natsort Documentation

Release 5.1.0

Seth M. Morton

Aug 20, 2017

Contents

Contents:

CHAPTER 1

The `natsort` module

Simple yet flexible natural sorting in Python.

- Source Code: https://github.com/SethMMorton/natsort
- Downloads: https://pypi.org/project/natsort/
- Documentation: http://natsort.readthedocs.io/
  - *Examples and Recipes*
  - *How Does Natsort Work?*
  - *API*
- Optional Dependencies:
  - fastnumbers >= 0.7.1
  - PyICU >= 1.0.0

`natsort` is a general utility for sorting lists *naturally*; the definition of "naturally" is not well-defined, but the most common definition is that numbers contained within the string should be sorted as numbers and not as you would other characters. If you need to present sorted output to a user, you probably want to sort it naturally.

`natsort` was initially created for sorting scientific output filenames that contained signed floating point numbers in the names. There was a lack of algorithms out there that could perform a natural sort on *floats* but plenty for *ints*; check out this StackOverflow question and its answers and links therein, this ActiveState forum, and of course this great article on natural sorting from CodingHorror.com for examples of what I mean. `natsort` was created to fill in this gap, but has since expanded to handle just about any definition of a number, as well as other sorting customizations.

Quick Description

When you try to sort a list of strings that contain numbers, the normal python sort algorithm sorts lexicographically, so you might not get the results that you expect:

```
>>> a = ['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> sorted(a)
['1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '2 ft 7 in', '7 ft 6 in']
```

Notice that it has the order ('1', '10', '2') - this is because the list is being sorted in lexicographical order, which sorts numbers like you would letters (i.e. 'b', 'ba', 'c').

natsort provides a function *natsorted()* that helps sort lists "naturally" ("naturally" is rather ill-defined, but in general it means sorting based on meaning and not computer code point).. Using *natsorted()* is simple:

```
>>> from natsort import natsorted
>>> a = ['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> natsorted(a)
['1 ft 5 in', '2 ft 7 in', '2 ft 11 in', '7 ft 6 in', '10 ft 2 in']
```

natsorted() identifies numbers anywhere in a string and sorts them naturally. Below are some other things you can do with *natsort* (please see the *Examples and Recipes* for a quick start guide, or the *natsort API* for more details).

Note: *natsorted()* is designed to be a drop-in replacement for the built-in `sorted()` function. Like `sorted()`, *natsorted() does not sort in-place*. To sort a list and assign the output to the same variable, you must explicitly assign the output to a variable:

```
>>> a = ['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> natsorted(a)
['1 ft 5 in', '2 ft 7 in', '2 ft 11 in', '7 ft 6 in', '10 ft 2 in']
>>> print(a)  # 'a' was not sorted; "natsorted" simply returned a sorted list
['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> a = natsorted(a)  # Now 'a' will be sorted because the sorted list was assigned␣
↪to 'a'
>>> print(a)
['1 ft 5 in', '2 ft 7 in', '2 ft 11 in', '7 ft 6 in', '10 ft 2 in']
```

Please see *Generating a Reusable Sorting Key and Sorting In-Place* for an alternate way to sort in-place naturally.

Sorting Versions

This is handled properly by default (as of *natsort* version >= 4.0.0):

```
>>> a = ['version-1.9', 'version-2.0', 'version-1.11', 'version-1.10']
>>> natsorted(a)
['version-1.9', 'version-1.10', 'version-1.11', 'version-2.0']
```

If you need to sort release candidates, please see *Sorting with Alpha, Beta, and Release Candidates* for a useful hack.

Sorting by Real Numbers (i.e. Signed Floats)

This is useful in scientific data analysis and was the default behavior of *natsorted()* for *natsort* version < 4.0.0. Use the *realsorted()* function:

```
>>> from natsort import realsorted, ns
>>> # Note that when interpreting as signed floats, the below numbers are
>>> #            +5.10,                -3.00,            +5.30,              +2.00
```

```
>>> a = ['position5.10.data', 'position-3.data', 'position5.3.data', 'position2.data']
>>> natsorted(a)
['position2.data', 'position5.3.data', 'position5.10.data', 'position-3.data']
>>> natsorted(a, alg=ns.REAL)
['position-3.data', 'position2.data', 'position5.10.data', 'position5.3.data']
>>> realsorted(a)  # shortcut for natsorted with alg=ns.REAL
['position-3.data', 'position2.data', 'position5.10.data', 'position5.3.data']
```

Locale-Aware Sorting (or "Human Sorting")

This is where the non-numeric characters are ordered based on their meaning, not on their ordinal value, and a locale-dependent thousands separator and decimal separator is accounted for in the number. This can be achieved with the *humansorted()* function:

```
>>> a = ['Apple', 'apple15', 'Banana', 'apple14,689', 'banana']
>>> natsorted(a)
['Apple', 'Banana', 'apple14,689', 'apple15', 'banana']
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
>>> natsorted(a, alg=ns.LOCALE)
['apple15', 'apple14,689', 'Apple', 'banana', 'Banana']
>>> from natsort import humansorted
>>> humansorted(a)
['apple15', 'apple14,689', 'Apple', 'banana', 'Banana']
```

You may find you need to explicitly set the locale to get this to work (as shown in the example). Please see *Possible Issues with humansorted() or ns.LOCALE* and the Installation section below before using the *humansorted()* function.

Further Customizing Natsort

If you need to combine multiple algorithm modifiers (such as `ns.REAL`, `ns.LOCALE`, and `ns.IGNORECASE`), you can combine the options using the bitwise OR operator (`|`). For example,

```
>>> a = ['Apple', 'apple15', 'Banana', 'apple14,689', 'banana']
>>> natsorted(a, alg=ns.REAL | ns.LOCALE | ns.IGNORECASE)
['Apple', 'apple15', 'apple14,689', 'Banana', 'banana']
>>> # The ns enum provides long and short forms for each option.
>>> ns.LOCALE == ns.L
True
>>> # You can also customize the convenience functions, too.
>>> natsorted(a, alg=ns.REAL | ns.LOCALE | ns.IGNORECASE) == realsorted(a, alg=ns.L |␣
↪ns.IC)
True
>>> natsorted(a, alg=ns.REAL | ns.LOCALE | ns.IGNORECASE) == humansorted(a, alg=ns.R␣
↪| ns.IC)
True
```

All of the available customizations can be found in the documentation for the `ns` enum *ns*.

Sorting Mixed Types

You can mix and match `int`, `float`, and `str` (or `unicode`) types when you sort:

```
>>> a = ['4.5', 6, 2.0, '5', 'a']
>>> natsorted(a)
[2.0, '4.5', '5', 6, 'a']
>>> # On Python 2, sorted(a) would return [2.0, 6, '4.5', '5', 'a']
>>> # On Python 3, sorted(a) would raise an "unorderable types" TypeError
```

Handling Bytes on Python 3

natsort does not officially support the *bytes* type on Python 3, but convenience functions are provided that help you decode to *str* first:

```
>>> from natsort import as_utf8
>>> a = [b'a', 14.0, 'b']
>>> # On Python 2, natsorted(a) would would work as expected.
>>> # On Python 3, natsorted(a) would raise a TypeError (bytes() < str())
>>> natsorted(a, key=as_utf8) == [14.0, b'a', 'b']
True
>>> a = [b'a56', b'a5', b'a6', b'a40']
>>> # On Python 2, natsorted(a) would would work as expected.
>>> # On Python 3, natsorted(a) would return the same results as sorted(a)
>>> natsorted(a, key=as_utf8) == [b'a5', b'a6', b'a40', b'a56']
True
```

Generating a Reusable Sorting Key and Sorting In-Place

Under the hood, *natsorted()* works by generating a custom sorting key using *natsort_keygen()* and then passes that to the built-in `sorted()`. You can use the *natsort_keygen()* function yourself to generate a custom sorting key to sort in-place using the `list.sort()` method.

```
>>> from natsort import natsort_keygen
>>> natsort_key = natsort_keygen()
>>> a = ['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> natsorted(a) == sorted(a, key=natsort_key)
True
>>> a.sort(key=natsort_key)
>>> a
['1 ft 5 in', '2 ft 7 in', '2 ft 11 in', '7 ft 6 in', '10 ft 2 in']
```

All of the algorithm customizations mentioned in the *Further Customizing Natsort* section can also be applied to *natsort_keygen()* through the *alg* keyword option.

Other Useful Things

- recursively descend into lists of lists
- controlling the case-sensitivity (see *Controlling Case When Sorting*)
- sorting file paths correctly (see *Sort OS-Generated Paths*)
- allow custom sorting keys (see *Using a Custom Sorting Key*)

Installation

Installation of *natsort* is ultra-easy. Simply execute from the command line:

```
pip install natsort
```

You can also download the source from https://pypi.org/project/natsort/, or browse the git repository at https://github.com/SethMMorton/natsort.

If you choose to install from source, you can unzip the source archive and enter the directory, and type:

```
python setup.py install
```

If you wish to run the unit tests, enter:

```
python setup.py test
```

If you want to build this documentation, enter:

```
python setup.py build_sphinx
```

natsort requires Python version 2.6 or greater or Python 3.2 or greater.

The most efficient sorting can occur if you install the fastnumbers package (it helps with the string to number conversions.) `natsort` will still run (efficiently) without the package, but if you need to squeeze out that extra juice it is recommended you include this as a dependency. `natsort` will not require (or check) that fastnumbers is installed.

It is recommended that you install PyICU if you wisht to sort in a locale-dependent manner, see *Possible Issues with humansorted() or ns.LOCALE* for an explanation why.

natsort comes with a shell script called *natsort*, or can also be called from the command line with `python -m natsort`. The command line script is only installed onto your `PATH` if you don't install via a wheel. There is apparently a known bug with the wheel installation process that will not create entry points.

CHAPTER 2

How Does Natsort Work?

`natsort` works by breaking strings into smaller sub-components (numbers or everything else), and returning these components in a tuple. Sorting tuples in Python is well-defined, and this fact is used to sort the input strings properly. But how does one break a string into sub-components? And what does one do to those components once they are split? Below I will explain the algorithm that was chosen for the `natsort` module, and some of the thinking that

went into those design decisions. I will also mention some of the stumbling blocks I ran into because getting sorting right is surprisingly hard.

If you are impatient, you can skip to *TL;DR 1 - The Simple "No Special Cases" Algorithm* for the algorithm in the simplest case, and *TL;DR 2 - Handling Crappy, Real-World Input* to see what extra code is needed to handle special cases.

First, How Does Natural Sorting Work At a High Level?

If I want to compare '2 ft 7 in' to '2 ft 11 in', I might do the following

```
>>> '2 ft 7 in' < '2 ft 11 in'
False
```

We as humans know that the above should be true, but why does Python think it is false? Here is how it is performing the comparison:

```
'2' <=> '2' ==> equal, so keep going
' ' <=> ' ' ==> equal, so keep going
'f' <=> 'f' ==> equal, so keep going
't' <=> 't' ==> equal, so keep going
' ' <=> ' ' ==> equal, so keep going
'7' <=> '1' ==> different, use result of '7' < '1'
```

'7' evaluates as greater than '1' so the statement is false. When sorting, if a value is less than another it is placed first, so in our above example '2 ft 11 in' would end up before '2 ft 7 in', which is not correct. What to do?

The best way to handle this is to break the string into sub-components of numbers and non-numbers, and then convert the numeric parts into `float()` or `int()` types. This will force Python to actually understand the context of what it is sorting and then "do the right thing." Luckily, it handles sorting lists of strings right out-of-the-box, so the only hard part is actually making this string-to-list transformation and then Python will handle the rest.

```
'2 ft 7 in'  ==> (2, ' ft ', 7,  ' in')
'2 ft 11 in' ==> (2, ' ft ', 11, ' in')
```

When Python compares the two, it roughly follows the below logic:

```
2       <=> 2       ==> equal, so keep going
' ft '  <=> ' ft ' ==> a string is a special type of sequence - evaluate each␣
↪character individually
                    ||
                    -->
                       ' ' <=> ' ' ==> equal, so keep going
                       'f' <=> 'f' ==> equal, so keep going
                       't' <=> 't' ==> equal, so keep going
                       ' ' <=> ' ' ==> equal, so keep going
                   <== Back to parent sequence
7 <=> 11 ==> different, use the result of 7 < 11
```

Clearly, seven is less than eleven, so our comparison is as we expect, and we would get the sorting order we wanted.

At its heart, `natsort` is simply a tool to break strings into tuples, turning numbers in strings (i.e. `'79'`) into *ints* and *floats* as it does this.

Natsort's Approach

Decomposing Strings Into Sub-Components

The first major hurtle to overcome is to decompose the string into sub-components. Remarkably, this turns out to be the easy part, owing mostly to Python's easy access to regular expressions. Breaking an arbitrary string based on a pattern is pretty straightforward.

```
>>> import re
>>> re.split(r'(\d+)', '2 ft 11 in')
['', '2', ' ft ', '11', ' in']
```

Clear (assuming you can read regular expressions) and concise.

The reason I began developing *natsort* in the first place was because I needed to handle the natural sorting of strings containing *real numbers*, not just unsigned integers as the above example contains. By real numbers, I mean those like `-45.4920E-23`. *natsort* can handle just about any number definition; to that end, here are all the regular expressions used in *natsort*:

```
>>> unsigned_int               = r'([0-9]+)'
>>> signed_int                 = r'([-+]?[0-9]+)'
>>> unsigned_float             = r'((?:[0-9]+\.?[0-9]*|\.[0-9]+)(?:[eE][-+]?[0-9]+)?)'
>>> signed_float               = r'([-+]?(?:[0-9]+\.?[0-9]*|\.[0-9]+)(?:[eE][-+]?[0-
↪9]+)?)'
>>> unsigned_float_no_exponent = r'((?:[0-9]+\.?[0-9]*|\.[0-9]+))'
>>> signed_float_no_exponent   = r'([-+]?(?:[0-9]+\.?[0-9]*|\.[0-9]+))'
```

Note that `"inf"` and `"nan"` are deliberately omitted from the float definition because you wouldn't want (for example) `"banana"` to be converted into `['ba', 'nan', 'a']`, Let's see an example:

```
>>> re.split(signed_float, 'The mass of 3 electrons is 2.732815068E-30 kg')
['The mass of ', '3', ' electrons is ', '2.732815068E-30', ' kg']
```

Note: It is a bit of a lie to say the above are the complete regular expressions. In the actual code there is also handling for non-ASCII unicode characters (such as), but I will ignore that aspect of *natsort* in this discussion.

Now, when the user wants to change the definition of a number, it is as easy as changing the pattern supplied to the regular expression engine.

Choosing the right default is hard, though (well, in this case it shouldn't have been but I was rather thick-headed). In retrospect, it should have been obvious that since essentially all the code examples I had/have seen for natural sorting were for *unsigned integers*, I should have made the default definition of a number an *unsigned integer*. But, in the brash days of my youth I assumed that since my use case was real numbers, everyone else would be happier sorting by real numbers; so, I made the default definition of a number a *signed float with exponent*. This astonished a lot of people (and some people aren't very nice when they are astonished). Starting with *natsort* version 4.0.0 the default

number definition was changed to an *unsigned integer* which satisfies the "least astonishment" principle, and I have not heard a complaint since.

Wouldn't itertools.groupby work as well as regex to split strings?

You *could* do it using something like `itertools.groupby()`, but it is not clearer nor more concise, *I promise*.

```
>>> import itertools
>>> import operator
>>> list(map(''.join, map(operator.itemgetter(1), itertools.groupby('2 ft 11 in', str.
↪isdigit))))
['2', ' ft ', '11', ' in']
```

OK, but let's assume for a moment that you *really* like itertools and think the above is fine. We still have lost a lot of flexibility here because of the `str.isdigit()` call which makes this method non-optimal; with a regular expression one can change the pattern string and split on much more complicated patterns, but with `itertools.groupby()` it becomes *much* more complicated to change it up; I implemented this strategy as part of my testing and it is anything but clear an concise.

Not to mention it's *way* slower than regex. Just the simple example above (unsigned integers) is 50% slower than regex...

Coercing Strings Containing Numbers Into Numbers

There has been some debate on Stack Overflow as to what method is best to coerce a string to a number if it can be coerced, and leaving it alone otherwise (see this one for coercion and this one for checking for some high traffic questions), but it mostly boils down to two different solutions, shown here:

```
>>> def coerce_try_except(x):
...     try:
...         return int(x)
...     except ValueError:
...         return x
...
>>> def coerce_regex(x):
...     # Note that precompiling the regex is more performant,
...     # but I do not show that here for clarity's sake.
...     return int(x) if re.match(r'[-+]?\d+$', x) else x
...
```

Here are some timing results run on my machine:

```
In [0]: numbers = list(map(str, range(100)))  # A list of numbers as strings

In [1]: not_numbers = ['banana' + x for x in numbers]

In [2]: %timeit [coerce_try_except(x) for x in numbers]
10000 loops, best of 3: 51.1 µs per loop

In [3]: %timeit [coerce_try_except(x) for x in not_numbers]
1000 loops, best of 3: 289 µs per loop

In [4]: %timeit [coerce_regex(x) for x in not_numbers]
10000 loops, best of 3: 67.6 µs per loop
```

```
In [5]: %timeit [coerce_regex(x) for x in numbers]
10000 loops, best of 3: 123 µs per loop
```

What can we learn from this? The `try: except` method (arguably the most "pythonic" of the solutions) is best for numeric input, but performs over 5X slower for non-numeric input. Conversely, the regular expression method, though slower than `try: except` for both input types, is more efficient for non-numeric input than for input that can be converted to an `int`. Further, even though the regular expression method is slower for both input types, it is always at least twice as fast as the worst case for the `try: except`.

Why do I care? Shouldn't I just pick a method and not worry about it? Probably. However, I am very conscious about the performance of *natsort*, and want it to be a true drop-in replacement for `sorted()` without having to incur a performance penalty. For the purposes of *natsort*, there is no clear winner between the two algorithms - the data being passed to this function will likely be a mix of numeric and non-numeric string content. Do I use the `try: except` method and hope the speed gains on numbers will offset the non-number performance, or do I use regular expressions and take the more stable performance?

It turns out that within the context of *natsort*, some assumptions can be made that make a hybrid approach attractive. Because all strings are pre-split into numeric and non-numeric content *before* being passed to this coercion function, the assumption can be made that *if a string begins with a digit or a sign, it can be coerced into a number*.

```
>>> def coerce_to_int(x):
...     if x[0] in '0123456789+-':
...         try:
...             return int(x)
...         except ValueError:
...             return x
...     else:
...         return x
...
```

So how does this perform compared to the standard coercion methods?

```
In [6]: %timeit [coerce_to_int(x) for x in numbers]
10000 loops, best of 3: 71.6 µs per loop

In [7]: %timeit [coerce_to_int(x) for x in not_numbers]
10000 loops, best of 3: 26.4 µs per loop
```

The hybrid method eliminates most of the time wasted on numbers checking that it is in fact a number before passing to `int()`, and eliminates the time wasted in the exception stack for input that is not a number.

That's as fast as we can get, right? In pure Python, probably. At least, it's close. But because I am crazy and a glutton for punishment, I decided to see if I could get any faster writing a C extension. It's called fastnumbers and contains a C implementation of the above coercion functions called `fast_int()`. How does it fair? Pretty well.

```
In [8]: %timeit [fast_int(x) for x in numbers]
10000 loops, best of 3: 30.9 µs per loop

In [9]: %timeit [fast_int(x) for x in not_numbers]
10000 loops, best of 3: 30 µs per loop
```

During development of *natsort*, I wanted to ensure that using it did not get in the way of a user's program by introducing a performance penalty to their code. To that end, I do not feel like my adventures down the rabbit hole of optimization of coercion functions was a waste; I can confidently look users in the eye and say I considered every option in ensuring *natsort* is as efficient as possible. This is why if fastnumbers is installed it will be used for this step, and otherwise the hybrid method will be used.

Note: Modifying the hybrid coercion function for floats is straightforward.

```
>>> def coerce_to_float(x):
...     if x[0] in '.0123456789+-' or x.lower().lstrip()[:3] in ('nan', 'inf'):
...         try:
...             return float(x)
...         except ValueError:
...             return x
...     else:
...         return x
...
```

TL;DR 1 - The Simple "No Special Cases" Algorithm

At this point, our *natsort* algorithm is essentially the following:

```
>>> import re
>>> def natsort_key(x, as_float=False, signed=False):
...     if as_float:
...         regex = signed_float if signed else unsigned_float
...     else:
...         regex = signed_int if signed else unsigned_int
...     split_input = re.split(regex, x)
...     split_input = filter(None, split_input)  # removes null strings
...     coerce = coerce_to_float if as_float else coerce_to_int
...     return tuple(coerce(s) for s in split_input)
...
```

I have written the above for clarity and not performance. This pretty much matches most natural sort solutions for python on Stack Overflow (except the above includes customization of the definition of a number).

Special Cases Everywhere!

If what I described in *TL;DR 1* were all that `natsort` needed to do then there probably wouldn't be much need for a third-party module, right? Probably. But it turns out that in real-world data there are a lot of special cases that need to be handled, and in true 80%/20% fashion, the majority of the code in `natsort` is devoted to handling special cases like those described below.

Sorting Filesystem Paths

The first major special case I encountered was sorting filesystem paths (if you go to the link, you will see I didn't handle it well for a year... this was before I fully realized how much functionality I could really add to `natsort`). Let's apply the `natsort_key()` from above to some filesystem paths that you might see being auto-generated from your operating system:

```
>>> paths = ['/p/Folder (10)/file.tar.gz',
...          '/p/Folder/file.tar.gz',
...          '/p/Folder (1)/file (1).tar.gz',
...          '/p/Folder (1)/file.tar.gz']
>>> sorted(paths, key=natsort_key)
['/p/Folder (1)/file (1).tar.gz', '/p/Folder (1)/file.tar.gz', '/p/Folder (10)/file.
↪tar.gz', '/p/Folder/file.tar.gz']
```

Well that's not right! What is `'/p/Folder/file.tar.gz'` doing at the end? It has to do with the numerical ASCII code assigned to the space and `/` characters in the ASCII table. According to the ASCII table, the space character (number 32) comes before the `/` character (number 47). If we remove the common prefix in all of the above

strings (`'/p/Folder'`), we can see why this happens:

```
>>> ' (1)/file.tar.gz' < '/file.tar.gz'
True
>>> ' ' < '/'
True
```

This isn't very convenient... how do we solve it? We can split the path across the path separators and then sort. A convenient way do to this is with the Path.parts method from `pathlib`:

```
>>> import pathlib
>>> sorted(paths, key=lambda x: tuple(natsort_key(s) for s in pathlib.Path(x).parts))
['/p/Folder/file.tar.gz', '/p/Folder (1)/file (1).tar.gz', '/p/Folder (1)/file.tar.gz
↪', '/p/Folder (10)/file.tar.gz']
```

Almost! It seems like there is some funny business going on in the final filename component as well. We can solve that nicely and quickly with Path.suffixes and Path.stem.

```
>>> def decompose_path_into_components(x):
...     path_split = list(pathlib.Path(x).parts)
...     # Remove the final filename component from the path.
...     final_component = pathlib.Path(path_split.pop())
...     # Split off all the extensions.
...     suffixes = final_component.suffixes
...     stem = final_component.name.replace(''.join(suffixes), '')
...     # Remove the '.' prefix of each extension, and make that
...     # final component a list of the stem and each suffix.
...     final_component = [stem] + [x[1:] for x in suffixes]
...     # Replace the split final filename component.
...     path_split.extend(final_component)
...     return path_split
...
>>> def natsort_key_with_path_support(x):
...     return tuple(natsort_key(s) for s in decompose_path_into_components(x))
...
>>> sorted(paths, key=natsort_key_with_path_support)
['/p/Folder/file.tar.gz', '/p/Folder (1)/file.tar.gz', '/p/Folder (1)/file (1).tar.gz
↪', '/p/Folder (10)/file.tar.gz']
```

This works because in addition to breaking the input by path separators, the final filename component is separated from its extensions as well[1]. *Then*, each of these separated components is sent to the *`natsort`* algorithm, so the result is a tuple of tuples. Once that is done, we can see how comparisons can be done in the expected manner.

```
>>> a = natsort_key_with_path_support('/p/Folder (1)/file (1).tar.gz')
>>> a
(('/',), ('p',), ('Folder (', 1, ')'), ('file (', 1, ')'), ('tar',), ('gz',))
>>>
>>> b = natsort_key_with_path_support('/p/Folder/file.tar.gz')
>>> b
(('/',), ('p',), ('Folder',), ('file',), ('tar',), ('gz',))
>>>
>>> a > b
True
```

[1] To anyone looking through the actual code, you will note that I don't actually use `pathlib` to split the paths... I wrote my own version to avoid adding an external dependency of `pathlib` on Python < 3.4.

Comparing Different Types on Python 3

The second major special case I encountered was sorting of different types. If you are on Python 2 (i.e. legacy Python), this mostly doesn't matter *too* much since it uses an arbitrary heuristic to allow traditionally un-comparable types to be compared (such as comparing `'a'` to `1`). However, on Python 3 (i.e. Python) it simply won't let you perform such nonsense, raising a `TypeError` instead.

You can imagine that a module that breaks strings into tuples of numbers and strings is walking a dangerous line if it does not have special handling for comparing numbers and strings. My imagination was not so great at first. Let's take a look at all the ways this can fail with real-world data.

```
>>> def natsort_key_with_poor_real_number_support(x):
...     split_input = re.split(signed_float, x)
...     split_input = filter(None, split_input)  # removes null strings
...     return tuple(coerce_to_float(s) for s in split_input)
>>>
>>> sorted([5, '4'], key=natsort_key_with_poor_real_number_support)
Traceback (most recent call last):
    ...
TypeError: ...
>>>
>>> sorted(['12 apples', 'apples'], key=natsort_key_with_poor_real_number_support)
Traceback (most recent call last):
    ...
TypeError: ...
>>>
>>> sorted(['version5.3.0', 'version5.3rc1'], key=natsort_key_with_poor_real_number_
↪support)
Traceback (most recent call last):
    ...
TypeError: ...
```

Let's break these down.

1. The integer `5` is sent to `re.split` which expects only strings or bytes, which is a no-no.
2. `natsort_key_with_poor_real_number_support('12 apples') < natsort_key_with_poor_real_number_support('apples')` is the same as `(12.0, 'apples') < ('apples',)`, and thus a number gets compared to a string[2] which also is a no-no.
3. This one scores big on the astonishment scale, especially if one accidentally uses signed integers or real numbers when they mean to use unsigned integers. `natsort_key_with_poor_real_number_support('version5.3.0') < natsort_key_with_poor_real_number_support('version5.3rc1')` is the same as `('version', 5.3, 0.0) < ('version', 5.3, 'rc', 1.0)`, so in the third element a number gets compared to a string, once again the same old no-no. (The same would happen with `'version5-3'` and `'version5-a'`, which would be come `('version', 5, -3)` and `('version', 5, '-a')`).

As you might expect, the solution to the first issue is to wrap the `re.split` call in a `try: except:` block and handle the number specially if a `TypeError` is raised. The second and third cases *could* be handled in a "special case" manner, meaning only respond and do something different if these problems are detected. But a less error-prone method is to ensure that the data is correct-by-construction, and this can be done by ensuring that the returned tuples *always* start with a string, and then alternate in a string-number-string-number-string patter;n this can be achieved by adding an empty string wherever the pattern is not followed[3]. This ends up working out pretty nicely because empty

[2] *"But if you hadn't removed the leading empty string from re.split this wouldn't have happened!!"* I can hear you saying. Well, that's true. I don't have a *great* reason for having done that except that in an earlier non-optimal incarnation of the algorithm I needed to it, and it kind of stuck, and it made other parts of the code easier if the assumption that there were no empty strings was valid.

[3] I'm not going to show how this is implemented in this document, but if you are interested you can look at the code to `_sep_inserter()` in util.py.

strings are always "less" than any non-empty string, and we typically want numbers to come before strings.

Let's take a look at how this works out.

```
>>> from natsort.utils import _sep_inserter
>>> list(_sep_inserter(iter(['apples']), ''))
['apples']
>>>
>>> list(_sep_inserter(iter([12, ' apples']), ''))
['', 12, ' apples']
>>>
>>> list(_sep_inserter(iter(['version', 5, -3]), ''))
['version', 5, '', -3]
>>>
>>> from natsort import natsort_keygen, ns
>>> natsort_key_with_good_real_number_support = natsort_keygen(alg=ns.REAL)
>>>
>>> sorted([5, '4'], key=natsort_key_with_good_real_number_support)
['4', 5]
>>>
>>> sorted(['12 apples', 'apples'], key=natsort_key_with_good_real_number_support)
['12 apples', 'apples']
>>>
>>> sorted(['version5.3.0', 'version5.3rc1'], key=natsort_key_with_good_real_number_
↪support)
['version5.3.0', 'version5.3rc1']
```

How the "good" version works will be given in *TL;DR 2 - Handling Crappy, Real-World Input*.

Handling NaN

A rather unexpected special case I encountered was sorting collections containing NaN. Let's see what happens when you try to sort a plain old list of numbers when there is a **NaN** floating around in there.

```
>>> danger = [7, float('nan'), 22.7, 19, -14, 59.123, 4]
>>> sorted(danger)
[7, nan, -14, 4, 19, 22.7, 59.123]
```

Clearly that isn't correct, and for once it isn't my fault! It's hard to compare floating point numbers. By definition, **NaN** is unorderable to any other number, and is never equal to any other number, including itself.

```
>>> nan = float('nan')
>>> 5 > nan
False
>>> 5 < nan
False
>>> 5 == nan
False
>>> 5 != nan
True
>>> nan == nan
False
>>> nan != nan
True
```

The implication of all this for us is that if there is an **NaN** in the data-set we are trying to sort, the data-set will end up being sorted in two separate yet individually sorted sequences - the one *before* the **NaN**, and the one *after*. This is because the < operation that is used to sort always returns `False` with **NaN**.

Because *natsort* aims to sort sequences in a way that does not surprise the user, keeping this behavior is not acceptable (I don't require my users to know how **NaN** will behave in a sorting algorithm). The simplest way to satisfy the "least astonishment" principle is to substitute **NaN** with some other value. But what value is *least* astonishing? I chose to replace **NaN** with $-\infty$ so that these poorly behaved elements always end up at the front where the users will most likely be alerted to their presence.

```
>>> def fix_nan(x):
...     if x != x:  # only true for NaN
...         return float('-inf')
...     else:
...         return x
...
```

Let's check out *TL;DR 2* to see how this can be incorporated into the simple key function from *TL;DR 1*.

TL;DR 2 - Handling Crappy, Real-World Input

Let's see how our elegant key function from *TL;DR 1* has become bastardized in order to support handling mixed real-world data and user customizations.

```
>>> def natsort_key(x, as_float=False, signed=False, as_path=False):
...     if as_float:
...         regex = signed_float if signed else unsigned_float
...     else:
...         regex = signed_int if signed else unsigned_int
...     try:
...         if as_path:
...             x = decompose_path_into_components(x)  # Decomposes into list of␣
↪strings
...         # If this raises a TypeError, input is not a string.
...         split_input = re.split(regex, x)
...     except TypeError:
...         try:
...             # Does this need to be applied recursively (list-of-list)?
...             return tuple(map(natsort_key, x))
...         except TypeError:
...             # Must be a number
...             ret = ('', fix_nan(x))  # Maintain string-number-string pattern
...             return (ret,) if as_path else ret  # as_path returns tuple-of-tuples
...     else:
...         split_input = filter(None, split_input)  # removes null strings
...         # Note that the coerce_to_int/coerce_to_float functions
...         # are also modified to use the fix_nan function.
...         if as_float:
...             coerced_input = (coerce_to_float(s) for s in split_input)
...         else:
...             coerced_input = (coerce_to_int(s) for s in split_input)
...         return tuple(_sep_inserter(coerced_input, ''))
...
```

And this doesn't even show handling `bytes` type! Notice that we have to do non-obvious things like modify the return form of numbers when `as_path` is given, just to avoid comparing strings and numbers for the case in which a user provides input like `['/home/me', 42]`.

Let's take it out for a spin!

```
>>> danger = [7, float('nan'), 22.7, '19', '-14', '59.123', 4]
>>> sorted(danger, key=lambda x: natsort_key(x, as_float=True, signed=True))
[nan, '-14', 4, 7, '19', 22.7, '59.123']
>>>
>>> paths = ['/p/Folder (1)/file.tar.gz',
...          '/p/Folder/file.tar.gz',
...          123456]
>>> sorted(paths, key=lambda x: natsort_key(x, as_path=True))
[123456, '/p/Folder/file.tar.gz', '/p/Folder (1)/file.tar.gz']
```

Here Be Dragons: Adding Locale Support

Probably the most challenging special case I had to handle was getting *natsort* to handle sorting the non-numerical parts of input correctly, and also allowing it to sort the numerical bits in different locales. This was in no way what I originally set out to do with this library, so I was caught a bit off guard when the request was initially made. I discovered the `locale` library, and assumed that if it's part of Python's StdLib there can't be too many dragons, right?

INCOMPLETE LIST OF DRAGONS

- https://github.com/SethMMorton/natsort/issues/21
- https://github.com/SethMMorton/natsort/issues/22
- https://github.com/SethMMorton/natsort/issues/23
- https://github.com/SethMMorton/natsort/issues/36
- https://github.com/SethMMorton/natsort/issues/44
- https://bugs.python.org/issue2481
- https://bugs.python.org/issue23195
- https://stackoverflow.com/questions/3412933/python-not-sorting-unicode-properly-strcoll-doesnt-help
- https://stackoverflow.com/questions/22203550/sort-dictionary-by-key-using-locale-collation
- https://stackoverflow.com/questions/33459384/unicode-character-not-in-range-when-calling-locale-strxfrm
- https://stackoverflow.com/questions/36431810/sort-numeric-lines-with-thousand-separators
- https://stackoverflow.com/questions/45734562/how-can-i-get-a-reasonable-string-sorting-with-python

---

These can be summed up as follows:

1. `locale` is a thin wrapper over your operating system's *locale* library, so if *that* is broken (like it is on BSD and OSX) then `locale` is broken in Python.

2. Because of a bug in legacy Python (i.e. Python 2), there is no uniform way to use the `locale` sorting functionality between legacy Python and Python 3.
3. People have differing opinions of how capitalization should affect word order.
4. There is no built-in way to handle locale-dependent thousands separators and decimal points *robustly*.
5. Proper handling of Unicode is complicated.
6. Proper handling of `locale` is complicated.

Easily over half of the the code in *natsort* is in some way dealing with some aspect of `locale` or basic case handling. It would have been impossible to get right without a really good testing strategy.

Don't expect any more TL;DR's... if you want to see how all this is fully incorporated into the *natsort* algorithm then please take a look at the code. However, I will hint at how specific steps are taken in each section.

Let's see how we can handle some of the dragons, one-by-one.

Basic Case Control Support

Without even thinking about the mess that is adding `locale` support, *natsort* can introduce support for controlling how case is interpreted.

First, let's take a look at how it is sorted by default (due to where characters lie on the ASCII table).

```
>>> a = ['Apple', 'corn', 'Corn', 'Banana', 'apple', 'banana']
>>> sorted(a)
['Apple', 'Banana', 'Corn', 'apple', 'banana', 'corn']
```

All uppercase letters come before lowercase letters in the ASCII table, so all capitalized words appear first. Not everyone agrees that this is the correct order. Some believe that the capitalized words should be last (`['apple', 'banana', 'corn', 'Apple', 'Banana', 'Corn']`). Some believe that both the lowercase and uppercase versions should appear together (`['Apple', 'apple', 'Banana', 'banana', 'Corn', 'corn']`). Some believe that both should be true . Some people don't care at all[4].

Solving the first case (I call it *LOWERCASEFIRST*) is actually pretty easy... just call the `str.swapcase()` method on the input.

```
>>> sorted(a, key=lambda x: x.swapcase())
['apple', 'banana', 'corn', 'Apple', 'Banana', 'Corn']
```

The last (i call it *IGNORECASE*) should be super easy, right? Simply call `str.lowercase()` on the input. This will work but may not always give the correct answer on non-latin character sets. It's a good thing that in Python 3.3 `str.casefold()` was introduced, which does a better job of removing all case information from unicode characters in non-latin alphabets.

```
>>> def remove_case(x):
...     try:
...         return x.casefold()
...     except AttributeError:  # Legacy Python backwards compatibility
...         return x.lowercase()
...
>>> sorted(a, key=remove_case)
['Apple', 'apple', 'Banana', 'banana', 'corn', 'Corn']
```

[4] Handling each of these is straightforward, but coupled with the rapidly fracturing execution paths presented in *TL;DR 2* one can imagine this will get out of hand quickly. If you take a look at natsort.py and util.py you can observe that to avoid this I take a more functional approach to construting the *natsort* algorithm as opposed to the procedural approach illustrated in *TL;DR 1* and *TL;DR 2*.

The middle case (I call it *GROUPLETTERS*) is less straightforward. The most efficient way to handle this is to duplicate each character with its lowercase version and then the original character.

```
>>> import itertools
>>> def groupletters(x):
...     return ''.join(itertools.chain.from_iterable((remove_case(y), y) for y in x))
...
>>> groupletters('Apple')
'aAppppllee'
>>> groupletters('apple')
'aappppllee'
>>> sorted(a, key=groupletters)
['Apple', 'apple', 'Banana', 'banana', 'Corn', 'corn']
```

The effect of this is that both `'Apple'` and `'apple'` are placed adjacent to each other because their transformations both begin with `'a'`, and then the second character can be used to order them appropriately with respect to each other.

There's a problem with this, though. Within the context of *natsort* we are trying to correctly sort numbers and those should be left alone.

```
>>> a = ['Apple5', 'apple', 'Apple4E10', 'Banana']
>>> sorted(a, key=lambda x: natsort_key(x, as_float=True))
['Apple5', 'Apple4E10', 'Banana', 'apple']
>>> sorted(a, key=lambda x: natsort_key(groupletters(x), as_float=True))
['Apple4E10', 'Apple5', 'apple', 'Banana']
>>> groupletters('Apple4E10')
'aAppppllee44eE1100'
```

We messed up the numbers! Looks like `groupletters()` needs to be applied *after* the strings are broken into their components. I'm not going to show how this is done here, but basically it requires applying the function in the `else:` block of `coerce_to_int()`/`coerce_to_float()`.

```
>>> better_groupletters = natsort_keygen(alg=ns.GROUPLETTERS | ns.REAL)
>>> better_groupletters('Apple4E10')
('aAppppllee', 40000000000.0)
>>> sorted(a, key=better_groupletters)
['Apple5', 'Apple4E10', 'apple', 'Banana']
```

Of course, applying both *LOWERCASEFIRST* and *GROUPLETTERS* is just a matter of turning on both functions.

Basic Unicode Support

Unicode is hard and complicated. Here's an example.

```
>>> b = [b'\x66', b'\x65', b'\xc3\xa9', b'\x65\xcc\x81', b'\x61', b'\x7a']
>>> a = [x.decode('utf8') for x in b]
>>> a
['f', 'e', 'é', 'é', 'a', 'z']
>>> sorted(a)
['a', 'e', 'é', 'f', 'z', 'é']
```

There are more than one way to represent the character 'é' in Unicode. In fact, many characters have multiple representations. This is a challenge because comparing the two representations would return `False` even though they *look* the same.

```
>>> a[2] == a[3]
False
```

Alas, since characters are compared based on the numerical value of their representation, sorting Unicode often gives unexpected results (like seeing 'é' come both *before* and *after* 'z').

The original approach that *natsort* took with respect to non-ASCII Unicode characters was to say "just use the `locale` or `PyICU` library" and then cross it's fingers and hope those libraries take care of it. As you will find in the following sections, that comes with its own baggage, and turned out to not always work anyway (see https://stackoverflow.com/q/45734562/1399279). A more robust approach is to handle the Unicode out-of-the-box without invoking a heavy-handed library like `locale` or `PyICU`. To do this, we must use *normalization*.

To fully understand Unicode normalization, check out some official Unicode documentation. Just kidding... that's too much text. The following StackOverflow answers do a good job at explaining Unicode normalization in simple terms: https://stackoverflow.com/a/7934397/1399279 and https://stackoverflow.com/a/7931547/1399279. Put simply, normalization ensures that Unicode characters with multiple representations are in some canonical and consistent representation so that (for example) comparisons of the characters can be performed in a sane way. The following discussion assumes you at least read the StackOverflow answers.

Looking back at our 'é' example, we can see that the two versions were constructed with the byte strings `b'\xc3\xa9'` and `b'\x65\xcc\x81'`. The former representation is actually LATIN SMALL LETTER E WITH ACUTE and is a single character in the Unicode standard. This is known as the *compressed form* and corresponds to the 'NFC' normalization scheme. The latter representation is actually the letter 'e' followed by COMBINING ACUTE ACCENT and so is two characters in the Unicode standard. This is known as the *decompressed form* and corresponds to the 'NFD' normalization scheme. Since the first character in the decompressed form is actually the letter 'e', when compared to other ASCII characters it fits where you might expect. Unfortunately, all Unicode compressed form characters come after the ASCII characters and so they always will be placed after 'z' when sorting.

It seems that most Unicode data is stored and shared in the compressed form which makes it challenging to sort. This can be solved by normalizing all incoming Unicode data to the decompressed form ('NFD') and *then* sorting.

```
>>> import unicodedata
>>> c = [unicodedata.normalize('NFD', x) for x in a]
>>> c
['f', 'e', 'é', 'é', 'a', 'z']
>>> sorted(c)
['a', 'e', 'é', 'é', 'f', 'z']
```

Huzzah! Sane sorting without having to resort to `locale`!

Using Locale to Compare Strings

The `locale` module is actually pretty cool, and provides lowly spare-time programmers like myself a way to handle the daunting task of proper locale-dependent support of their libraries and utilities. Having said that, it can be a bit of a bear to get right, although they do point out in the documentation that it will be painful to use. Aside from the caveats spelled out in that link, it turns out that just comparing strings with `locale` in a cross-platform and cross-python-version manner is not as straightforward as one might hope.

First, how to use `locale` to compare strings? It's actually pretty straightforward. Simply run the input through the `locale` transformation function `locale.strxfrm()`.

```
>>> import locale, sys
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
>>> a = ['a', 'b', 'ä']
>>> sorted(a)
['a', 'b', 'ä']
>>> # The below fails on OSX, so don't run doctest on darwin.
>>> is_osx = sys.platform == 'darwin'
>>> sorted(a, key=locale.strxfrm) if not is_osx else ['a', 'ä', 'b']
```

```
['a', 'ä', 'b']
>>>
>>> a = ['apple', 'Banana', 'banana', 'Apple']
>>> sorted(a, key=locale.strxfrm) if not is_osx else ['apple', 'Apple', 'banana',
↪'Banana']
['apple', 'Apple', 'banana', 'Banana']
```

It turns out that locale-aware sorting groups numbers in the same way as turning on *GROUPLETTERS* and *LOWERCASEFIRST*. The trick is that you have to apply `locale.strxfrm()` only to non-numeric characters; otherwise, numbers won't be parsed properly. Therefore, it must be applied as part of the `coerce_to_int()`/`coerce_to_float()` functions in a manner similar to `groupletters()`.

As you might have guessed, there is a small problem. It turns out the there is a bug in the legacy Python implementation of `locale.strxfrm()` that causes it to outright fail for `unicode()` input (https://bugs.python.org/issue2481). `locale.strcoll()` works, but is intended for use with `cmp`, which does not exist in current Python implementations. Luckily, the `functools.cmp_to_key()` function makes `locale.strcoll()` behave like `locale.strxfrm()` (that is, of course, unless you are on Python 2.6 where `functools.cmp_to_key()` doesn't exist, in which case you simply copy-paste the implementation from Python 2.7 directly into your code).

Handling Broken Locale On OSX

But what if the underlying *locale* implementation that `locale` relies upon is simply broken? It turns out that the *locale* library on OSX (and other BSD systems) is broken (and for some reason has never been fixed?), and so `locale` does not work as expected.

How do I define doesn't work as expected?

```
>>> a = ['apple', 'Banana', 'banana', 'Apple']
>>> sorted(a)
['Apple', 'Banana', 'apple', 'banana']
>>>
>>> sorted(a, key=locale.strxfrm) if is_osx else sorted(a)
['Apple', 'Banana', 'apple', 'banana']
```

IT'S SORTING AS IF `locale.stfxfrm()` WAS NEVER USED!! (and it's worse once non-ASCII characters get thrown into the mix.) I'm really not sure why this is considered OK for the OSX/BSD maintainers to not fix, but it's more than frustrating for poor developers who have been dragged into the *locale* game kicking and screaming. <*deep breath*>.

So, how to deal with this situation? There are two ways to do so.

1. Detect if `locale` is sorting incorrectly (i.e. `dumb`) by seeing if `'A'` is sorted before `'a'` (incorrect) or not.

   ```
   >>> # This is genuinely the name of this function.
   >>> # See natsort.compat.locale.py
   >>> def dumb_sort():
   ...     return locale.strxfrm('A') < locale.strxfrm('a')
   ...
   ```

   If a `dumb` *locale* implementation is found, then automatically turn on *LOWERCASEFIRST* and *GROUPLETTERS*.

2. Use an alternate library if installed. ICU is a great and powerful library that has a pretty decent Python port called (you guessed it) PyICU. If a user has this library installed on their computer, *`natsort`* chooses to use that instead of `locale`. With a little bit of planning, one can write a set of wrapper functions that call the correct library under the hood such that the business logic never has to know what library is being used (see natsort.compat.locale.py).

Let me tell you, this little complication really makes a challenge of testing the code, since one must set up different environments on different operating systems in order to test all possible code paths. Not to mention that certain checks *will* fail for certain operating systems and environments so one must be diligent in either writing the tests not to fail, or ignoring those tests when on offending environments.

Handling Locale-Aware Numbers

Thousands separator support is a problem that I knew would someday be requested but had decided to push off until a rainy day. One day it finally rained, and I decided to tackle the problem.

So what is the problem? Consider the number `1,234,567` (assuming the `','` is the thousands separator). Try to run that through `int()` and you will get a `ValueError`. To handle this properly the thousands separators must be removed.

```
>>> float('1,234,567'.replace(',', ''))
1234567.0
```

What if, in our current locale, the thousands separator is `'.'` and the `','` is the decimal separator (like for the German locale *de_DE*)?

```
>>> float('1.234.567'.replace('.', '').replace(',', '.'))
1234567.0
>>> float('1.234.567,89'.replace('.', '').replace(',', '.'))
1234567.89
```

This is pretty much what `locale.atoi()` and `locale.atof()` do under the hood. So what's the problem? Why doesn't *natsort* just use this method under its hood? Well, let's take a look at what would happen if we send some possible *natsort* input through our the above function:

```
>>> natsort_key('1,234 apples, please.'.replace(',', ''))
('', 1234, ' apples please.')
>>> natsort_key('Sir, €1.234,50 please.'.replace('.', '').replace(',', '.'), as_
↪float=True)
('Sir. €', 1234.5, ' please')
```

Any character matching the thousands separator was dropped, and anything matching the decimal separator was changed to `'.'`! If these characters were critical to how your data was ordered, this would break *natsort*.

The first solution one might consider would be to first decompose the input into sub-components (like we did for the *GROUPLETTERS* method above) and then only apply these transformations on the number components. This is a chicken-and-egg problem, though, because *we cannot appropriately separate out the numbers because of the thousands separators and non-'.' decimal separators* (well, at least not without making multiple passes over the data which I do not consider to be a valid option).

Regular expressions to the rescue! With regular expressions, we can remove the thousands separators and change the decimal separator only when they are actually within a number. Once the input has been pre-processed with this regular expression, all the infrastructure shown previously will work.

Beware, these regular expressions will make your eyes bleed.

```
>>> decimal = ','  # Assume German locale, so decimal separator is ','
>>> # Look-behind assertions cannot accept range modifiers, so instead of i.e.
>>> # (?<!\.[0-9]{1,3}) I have to repeat the look-behind for 1, 2, and 3.
>>> nodecimal = r'(?<!{dec}[0-9])(?<!{dec}[0-9]{{2}})(?<!{dec}[0-9]{{3}})'.
↪format(dec=decimal)
>>> strip_thousands = r'''
...     (?<=[0-9]{{1}})  # At least 1 number
```

```
...     (?<![0-9]{{4}})  # No more than 3 numbers
...     {nodecimal}      # Cannot follow decimal
...     {thou}           # The thousands separator
...     (?=[0-9]{{3}}    # Three numbers must follow
...      ([^0-9]|$)      # But a non-number after that
...     )
... '''.format(nodecimal=nodecimal, thou='.')  # Thousands separator is '.' in German␣
↪locale.
...
>>> re.sub(strip_thousands, '', 'Sir, €1.234,50 please.', flags=re.X)
'Sir, €1234,50 please.'
>>>
>>> # The decimal point must be preceded by a number or after
>>> # a number. This option only needs to be performed in the
>>> # case when the decimal separator for the locale is not '.'.
>>> switch_decimal = r'(?<=[0-9]){decimal}|{decimal}(?=[0-9])'
>>> switch_decimal = switch_decimal.format(decimal=decimal)
>>> re.sub(switch_decimal, '.', 'Sir, €1234,50 please.', flags=re.X)
'Sir, €1234.50 please.'
>>>
>>> natsort_key('Sir, €1234.50 please.', as_float=True)
('Sir, €', 1234.5, ' please.')
```

Final Thoughts

My hope is that users of *natsort* never have to think about or worry about all the bookkeeping or any of the details described above, and that using *natsort* seems to magically "just work". For those of you who took the time to read this engineering description, I hope it has enlightened you to some of the issues that can be encountered when code is released into the wild and has to accept "real-world data", or to what happens to developers who naïvely make bold assumptions that are counter to what the rest of the world assumes.

CHAPTER 3

Examples and Recipes

If you want more detailed examples than given on this page, please see https://github.com/SethMMorton/natsort/tree/master/test_natsort.

Basic Usage

In the most basic use case, simply import `natsorted()` and use it as you would `sorted()`:

```
>>> a = ['2 ft 7 in', '1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '7 ft 6 in']
>>> sorted(a)
```

```
['1 ft 5 in', '10 ft 2 in', '2 ft 11 in', '2 ft 7 in', '7 ft 6 in']
>>> from natsort import natsorted, ns
>>> natsorted(a)
['1 ft 5 in', '2 ft 7 in', '2 ft 11 in', '7 ft 6 in', '10 ft 2 in']
```

Sort Version Numbers

As of *natsort* version >= 4.0.0, *natsorted()* will now properly sort version numbers. The old function *versorted()* exists for backwards compatibility but new development should use *natsorted()*.

Sorting with Alpha, Beta, and Release Candidates

By default, if you wish to sort versions with a non-strict versioning scheme, you may not get the results you expect:

```
>>> a = ['1.2', '1.2rc1', '1.2beta2', '1.2beta1', '1.2alpha', '1.2.1', '1.1', '1.3']
>>> natsorted(a)
['1.1', '1.2', '1.2.1', '1.2alpha', '1.2beta1', '1.2beta2', '1.2rc1', '1.3']
```

To make the '1.2' pre-releases come before '1.2.1', you need to use the following recipe:

```
>>> natsorted(a, key=lambda x: x.replace('.', '~'))
['1.1', '1.2', '1.2alpha', '1.2beta1', '1.2beta2', '1.2rc1', '1.2.1', '1.3']
```

If you also want '1.2' after all the alpha, beta, and rc candidates, you can modify the above recipe:

```
>>> natsorted(a, key=lambda x: x.replace('.', '~')+'z')
['1.1', '1.2alpha', '1.2beta1', '1.2beta2', '1.2rc1', '1.2', '1.2.1', '1.3']
```

Please see this issue to see why this works.

Sort OS-Generated Paths

In some cases when sorting file paths with OS-Generated names, the default `natsorted` algorithm may not be sufficient. In cases like these, you may need to use the `ns.PATH` option:

```
>>> a = ['./folder/file (1).txt',
...      './folder/file.txt',
...      './folder (1)/file.txt',
...      './folder (10)/file.txt']
>>> natsorted(a)
['./folder (1)/file.txt', './folder (10)/file.txt', './folder/file (1).txt', './
↪folder/file.txt']
>>> natsorted(a, alg=ns.PATH)
['./folder/file.txt', './folder/file (1).txt', './folder (1)/file.txt', './folder␣
↪(10)/file.txt']
```

Locale-Aware Sorting (Human Sorting)

Note: Please read *Possible Issues with humansorted() or ns.LOCALE* before using `ns.LOCALE`, *`humansorted()`*, or *`index_humansorted()`*.

You can instruct *`natsort`* to use locale-aware sorting with the `ns.LOCALE` option. In addition to making this understand non-ASCII characters, it will also properly interpret non-'.' decimal separators and also properly order case. It may be more convenient to just use the *`humansorted()`* function:

```
>>> from natsort import humansorted
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
>>> a = ['Apple', 'corn', 'Corn', 'Banana', 'apple', 'banana']
>>> natsorted(a, alg=ns.LOCALE)
['apple', 'Apple', 'banana', 'Banana', 'corn', 'Corn']
>>> humansorted(a)
['apple', 'Apple', 'banana', 'Banana', 'corn', 'Corn']
```

You may find that if you do not explicitly set the locale your results may not be as you expect... I have found that it depends on the system you are on. If you use PyICU (see below) then you should not need to do this.

Controlling Case When Sorting

For non-numbers, by default *`natsort`* used ordinal sorting (i.e. it sorts by the character's value in the ASCII table). For example:

```
>>> a = ['Apple', 'corn', 'Corn', 'Banana', 'apple', 'banana']
>>> natsorted(a)
['Apple', 'Banana', 'Corn', 'apple', 'banana', 'corn']
```

There are times when you wish to ignore the case when sorting, you can easily do this with the `ns.IGNORECASE` option:

```
>>> natsorted(a, alg=ns.IGNORECASE)
['Apple', 'apple', 'Banana', 'banana', 'corn', 'Corn']
```

Note thats since Python's sorting is stable, the order of equivalent elements after lowering the case is the same order they appear in the original list.

Upper-case letters appear first in the ASCII table, but many natural sorting methods place lower-case first. To do this, use `ns.LOWERCASEFIRST`:

```
>>> natsorted(a, alg=ns.LOWERCASEFIRST)
['apple', 'banana', 'corn', 'Apple', 'Banana', 'Corn']
```

It may be undesirable to have the upper-case letters grouped together and the lower-case letters grouped together; most would expect all "a"s to bet together regardless of case, and all "b"s, and so on. To achieve this, use `ns.GROUPLETTERS`:

```
>>> natsorted(a, alg=ns.GROUPLETTERS)
['Apple', 'apple', 'Banana', 'banana', 'Corn', 'corn']
```

You might combine this with `ns.LOWERCASEFIRST` to get what most would expect to be "natural" sorting:

```
>>> natsorted(a, alg=ns.G | ns.LF)
['apple', 'Apple', 'banana', 'Banana', 'corn', 'Corn']
```

Customizing Float Definition

You can make *natsorted()* search for any float that would be a valid Python float literal, such as 5, 0.4, -4.78, +4.2E-34, etc. using the `ns.FLOAT` key. You can disable the exponential component of the number with `ns.NOEXP`.

```
>>> a = ['a50', 'a51.', 'a+50.4', 'a5.034e1', 'a+50.300']
>>> natsorted(a, alg=ns.FLOAT)
['a50', 'a5.034e1', 'a51.', 'a+50.300', 'a+50.4']
>>> natsorted(a, alg=ns.FLOAT | ns.SIGNED)
['a50', 'a+50.300', 'a5.034e1', 'a+50.4', 'a51.']
>>> natsorted(a, alg=ns.FLOAT | ns.SIGNED | ns.NOEXP)
['a5.034e1', 'a50', 'a+50.300', 'a+50.4', 'a51.']
```

For convenience, the `ns.REAL` option is provided which is a shortcut for `ns.FLOAT | ns.SIGNED` and can be used to sort on real numbers. This can be easily accessed with the *realsorted()* convenience function. Please note that the behavior of the *realsorted()* function was the default behavior of *natsorted()* for *natsort* version < 4.0.0:

```
>>> natsorted(a, alg=ns.REAL)
['a50', 'a+50.300', 'a5.034e1', 'a+50.4', 'a51.']
>>> from natsort import realsorted
>>> realsorted(a)
['a50', 'a+50.300', 'a5.034e1', 'a+50.4', 'a51.']
```

Using a Custom Sorting Key

Like the built-in `sorted` function, `natsorted` can accept a custom sort key so that:

```
>>> from operator import attrgetter, itemgetter
>>> a = [['a', 'num4'], ['b', 'num8'], ['c', 'num2']]
>>> natsorted(a, key=itemgetter(1))
[['c', 'num2'], ['a', 'num4'], ['b', 'num8']]
>>> class Foo:
...     def __init__(self, bar):
...         self.bar = bar
...     def __repr__(self):
...         return "Foo('{0}')".format(self.bar)
>>> b = [Foo('num3'), Foo('num5'), Foo('num2')]
>>> natsorted(b, key=attrgetter('bar'))
[Foo('num2'), Foo('num3'), Foo('num5')]
```

Generating a Natsort Key

If you need to sort a list in-place, you cannot use *natsorted()*; you need to pass a key to the `list.sort()` method. The function *natsort_keygen()* is a convenient way to generate these keys for you:

```
>>> from natsort import natsort_keygen
>>> a = ['a50', 'a51.', 'a50.4', 'a5.034e1', 'a50.300']
>>> natsort_key = natsort_keygen(alg=ns.FLOAT)
>>> a.sort(key=natsort_key)
>>> a
['a50', 'a50.300', 'a5.034e1', 'a50.4', 'a51.']
```

natsort_keygen() has the same API as *natsorted()* (minus the *reverse* option).

Sorting Multiple Lists According to a Single List

Sometimes you have multiple lists, and you want to sort one of those lists and reorder the other lists according to how the first was sorted. To achieve this you could use the *index_natsorted()* in combination with the convenience function *order_by_index()*:

```
>>> from natsort import index_natsorted, order_by_index
>>> a = ['a2', 'a9', 'a1', 'a4', 'a10']
>>> b = [4,    5,    6,    7,    8]
>>> c = ['hi', 'lo', 'ah', 'do', 'up']
>>> index = index_natsorted(a)
>>> order_by_index(a, index)
['a1', 'a2', 'a4', 'a9', 'a10']
>>> order_by_index(b, index)
[6, 4, 7, 5, 8]
>>> order_by_index(c, index)
['ah', 'hi', 'do', 'lo', 'up']
```

Returning Results in Reverse Order

Just like the `sorted()` built-in function, you can supply the `reverse` option to return the results in reverse order:

```
>>> a = ['a2', 'a9', 'a1', 'a4', 'a10']
>>> natsorted(a, reverse=True)
['a10', 'a9', 'a4', 'a2', 'a1']
```

Sorting Bytes on Python 3

Python 3 is rather strict about comparing strings and bytes, and this can make it difficult to deal with collections of both. Because of the challenge of guessing which encoding should be used to decode a bytes array to a string, *natsort* does *not* try to guess and automatically convert for you; in fact, the official stance of *natsort* is to not support sorting bytes. Instead, some decoding convenience functions have been provided to you (see *Help With Bytes On Python 3*) that allow you to provide a codec for decoding bytes through the `key` argument that will allow *natsort* to convert byte arrays to strings for sorting; these functions know not to raise an error if the input is not a byte array, so you can use the key on any arbitrary collection of data.

```
>>> from natsort import as_ascii
>>> a = [b'a', 14.0, 'b']
>>> # On Python 2, natsorted(a) would would work as expected.
>>> # On Python 3, natsorted(a) would raise a TypeError (bytes() < str())
```

```
>>> natsorted(a, key=as_ascii) == [14.0, b'a', 'b']
True
```

Additionally, regular expressions cannot be run on byte arrays, making it so that *natsort* cannot parse them for numbers. As a result, if you run *natsort* on a list of bytes, you will get results that are like Python's default sorting behavior. Of course, you can use the decoding functions to solve this:

```
>>> from natsort import as_utf8
>>> a = [b'a56', b'a5', b'a6', b'a40']
>>> natsorted(a)
[b'a40', b'a5', b'a56', b'a6']
>>> natsorted(a, key=as_utf8) == [b'a5', b'a6', b'a40', b'a56']
True
```

If you need a codec different from ASCII or UTF-8, you can use *decoder()* to generate a custom key:

```
>>> from natsort import decoder
>>> a = [b'a56', b'a5', b'a6', b'a40']
>>> natsorted(a, key=decoder('latin1')) == [b'a5', b'a6', b'a40', b'a56']
True
```

Sorting a Pandas DataFrame

As of Pandas version 0.16.0, the sorting methods do not accept a `key` argument, so you cannot simply pass *natsort_keygen()* to a Pandas DataFrame and sort. This request has been made to the Pandas devs; see issue 3942 if you are interested. If you need to sort a Pandas DataFrame, please check out this answer on StackOverflow for ways to do this without the `key` argument to `sort`.

CHAPTER 4

natsort API

natsort_keygen()

natsort.**natsort_keygen**(*key=None*, *alg=0*, ***_kwargs*)

Generate a key to sort strings and numbers naturally.

Generate a key to sort strings and numbers naturally, not lexicographically. This key is designed for use as the *key* argument to functions such as the *sorted* builtin.

The user may customize the generated function with the arguments to *natsort_keygen*, including an optional *key* function.

Parameters

- **key** (*callable, optional*) – A key used to manipulate the input value before parsing for numbers. It is **not** applied recursively. It should accept a single argument and return a single value.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.INT*.

Returns **out** – A function that parses input for natural sorting that is suitable for passing as the *key* argument to functions such as *sorted*.

Return type function

See also:

natsorted()

Examples

natsort_keygen is a convenient way to create a custom key to sort lists in-place (for example).:

```
>>> a = ['num5.10', 'num-3', 'num5.3', 'num2']
>>> a.sort(key=natsort_keygen(alg=ns.REAL))
>>> a
['num-3', 'num2', 'num5.10', 'num5.3']
```

natsorted()

natsort.**natsorted**(*seq*, *key=None*, *reverse=False*, *alg=0*, ***_kwargs*)

Sorts an iterable naturally.

Sorts an iterable naturally (alphabetically and numerically), not lexicographically. Returns a list containing a sorted copy of the iterable.

Parameters

- **seq** (*iterable*) – The iterable to sort.
- **key** (*callable, optional*) – A key used to determine how to sort each element of the iterable. It is **not** applied recursively. It should accept a single argument and return a single value.
- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.INT*.

Returns **out** – The sorted sequence.

Return type list

See also:

natsort_keygen() Generates the key that makes natural sorting possible.

realsorted() A wrapper for `natsorted(seq, alg=ns.REAL)`.

humansorted() A wrapper for `natsorted(seq, alg=ns.LOCALE)`.

index_natsorted() Returns the sorted indexes from *natsorted*.

Examples

Use *natsorted* just like the builtin *sorted*:

```
>>> a = ['num3', 'num5', 'num2']
>>> natsorted(a)
['num2', 'num3', 'num5']
```

versorted()

natsort.**versorted**(*seq*, *key=None*, *reverse=False*, *alg=0*, ***_kwargs*)

Identical to *natsorted()*.

This function exists for backwards compatibility with *natsort* version < 4.0.0. Future development should use *natsorted()*.

See also:

natsorted()

humansorted()

natsort.**humansorted**(*seq*, *key=None*, *reverse=False*, *alg=0*)

Convenience function to properly sort non-numeric characters.

Convenience function to properly sort non-numeric characters in a locale-aware fashion (a.k.a "human sorting"). This is a wrapper around `natsorted(seq, alg=ns.LOCALE)`.

Parameters

- **seq** (*iterable*) – The sequence to sort.
- **key** (*callable, optional*) – A key used to determine how to sort each element of the sequence. It is **not** applied recursively. It should accept a single argument and return a single value.
- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.LOCALE*.

Returns **out** – The sorted sequence.

Return type list

See also:

index_humansorted() Returns the sorted indexes from *humansorted*.

Notes

Please read *Possible Issues with humansorted() or ns.LOCALE* before using *humansorted*.

Examples

Use *humansorted* just like the builtin *sorted*:

```
>>> a = ['Apple', 'Banana', 'apple', 'banana']
>>> natsorted(a)
['Apple', 'Banana', 'apple', 'banana']
>>> humansorted(a)
['apple', 'Apple', 'banana', 'Banana']
```

realsorted()

natsort.**realsorted**(*seq*, *key=None*, *reverse=False*, *alg=0*)

Convenience function to properly sort signed floats.

Convenience function to properly sort signed floats within strings (i.e. "a-5.7"). This is a wrapper around `natsorted(seq, alg=ns.REAL)`.

The behavior of *realsorted()* for *natsort* version >= 4.0.0 was the default behavior of *natsorted()* for *natsort* version < 4.0.0.

Parameters

- **seq** (*iterable*) – The sequence to sort.
- **key** (*callable, optional*) – A key used to determine how to sort each element of the sequence. It is **not** applied recursively. It should accept a single argument and return a single value.
- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.REAL*.

Returns **out** – The sorted sequence.

Return type list

See also:

index_realsorted() Returns the sorted indexes from *realsorted*.

Examples

Use *realsorted* just like the builtin *sorted*:

```
>>> a = ['num5.10', 'num-3', 'num5.3', 'num2']
>>> natsorted(a)
['num2', 'num5.3', 'num5.10', 'num-3']
>>> realsorted(a)
['num-3', 'num2', 'num5.10', 'num5.3']
```

index_natsorted()

natsort.**index_natsorted**(*seq*, *key=None*, *reverse=False*, *alg=0*, ***_kwargs*)

Return the list of the indexes used to sort the input sequence.

Sorts a sequence naturally, but returns a list of sorted the indexes and not the sorted list. This list of indexes can be used to sort multiple lists by the sorted order of the given sequence.

Parameters

- **seq** (*iterable*) – The sequence to sort.

- **key** (*callable, optional*) – A key used to determine how to sort each element of the sequence. It is **not** applied recursively. It should accept a single argument and return a single value.
- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.INT*.

Returns **out** – The ordered indexes of the sequence.

Return type tuple

See also:

natsorted(), *order_by_index()*

Examples

Use index_natsorted if you want to sort multiple lists by the sorted order of one list:

```
>>> a = ['num3', 'num5', 'num2']
>>> b = ['foo', 'bar', 'baz']
>>> index = index_natsorted(a)
>>> index
[2, 0, 1]
>>> # Sort both lists by the sort order of a
>>> order_by_index(a, index)
['num2', 'num3', 'num5']
>>> order_by_index(b, index)
['baz', 'foo', 'bar']
```

index_versorted()

natsort.**index_versorted**(*seq*, *key=None*, *reverse=False*, *alg=0*, ***_kwargs*)

Identical to *index_natsorted()*.

This function exists for backwards compatibility with `index_natsort` version < 4.0.0. Future development should use *index_natsorted()*.

Please see the *index_natsorted()* documentation for use.

See also:

index_natsorted()

index_humansorted()

natsort.**index_humansorted**(*seq*, *key=None*, *reverse=False*, *alg=0*)

Return the list of the indexes used to sort the input sequence in a locale-aware manner.

Sorts a sequence in a locale-aware manner, but returns a list of sorted the indexes and not the sorted list. This list of indexes can be used to sort multiple lists by the sorted order of the given sequence.

This is a wrapper around `index_natsorted(seq, alg=ns.LOCALE)`.

Parameters

- **seq** (*iterable*) – The sequence to sort.
- **key** (*callable, optional*) – A key used to determine how to sort each element of the sequence. It is **not** applied recursively. It should accept a single argument and return a single value.
- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.LOCALE*.

Returns **out** – The ordered indexes of the sequence.

Return type tuple

See also:

humansorted(), *order_by_index()*

Notes

Please read *Possible Issues with humansorted() or ns.LOCALE* before using *humansorted*.

Examples

Use *index_humansorted* just like the builtin *sorted*:

```
>>> a = ['Apple', 'Banana', 'apple', 'banana']
>>> index_humansorted(a)
[2, 0, 3, 1]
```

index_realsorted()

natsort.**index_realsorted**(*seq*, *key=None*, *reverse=False*, *alg=0*)

Return the list of the indexes used to sort the input sequence in a locale-aware manner.

Sorts a sequence in a locale-aware manner, but returns a list of sorted the indexes and not the sorted list. This list of indexes can be used to sort multiple lists by the sorted order of the given sequence.

This is a wrapper around `index_natsorted(seq, alg=ns.REAL)`.

The behavior of *index_realsorted()* in *natsort* version >= 4.0.0 was the default behavior of *index_natsorted()* for *natsort* version < 4.0.0.

Parameters

- **seq** (*iterable*) – The sequence to sort.
- **key** (*callable, optional*) – A key used to determine how to sort each element of the sequence. It is **not** applied recursively. It should accept a single argument and return a single value.

- **reverse** (*{True, False}, optional*) – Return the list in reversed sorted order. The default is *False*.
- **alg** (*ns enum, optional*) – This option is used to control which algorithm *natsort* uses when sorting. For details into these options, please see the *ns* class documentation. The default is *ns.REAL*.

Returns **out** – The ordered indexes of the sequence.

Return type tuple

See also:

realsorted(), *order_by_index()*

Examples

Use *index_realsorted* just like the builtin *sorted*:

```
>>> a = ['num5.10', 'num-3', 'num5.3', 'num2']
>>> index_realsorted(a)
[1, 3, 0, 2]
```

order_by_index()

natsort.**order_by_index**(*seq*, *index*, *iter=False*)

Order a given sequence by an index sequence.

The output of *index_natsorted* is a sequence of integers (index) that correspond to how its input sequence **would** be sorted. The idea is that this index can be used to reorder multiple sequences by the sorted order of the first sequence. This function is a convenient wrapper to apply this ordering to a sequence.

Parameters

- **seq** (*sequence*) – The sequence to order.
- **index** (*iterable*) – The iterable that indicates how to order *seq*. It should be the same length as *seq* and consist of integers only.
- **iter** (*{True, False}, optional*) – If *True*, the ordered sequence is returned as a iterator; otherwise it is returned as a list. The default is *False*.

Returns **out** – The sequence ordered by *index*, as a *list* or as an iterator (depending on the value of *iter*).

Return type {list, iterator}

See also:

index_natsorted(), *index_humansorted()*, *index_realsorted()*

Examples

order_by_index is a convenience function that helps you apply the result of *index_natsorted*:

```
>>> a = ['num3', 'num5', 'num2']
>>> b = ['foo', 'bar', 'baz']
>>> index = index_natsorted(a)
>>> index
[2, 0, 1]
>>> # Sort both lists by the sort order of a
>>> order_by_index(a, index)
['num2', 'num3', 'num5']
>>> order_by_index(b, index)
['baz', 'foo', 'bar']
```

ns

class natsort.**ns**

Enum to control the *natsort* algorithm.

This class acts like an enum to control the *natsort* algorithm. The user may select several options simultaneously by or'ing the options together. For example, to choose `ns.INT`, `ns.PATH`, and `ns.LOCALE`, you could do `ns.INT | ns.LOCALE | ns.PATH`. Each function in the *natsort* package has an *alg* option that accepts this enum to allow fine control over how your input is sorted.

Each option has a shortened 1- or 2-letter form.

Note: Please read *Possible Issues with humansorted() or ns.LOCALE* before using `ns.LOCALE`.

INT, I (default)
The default - parse numbers as integers.

FLOAT, F
Tell *natsort* to parse numbers as floats.

UNSIGNED, U (default)
Tell *natsort* to ignore any sign (i.e. "-" or "+") to the immediate left of a number. This is the default.

SIGNED, S
Tell *natsort* to take into account any sign (i.e. "-" or "+") to the immediate left of a number.

REAL, R
This is a shortcut for `ns.FLOAT | ns.SIGNED`, which is useful when attempting to sort real numbers.

NOEXP, N
Tell *natsort* to not search for exponents as part of a float number. For example, with *NOEXP* the number "5.6E5" would be interpreted as *5.6*, *"E"*, and *5* instead of *560000*.

PATH, P
Tell *natsort* to interpret strings as filesystem paths, so they will be split according to the filesystem separator (i.e. '/' on UNIX, '' on Windows), as well as splitting on the file extension, if any. Without this, lists of file paths like `['Folder/', 'Folder (1)/', 'Folder (10)/']` will not be sorted properly; 'Folder/' will be placed at the end, not at the front. It is the same as setting the old *as_path* option to *True*.

COMPATIBILITYNORMALIZE, CN
Use the "NFKD" unicode normalization form on input rather than the default "NFD". This will transform characters such as '' into '7'. Please see https://stackoverflow.com/a/7934397/1399279, https://stackoverflow.com/a/7931547/1399279, and http://unicode.org/reports/tr15/ full details into unicode normalization.

LOCALE, L
Tell *natsort* to be locale-aware when sorting. This includes both proper sorting of alphabetical characters as well as proper handling of locale-dependent decimal separators and thousands separators. This is a shortcut for `ns.LOCALEALPHA | ns.LOCALENUM`. Your sorting results will vary depending on your current locale.

LOCALEALPHA, LA
Tell *natsort* to be locale-aware when sorting, but only for alphabetical characters.

LOCALENUM, LN
Tell *natsort* to be locale-aware when sorting, but only for decimal separators and thousands separators.

IGNORECASE, IC
Tell *natsort* to ignore case when sorting. For example, `['Banana', 'apple', 'banana', 'Apple']` would be sorted as `['apple', 'Apple', 'Banana', 'banana']`.

LOWERCASEFIRST, LF
Tell *natsort* to put lowercase letters before uppercase letters when sorting. For example, `['Banana', 'apple', 'banana', 'Apple']` would be sorted as `['apple', 'banana', 'Apple', 'Banana']` (the default order would be `['Apple', 'Banana', 'apple', 'banana']` which is the order from a purely ordinal sort). Useless when used with *IGNORECASE*. Please note that if used with `LOCALE`, this actually has the reverse effect and will put uppercase first (this is because `LOCALE` already puts lowercase first); you may use this to your advantage if you need to modify the order returned with `LOCALE`.

GROUPLETTERS, G
Tell *natsort* to group lowercase and uppercase letters together when sorting. For example, `['Banana', 'apple', 'banana', 'Apple']` would be sorted as `['Apple', 'apple', 'Banana', 'banana']`. Useless when used with *IGNORECASE*; use with *LOWERCASEFIRST* to reverse the order of upper and lower case. Generally not needed with *LOCALE*.

CAPITALFIRST, C
Only used when *LOCALE* is enabled. Tell *natsort* to put all capitalized words before non-capitalized words. This is essentially the inverse of *GROUPLETTERS*, and is the default Python sorting behavior without *LOCALE*.

UNGROUPLETTERS, UG
An alias for *CAPITALFIRST*.

NANLAST, NL
If an NaN shows up in the input, this instructs *natsort* to treat these as +Infinity and place them after all the other numbers. By default, an NaN be treated as -Infinity and be placed first.

TYPESAFE, T
Deprecated as of *natsort* version 5.0.0; this option is now a no-op because it is always true.

VERSION, V
Deprecated as of *natsort* version 5.0.0; this option is now a no-op because it is the default.

DIGIT, D
Same as *VERSION* above.

Notes

If you prefer to use *import natsort as ns* as opposed to *from natsort import natsorted, ns*, the *ns* options are available as top-level imports.

```
>>> import natsort as ns
>>> a = ['num5.10', 'num-3', 'num5.3', 'num2']
>>> ns.natsorted(a, alg=ns.REAL) == ns.natsorted(a, alg=ns.ns.REAL)
True
```

Help With Bytes On Python 3

The official stance of *natsort* is to not support *bytes* for sorting; there is just too much that can go wrong when trying to automate conversion between *bytes* and *str*. But rather than completely give up on *bytes*, *natsort* provides three functions that make it easy to quickly decode *bytes* to *str* so that sorting is possible.

natsort.**decoder**(*encoding*)

Return a function that can be used to decode bytes to unicode.

Parameters **encoding** (*str*) – The codec to use for decoding. This must be a valid unicode codec.

Returns A function that takes a single argument and attempts to decode it using the supplied codec. Any *UnicodeErrors* are raised. If the argument was not of *bytes* type, it is simply returned as-is.

Return type decode_function

See also:

as_ascii(), *as_utf8()*

Examples

```
>>> f = decoder('utf8')
>>> f(b'bytes') == 'bytes'
True
>>> f(12345) == 12345
True
>>> # On Python 3, without decoder this would return [b'a10', b'a2']
>>> natsorted([b'a10', b'a2'], key=decoder('utf8')) == [b'a2', b'a10']
True
>>> # On Python 3, without decoder this would raise a TypeError.
>>> natsorted([b'a10', 'a2'], key=decoder('utf8')) == ['a2', b'a10']
True
```

natsort.**as_ascii**(*s*)

Function to decode an input with the ASCII codec, or return as-is.

Parameters **s** – Any object.

Returns If the input was of type *bytes*, the return value is a *str* decoded with the ASCII codec. Otherwise, the return value is identically the input.

Return type output

See also:

decoder()

natsort.**as_utf8**(*s*)

Function to decode an input with the UTF-8 codec, or return as-is.

Parameters **s** – Any object.

Returns If the input was of type *bytes*, the return value is a *str* decoded with the UTF-8 codec. Otherwise, the return value is identically the input.

Return type output

See also:

decoder()

Help With Creating Function Keys

If you need to create a complicated *key* argument to (for example) *natsorted()* that is actually multiple functions called one after the other, the following function can help you easily perform this action. It is used internally to *natsort*, and has been exposed publically for the convenience of the user.

natsort.**chain_functions**(*functions*)

Chain a list of single-argument functions together and return.

The functions are applied in list order, and the output of the previous functions is passed to the next function.

Parameters **functions** (*list*) – A list of single-argument functions to chain together.

Returns

Return type A single argument function.

Examples

Chain several functions together!

```
>>> funcs = [lambda x: x * 4, len, lambda x: x + 5]
>>> func = chain_functions(funcs)
>>> func('hey')
17
```

Possible Issues with `humansorted()` or `ns.LOCALE`

Being Locale-Aware Means Both Numbers and Non-Numbers

In addition to modifying how characters are sorted, `ns.LOCALE` will take into account locale-dependent thousands separators (and locale-dependent decimal separators if `ns.FLOAT` is enabled). This means that if you are in a locale that uses commas as the thousands separator, a number like `123,456` will be interpreted as `123456`. If this is not what you want, you may consider using `ns.LOCALEALPHA` which will only enable locale-aware sorting for non-numbers (similarly, `ns.LOCALENUM` enables locale-aware sorting only for numbers).

Regenerate Key With `natsort_keygen()` After Changing Locale

When *natsort_keygen()* is called it returns a key function that hard-codes the provided settings. This means that the key returned when `ns.LOCALE` is used contins the settings specifed by the locale *loaded at the time the key is generated*. If you change the locale, you should regenerate the key to account for the new locale.

Corollary: Do Not Reuse `natsort_keygen()` After Changing Locale

If you change locale, the old function will not work as expected. The locale library works with a global state. When *`natsort_keygen()`* is called it does the best job that it can to make the returned function as static as possible and independent of the global state, but the strxfrm function must access this global state to work; therefore, if you change locale and use `ns.LOCALE` then you should discard the old key.

Note: If you use PyICU then you may be able to reuse keys after changing locale.

The locale Module From the StdLib Has Issues

`natsort` will use PyICU for *`humansorted()`* or `ns.LOCALE` if it is installed. If not, it will fall back on the locale library from the Python stdlib. If you do not have PyICU installed, please keep the following known problems and issues in mind.

Note: Remember, if you have PyICU installed you shouldn't need to worry about any of these.

Explicitly Set the Locale Before Using `humansorted()` or `ns.LOCALE`

I have found that unless you explicitly set a locale, the sorted order may not be what you expect. Setting this is straightforward (in the below example I use 'en_US.UTF-8', but you should use your locale):

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
```

locale Is Broken on Mac OS X

It's not Python's fault, but the OS... the locale library for BSD-based systems (of which Mac OS X is one) is broken. See the following links:

- http://stackoverflow.com/questions/3412933/python-not-sorting-unicode-properly-strcoll-doesnt-help
- http://bugs.python.org/issue23195
- https://github.com/SethMMorton/natsort/issues/21 (contains instructons on installing)
- http://stackoverflow.com/questions/33459384/unicode-character-not-in-range-when-calling-locale-strxfrm
- https://github.com/SethMMorton/natsort/issues/34

Of course, installing PyICU fixes this, but if you don't want to or cannot install this there is some hope.

1. As of `natsort` version 4.0.0, `natsort` is configured to compensate for a broken `locale` library. When sorting non-numbers it will handle case as you expect, but it will still not be able to comprehend non-ASCII characters properly. Additionally, it has a built-in lookup table of thousands separators that are incorrect on OS X/BSD (but is possible it is not complete... please file an issue if you see it is not complete)
2. Use "*.ISO8859-1" locale (i.e. 'en_US.ISO8859-1') rather than "*.UTF-8" locale. I have found that these have fewer issues than "UTF-8", but your mileage may vary.

CHAPTER 5

Shell Script

The `natsort` shell script is automatically installed when you install *`natsort`* with pip.

Below is the usage and some usage examples for the `natsort` shell script.

Usage

```
usage: natsort [-h] [--version] [-p] [-f LOW HIGH] [-F LOW HIGH] [-e EXCLUDE]
               [-r] [-t {digit,int,float,version,ver}] [--nosign] [--noexp]
               [--locale]
               [entries [entries ...]]

Performs a natural sort on entries given on the command-line.
A natural sort sorts numerically then alphabetically, and will sort
by numbers in the middle of an entry.

positional arguments:
  entries               The entries to sort. Taken from stdin if nothing is
                        given on the command line.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -p, --paths           Interpret the input as file paths. This is not
                        strictly necessary to sort all file paths, but in
                        cases where there are OS-generated file paths like
                        "Folder/" and "Folder (1)/", this option is needed to
                        make the paths sorted in the order you expect
                        ("Folder/" before "Folder (1)/").
  -f LOW HIGH, --filter LOW HIGH
                        Used for keeping only the entries that have a number
                        falling in the given range.
  -F LOW HIGH, --reverse-filter LOW HIGH
                        Used for excluding the entries that have a number
```

```
                        falling in the given range.
  -e EXCLUDE, --exclude EXCLUDE
                        Used to exclude an entry that contains a specific
                        number.
  -r, --reverse         Returns in reversed order.
  -t {digit,int,float,version,ver,real,f,i,r,d},
  --number-type {digit,int,float,version,ver,real,f,i,r,d},
  --number_type {digit,int,float,version,ver,real,f,i,r,d}
                        Choose the type of number to search for. "float" will
                        search for floating-point numbers. "int" will only
                        search for integers. "digit", "version", and "ver" are
                        synonyms for "int"."real" is a shortcut for "float"
                        with --sign. "i" and "d" are synonyms for "int", "f"
                        is a synonym for "float", and "r" is a synonym for
                        "real".The default is int.
  --nosign              Do not consider "+" or "-" as part of a number, i.e.
                        do not take sign into consideration. This is the
                        default.
  -s, --sign            Consider "+" or "-" as part of a number, i.e. take
                        sign into consideration. The default is unsigned.
  --noexp               Do not consider an exponential as part of a number,
                        i.e. 1e4, would be considered as 1, "e", and 4, not as
                        10000. This only effects the --number-type=float.
  -l, --locale          Causes natsort to use locale-aware sorting. You will
                        get the best results if you install PyICU.
```

Description

`natsort` was originally written to aid in computational chemistry research so that it would be easy to analyze large sets of output files named after the parameter used:

```
$ ls *.out
mode1000.35.out mode1243.34.out mode744.43.out mode943.54.out
```

(Obviously, in reality there would be more files, but you get the idea.) Notice that the shell sorts in lexicographical order. This is the behavior of programs like `find` as well as `ls`. The problem is passing these files to an analysis program causes them not to appear in numerical order, which can lead to bad analysis. To remedy this, use `natsort`:

```
$ natsort *.out
mode744.43.out
mode943.54.out
mode1000.35.out
mode1243.34.out
$ natsort -t r *.out | xargs your_program
```

`-t r` is short for `--number-type real`. You can also place natsort in the middle of a pipe:

```
$ find . -name "*.out" | natsort -t r | xargs your_program
```

To sort version numbers, use the default `--number-type`:

```
$ ls *
prog-1.10.zip prog-1.9.zip prog-2.0.zip
$ natsort *
prog-1.9.zip
```

```
prog-1.10.zip
prog-2.0.zip
```

In general, all `natsort` shell script options mirror the *`natsorted()`* API, with notable exception of the `--filter`, `--reverse-filter`, and `--exclude` options. These three options are used as follows:

```
$ ls *.out
mode1000.35.out mode1243.34.out mode744.43.out mode943.54.out
$ natsort -t r *.out -f 900 1100 # Select only numbers between 900-1100
mode943.54.out
mode1000.35.out
$ natsort -t r *.out -F 900 1100 # Select only numbers NOT between 900-1100
mode744.43.out
mode1243.34.out
$ natsort -t r *.out -e 1000.35 # Exclude 1000.35 from search
mode744.43.out
mode943.54.out
mode1243.34.out
```

If you are sorting paths with OS-generated filenames, you may require the `--paths`/`-p` option:

```
$ find . ! -path . -type f
./folder/file (1).txt
./folder/file.txt
./folder (1)/file.txt
./folder (10)/file.txt
./folder (2)/file.txt
$ find . ! -path . -type f | natsort
./folder (1)/file.txt
./folder (2)/file.txt
./folder (10)/file.txt
./folder/file (1).txt
./folder/file.txt
$ find . ! -path . -type f | natsort -p
./folder/file.txt
./folder/file (1).txt
./folder (1)/file.txt
./folder (2)/file.txt
./folder (10)/file.txt
```

CHAPTER 6

Changelog

08-19-2017 v. 5.1.0

- Fixed `StopIteration` warning on Python 3.6+.
- All Unicode input is now normalized.

04-30-2017 v. 5.0.3

- Improved development infrastructure.
- Migrated documentation to ReadTheDocs.

01-02-2017 v. 5.0.2

- Added additional unicode number support for Python 3.6.
- Renamed several internal functions and variables to improve clarity.
- Improved documentation examples.
- Added a "how does it work?" section to the documentation.

06-04-2016 v. 5.0.1

- The `ns` enum attributes can now be imported from the top-level namespace.
- Fixed a bug with the `from natsort import *` mechanism.
- Fixed bug with using `natsort` with `python -OO`.

05-08-2016 v. 5.0.0

- `ns.LOCALE`/`humansorted` now accounts for thousands separators.
- Refactored entire codebase to be more functional (as in use functions as units). Previously, the code was rather monolithic and difficult to follow. The goal is that with the code existing in smaller units, contributing will be easier.
- Deprecated `ns.TYPESAFE` option as it is now always on (due to a new iterator-based algorithm, the typesafe function is now cheap).
- Increased speed of execution (came for free with the new functional approach because the new factory function paradigm eliminates most `if` branches during execution).
  - For the most cases, the code is 30-40% faster than version 4.0.4.
  - If using `ns.LOCALE` or `humansorted`, the code is 1100% faster than version 4.0.4.
- Improved clarity of documentaion with regards to locale-aware sorting.
- Added a new `chain_functions` function for convenience in creating a complex user-given `key` from several existing functions.

11-01-2015 v. 4.0.4

- Improved coverage of unit tests.
- Unit tests use new and improved hypothesis library.
- Fixed compatibility issues with Python 3.5

06-25-2015 v. 4.0.3

- Fixed bad install on last release (sorry guys!).

06-24-2015 v. 4.0.2

- Added back Python 2.6 and Python 3.2 compatibility. Unit testing is now performed for these versions.
- Consolidated under-the-hood compatibility functionality.

06-04-2015 v. 4.0.1

- Added support for sorting NaN by internally converting to -Infinity or +Infinity

05-17-2015 v. 4.0.0

- Made default behavior of 'natsort' search for unsigned ints, rather than signed floats. This is a backwards-incompatible change but in 99% of use cases it should not require any end-user changes.

- Improved handling of locale-aware sorting on systems where the underlying locale library is broken.
- Greatly improved all unit tests by adding the hypothesis library.

04-06-2015 v. 3.5.6

- Added 'UNGROUPLETTERS' algorithm to get the case-grouping behavior of an ordinal sort when using 'LOCALE'.
- Added convenience functions 'decoder', 'as_ascii', and 'as_utf8' for dealing with bytes types.

04-04-2015 v. 3.5.5

- Added 'realsorted' and 'index_realsorted' functions for forward-compatibility with >= 4.0.0.
- Made explanation of when to use "TYPESAFE" more clear in the docs.

04-02-2015 v. 3.5.4

- Fixed bug where a 'TypeError' was raised if a string containing a leading number was sorted with alpha-only strings when 'LOCALE' is used.

03-26-2015 v. 3.5.3

- Fixed bug where '–reverse-filter' option in shell script was not getting checked for correctness.
- Documentation updates to better describe locale bug, and illustrate upcoming default behavior change.
- Internal improvements, including making test suite more granular.

01-13-2015 v. 3.5.2

- Enhancement that will convert a 'pathlib.Path' object to a 'str' if 'ns.PATH' is enabled.

09-25-2014 v. 3.5.1

- Fixed bug that caused list/tuples to fail when using 'ns.LOWECASEFIRST' or 'ns.IGNORECASE'.
- Refactored modules so that only the public API was in natsort.py and ns_enum.py.
- Refactored all import statements to be absolute, not relative.

09-02-2014 v. 3.5.0

- Added the 'alg' argument to the 'natsort' functions. This argument accepts an enum that is used to indicate the options the user wishes to use. The 'number_type', 'signed', 'exp', 'as_path', and 'py3_safe' options are being deprecated and will become (undocumented) keyword-only options in natsort version 4.0.0.
- The user can now modify how 'natsort' handles the case of non-numeric characters.
- The user can now instruct 'natsort' to use locale-aware sorting, which allows 'natsort' to perform true "human sorting".
  - The *humansorted* convenience function has been included to make this easier.
- Updated shell script with locale functionality.

08-12-2014 v. 3.4.1

- 'natsort' will now use the 'fastnumbers' module if it is installed. This gives up to an extra 30% boost in speed over the previous performance enhancements.
- Made documentation point to more 'natsort' resources, and also added a new example in the examples section.

07-19-2014 v. 3.4.0

- Fixed a bug that caused user's options to the 'natsort_key' to not be passed on to recursive calls of 'natsort_key'.
- Added a 'natsort_keygen' function that will generate a wrapped version of 'natsort_key' that is easier to call. 'natsort_key' is now set to deprecate at natsort version 4.0.0.
- Added an 'as_path' option to 'natsorted' & co. that will try to treat input strings as filepaths. This will help yield correct results for OS-generated inputs like `['/p/q/o.x', '/p/q (1)/o.x', '/p/q (10)/o.x', '/p/q/o (1).x']`.
- Massive performance enhancements for string input (1.8x-2.0x), at the expense of reduction in speed for numeric input (~2.0x).
  - This is a good compromise because the most common input will be strings, not numbers, and sorting numbers still only takes 0.6x the time of sorting strings. If you are sorting only numbers, you would use 'sorted' anyway.
- Added the 'order_by_index' function to help in using the output of 'index_natsorted' and 'index_versorted'.
- Added the 'reverse' option to 'natsorted' & co. to make it's API more similar to the builtin 'sorted'.
- Added more unit tests.
- Added auxillary test code that helps in profiling and stress-testing.
- Reworked the documentation, moving most of it to PyPI's hosting platform.
- Added support for coveralls.io.
- Entire codebase is now PyFlakes and PEP8 compliant.

06-28-2014 v. 3.3.0

- Added a 'versorted' method for more convenient sorting of versions.
- Updated command-line tool –number_type option with 'version' and 'ver' to make it more clear how to sort version numbers.
- Moved unit-testing mechanism from being docstring-based to actual unit tests in actual functions.
  - This has provided the ability determine the coverage of the unit tests (99%).
  - This also makes the pydoc documentation a bit more clear.
- Made docstrings for public functions mirror the README API.
- Connected natsort development to Travis-CI to help ensure quality releases.

06-20-2014 v. 3.2.1

- Re-"Fixed" unorderable types issue on Python 3.x - this workaround is for when the problem occurs in the middle of the string.

05-07-2014 v. 3.2.0

- "Fixed" unorderable types issue on Python 3.x with a workaround that attempts to replicate the Python 2.x behavior by putting all the numbers (or strings that begin with numbers) first.
- Now explicitly excluding __pycache__ from releases by adding a prune statement to MANIFEST.in.

05-05-2014 v. 3.1.2

- Added setup.cfg to support universal wheels.
- Added Python 3.0 and Python 3.1 as requiring the argparse module.

03-01-2014 v. 3.1.1

- Added ability to sort lists of lists.
- Cleaned up import statements.

01-20-2014 v. 3.1.0

- Added the `signed` and `exp` options to allow finer tuning of the sorting
- Entire codebase now works for both Python 2 and Python 3 without needing to run `2to3`.
- Updated all doctests.
- Further simplified the `natsort` base code by removing unneeded functions.

- Simplified documentation where possible.
- Improved the shell script code
  - Made the documentation less "path"-centric to make it clear it is not just for sorting file paths.
  - Removed the filesystem-based options because these can be achieved better though a pipeline.
  - Added doctests.
  - Added new options that correspond to `signed` and `exp`.
  - The user can now specify multiple numbers to exclude or multiple ranges to filter by.

10-01-2013 v. 3.0.2

- Made float, int, and digit searching algorithms all share the same base function.
- Fixed some outdated comments.
- Made the `__version__` variable available when importing the module.

8-15-2013 v. 3.0.1

- Added support for unicode strings.
- Removed extraneous `string2int` function.
- Fixed empty string removal function.

7-13-2013 v. 3.0.0

- Added a `number_type` argument to the sorting functions to specify how liberal to be when deciding what a number is.
- Reworked the documentation.

6-25-2013 v. 2.2.0

- Added `key` attribute to `natsorted` and `index_natsorted` so that it mimics the functionality of the built-in `sorted`
- Added tests to reflect the new functionality, as well as tests demonstrating how to get similar functionality using `natsort_key`.

12-5-2012 v. 2.1.0

- Reorganized package.
- Now using a platform independent shell script generator (entry_points from distribute).
- Can now execute natsort from command line with `python -m natsort` as well.

11-30-2012 v. 2.0.2

- Added the use_2to3 option to setup.py.
- Added distribute_setup.py to the distribution.
- Added dependency to the argparse module (for python2.6).

11-21-2012 v. 2.0.1

- Reorganized directory structure.
- Added tests into the natsort.py file iteself.

11-16-2012, v. 2.0.0

- Updated sorting algorithm to support floats (including exponentials) and basic version number support.
- Added better README documentation.
- Added doctests.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index