listools Documentation

Release 2.2.2

Gilberto Agostinho

Mar 03, 2020

CONTENTS:

CHAPTER

ONE

THE *LISTOOLS* PACKAGE

listools is a Python 3 package of which provides utility functions for dealing with lists in Python 3. *listools* supports Python version 3.5 and newer. You can install it using *pip install listools*.

This package contains four modules: *flatools*, *iterz*, *listutils* and *llogic*. All functions have a *__doc__* attribute with usage instructions.

Documentation is available at https://gilbertohasnofb.github.io/listools-docs/.

A pdf version of the documentation is also available in the *docs* directory.

Bugs can be reported to https://github.com/gilbertohasnofb/listools/issues.

This library is published under the MIT License.

CHAPTER

TWO

FLATOOLS MODULE

The module *flatools* contains functions that deal with flatten lists.

All functions have a *__doc__* attribute with usage instructions.

This library is published under the MIT License.

`listools.flatools.`**`flatten`**(*input_list*)

Completely flattens a list containing any number of nested sublists into a one dimensional list. It is equivalent to flatools.pflatten() with an infinitely large depth. Usage:

```
>>> alist = [[1, 2], [3, 4], [5], [6, 7, 8], [9, 10]]
>>> flatools.flatten(alist)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> alist = [1, 2, [3, [4, 5]]]
>>> flatools.flatten(alist)
[1, 2, 3, 4, 5]
```

Notice that the list themselves can be made out of any datatypes:

```
>>> alist = [1, [2.2, True], ['foo', [(1, 4), None]], [3+2j, {'a': 1}]]
>>> flatools.flatten(alist)
[1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
```

`listools.flatools.`**`flatten_index`**(*input_list*)

Returns the index of the first instance of an element in a flatten list. Usage:

```
>>> alist = [[1, 2], [3, 4], [5, 6]]
>>> flatools.flatten_index(3, alist)
2
```

The datatypes of the elements of the list do not matter:

```
>>> alist = [1, [2.2, True], ['foo', [(1, 4), None]], [3+2j, {'a': 1}]]
>>> flatools.flatten_index(None, alist)
5
```

Just like the default behaviour of the flatten_index() method of a list, flatten_index raises a ValueError if an element is not found in a list:

```
>>> alist = [[1, 2], [3, 4], [5, 6]]
>>> flatools.flatten_index(7, alist)
ValueError: 7 is not in list
```

listools.flatools.**flatten_join**(**input_lists*)

Completely flattens and concatenates an arbitrary number of input lists containing any number of nested sublists. Usage:

```
>>> alist = [[1, 2], [3, 4]]
>>> blist = [[5, 6], [7, 8]]
>>> flatools.flatten_join(alist, blist)
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> alist = [1, [2, [3]]]
>>> blist = [[[4], 5], 6]
>>> flatools.flatten_join(alist, blist)
[1, 2, 3, 4, 5, 6]
```

```
>>> alist = [[1, 2], [3, 4]]
>>> blist = [[5, 6], [7, 8]]
>>> clist = [[[9], 10], 11]
>>> flatools.flatten_join(alist, blist, clist)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Notice that the list themselves can be made out of any datatypes:

```
>>> alist = [1, [2.2, True]]
>>> blist = ['foo', [(1, 4), None]]
>>> clist = [3+2j, {'a': 1}]
>>> flatools.flatten_join(alist, blist, clist)
[1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
```

listools.flatools.**flatten_len**(*input_list*)

Returns the length of a flatten list (that is, it counts all elements in all of its subslists). Usage:

```
>>> alist = [[1, 2], [3, 4], [5, 6]]
>>> flatools.flatten_len(alist)
6
```

The datatypes of the elements of the list do not matter:

```
>>> alist = [1, [2.2, True], ['foo', [(1, 4), None]], [3+2j, {'a': 1}]]
>>> flatools.flatten_len(alist)
8
```

listools.flatools.**flatten_max**(*input_list*, *[, *key*, *default*])

Finds the largest element of a flattened list containing any number of nested subslists. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_max(alist)
10
```

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> flatools.flatten_max(alist)
5
```

The list can also be made out of floats:

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_max(alist)
9.41
```

Or it can be made out of a mixture of integers and floats:

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_max(alist)
7.8
```

There are two optional arguments that can be used. The first is a called 'key' and takes a function that serves as a key for the sort comparison.

```
>>> alist = [-1, -5, [3, [-2, 4]]]
>>> print(flatten_max(alist))
4
>>> print(flatten_max(alist, key=abs))
-5
```

The second is 'default', which is the value that the function defaults to when the input is an empty list:

```
>>> alist = [1, 2, 3]
>>> blist = []
>>> print(flatten_max(alist, default=-100))
3
>>> print(flatten_max(blist, default=-100))
-100
```

listools.flatools.**flatten_min**(*input_list*, *[, *key*, *default*])

Finds the smallest element of a flattened list containing any number of nested sublists. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_min(alist)
1
```

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> flatools.flatten_min(alist)
1
```

The list can also be made out of floats:

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_min(alist)
-3.14
```

Or it can be made out of a mixture of integers and floats:

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_min(alist)
-3.1
```

There are two optional arguments that can be used. The first is a called 'key' and takes a function that serves as a key for the sort comparison.

```
>>> alist = [-1, -5, [3, [-2, 4]]]
>>> print(flatten_min(alist))
-5
>>> print(flatten_min(alist, key=abs))
-1
```

The second is 'default', which is the value that the function defaults to when the input is an empty list:

```
>>> alist = [1, 2, 3]
>>> blist = []
>>> print(flatten_min(alist, default=-100))
1
>>> print(flatten_min(blist, default=-100))
-100
```

listools.flatools.**flatten_mixed_type**(*input_list*)

Returns False if all elements of the flattened input_list are of the same type and True if they are not. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_mixed_type(alist)
False
```

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> flatools.flatten_mixed_type(alist)
False
```

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_mixed_type(alist)
False
```

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_mixed_type(alist)
True
```

```
>>> alist = ['foo', ['bar', ('foo', 'bar')]]
>>> flatools.flatten_mixed_type(alist)
True
```

Note that empty lists return False:

```
>>> alist = []
>>> flatools.flatten_mixed_type(alist)
False
```

listools.flatools.**flatten_reverse**(*input_list*)

Completely flattens a list containing any number of nested subslists into a flatten_reversely sorted one dimensional list. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_reverse(alist)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> alist = [1, 5, [3, [2, 4]]]
>>> flatools.flatten_reverse(alist)
[5, 4, 3, 2, 1]
```

The list can also be made out of floats:

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_reverse(alist)
[9.41, 5.56, 1.73, -1.03, -3.14]
```

Or it can be made out of a mixture of integers and floats:

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_reverse(alist)
[7.8, 6.6, 5, 3, 1.4, -3.1]
```

listools.flatools.**flatten_single_type**(*input_list*)

Returns True if all elements of the flattened input_list are of the same type and False if they are not. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_single_type(alist)
True
```

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> flatools.flatten_single_type(alist)
True
```

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_single_type(alist)
True
```

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_single_type(alist)
False
```

```
>>> alist = ['foo', ['bar', ('foo', 'bar')]]
>>> flatools.flatten_single_type(alist)
False
```

Note that empty lists return False:

```
>>> alist = []
>>> flatools.flatten_single_type(alist)
False
```

listools.flatools.**flatten_sorted**(*input_list*, *[, *key*, *reverse*])

Completely flattens a list containing any number of nested subslists into a flatten_sorted one dimensional list. Usage:

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> flatools.flatten_sorted(alist)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> alist = [1, 5, [3, [2, 4]]]
>>> flatools.flatten_sorted(alist)
[1, 2, 3, 4, 5]
```

The list can also be made out of floats:

```
>>> alist = [[1.73, -3.14, 9.41], [5.56, -1.03]]
>>> flatools.flatten_sorted(alist)
[-3.14, -1.03, 1.73, 5.56, 9.41]
```

Or it can be made out of a mixture of integers and floats:

```
>>> alist = [[3, 1.4], [5, 7.8], [-3.1, 6.6]]
>>> flatools.flatten_sorted(alist)
[-3.1, 1.4, 3, 5, 6.6, 7.8]
```

There are two optional arguments that can be used. The first is a called 'key' and takes a function that serves as a key for the sort comparison.

```
>>> alist = [-1, -5, [3, [-2, 4]]]
>>> print(flatten_sorted(alist))
[-5, -2, -1, 3, 4]
>>> print(flatten_sorted(alist, key=abs))
[-1, -2, 3, 4, -5]
```

The second is 'reverse', which reverses the order of the output list:

```
>>> alist = [1, 5, [3, [2, 4]]]
>>> print(flatten_sorted(alist, reverse=True))
[5, 4, 3, 2, 1]
```

listools.flatools.**flatten_sum**(*input_list*[, *start*])

Sums all values of the list, including any nested subslists. Usage:

```
>>> alist = [[1, 2], [3, 4], [5, 6]]
>>> flatools.flatten_sum(alist)
21
```

```
>>> alist = [1, [2, [3]]]
>>> flatools.flatten_sum(alist)
6
```

The list can also be made out of floats:

```
>>> alist = [1.1, [2.2, [3.3]]]
>>> flatools.flatten_sum(alist)
6.6
```

Or it can contain a mix of integers and floats:

```
>>> alist = [1, [2.1, [3, [4.1]]]]
>>> flatools.flatten_sum(alist)
10.2
```

It can also take an optional argument named 'start' which defines the starting value of the sum.

```
>>> alist = [1, [2, [3]]]
>>> flatools.flatten_sum(alist, start=4)
10
```

listools.flatools.**flatten_zip_cycle**(**input_lists*)

This function is nearly identical to iterz.zip_cycle except that it also flattens all lists before zipping and cycling them. Usage:

```
>>> alist = [1, 2]
>>> blist = [4, [5, 6, 7], 8]
>>> for i, j in flatools.flatten_zip_cycle(alist, blist):
...     print(i, j)
1 4
2 5
1 6
2 7
1 8
```

It also works with multiple lists:

```
>>> a = [1, 2]
>>> b = [1, [2, 3]]
>>> c = [[[1], 2, 3], 4]
>>> d = [1, [2, [3, 4]], 5]
>>> for i, j, k, l in flatools.flatten_zip_cycle(a, b, c, d):
...     print(i, j, k, l)
1 1 1 1
2 2 2 2
1 3 3 3
2 1 4 4
1 2 1 5
```

It also works with lists containing any datatypes:

```
>>> alist = [1, 2.0, 'foo', True, None]
>>> blist = [False, 'bar', (1, 4)]
>>> for i, j in flatools.flatten_zip_cycle(alist, blist):
...     print(i, j)
1 False
2.0 bar
foo (1, 4)
True False
None bar
```

Note that unlike flatools.flatten_zip_cycle(), this function accepts only lists as input due to its flatenning function.

listools.flatools.**pflatten**(*input_list*[, *depth*])

Partially flattens a list containing subslists as elements. Usage:

```
>>> alist = [[1, 2], [3, 4], [5], [6, 7, 8], [9, 10]]
>>> flatools.pflatten(alist)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> alist = [1, 2, [3, [[4], 5]]]
>>> flatools.pflatten(alist)
[1, 2, 3, [[4], 5]]
```

Use the depth argument (which should always be an integer) when wanting to flatten nested sublists:

```
>>> alist = [1, 2, [3, [4, 5]]]
>>> flatools.pflatten(alist, depth=2)
[1, 2, 3, [4], 5]
```

```
>>> alist = [1, 2, [3, [4, 5]]]
>>> flatools.pflatten(alist, depth=3)
[1, 2, 3, 4, 5]
```

```
>>> alist = [1, 2, [3, [4, 5]]]
>>> flatools.pflatten(alist, depth=4)
[1, 2, 3, 4, 5]
```

Notice that the list themselves can be made out of any datatypes:

```
>>> alist = [1, [2.2, True], ['foo', [(1, 4), None]], [3+2j, {'a': 1}]]
>>> flatools.flatten(alist, depth=3)
[1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
```

CHAPTER

THREE

ITERZ MODULE

The module *iterz* contains functions that manipualte lists as iterators.

All functions have a *__doc__* attribute with usage instructions.

This library is published under the MIT License.

listools.iterz.**cycle_until_index**(*input_iter*, *i*)

This will cycle an iterator up to a certain index (inclusive). Usage:

```
>>> alist = [1, 2, 4, 8, 16, 32]
>>> for item in iterz.cycle_until_index(atuple, 4):
...     print(item)
1
2
4
8
16
```

In fact, it works with any iterable containing any datatypes:

```
>>> atuple = (1, 'foo', 3.0, 3+2j, [0.0])
>>> for item in iterz.cycle_until_index(atuple, 3):
...     print(item)
1
foo
3.0
3+2j
```

listools.iterz.**inf_cycle**(*input_iter*)

This will cycle an iterator indefinitely. Usage:

```
>>> alist = [1, 2, 4, 8]
>>> inf_cycle_iter = iterz.inf_cycle(alist)
>>> for _ in range(9):
...     print(inf_cycle_iter.__next__())
1
2
4
8
1
2
4
8
1
```

In fact, it works with any iterable containing any datatypes:

```
>>> atuple = (1, 'foo', 3.0)
>>> inf_cycle_iter = iterz.inf_cycle(atuple)
>>> for i in range(5):
...     print(inf_cycle_iter.__next__())
1
foo
3.0
1
foo
```

listools.iterz.**iter_mask**(*input_iter*, *mask*)

This function takes an input iterator and applies a mask to it, yielding values according to the mastk. The mask should be a list containing 1's and 0's, or alternatively True's and False's. Usage:

```
>>> alist = [1, 2, 3]
>>> mask = [True, False, True]
>>> for item in iterz.iter_mask(alist, mask):
...     print(item)
1
3
```

```
>>> alist = [1, 2, 3, 4, 5]
>>> mask = [1, 0, 0, 1, 0]
>>> for item in iterz.iter_mask(alist, mask):
...     print(item)
1
4
```

If the mask is shorter than the input iterator, it loops:

```
>>> alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> mask = [1, 0]
>>> for item in iterz.iter_mask(alist, mask):
...     print(item)
1
3
5
7
9
```

The input iterator can contain any datatypes:

```
>>> alist = [1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
>>> mask = [True, False, True]
>>> for item in iterz.iter_mask(alist, mask):
...     print(item)
1
True
'foo'
None
3+2j
```

listools.iterz.**ncycle**(*input_iter*)

This will cycle an iterator a certain number of times. Usage:

```
>>> alist = [1, 2, 4, 8]
>>> for item in iterz.ncycle(alist, 2)
```

(continues on next page)

(continued from previous page)

```
...     print(item)
1
2
4
8
1
2
4
8
```

In fact, it works with any iterable containing any datatypes:

```
>>> atuple = (1, 'foo', 3.0)
>>> for item in iterz.ncycle(atuple, 3):
...     print(item)
1
foo
3.0
1
foo
3.0
1
foo
3.0
```

listools.iterz.**zip_cycle**(**input_iters*)

Similar to zip but cycles smaller lists or iterables until the longest one is output. Usage:

```
>>> alist = [1, 2]
>>> blist = [4, 5, 6, 7, 8]
>>> for i, j in iterz.zip_cycle(alist, blist):
...     print(i, j)
1 4
2 5
1 6
2 7
1 8
```

It also works with multiple lists:

```
>>> alist = [1, 2]
>>> blist = [1, 2, 3]
>>> clist = [1, 2, 3, 4]
>>> dlist = [1, 2, 3, 4, 5]
>>> for i, j, k, l in iterz.zip_cycle(alist, blist, clist, dlist):
...     print(i, j, k, l)
1 1 1 1
2 2 2 2
1 3 3 3
2 1 4 4
1 2 1 5
```

In fact, it works with any iterable containing any datatypes:

```
>>> a = (1, 2, 3)
>>> b = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

(continues on next page)

(continued from previous page)

```
>>> c = 'abcde'
>>> for i, j, k in iterz.zip_cycle(a, b, c):
...     print(i, j, k)
1 1.0 a
2 2.0 b
3 3.0 c
1 4.0 d
2 5.0 e
3 6.0 a
1 7.0 b
```

listools.iterz.**zip_inf_cycle**(**input_iters*)

Similar to zip but cycles all lists indefinitely. Usage:

```
>>> alist = [1, 2]
>>> blist = [4, 5, 6, 7, 8]
>>> zip_inf_cycle_iter = iterz.zip_inf_cycle(alist, blist)
>>> for _ in range(9):
...     print(zip_inf_cycle_iter.__next__())
1 4
2 5
1 6
2 7
1 8
2 4
1 5
2 6
1 7
```

It also works with multiple lists:

```
>>> alist = [1, 2]
>>> blist = [1, 2, 3]
>>> clist = [1, 2, 3, 4]
>>> dlist = [1, 2, 3, 4, 5]
>>> zip_inf_cycle_iter = iterz.zip_inf_cycle(alist, blist, clist, dlist)
>>> for i in range(7):
...     print(zip_inf_cycle_iter.__next__())
1 1 1 1
2 2 2 2
1 3 3 3
2 1 4 4
1 2 1 5
1 3 2 1
2 1 3 2
```

In fact, it works with any iterable containing any datatypes:

```
>>> a = (1, 2, 3)
>>> b = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> c = 'abcde'
>>> zip_inf_cycle_iter = iterz.zip_inf_cycle(a, b, c)
>>> for i in range(10):
...     print(zip_inf_cycle_iter.__next__())
1 1.0 a
2 2.0 b
```

(continues on next page)

(continued from previous page)

```
3 3.0 c
1 4.0 d
2 5.0 e
3 6.0 a
1 7.0 b
2 1.0 c
3 2.0 d
1 3.0 e
```

listools.iterz.**zip_longest**(**input_iters*[, *default*])

Similar to zip_cycle but yields values until the longest of the input iterators is exhausted. Shorter iterators yields None when exhausted. Usage:

```
>>> alist = [1, 2]
>>> blist = [4, 5, 6, 7, 8]
>>> for i, j in iterz.zip_longest(alist, blist):
...     print(i, j)
1 4
2 5
None 6
None 7
None 8
```

It also works with multiple lists:

```
>>> alist = [1, 2]
>>> blist = [1, 2, 3]
>>> clist = [1, 2, 3, 4]
>>> dlist = [1, 2, 3, 4, 5]
>>> for i, j, k, l in iterz.zip_longest(alist, blist, clist, dlist):
...     print(i, j, k, l)
1 1 1 1
2 2 2 2
None 3 3 3
None None 4 4
None None None 5
```

In fact, it works with any iterable containing any datatypes:

```
>>> a = (1, 2, 3)
>>> b = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> c = 'abcde'
>>> for i, j, k in iterz.zip_longest(a, b, c):
...     print(i, j, k)
1 1.0 a
2 2.0 b
3 3.0 c
None 4.0 d
None 5.0 e
None 6.0 None
None 7.0 None
```

The value None can be changed using the keyword argument default:

```
>>> alist = [1, 2]
>>> blist = [1, 2, 3, 4]
```

(continues on next page)

(continued from previous page)

```
>>> clist = [1, 2, 3, 4, 5, 6]
>>> for i, j, k, l in iterz.zip_longest(alist, blist, clist, default=0):
...     print(i, j, k, l)
1 1 1
2 2 2
0 3 3
0 4 4
0 0 5
0 0 6
```

listools.iterz.**zip_syzygy**(**input_iters*)

Similar to zip but cycles lists until all of them are exhausted at the same time (that is, when the next output tuple would be the same as the very first yielded one). Usage:

```
>>> alist = [1, 2]
>>> blist = [4, 5, 6, 7, 8]
>>> for i, j in iterz.zip_syzygy(alist, blist):
...     print(i, j)
1 4
2 5
1 6
2 7
1 8
2 4
1 5
2 6
1 7
2 8
```

It also works with multiple lists:

```
>>> alist = [1, 2]
>>> blist = [1, 2, 3]
>>> clist = [1, 2, 3, 4]
>>> dlist = [4, 5, 6]
>>> for i, j, k, l in iterz.zip_syzygy(alist, blist, clist, dlist):
...     print(i, j, k, l)
1 1 1 4
2 2 2 5
1 3 3 6
2 1 4 4
1 2 1 5
2 3 2 6
1 1 3 4
2 2 4 5
1 3 1 6
2 1 2 4
1 2 3 5
2 3 4 6
```

In fact, it works with any iterable containing any datatypes:

```
>>> a = (1, 2)
>>> b = [1.0, 2.0, 3.0]
>>> c = 'abc'
>>> for i, j, k in iterz.zip_syzygy(a, b, c):
```

(continues on next page)

(continued from previous page)

```
...     print(i, j, k)
1 1.0 a
2 2.0 b
1 3.0 c
2 1.0 a
1 2.0 b
2 3.0 c
```

CHAPTER

FOUR

LISTUTILS MODULE

The module *listutils* contains functions that apply simple mathematical operations to lists.

All functions have a *__doc__* attribute with usage instructions.

This library is published under the MIT License.

listools.listutils.**list_gcd**(*input_list*)

This function returns the greatest common divisor of a list of integers. Usage:

```
>>> alist = [8, 12]
>>> listutils.list_gcd(alist)
4
```

```
>>> alist = [74, 259, 185, 333]
>>> listutils.list_gcd(alist)
37
```

listools.listutils.**list_lcm**(*input_list*)

This function returns the least common multiple of a list of integers. Usage:

```
>>> alist = [1, 2, 3]
>>> listutils.list_lcm(alist)
6
```

```
>>> alist = [7, 8, 4, 3]
>>> listutils.list_lcm(alist)
168
```

listools.listutils.**list_mask**(*input_list*, *mask*)

This function takes an input list and applies a mask to it, outputting a new list. The mask should be a list containing 1's and 0's, or alternatively True's and False's. If the mask is shorter than the input list then the input list will be considered only up to the mask length. Usage:

```
>>> alist = [1, 2, 3]
>>> mask = [True, False, True]
>>> listutils.list_mask(alist, mask)
[1, 3]
```

```
>>> alist = [1, 2, 3, 4, 5]
>>> mask = [1, 0, 0, 1, 0]
>>> listutils.list_mask(alist, mask)
[1, 4]
```

If the mask is shorter than the list, the :

```
>>> alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> mask = [1, 0]
>>> listutils.list_mask(alist, mask)
[1]
```

If the mask is shorter than the input list then the input list will be considered only up to the mask length:

```
>>> alist = [1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
>>> mask = [True, False, True]
>>> listutils.list_mask(alist, mask)
[1, True]
```

The input list can be empty, in which case an empty list is return. On the other hand, the mask argument cannot be an empty list.

listools.listutils.**list_mask_cycle**(*input_list*, *mask*)

This function takes an input list and applies a mask to it, outputting a new list. The mask should be a list containing 1's and 0's, or alternatively True's and False's. If the mask is shorter than the list, the mask is cycled. Usage:

```
>>> alist = [1, 2, 3]
>>> mask = [True, False, True]
>>> listutils.list_mask_cycle(alist, mask)
[1, 3]
```

```
>>> alist = [1, 2, 3, 4, 5]
>>> mask = [1, 0, 0, 1, 0]
>>> listutils.list_mask_cycle(alist, mask)
[1, 4]
```

If the mask is shorter than the list, it loops:

```
>>> alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> mask = [1, 0]
>>> listutils.list_mask_cycle(alist, mask)
[1, 3, 5, 7, 9]
```

The input list can contain any datatypes:

```
>>> alist = [1, 2.2, True, 'foo', (1, 4), None, 3+2j, {'a': 1}]
>>> mask = [True, False, True]
>>> listutils.list_mask_cycle(alist, mask)
[1, True, 'foo', None, 3+2j]
```

The input list can be empty, in which case an empty list is return. On the other hand, the mask argument cannot be an empty list.

listools.listutils.**period_len**(*input_list*[, *ignore_partial_cycles*])

This function returns the length of the period of an input list. Usage:

```
>>> alist = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> listutils.period_len(alist)
3
```

If a list is not periodic, the period length equals to the list size:

```
>>> alist = [3, 1, 4, 1, 5, 9, 2, 6]
>>> listutils.period_len(alist)
8
```

This function detects periodicity in lists with partial cycles:

```
>>> alist = [1, 2, 3, 1, 2, 3, 1]
>>> listutils.period_len(alist)
3
```

To disable this behaviour, use the ignore_partial_cycles argument:

```
>>> alist = [1, 2, 3, 1, 2, 3, 1]
>>> listutils.period_len(alist, ignore_partial_cycles=True)
7
```

If a list does not contain partial cycles, the ignore_partial_cycles argument does not affect the result:

```
>>> alist = [1, 2, 3, 1, 2, 3]
>>> listutils.period_len(alist, ignore_partial_cycles=True)
3
```

listools.listutils.**shuffle**(*input_list*)

This function returns a shuffled list with the same elements as the input list. Usage:

```
>>> alist = [0, 1, 2, 3, 4]
>>> listutils.shuffle(alist)
[2, 1, 4, 0, 3]
```

It differs from random.shuffle() since listutils.shuffle() outputs a new list, preserving the input one:

```
>>> alist = [0, 1, 2, 3, 4]
>>> listutils.shuffle(alist)
[2, 1, 4, 0, 3]
>>> alist
[0, 1, 2, 3, 4]
>>> import random
>>> random.shuffle(alist)
>>> alist
[3, 2, 1, 4, 0]
```

CHAPTER

FIVE

LLOGIC MODULE

The module *llogic* contains functions that deal with logic operations on lists.

All functions have a *__doc__* attribute with usage instructions.

This library is published under the MIT License.

`listools.llogic.`**`difference`**(*list_1*, *list_2*)

Returns the difference of two lists (omitting repetitions). The order of the elements of the output depends on their order in the lists. The order of the inputs lists does affect the result. Usage:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = [7, 6, 5, 4, 3]
>>> llogic.difference(alist, blist)
[1, 2]
>>> llogic.difference(blist, alist)
[7, 6]
```

```
>>> alist = [1, 2, 3, 3, 4, 4, 5, 5, 5]
>>> blist = [3, 3, 4, 5, 5, 6]
>>> llogic.difference(alist, blist)
[1, 2]
```

Note that llogic.difference does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, 1, 5, 2]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.difference(alist, blist)
[]
>>> alist = [3, 4, [1, [5, 2]]]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.difference(alist, blist)
[[1, [5, 2]]]
```

The lists can contain any datatype:

```
>>> alist = [1, 2.3, 'foo', (3, 7)]
>>> blist = ['foo', 7+3j, (3, 7)]
>>> llogic.difference(alist, blist)
[1, 2.3]
```

`listools.llogic.`**`intersection`**(*list_1*, *list_2*)

Returns the intersection of two lists (omitting repetitions). The order of the elements of the output depends on the order they are found in the first list. Usage:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = [7, 6, 5, 4, 3]
>>> llogic.intersection(alist, blist)
[3, 4, 5]
```

```
>>> alist = [1, 2, 3, 3, 4, 4, 5, 5, 5]
>>> blist = [3, 3, 4, 5, 5, 6]
>>> llogic.intersection(alist, blist)
[3, 4, 5]
```

Note that llogic.intersection does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.intersection(alist, blist)
[3, 4]
```

The lists can contain any datatype:

```
>>> alist = [1, 2.3, 'foo', (3, 7)]
>>> blist = ['foo', 7+3j, (3, 7)]
>>> llogic.intersection(alist, blist)
['foo', (3, 7)]
```

If either list is empty then the result is an empty list:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = []
>>> llogic.intersection(alist, blist)
[]
```

listools.llogic.**is_ascending**(*input_list*[, *step*])

This function returns True if the input list is ascending with a fixed step, otherwise it returns False. Usage:

```
>>> alist = [0, 1, 2, 3]
>>> llogic.is_ascending(alist)
True
```

The initial value can be other than zero:

```
>>> alist = [10, 11, 12]
>>> llogic.is_ascending(alist)
True
```

The list can also have negative elements:

```
>>> alist = [-2, -1, 0, 1, 2]
>>> llogic.is_ascending(alist)
True
```

It will return False if the list is not ascending:

```
>>> alist = [6, 5, 9, 2]
>>> llogic.is_ascending(alist)
False
```

By default, the function uses steps of size 1 so the list below is not considered as ascending:

```
>>> alist = [1, 3, 5, 7]
>>> llogic.is_ascending(alist)
False
```

But the user can set the step argument to any value greater than one:

```
>>> alist = [1, 3, 5, 7]
>>> step = 2
>>> llogic.is_ascending(alist, step)
True
```

listools.llogic.**is_contained**(*list_1*, *list_2*)

Returns True if all unique elements of list_1 are also present in list_2 and returns False when that's not the case. Usage:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = [2, 3, 4]
>>> llogic.is_contained(blist, alist)
True
>>> llogic.is_contained(alist, blist)
False
```

```
>>> alist = [1, 2, 3, 3, 4, 4, 5, 5, 5, 6, 7]
>>> blist = [3, 3, 4, 5, 5, 6]
>>> llogic.is_contained(blist, alist)
True
```

Lists are not flattened, so sublists are considered as a single element:

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> blist = [1, 2]
>>> llogic.is_contained(blist, alist)
False
```

The lists can contain any datatype:

```
>>> alist = [1, 2.3, 'foo', (3, 7), 7+3j]
>>> blist = ['foo', 7+3j, (3, 7)]
>>> llogic.is_contained(blist, alist)
True
```

listools.llogic.**is_descending**(*input_list*[, *step*])

This function returns True if the input list is descending with a fixed step, otherwise it returns False. Usage:

```
>>> alist = [3, 2, 1, 0]
>>> llogic.is_descending(alist)
True
```

The final value can be other than zero:

```
>>> alist = [12, 11, 10]
>>> llogic.is_descending(alist)
True
```

The list can also have negative elements:

```
>>> alist = [2, 1, 0, -1, -2]
>>> llogic.is_descending(alist)
True
```

It will return False if the list is not ascending:

```
>>> alist = [6, 5, 9, 2]
>>> llogic.is_descending(alist)
False
```

By default, the function uses steps of size 1 so the list below is not considered as ascending:

```
>>> alist = [7, 5, 3, 1]
>>> llogic.is_descending(alist)
False
```

But the user can set the step argument to any value less than one:

```
>>> alist = [7, 5, 3, 1]
>>> step = -2
>>> llogic.is_descending(alist, step)
True
```

listools.llogic.**mixed_type**(*input_list*)

Returns False if all elements of an input_list are of the same type and True if they are not. Usage:

```
>>> alist = [3, 4, 1, 5, 2]
>>> llogic.single_type(alist)
False
```

```
>>> alist = [3.1, 4, 1, 5, '2']
>>> llogic.single_type(alist)
True
```

Note that llogic.mixed_type does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> llogic.single_type(alist)
True
```

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> llogic.single_type(alist)
False
```

Also note that empty lists return False:

```
>>> alist = []
>>> llogic.single_type(alist)
False
```

listools.llogic.**single_type**(*input_list*)

Returns True if all elements of an input_list are of the same type and False if they are not. Usage:

```
>>> alist = [3, 4, 1, 5, 2]
>>> llogic.single_type(alist)
True
```

```
>>> alist = [3.1, 4, 1, 5, '2']
>>> llogic.single_type(alist)
False
```

Note that llogic.single_type does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> llogic.single_type(alist)
False
```

```
>>> alist = [[1, 4], [5, 7], [2], [9, 6, 10], [8, 3]]
>>> llogic.single_type(alist)
True
```

Also note that empty lists return False:

```
>>> alist = []
>>> llogic.single_type(alist)
False
```

listools.llogic.**symmetric_difference**(*list_1*, *list_2*)

Returns the symmetric difference of two lists (omitting repetitions). The order of the elements of the output depends on their order in the lists. The order of the inputs lists does affect the result. Usage:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = [7, 6, 5, 4, 3]
>>> llogic.symmetric_difference(alist, blist)
[1, 2, 7, 6]
>>> llogic.symmetric_difference(blist, alist)
[7, 6, 1, 2]
```

```
>>> alist = [1, 2, 3, 3, 4, 4, 5, 5, 5]
>>> blist = [3, 3, 4, 5, 5, 6]
>>> llogic.symmetric_difference(alist, blist)
[1, 2, 6]
```

Note that llogic.symmetric_difference does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, 1, 5, 2]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.symmetric_difference(alist, blist)
[]
>>> alist = [3, 4, [1, [5, 2]]]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.symmetric_difference(alist, blist)
[[1, [5, 2]], 1, 2, 5]
```

The lists can contain any datatype:

```
>>> alist = [1, 2.3, 'foo', (3, 7)]
>>> blist = ['foo', 7+3j, (3, 7)]
>>> llogic.symmetric_difference(alist, blist)
[1, 2.3, 7+3j]
```

listools.llogic.**union**(*list_1*, *list_2*)

Returns the union of two lists (omitting repetitions). The order of the elements of the output depends on the order they are found in the first and then in the second lists. Usage:

```
>>> alist = [1, 2, 3, 4, 5]
>>> blist = [7, 6, 5, 4, 3]
>>> llogic.union(alist, blist)
[1, 2, 3, 4, 5, 7, 6]
```

```
>>> alist = [1, 2, 3, 3, 4, 4, 5, 5, 5]
>>> blist = [7, 6, 6, 5, 5, 5, 4]
>>> llogic.union(alist, blist)
[1, 2, 3, 4, 5, 7, 6]
```

Note that llogic.union does not flatten the lists so nested lists are of type list:

```
>>> alist = [3, 4, [1, [5, 2]]]
>>> blist = [1, 2, 3, 4, 5]
>>> llogic.union(alist, blist)
[3, 4, [1, [5, 2]], 1, 2, 5]
```

The lists can contain any datatype:

```
>>> alist = [1, 2.3, 'foo', (3, 7)]
>>> blist = ['foo', 7+3j, (3, 7)]
>>> llogic.union(alist, blist)
[1, 2.3, 'foo', (3, 7), 7+3j]
```

PYTHON MODULE INDEX

l

INDEX