarning1\ninfo1\nwarning2\ninfo2'
>>> input | wcountvi('ERROR')
4
```



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

## 7.1 bzcat

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

Uncompress the bz2 file(s) with the name(s) given in input text

If a context dict is specified, the path is formatted with that context (`str.format`) The gzipped file must have a textual content.

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

### Examples

```
>>> '/var/log/dmesg' | cat() | grep('error') | togzfile('/tmp/errors.log.  
↪gz')
```

---

**Note:** A list of filename can be given as input text : all specified files will be uncompressed

---

## 7.2 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`) The file must have a textual content.

**Parameters**

- **context** (*dict*) – The context to format the file path (Optionnal)
- **encoding** (*str*) – file encoding (Default: utf-8)
- **encoding\_errors** (*str*) – ‘strict’, ‘ignore’, ‘replace’, ‘xmlcharrefreplace’, ‘backslashreplace’ (Default : ‘replace’)

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

## Examples

```
>>> open('/tmp/testfile.txt','w').write('here is the file content')
24
>>> '/tmp/testfile.txt' | cat()                                #doctest: +ELLIPSIS
<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)                        #doctest: _
↪+ELLIPSIS
<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                                #doctest: +ELLIPSIS
<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')
37
>>> '/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():
```

(continues on next page)

(continued from previous page)

```

...     print(bits)
...
['here', 'is', 'the', 'file', 'content']

```

## 7.3 find

**class** `textops.find` (*pattern*='\*', *context*={}, *only\_files*=False, *only\_dirs*=False)

Return a list of files/dirs matching a pattern

find recursively files/dirs matching a pattern. The pattern is a unix-like pattern, it searches only against the last part of the path (basename)

### Parameters

- **pattern** (*str*) – the file pattern to search
- **context** (*dict*) – The context to format the file path (Optionnal)
- **only\_files** (*bool*) – get only files (Default : False)
- **only\_dirs** (*bool*) – get only dirs (Default : False)

**Yields** *str* – file name matching the pattern

### Examples

To come ...

## 7.4 findre

**class** `textops.findre` (*pattern*="", *context*={}, *only\_files*=False, *only\_dirs*=False)

Return a list of files/dirs matching a pattern

find recursively files/dirs matching a pattern. The pattern is a python regex, it searches against the whole file path

### Parameters

- **pattern** (*str or regex*) – the file pattern to search
- **context** (*dict*) – The context to format the file path (Optionnal)
- **only\_files** (*bool*) – get only files (Default : False)
- **only\_dirs** (*bool*) – get only dirs (Default : False)

**Yields** *str* – file name matching the pattern

### Examples

To come ...

## 7.5 gzcat

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

Uncompress the gzfile(s) with the name(s) given in input text

If a context dict is specified, the path is formatted with that context (`str.format`) The gzipped file must have a textual content.

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

---

**Note:** A list of filename can be given as input text : all specified files will be uncompressed

---

---

**Note:** Password encrypted zip creation is not supported.

---

## 7.6 ls

**class** `textops.ls` (*pattern*='\*', *context*={}, *only\_files*=False, *only\_dirs*=False)

Return a list of files/dirs

it uses the python `glob.glob()` so it will do a Unix style pathname pattern expansion

**Parameters**

- **pattern** (*str*) – the file pattern to search
- **context** (*dict*) – The context to format the file path (Optionnal)
- **only\_files** (*bool*) – get only files (Default : False)
- **only\_dirs** (*bool*) – get only dirs (Default : False)

**Yields** *str* – file name matching the pattern

### Examples

To come ...

## 7.7 replacefile

**class** `textops.replacefile` (*filename*, *mode*='w', *newline*='n')

send input to file

Works like `textops.to_file` except it takes care to consume input text generators before writing the file. This is mandatory when doing some in-file textops. The drawback is that the data to write to file is stored temporarily in memory.

This does not work:

```
cat('myfile').sed('from_patter','to_pattern').tofile('myfile').n
```

This works:



```
cat('myfile').sed('from_patter', 'to_pattern').replacefile('myfile').n
```

#### Parameters

- **filename** (*str*) – The file to send output to
- **mode** (*str*) – File open mode (Default : 'w')
- **newline** (*str*) – The newline string to add for each line (default: 'n')

#### Examples

```
>>> cat('myfile').sed('from_patter', 'to_pattern').replacefile('myfile').n
```

## 7.8 stats

**class** textops.**stats**

Return a dict or a list of dicts containing the filename and its statistics

it uses the python `os.stat()` to get file statistics, the filename is stored in 'filename' key

**Yields** *dict* – file name and file statistics in the same dict

#### Examples

To come ...

## 7.9 tee file

**class** textops.**tee file** (*filename, mode='w', newline='n'*)

send input to file AND yield the same input text

#### Parameters

- **filename** (*str*) – The file to send output to
- **mode** (*str*) – File open mode (Default : 'w')
- **newline** (*str*) – The newline string to add for each line (default: 'n')

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

#### Examples

```
>>> '/var/log/dmesg' | cat() | tee file('/tmp/dmesg_before') | grep('error
↪') | tofile('/tmp/dmesg_after')
```

## 7.10 tobz2file

**class** textops.tobz2file(*filename*, *mode*='w', *newline*='n')

send input to gz file

tobz2file() must be the last text operation

### Parameters

- **filename** (*str*) – The gz file to send COMPRESSED output to
- **mode** (*str*) – File open mode (Default : 'wb')
- **newline** (*str*) – The newline string to add for each line (default: 'n')

### Examples

```
>>> '/var/log/dmesg' | cat() | grep('error') | togzfile('/tmp/errors.log.  
↪gz')
```

---

**Note:** Password encrypted zip creation is not supported.

---

## 7.11 tofile

**class** textops.tofile(*filename*, *mode*='w', *newline*='n')

send input to file

tofile() must be the last text operation, if you want to write to file AND continue some text operations, use *textops.teefile* instead. if you want to write the same file than the one opened, please use *textops.replacefile* instead.

### Parameters

- **filename** (*str*) – The file to send output to
- **mode** (*str*) – File open mode (Default : 'w')
- **newline** (*str*) – The newline string to add for each line (default: 'n')

### Examples

```
>>> '/var/log/dmesg' | cat() | grep('error') | tofile('/tmp/errors.log')
```

## 7.12 togzfile

**class** textops.togzfile(*filename*, *mode*='wb', *newline*='n')

send input to gz file

togzfile() must be the last text operation

### Parameters

- **filename** (*str*) – The gz file to send COMPRESSED output to
- **mode** (*str*) – File open mode (Default : 'wb')
- **newline** (*str*) – The newline string to add for each line (default: 'n')

### Examples

```
>>> '/var/log/dmesg' | cat() | grep('error') | togzfile('/tmp/errors.log.  
↪gz')
```

---

**Note:** Password encrypted zip creation is not supported.

---

## 7.13 tozipfile

**class** textops.**tozipfile** (*filename, member, mode='w', newline='n'*)

send input to zip file

tozipfile() must be the last text operation.

### Parameters

- **filename** (*str*) – The zip file to send COMPRESSED output to
- **member** (*str*) – The name of the file INSIDE the zip file to send UNCOMPRESSED output to
- **mode** (*str*) – File open mode (Default : 'w', use 'a' to append an existing zip or create it if not present)
- **newline** (*str*) – The newline string to add for each line (default: 'n')

### Examples

```
>>> '/var/log/dmesg' | cat() | grep('error') | tozipfile('/tmp/errors.  
↪log.zip', '/tmp/errors.log')
```

---

**Note:** Password encrypted zip creation is not supported.

---

## 7.14 unzip

**class** textops.**unzip** (*member, topath=None, password=None, context={}, ignore=False*)

Extract ONE file from a zip archive

The zip file name is taken from text input.

### Parameters

- **member** (*str*) – the file name to extract from the zip archive
- **topath** (*str*) – the directory path to extract to (Default : current directory)

- **password** (*str*) – The password to open zip if it is encrypted (Optionnal)
- **context** (*dict*) – The context to format the file path and topath argument (Optionnal)
- **ignore** (*bool*) – If True do not raise exception when member does not exist (Default : False)

**Yields** *str* – the member file path

### Examples

To come ...

## 7.15 unzipre

**class** textops.**unzipre** (*member\_regex*, *topath=None*, *password=None*, *context={}, ignore=False*)

Extract files having a specified name pattern from a zip archive

The zip file name is taken from text input.

### Parameters

- **member\_regex** (*str or regex*) – the regex to find the first file inside the zip to read
- **topath** (*str*) – the directory path to extract to (Default : current directory)
- **password** (*str*) – The password to open zip if it is encrypted (Optionnal)
- **context** (*dict*) – The context to format the file path and topath argument (Optionnal)

**Yields** *str* – the extracted files path

### Examples

To come ...

## 7.16 zipcat

**class** textops.**zipcat** (*member*, *context={}, password=None*)

Return the content of the zip 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

- **member** (*str*) – the file inside the zip to read
- **context** (*dict*) – The context to format the file path (Optionnal)
- **password** (*str*) – The password to open zip if it is encrypted (Optionnal)

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

## Examples

To come ...

## 7.17 zipcatre

**class** `textops.zipcatre` (*member\_regex*, *context*={}, *password*=None)

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

If a context dict is specified, the path is formatted with that context (`str.format`) Works like `textops.zipcat` except that the member name is a regular expression : it will cat all member matching the regex

### Parameters

- **member\_regex** (*str* or *regex*) – the regex to find the files inside the zip to read
- **context** (*dict*) – The context to format the file path (Optionnal)
- **password** (*str*) – The password to open zip if it is encrypted (Optionnal)

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

## Examples

To come ...

## 7.18 ziplist

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

Return the name of the files included within the zip file

The zip file name is taken from text input

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

**Yields** *str* – the file names

## Examples

To come ...



This module gathers list/line operations

## 8.1 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')
```

(continues on next page)

(continued from previous page)

```
>>> 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 !
```

## 8.2 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
```



## 8.3 xrun

**class** textops.**xrun** (*cmd*, *context*={}, *defvalue*='unknwon')

Run the command formatted with the context taken from the input text

### Parameters

- **command** (*str*) – The command pattern to run (formatted against the context)
- **context** (*dict*) – additional context dictionary
- **defvalue** (*str* or *callable*) – default string to display when a key or an index is unreachable.

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

### Examples

to come...



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

### 9.1 WrapOp

```
class textops.WrapOp
```

### 9.2 WrapOpIter

```
class textops.WrapOpIter
```

### 9.3 WrapOpStr

```
class textops.WrapOpStr
```

### 9.4 resub

```
class textops.resub (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.']
```

This module gathers parsers to handle whole input text

## 10.1 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', 'month': '11', 'day': '16'}
```

## 10.2 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 insensitive.

**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'
```

## 10.3 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
```

## 10.4 find\_patterni

**class** textops.**find\_patterni** (*pattern*)

Fast pattern search (case insensitive)

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

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

**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'
```

## 10.5 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', 'minor': '2', 'build': '3'}, 'format': 'json'}
>>> r.version.major
'1'
>>> s | find_patterns({
... 'version': r'^Version:\s*(\d+)\.(\d+)\.(\d+)',
... 'format': r'^Format:\s*(.*)',
... })
{'version': {'group0': '1', 'group1': '2', 'group2': '3'}, 'format':
↪ 'json'}
```

(continues on next page)

(continued from previous page)

```

>>> 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*(.*)'])
[{'year': '2015', 'month': '11', 'day': '16'}, '2015-11-17', '2015-10-14
↪']

```

## 10.6 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'}

```

## 10.7 keyval

**class** textops.**keyval** (*pattern*, *key\_name*='key', *key\_update*=None, *val\_name*=None)

Return a dictionary where keys and values are taken from the pattern specify

It is a shortcut for *textops.parsekv* with *val\_name*='val' The input can be a string or a list of strings.

### Parameters

- **pattern** (*str*) – a regular expression string.
- **key\_name** (*str*) – The key name to obtain the value that will be the key of the groupdict ('key' by default)
- **key\_update** (*callable*) – function to convert/normalize the calculated key. If None, the keys is normalized. If not None but not callable ,the key is unchanged.



- **val\_name** (*str*) – instead of storing the groupdict, one can choose to select the value at the key “val\_name”. (by default, None means ‘val’)

**Returns** A dict of key:val from the matched pattern groupdict or a list of dicts if the input is a list of strings

**Return type** dict

### Examples

```
>>> s = '''name: Lapouyade
... first name: Eric
... country: France'''
>>> s | keyval(r'(?P<key>.*):(?P<val>.*)') #doctest:
↪+NORMALIZE_WHITESPACE
{'name': 'Lapouyade', 'first_name': 'Eric', 'country': 'France'}
```

```
>>> s = [ '''name: Lapouyade
... first name: Eric ''',
...       '''name: Python
... first name: Guido''' ]
>>> s | keyval(r'(?P<key>.*):(?P<val>.*)') #doctest:
↪+NORMALIZE_WHITESPACE
[{'name': 'Lapouyade', 'first_name': 'Eric '}, {'name': 'Python', 'first_
↪name': 'Guido'}]
```

## 10.8 keyvali

**class** textops.**keyvali** (*pattern*, *key\_name*=‘key’, *key\_update*=None, *val\_name*=None)

Return a dictionary where keys and values are taken from the pattern specify

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

#### Parameters

- **pattern** (*str*) – a regular expression string (case insensitive).
- **key\_name** (*str*) – The key name to obtain the value that will be the key of the groupdict (‘key’ by default)
- **key\_update** (*callable*) – function to convert/normalize the calculated key. If None, the key is normalized. If not None but not callable, the key is unchanged.
- **val\_name** (*str*) – instead of storing the groupdict, one can choose to select the value at the key “val\_name”. (by default, None means ‘val’)

**Returns** A dict of key:val from the matched pattern groupdict

**Return type** dict

### Examples

```
>>> s = '''name IS Lapouyade
... first name IS Eric
... country IS France'''
```

(continues on next page)

(continued from previous page)

```
>>> s | keyvali(r'(?P<key>.*) is (?P<val>.*)') #doctest:␣
↪+NORMALIZE_WHITESPACE
{'name': 'Lapouyade', 'first_name': 'Eric', 'country': 'France'}
```

## 10.9 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) #doctest: +NORMALIZE_WHITESPACE
{'errors': ['error 1', 'error 2'],
 'warnings': ['warning 1', 'warning 2'],
 'infos': ['info 1', 'info 2']}
```

```
>>> s = '''
... Disk states
... -----
... name: clt0d0s0
... state: good
... fs: /
... name: clt0d0s4
... state: failed
... fs: /home
```

(continues on next page)

(continued from previous page)

```

...
... '''
>>> t = s | mgrep({
... 'disks' : r'^name:',
... 'states' : r'^state:',
... 'fss' : r'^fs:',
... })
>>> print(t) #doctest: +NORMALIZE_WHITESPACE
{'disks': ['name: clt0d0s0', 'name: clt0d0s4'],
'states': ['state: good', 'state: failed'],
'fss': ['fs: /', 'fs: /home']}
>>> dict(zip(t.disks.cutre(':', '*', 1), zip(t.states.cutre(':', '*', 1), t.fss.
↵cutre(':', '*', 1))))
{'clt0d0s0': ('good', '/'), 'clt0d0s4': ('failed', '/home')}

```

## 10.10 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']}

```

## 10.11 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)                                     ) #doctest:␣
↪+NORMALIZE_WHITESPACE
{'not_warnings': ['error 1', 'error 2'], 'not_errors': ['warning 1',
↪'warning 2']}
```

## 10.12 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',
```

(continues on next page)

(continued from previous page)

```

... })
>>> print(t)                                     )#doctest:␣
↪+NORMALIZE_WHITESPACE
{'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)                                     )#doctest:␣
↪+NORMALIZE_WHITESPACE
{'not_warnings': ['error 1', 'ERROR 2'], 'not_errors': ['WARNING 1',
↪'warning 2']}

```

## 10.13 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'}

```

## 10.14 parse\_smart

**class** textops.parse\_smart

Try to automatically parse a text

It looks for key/value patterns, store found values in a dictionary. It tries to respect indents by creating sub-dictionaries. The keys are normalized (only keep A-Za-z0-9\_, the original key value is stored into the inner dict under the ‘\_original\_key’ key), the values are stripped.

### Parameters

- **key\_filter** (*func*) – a function that will receive a key before
- **and will return a new key string. The could be useful** (*normalization*) –
- **a chapter title is too long. (Default** (*when*) – no filtering)

**Returns** structured keys:values

**Return type** dict

### Examples

```
>>> s = '''
... Date/Time:      Wed Dec  2 09:51:17 NFT 2015
... Sequence Number: 156637
... Machine Id:     00F7B0114C00
...   Node Id:      xvio6
... Class:          H
... Type:           PERM
...   WPAR:         Global
...   Resource Name: hdisk21
...   Resource Class: disk
... Resource Type:  mpioapdisk
... Location:       U78AA.001.WZSHMOM-P1-C6-T1-W201400A0B8292A18-
↳ L130000000000000
...
... VPD:
...   Manufacturer.....IBM
...   Machine Type and Model.....1815      FAStT
...   ROS Level and ID.....30393134
...   Serial Number.....
...   Device Specific.(Z0).....0000053245004032
...   Device Specific.(Z1).....
...
... Description
... DISK OPERATION ERROR
...
... Probable Causes
... DASD DEVICE
... '''
>>> parsed = s >> parse_smart()
>>> print(parsed.pretty())
{  'class': 'H',
   'date_time': 'Wed Dec  2 09:51:17 NFT 2015',
   'description': ['DISK OPERATION ERROR'],
   'location': 'U78AA.001.WZSHMOM-P1-C6-T1-W201400A0B8292A18-
↳ L130000000000000',
```

(continues on next page)

(continued from previous page)

```

'machine_id': { '_original_key': 'Machine Id',
                'machine_id': '00F7B0114C00',
                'node_id': 'xvio6'},
'probable_causes': ['DASD DEVICE'],
'resource_type': 'mpioapdisk',
'sequence_number': '156637',
'type': { '_original_key': 'Type',
          'resource_name': { '_original_key': 'Resource Name',
                              'resource_class': 'disk',
                              'resource_name': 'hdisk21'},
          'type': 'PERM',
          'wpar': 'Global'},
'vpd': { '_original_key': 'VPD',
         'device_specific_z0': '0000053245004032',
         'device_specific_z1': '',
         'machine_type_and_model': '1815          FAStT',
         'manufacturer': 'IBM',
         'ros_level_and_id': '30393134',
         'serial_number': ''}}
>>> print(parsed.vpd.device_specific_z0)
0000053245004032

```

## 10.15 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>.*)')           #doctest:␣
↪+NORMALIZE_WHITESPACE
[{'key': 'name', 'val': 'Lapouyade'},
 {'key': 'first name', 'val': 'Eric'},
 {'key': 'country', 'val': 'France'}]

```

## 10.16 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>.*)')
↪ #doctest: +NORMALIZE_WHITESPACE
[{'level': 'Error', 'msg': 'System will reboot'},
 {'level': 'Warning', 'msg': 'Python must be used without moderation'}]
```

## 10.17 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 corresponding 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']
```

## 10.18 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 corresponding 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']
```

## 10.19 parsekv

**class** `textops.parsekv` (*pattern*, *key\_name*=*'key'*, *key\_update*=*None*, *val\_name*=*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 obtain the value that will be the key of the groupdict (*'key'* by default)
- **key\_update** (*callable*) – function to convert/normalize the calculated key. If *None*, the keys is normalized. If not *None* but not callable ,the key is unchanged.
- **val\_name** (*str*) – instead of storing the groupdict, on can choose to select the value at the key *"val\_name"*. (by default, *None* : means the whole groupdict)

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

(continues on next page)

(continued from previous page)

```
>>> s | parsekv(r'(?P<key>.*):\s*(?P<val>.*)',key_update=0)
↪#doctest: +NORMALIZE_WHITESPACE
{'name': {'key': 'name', 'val': 'Lapouyade'},
'first name': {'key': 'first name', 'val': 'Eric'},
'country': {'key': 'country', 'val': 'France'}}
>>> s | parsekv(r'(?P<key>.*):\s*(?P<val>.*)',val_name='val')
↪#doctest: +NORMALIZE_WHITESPACE
{'name': 'Lapouyade', 'first_name': 'Eric', 'country': 'France'}
```

## 10.20 parsekvi

**class** `textops.parsekvi` (*pattern*, *key\_name*='key', *key\_update*=None, *val\_name*=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 obtain the value that will be the key of the groupdict ('key' by default)
- **key\_update** (*callable*) – function to convert/normalize the calculated key. If None, the keys is normalized. If not None but not callable ,the key is unchanged.
- **val\_name** (*str*) – instead of storing the groupdict, on can choose to select the value at the key “val\_name”. (by default, None : means the whole groupdict)

**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': {'key': 'name', 'val': 'Lapouyade'}}
```

## 10.21 sgrep

**class** `textops.sgrep` (*patterns*, *key*=None)

Switch grep

This works like `textops.mgrep` except that it returns a list of lists. `sgrep` dispatches lines matching a pattern to the list corresponding to the pattern order. If a line matches the third pattern, it will be dispatched to the third returned list. If N patterns are given to search, it will return N+1 lists, where the last list will be filled of lines that does not match any pattern in the given patterns list. The patterns list order is important : only the first matching pattern will be taken in account. One can consider that `sgrep` works like a `switch()` : it will do for each line a kind of

```

if pattern1 matches:
    put line in list1
elif pattern2 matches:
    put line in list2
elif patternN matches:
    put line in listN
else:
    put line in listN+1

```

#### Parameters

- **patterns** (*list*) – a list of patterns to search.
- **key** (*int* or *str*) – test only one column or one key (optional)

**Returns** a list of lists (nb patterns + 1)

**Return type** list

#### Examples

```

>>> logs = '''
... error 1
... warning 1
... warning 2
... info 1
... error 2
... info 2
... '''
>>> t = logs | sgrep(('^err','^warn'))
>>> print(t)                                     ) #doctest:␣
↪ +NORMALIZE_WHITESPACE
[['error 1', 'error 2'], ['warning 1', 'warning 2'], ['', 'info 1',
↪ 'info 2']]

```

## 10.22 sgrep\_i

**class** textops.**sgrep\_i** (*patterns, key=None*)

Switch grep case insensitive

This works like `textops.sgrep` but is case insensitive

## 10.23 sgrep\_v

**class** textops.**sgrep\_v** (*patterns, key=None*)

Switch grep reversed

This works like `textops.sgrep` except that it tests that patterns DOES NOT match the line.

## 10.24 sgrepvi

**class** textops.**sgrepvi** (*patterns*, *key=None*)

Switch grep reversed case insensitive

This works like `textops.sgrepv` but is case insensitive

## 10.25 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** (*tuple*) – description of states and patterns : see below for explanation
- **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 tuple. 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. if the pattern matches, no more rules are used for the current line except when you specify `__continue__` for the goto state. This is useful when you want to apply several rules on the same line.

**<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 matching process, The syntax is:

```

'{contextkey1}.{contextkey2}. ... .{contextkeyN}'
or
('{contextkey1}','{contextkey2}', ... ,'{contextkeyN}')
or
'key1.key2.keyN'
or
'key1.key2.keyN['
or
'{contextkey1}.{contextkey2}. ... .keyN['
or
'>context_dict_key'
or
'>>context_dict_key'
or
'>context_dict_key.{contextkey1}. ... .keyN'
or
'>>context_dict_key.{contextkey1}. ... .keyN'
or
None

```

The contextdict (see after the definition) is used to format strings with {contextkeyN} syntax. instead of {contextkeyN}, one can use a simple string to put data in a static path.

Once the path fully formatted, let's say to key1.key2.keyN, the parser will store the value into the result dictionary at : {'key1':{'key2':{'keyN' : thevalue }}}}

Example, Let's take the following data path

```

data path : 'disks.{name}.{var}'

if contextdict = {'name':'disk1','var':'size'}

then the formatted data path is : 'disks.disk1.size',
This means that the parsed data will be stored at :
``{'disks':{'disk1':{'size' : theparsedvalue depending on <out filter>
↪ }}}``

```

One can use the string [] at the end of the path : the groupdict will be appended in a list ie : {'key1':{'key2':{'keyN' : [thevalue,...] }}}}

if '>context\_dict\_key' is used, data is not store in parsed data but will be stored in context dict at context\_dict\_key key. by this way, you can differ the parsed date storage. To finally store to the parsed data use '<context\_dict\_key' for <out filter> in some other rule. '>>context\_dict\_key' works like '>context\_dict\_key' but it updates data instead of replacing them (in others words : use > to start with an empty set of data, then use >> to update the data set). One can add dotted notation to complete data path: >>context\_dict\_key.{contextkey1}. ... .keyN

if None is used : nothing is stored

**<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 dict : mainly to initialize the differed data set when using '>context\_dict\_key' in <out data path>
- '<context\_dict\_key' to store data from context dict at key context\_dict\_key

- a string : used as a format string with context dict, the formatted string is stored
- a callable : to calculate the value to be stored and modify context dict if needed. the `re.MatchObject` and the context dict are given as arguments, it must return a tuple : the value to store AND the new context dict or `None` if unchanged

### 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 example, 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'),
↳) ) #doctest: +NORMALIZE_WHITESPACE
{'first_name': {'key': 'first name', 'val': 'Eric'},
 'last_name': {'key': 'last name', 'val': 'Lapouyade'}}
>>> 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( (                                     #doctest:_
↪+NORMALIZE_WHITESPACE
... ('','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}'
↪')) )
{'section1': {'email': 'ericdupo@gmail.com'},
'section2': {'first_name': 'Eric', 'last_name': 'Dupont'}}

```

```

>>> s = '''
... Disk states
... -----
... name: clt0d0s0
... state: good
... fs: /
... name: clt0d0s4
... state: failed
... fs: /home
...
... '''
>>> s | state_pattern( (                                     #doctest:_
↪+NORMALIZE_WHITESPACE
... ('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': {'clt0d0s0': {'state': 'good', 'fs': '/'},
'clt0d0s4': {'state': 'failed', 'fs': '/home'}}}

```

```

>>> s = '''
... {
... name: clt0d0s0
... state: good
... fs: /
... },
... {
... fs: /home
... name: clt0d0s4
... }
... '''
>>> s | state_pattern( (                                     #doctest:_
↪+NORMALIZE_WHITESPACE
... ('top','disk',r'{'','>disk_info',{}),
... ('disk',' ', r'(?P<key>.*):(?P<val>.*)','>>disk_info.{key}','{val}'
↪'),

```

(continues on next page)

(continued from previous page)

```
... ('disk', 'top', 'r')', 'disks.{disk_info[name]}', '<disk_info'),
... ) )
{'disks': {'c1t0d0s0': {'name': 'c1t0d0s0', 'state': 'good', 'fs': '/'},
'c1t0d0s4': {'fs': '/home', 'name': 'c1t0d0s4'}}
```

```
>>> s='firstname:Eric lastname=Lapouyade'
>>> s | state_pattern((
... ('top', '', r'firstname:(?P<val>\S+)', 'firstname', '{val}'),
... ('top', '', r'.*lastname=(?P<val>\S+)', 'lastname', '{val}'),
... ))
{'firstname': 'Eric'}
```

```
>>> s='firstname:Eric lastname=Lapouyade'
>>> s | state_pattern((
... ('top', '__continue__', r'firstname:(?P<val>\S+)', 'firstname', '{val}'),
... ('top', '', r'.*lastname=(?P<val>\S+)', 'lastname', '{val}'),
... ))
{'firstname': 'Eric', 'lastname': 'Lapouyade'}
```



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

## 11.1 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'}
```

## 11.2 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)
```

## 11.3 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, 'b': 2, 'c': 3}
```

## 11.4 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]
```

## 11.5 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]
```

## 11.6 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())
<class 'NoneType'>
>>> def g(): yield 'hello'
...
>>> g() | tolist()
['hello']
```

## 11.7 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()
[]
```

## 11.8 tonull

**class** textops.**tonull**  
Consume generator if any and then return nothing (aka None)

Examples:

```
>>> [ 1, 2, 3 ] | echo().tonull()
```

## 11.9 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'
```

## 11.10 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()
<class 'NoneType'>
>>> None | tostr(return_if_none='N/A')
'N/A'
```

## 11.11 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()
''
```

This module gathers operations for text recoding

## 12.1 list\_to\_multilinestring

```
class textops.list_to_multilinestring(in_place=False, tag='-----  
                                     < Multiline string as list >-----  
                                     ---)
```

In a data structure, change all tagged list of strings into multiline strings

This is useful to undo data recoded by *multilinestring\_to\_list*.

**Returns** Same data structure with tagged lists replaced by multiple line strings.

### Examples

```
>>> data=[
...     [
...         "-----< Multiline string as list >-----"
...         ↪-----",
...         "line1",
...         "line2",
...         "line3"
...     ],
...     {
...         "key2": [
...             "-----< Multiline string as list >-----"
...             ↪-----",
...             "lineA",
...             "lineB",
...             "lineC",
...             "lineD"
```

(continues on next page)

(continued from previous page)

```

...     ],
...     "key1": "one line"
... }
... ]
>>> data | list_to_multilinesstring()
['line1\nline2\nline3', {'key2': 'lineA\nlineB\nlineC\nlineD', 'key1':
↪ 'one line'}]

```

## 12.2 multilinesstring\_to\_list

**class** `textops.multilinesstring_to_list` (*in\_place=False*)

In a data structure, change all multiline strings into a list of strings

This is useful for `json.dump()` or `dumps()` in order to have a readable json data when the structure has some strings with many lines. This works on imbricated dict and list. Dictionary keys are not changed, only their values. Each generated list of strings is tagged with a first item (`MULTIPLELINESTRING_TAG`). By this way the process is reversible : see `list_to_multilinesstring`

**Returns** Same data structure with multiple line strings replaced by lists.

### Examples

```

>>> data=['line1\nline2\nline3',{'key1':'one line','key2':
↪ 'lineA\nlineB\nlineC\nlineD'}]
>>> print(json.dumps(data,indent=4)) #doctest: +NORMALIZE_WHITESPACE
[
  "line1\nline2\nline3",
  {
    "key1": "one line",
    "key2": "lineA\nlineB\nlineC\nlineD"
  }
]
>>> print(json.dumps(data | multilinesstring_to_list(),indent=4) )
↪ #doctest: +NORMALIZE_WHITESPACE
[
  [
    "-----< Multiline string as list >-----",
    ↪ "line1",
    "line2",
    "line3"
  ],
  {
    "key1": "one line",
    "key2": [
      "-----< Multiline string as list >-----",
      ↪ "lineA",
      "lineB",
      "lineC",
      "lineD"
    ]
  }
]

```

(continues on next page)



(continued from previous page)

```
}  
]
```



This module defines base classes for python-textops3

## 13.1 activate\_debug

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

This function is useful when playing with python-textops3 through a python console. It is not recommended to use this function in a real application : use standard logging functions instead.

## 13.2 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
```

(continues on next page)

(continued from previous page)

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

## 13.3 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
```

## 13.4 dictmerge

`textops.dictmerge(*dict_args)`

Merge as many dicts you want

Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

**Parameters** `*dict_args` (*dict*) – List of dicts

**Returns** a new merged dict

**Return type** *dict*

## Examples

```
>>> dictmerge({'a':1, 'b':2}, {'b':3, 'c':4})
{'a': 1, 'b': 3, 'c': 4}
```

## 13.5 dformat

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

Formats a dictionary, manages unknown 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()`
- **dct** (*dict*) – the dict to 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} downloads', d)
'textops : 32591 downloads'
>>> dformat('{software} : {count} downloads', d, 'N/A')
'N/A : 32591 downloads'
>>> dformat('{software} : {count} downloads', d, lambda k: 'unknown_tag_{}'.format(k))
'unknown_tag_software : 32591 downloads'
```

## 13.6 eformat

`textops.eformat (format_str, lst, dct, defvalue='-')`

Formats a list and a dictionary, manages unknown keys

It works like `string.Formatter.vformat()` except that it accepts 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()`
- **lst** (*dict*) – the list to format
- **dct** (*dict*) – the dict to format
- **defvalue** (*str or callable*) – the default value to display when the data is not in the dict

## Examples

```
>>> d = {'count': '32591', 'soft': 'textops'}
>>> l = ['Eric', 'Guido']
>>> eformat('{0} => {soft} : {count} dowloads', l, d)
'Eric => textops : 32591 dowloads'
>>> eformat('{2} => {software} : {count} dowloads', l, d, 'N/A')
'N/A => N/A : 32591 dowloads'
>>> eformat('{2} => {software} : {count} dowloads', l, d, lambda k: 'unknown_
↳ tag_%s' % k)
'unknown_tag_2 => unknown_tag_software : 32591 dowloads'
```

## 13.7 vformat

Same syntax as `string.Formatter.vformat()`

## 13.8 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, bytes) 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 for-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: 'dict_items' 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'])
<class 'str'>
>>> d = DictExt(d)
>>> d['mykey']
'myval'
>>> type(d['mykey'])
<class 'textops.base.StrExt'>
```

```
>>> d=DictExt()
>>> d[0]=[]
>>> d
{0: []}
>>> d[0].append(3)
>>> d
{0: [3]}
>>> type(d[0])
<class 'textops.base.ListExt'>
```

**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()
{'soft': 'textops', 'count': '32591'}
```

(continues on next page)

(continued from previous page)

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

**as\_list**

Convert to ListExt object

**render** (*format\_string*, *defvalue*='-')

Render a DictExt as a string

It uses the function *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'
```

## 13.9 ListExt

**class** textops.ListExt

Extend list class to gain access to textops as attributes

In addition, all list items (dict, list, str, bytes) 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

## 13.10 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

## 13.11 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')                                     # doctest:␣
↪+ELLIPSIS
cut(<generator object mygen at ...>, 'l')
>>> cut(mygen(), 'l').g                                     # doctest:␣
↪+ELLIPSIS
<generator object extend_type_gen at ...>
>>> def mygen(): yield None
```

(continues on next page)

(continued from previous page)

```
>>> type(echo(None).g) # doctest:␣
↳+ELLIPSIS
<class '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 # doctest:␣
↳+ELLIPSIS
[]
```

**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)
<class '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)
<class '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
[]
```

**n**

Execute operations, do not convert, do not return anything

If `_process()` returns a generator, it is consumed**Examples**

```
>>> echo('1789').length().n
```

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

This method must be overridden 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': 'val8'}
>>> print(parse_indented(s).pp)
{  'a': 'val1',
```

(continues on next page)

(continued from previous page)

```
'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 joining items with a newline.

### Examples

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

## 13.12 BytesExt

**class** textops.BytesExt

Extend bytes class to gain access to textops as attributes

## Examples

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

```
>>> BytesExt(b'extended bytes').cut()
[b'extended', b'bytes']
```

### **as\_list**

Convert to ListExt object

- genindex
- modindex
- search



## CHAPTER 14

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### t

- `textops.base`, [119](#)
- `textops.ops.cast`, [109](#)
- `textops.ops.fileops`, [73](#)
- `textops.ops.listops`, [27](#)
- `textops.ops.parse`, [89](#)
- `textops.ops.recode`, [115](#)
- `textops.ops.runops`, [83](#)
- `textops.ops.strops`, [15](#)
- `textops.ops.wrapops`, [87](#)



## A

activate\_debug() (in module textops), 119  
add\_textop() (in module textops), 119  
add\_textop\_iter() (in module textops), 120  
after (class in textops), 27  
afteri (class in textops), 28  
aggregate (class in textops), 28  
amend() (textops.DictExt method), 123  
as\_list (textops.BytesExt attribute), 129  
as\_list (textops.DictExt attribute), 124  
as\_list (textops.ListExt attribute), 124  
as\_list (textops.StrExt attribute), 125

## B

before (class in textops), 29  
beforei (class in textops), 30  
between (class in textops), 30  
betweenb (class in textops), 32  
betweenbi (class in textops), 32  
betweeni (class in textops), 33  
BytesExt (class in textops), 128  
bzcat (class in textops), 73

## C

cat (class in textops), 73  
cut (class in textops), 15  
cutca (class in textops), 16  
cutdct (class in textops), 17  
cutkv (class in textops), 18  
cutm (class in textops), 19  
cutmi (class in textops), 19  
cutre (class in textops), 20  
cuts (class in textops), 21  
cutsa (class in textops), 22  
cutsai (class in textops), 23  
cutsi (class in textops), 23

## D

dformat() (in module textops), 121

DictExt (class in textops), 122  
dictmerge() (in module textops), 120  
doformat (class in textops), 34  
doreduce (class in textops), 34  
dorender (class in textops), 35  
doslice (class in textops), 35  
dostrstrip (class in textops), 36

## E

echo (class in textops), 24  
eformat() (in module textops), 121

## F

f (textops.TextOp attribute), 125  
find (class in textops), 75  
find\_first\_pattern (class in textops), 89  
find\_first\_patterni (class in textops), 90  
find\_pattern (class in textops), 90  
find\_patterni (class in textops), 90  
find\_patterns (class in textops), 91  
find\_patternsi (class in textops), 92  
findhighlight (class in textops), 36  
findre (class in textops), 75  
first (class in textops), 38  
formatdicts (class in textops), 38  
formatitems (class in textops), 39  
formatlists (class in textops), 40

## G

g (textops.TextOp attribute), 125  
ge (textops.TextOp attribute), 126  
greaterequal (class in textops), 41  
greaterthan (class in textops), 42  
grep (class in textops), 42  
grepc (class in textops), 43  
grepci (class in textops), 44  
grepcv (class in textops), 45  
grepcvi (class in textops), 45  
gredi (class in textops), 45

grepv (class in textops), 46  
grepvi (class in textops), 46  
gzcat (class in textops), 76

## H

haspattern (class in textops), 47  
haspatterni (class in textops), 47  
head (class in textops), 48

## I

i (textops.TextOp attribute), 126  
iffn (class in textops), 48  
inrange (class in textops), 49

## J

j (textops.TextOp attribute), 126  
je (textops.TextOp attribute), 126

## K

keyval (class in textops), 92  
keyvali (class in textops), 93

## L

l (textops.TextOp attribute), 126  
last (class in textops), 50  
lcount (class in textops), 51  
le (textops.TextOp attribute), 127  
length (class in textops), 24  
less (class in textops), 51  
lessequal (class in textops), 52  
lessthan (class in textops), 52  
linetester (class in textops), 53  
list\_to\_multilinestring (class in textops), 115  
ListExt (class in textops), 124  
ls (class in textops), 76

## M

mapfn (class in textops), 53  
mapif (class in textops), 54  
matches (class in textops), 25  
merge\_dicts (class in textops), 54  
mgrep (class in textops), 94  
mgrepi (class in textops), 95  
mgrepv (class in textops), 95  
mgrepvi (class in textops), 96  
mrun (class in textops), 83  
multilinestring\_to\_list (class in textops), 116

## N

n (textops.TextOp attribute), 127  
norepeat (class in textops), 55

## O

op() (textops.TextOp class method), 127  
outrange (class in textops), 55

## P

parse\_indented (class in textops), 97  
parse\_smart (class in textops), 98  
parseg (class in textops), 99  
parsegi (class in textops), 99  
parsek (class in textops), 100  
parseki (class in textops), 100  
parsekv (class in textops), 101  
parsekvi (class in textops), 102  
pp (textops.TextOp attribute), 127  
pretty (class in textops), 109

## R

r (textops.TextOp attribute), 128  
render() (textops.DictExt method), 124  
renderdicts (class in textops), 56  
renderitems (class in textops), 57  
renderlists (class in textops), 57  
replacefile (class in textops), 76  
resplitblock (class in textops), 58  
resub (class in textops), 87  
rmblank (class in textops), 60  
run (class in textops), 84

## S

s (textops.TextOp attribute), 128  
se (textops.TextOp attribute), 128  
searches (class in textops), 26  
sed (class in textops), 60  
sedi (class in textops), 61  
sgrep (class in textops), 102  
sgrepi (class in textops), 103  
sgrepv (class in textops), 103  
sgrepvi (class in textops), 104  
since (class in textops), 61  
skess (class in textops), 62  
skip (class in textops), 62  
sortdicts (class in textops), 63  
sortlists (class in textops), 63  
span (class in textops), 64  
splitblock (class in textops), 64  
splitln (class in textops), 26  
state\_pattern (class in textops), 104  
stats (class in textops), 77  
StrExt (class in textops), 124  
StrOp (class in textops), 15  
subitem (class in textops), 66  
subitems (class in textops), 66  
subslicing (class in textops), 67

## T

- [tail](#) (class in textops), 67
- [teefile](#) (class in textops), 77
- [TextOp](#) (class in textops), 125
- [textops.base](#) (module), 119
- [textops.ops.cast](#) (module), 109
- [textops.ops.fileops](#) (module), 73
- [textops.ops.listops](#) (module), 27
- [textops.ops.parse](#) (module), 89
- [textops.ops.recode](#) (module), 115
- [textops.ops.runops](#) (module), 83
- [textops.ops.strops](#) (module), 15
- [textops.ops.wrapops](#) (module), 87
- [tobz2file](#) (class in textops), 78
- [todatetime](#) (class in textops), 110
- [todict](#) (class in textops), 110
- [tofile](#) (class in textops), 78
- [tofloat](#) (class in textops), 110
- [togzfile](#) (class in textops), 78
- [toint](#) (class in textops), 111
- [tolist](#) (class in textops), 111
- [toliste](#) (class in textops), 112
- [tonull](#) (class in textops), 112
- [toslug](#) (class in textops), 113
- [tostr](#) (class in textops), 113
- [tostre](#) (class in textops), 114
- [tozipfile](#) (class in textops), 79

## U

- [uniq](#) (class in textops), 68
- [until](#) (class in textops), 69
- [unzip](#) (class in textops), 79
- [unzipre](#) (class in textops), 80

## W

- [wcount](#) (class in textops), 69
- [wcounti](#) (class in textops), 69
- [wcountv](#) (class in textops), 70
- [wcountvi](#) (class in textops), 70
- [WrapOp](#) (class in textops), 87
- [WrapOpIter](#) (class in textops), 87
- [WrapOpStr](#) (class in textops), 87

## X

- [xrun](#) (class in textops), 85

## Z

- [zipcat](#) (class in textops), 80
- [zipcatre](#) (class in textops), 81
- [ziplist](#) (class in textops), 81