fastnumbers Documentation

Release 2.0.1

Seth M. Morton

Aug 20, 2017

Contents

Contents:

CHAPTER 1

The `fastnumbers` module

Super-fast and clean conversions to numbers.

- Source Code: https://github.com/SethMMorton/fastnumbers
- Downloads: https://pypi.python.org/pypi/fastnumbers
- Documentation: http://fastnumbers.readthedocs.io/

Please see the *Timing Documentation* for timing details. Check out the *API*.

Quick Description

The below examples showcase the `fast_float()` function, which is a fast conversion functions with error-handling. Please see the *API Documentation* for other functions that are available from `fastnumbers`.

```
>>> from fastnumbers import fast_float, float as fnfloat
>>> # Convert string to a float
>>> fast_float('56.07')
56.07
>>> # Unconvertable string returned as-is by default
>>> fast_float('bad input')
'bad input'
>>> # Unconvertable strings can trigger a default value
>>> fast_float('bad input', default=0)
0
>>> # 'default' is also the first optional positional arg
>>> fast_float('bad input', 0)
0
>>> # Integers are converted to floats
>>> fast_float(54)
54.0
>>> # One can ask inf or nan to be substituted with another value
>>> fast_float('nan')
nan
```

```
>>> fast_float('nan', nan=0.0)
0.0
>>> fast_float(float('nan'), nan=0.0)
0.0
>>> fast_float('56.07', nan=0.0)
56.07
>>> # The default built-in float behavior can be triggered with
>>> # "raise_on_invalid" set to True.
>>> fast_float('bad input', raise_on_invalid=True)
Traceback (most recent call last):
  ...
ValueError: invalid literal for float(): bad input
>>> # A key function can be used to return an alternate value for invalid input
>>> fast_float('bad input', key=len)
9
>>> fast_float(54, key=len)
54.0
>>> # Single unicode characters can be converted.
>>> fast_float(u'\u2164')  # Roman numeral 5 (V)
5.0
>>> fast_float(u'\u2466')  # 7 enclosed in a circle
7.0
```

NOTE: If you need locale-dependent conversions, supply the *fastnumbers* function of your choice to `locale.atof()`.

```
import locale
locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
print(atof('468,5', func=fast_float))  # Prints 468.5
```

Installation

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

```
easy_install fastnumbers
```

or, if you have `pip` (preferred over `easy_install`):

```
pip install fastnumbers
```

Both of the above commands will download the source for you.

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

If you choose to install from source (will need a C compiler and the Python headers), 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
```

fastnumbers requires python version 2.6 or greater (this includes python 3.x). Unit tests are only run on 2.6, 2.7 and >= 3.3.

CHAPTER 2

Timing

In order for you to see the benefit of `fastnumbers`, some timings are collected below for comparison to equivalent python implementations. The numbers may change depending on the machine you are on. Feel free to download the source code to run all timing tests.

Note that the test results are for Python 2.7. The trends are the same for Python 3.x, but the in general Python 3.x is faster at converting strings to integers than Python 2.7 so rather than a 4x speedup you will only see a 2x speedup for integers on Python 3.x compared to Python 2.7.

Additionally, because of the addition of allowing underscores in numbers in Python 3.6, conversions for both the Python built-in and `fastnumbers` are slower than earlier versions.

Built-in Functions Drop-in Replacement Timing Results

The following timing tests compare conversions to numbers using both the built-in Python *float* and *int* functions, and the `fastnumbers` drop-in replacement functions.

Timing Runner

```
import sys
from timeit import repeat

def mean(x):
    return sum(x) / len(x)

def time_input_against_all_functions(value, label, with_int=False):
    """Run the given input on all function types."""
    funcs = ('int', 'float',) if with_int else ('float',)
    fmt = '{func}({value!r})'
    for func in funcs:
        print(label + ',', "fn"+func + ':', end=' ')
        sys.stdout.flush()
        time_results = repeat(fmt.format(func=func, value=value),
                              'from fastnumbers import {}'.format(func),
                              repeat=10)
        time_results = mean(time_results)
        print(time_results, 'seconds')
        print(label + ',', func + ':', end=' ')
        sys.stdout.flush()
        time_results = repeat(fmt.format(func=func, value=value),
                              repeat=10)
        time_results = mean(time_results)
        print(time_results, 'seconds')
    print()


print('All timing results are the average of 10 runs.')
print()
time_input_against_all_functions('-41053', 'Int String', with_int=True)
time_input_against_all_functions('35892482945872302493947939485729', 'Large Int String
↪', with_int=True)
time_input_against_all_functions('-41053.543034e34', 'Float String')
time_input_against_all_functions('-41053.543028758302e256', 'Large Float String')
time_input_against_all_functions(-41053, 'Int', with_int=True)
time_input_against_all_functions(-41053.543028758302e100, 'Float')
```

The following are the results:

```
All timing results are the average of 10 runs.

Int String, fnint: 0.186911249161 seconds
Int String, int: 0.660544228554 seconds
Int String, fnfloat: 0.198802471161 seconds
Int String, float: 0.219639778137 seconds

Large Int String, fnint: 0.574236178398 seconds
Large Int String, int: 0.854444551468 seconds
Large Int String, fnfloat: 0.681066322327 seconds
Large Int String, float: 0.637784719467 seconds

Float String, fnfloat: 0.252382349968 seconds
Float String, float: 0.542794394493 seconds

Large Float String, fnfloat: 1.09777746201 seconds
Large Float String, float: 0.96131811142 seconds
```

```
Int, fnint: 0.153172326088 seconds
Int, int: 0.191195559502 seconds
Int, fnfloat: 0.157608604431 seconds
Int, float: 0.182482266426 seconds

Float, fnfloat: 0.141705727577 seconds
Float, float: 0.185694789886 seconds
```

Error-Handling and Checking Functions Timing Results

The following timing tests are for the functions that convert input to numbers but adds extra error-handing, and also the functions that check that an input can be converted to a number.

Timing Runner

The timing tests presented below use the following function to run the timings:

```
from timeit import repeat


def mean(x):
    return sum(x) / len(x)


def time_conv(regex, try_, fast):
    """\
    Run timing tests on multiple types of input,
    for multiple types of functions.
    """

    print('All timing results are the average of 10 runs.')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Non-number String,', x, mean(repeat('{}("not_a_number")'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Int String,', x, mean(repeat('{}("-41053")'.format(y[0]), y[1],␣
↪repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Large Int String,', x, mean(repeat('{}(
↪"35892482945872302493947939485729")'.format(y[0]), y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Float String,', x, mean(repeat('{}("-41053.543034e34")'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Large Float String,', x, mean(repeat('{}("-41053.543028758302e256")'.
↪format(y[0]), y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
```

```
        print('Float,', x, mean(repeat('{}(-41053.543028758302e100)'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Int,', x, mean(repeat('{}(-41053)'.format(y[0]), y[1], repeat=10)),
↪'seconds')


def time_test(regex, try_, fast):
    """\
    Run timing tests on multiple types of input,
    for multiple types of functions.
    """

    print('All timing results are the average of 10 runs.')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Non-number String,', x, mean(repeat('{}("not_a_number")'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Int String,', x, mean(repeat('{}("-41053")'.format(y[0]), y[1],␣
↪repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Large Int String,', x, mean(repeat('{}(
↪"35892482945872302493947939485729")'.format(y[0]), y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Float String,', x, mean(repeat('{}("-41053.543034e34")'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Large Float String,', x, mean(repeat('{}("-41053.543028758302e256")'.
↪format(y[0]), y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Float,', x, mean(repeat('{}(-41053.543028758302e100)'.format(y[0]),␣
↪y[1], repeat=10)), 'seconds')
    print()
    for x, y in zip(('re:', 'try:', 'fast:'), (regex, try_, fast)):
        print('Int,', x, mean(repeat('{}(-41053)'.format(y[0]), y[1], repeat=10)),
↪'seconds')
```

Converting Floats

The code to perform the *float* conversion timings is given below:

```
float_re = '''\
import re
float_regex = re.compile(r'[-+]?\d*\.?\d+(?:[eE][-+]?\d+)?$')
float_match = float_regex.match
def float_re(x):
    """Function to simulate safe_float but with regular expressions."""
    try:
        if float_match(x):
```

```
            return float(x)
        else:
            return x
    except TypeError:
        return float(x)
'''

float_try = '''\
def float_try(x):
    """Function to simulate safe_float but with try/except."""
    try:
        return float(x)
    except ValueError:
        return x
'''

time_conv(['float_re', float_re],
          ['float_try', float_try],
          ['fast_float', 'from fastnumbers import fast_float'])
```

The following are the results:

```
All timing results are the average of 10 runs.

Non-number String, re: 0.573668313026 seconds
Non-number String, try: 2.18123717308 seconds
Non-number String, fast: 0.169103908539 seconds

Int String, re: 1.14506702423 seconds
Int String, try: 0.376295471191 seconds
Int String, fast: 0.198381090164 seconds

Large Int String, re: 1.92678444386 seconds
Large Int String, try: 0.756599807739 seconds
Large Int String, fast: 0.650611639023 seconds

Float String, re: 1.82797636986 seconds
Float String, try: 0.687785792351 seconds
Float String, fast: 0.242390632629 seconds

Large Float String, re: 2.31674897671 seconds
Large Float String, try: 1.07994887829 seconds
Large Float String, fast: 0.970883083344 seconds

Float, re: 1.93790380955 seconds
Float, try: 0.333303022385 seconds
Float, fast: 0.133689498901 seconds

Int, re: 2.0137783289 seconds
Int, try: 0.326768898964 seconds
Int, fast: 0.140902447701 seconds
```

Converting Ints

The code to perform the *int* conversion timings is given below:

```
int_re = '''\
import re
int_regex = re.compile(r'[-+]?\d+$')
int_match = int_regex.match
def int_re(x):
    """Function to simulate safe_int but with regular expressions."""
    try:
        if int_match(x):
            return int(x)
        else:
            return x
    except TypeError:
        return int(x)
'''

int_try = '''\
def int_try(x):
    """Function to simulate safe_int but with try/except."""
    try:
        return int(x)
    except ValueError:
        return x
'''

time_conv(['int_re', int_re],
          ['int_try', int_try],
          ['fast_int', 'from fastnumbers import fast_int'])
```

The following are the results:

```
All timing results are the average of 10 runs.

Non-number String, re: 0.513276076317 seconds
Non-number String, try: 3.59187278748 seconds
Non-number String, fast: 0.154757094383 seconds

Int String, re: 1.62476005554 seconds
Int String, try: 0.857665300369 seconds
Int String, fast: 0.188158082962 seconds

Large Int String, re: 2.24101734161 seconds
Large Int String, try: 1.12569539547 seconds
Large Int String, fast: 0.643681025505 seconds

Float String, re: 0.732820224762 seconds
Float String, try: 3.60975520611 seconds
Float String, fast: 0.163253188133 seconds

Large Float String, re: 0.729602479935 seconds
Large Float String, try: 3.65623159409 seconds
Large Float String, fast: 0.163104772568 seconds

Float, re: 2.32226393223 seconds
Float, try: 0.539329576492 seconds
Float, fast: 0.290531635284 seconds

Int, re: 2.01958138943 seconds
Int, try: 0.326728582382 seconds
```

```
Int, fast: 0.130790877342 seconds
```

Checking Floats

The code to perform the *float* checking timings is given below:

```
isfloat_re = '''\
import re
float_regex = re.compile(r'[-+]?\d*\.?\d+(?:[eE][-+]?\d+)?$')
float_match = float_regex.match
nums = set([float, int])
def isfloat_re(x):
    """Function to simulate isfloat but with regular expressions."""
    t = type(x)
    return t == float if t in nums else bool(float_match(x))
'''

isfloat_try = '''\
def isfloat_try(x):
    """Function to simulate isfloat but with try/except."""
    try:
        float(x)
    except ValueError:
        return False
    else:
        return type(x) != int
'''

time_test(['isfloat_re', isfloat_re],
          ['isfloat_try', isfloat_try],
          ['isfloat', 'from fastnumbers import isfloat'])
```

The following are the results:

```
All timing results are the average of 10 runs.

Non-number String, re: 1.00798327923 seconds
Non-number String, try: 2.1671749115 seconds
Non-number String, fast: 0.161317801476 seconds

Int String, re: 1.25955090523 seconds
Int String, try: 0.568465399742 seconds
Int String, fast: 0.174383997917 seconds

Large Int String, re: 1.58316028118 seconds
Large Int String, try: 0.962632489204 seconds
Large Int String, fast: 0.204193854332 seconds

Float String, re: 1.43826227188 seconds
Float String, try: 0.947708177567 seconds
Float String, fast: 0.176113319397 seconds

Large Float String, re: 1.50998635292 seconds
Large Float String, try: 1.2951467514 seconds
Large Float String, fast: 0.182495999336 seconds
```

```
Float, re: 0.384200811386 seconds
Float, try: 0.526714110374 seconds
Float, fast: 0.132399153709 seconds

Int, re: 0.385046958923 seconds
Int, try: 0.525576925278 seconds
Int, fast: 0.138587331772 seconds
```

Checking Ints

The code to perform the *int* checking timings is given below:

```
isint_re = '''\
import re
int_regex = re.compile(r'[-+]?\d+$')
int_match = int_regex.match
nums = set([float, int])
def isint_re(x):
    """Function to simulate isint but with regular expressions."""
    t = type(x)
    return t == int if t in nums else bool(int_match(x))
'''

isint_try = '''\
def isint_try(x):
    """Function to simulate isint but with try/except."""
    try:
        int(x)
    except ValueError:
        return False
    else:
        return type(x) != float
'''

time_test(['isint_re', isint_re],
          ['isint_try', isint_try],
          ['isint', 'from fastnumbers import isint'])
```

The following are the results:

```
All timing results are the average of 10 runs.

Non-number String, re: 0.967393517494 seconds
Non-number String, try: 3.5071721077 seconds
Non-number String, fast: 0.142234826088 seconds

Int String, re: 1.07628540993 seconds
Int String, try: 1.07323606014 seconds
Int String, fast: 0.146803045273 seconds

Large Int String, re: 1.39286680222 seconds
Large Int String, try: 1.29413485527 seconds
Large Int String, fast: 0.190896821022 seconds

Float String, re: 1.15165970325 seconds
Float String, try: 3.55873417854 seconds
```

```
Float String, fast: 0.148036885262 seconds

Large Float String, re: 1.15159604549 seconds
Large Float String, try: 3.60109534264 seconds
Large Float String, fast: 0.145488548279 seconds

Float, re: 0.400713467598 seconds
Float, try: 0.685648226738 seconds
Float, fast: 0.129911446571 seconds

Int, re: 0.405667829514 seconds
Int, try: 0.530900597572 seconds
Int, fast: 0.127823591232 seconds
```

CHAPTER 3

fastnumbers API

The "Built-In Replacement" Functions

Each of these functions acts as a faster drop-in replacement for the equivalent Python built-in function.

float()

`fastnumbers.float(x=0)`

Drop-in but faster replacement for the built-in *float*.

Behaves identically to the built-in *float* except for the following:

- Is implemented as a function, not a class, which means it cannot be sub-classed, and has no *fromhex* classmethod.
- A *ValueError* will be raised instead of a *UnicodeEncodeError* if a partial surrogate is given as input.

int()

`fastnumbers.int(x=0, base=10)`

Drop-in but faster replacement for the built-in *int*.

Behaves identically to the built-in *int* except for the following:

- Cannot convert from the `__trunc__` special method of an object.
- Is implemented as a function, not a class, which means it cannot be sub-classed, and has no *from_bytes* classmethod.

real()

`fastnumbers.real(x=0.0, coerce=True)`

Convert to *float* or *int*, whichever is most appropriate.

If an *int* literal or string containing an *int* is provided, then an *int* will be returned. If a *float* literal or a string containing a non-*int* and non-*complex* number is provided, a *float* will be returned.

If *coerce* is *True* (the default), then if a *float* is given that has no decimal places after conversion or only zeros after the decimal point, it will be returned as an *int* instead of a *float*.

The "Error-Handling" Functions

Each of these functions will quickly convert strings to numbers (and also numbers to numbers) with fast and convenient error handling. They are guaranteed to return results identical to the built-in `float` or `int` functions.

fast_real()

`fastnumbers.fast_real(x, default=None, raise_on_invalid=False, key=None, nan=None, inf=None, coerce=True)`

Quickly convert input to an *int* or *float* depending on value.

Any input that is valid for the built-in *float* or *int* functions will be converted to either a *float* or *int*. An input of a single numeric unicode character is also valid. The return value is guaranteed to be of type *str*, *int*, or *float* (or *long* on Python2).

If the given input is a string and cannot be converted to a *float* or *int*, it will be returned as-is unless *default* or *raise_on_invalid* is given.

Parameters

- **input** (*{str, float, int, long}*) – The input you wish to convert to a real number.
- **default** (*optional*) – This value will be returned instead of the input when the input cannot be converted. Has no effect if *raise_on_invalid* is *True*.
- **raise_on_invalid** (*bool, optional*) – If *True*, a *ValueError* will be raised if string input cannot be converted to a *float* or *int*. If *False*, the string will be returned as-is. The default is *False*.
- **key** (*callable, optional*) – If given and the *input* cannot be converted, the input will be passed to the callable object and its return value will be returned. The function must take one and only one required argument.
- **nan** (*optional*) – If the input value is NAN or can be parsed as NAN, return this value instead of NAN.
- **inf** (*optional*) – If the input value is INF or can be parsed as INF, return this value instead of INF.
- **coerce** (*bool, optional*) – If the input can be converted to an *int* without loss of precision (even if the input was a *float* or float-containing *str*) coerce to an *int* rather than returning a *float*.

Returns **out** – If the input could be converted to an *int*, the return type will be *int* (or *long* on Python2 if the integer is large enough). If the input could be converted to a *float* but not an *int*, the return type will be *float*. Otherwise, the input *str* will be returned as-is (if *raise_on_invalid* is *False*) or whatever value is assigned to *default* if *default* is not *None*.

Return type {str, *float*, *int*, long}

Raises

- `TypeError` – If the input is not one of *str*, *float*, or *int* (or *long* on Python2).
- `ValueError` – If *raise_on_invalid* is *True*, this will be raised if the input string cannot be converted to a *float* or *int*.

See also:

isreal(), *real()*

Examples

```
>>> from fastnumbers import fast_real
>>> fast_real('56')
56
>>> fast_real('56.0')
56
>>> fast_real('56.0', coerce=False)
56.0
>>> fast_real('56.07')
56.07
>>> fast_real('56.07 lb')
'56.07 lb'
>>> fast_real(56.07)
56.07
>>> fast_real(56.0)
56
>>> fast_real(56.0, coerce=False)
```

```
56.0
>>> fast_real(56)
56
>>> fast_real('invalid', default=50)
50
>>> fast_real('invalid', 50)  # 'default' is first optional positional arg
50
>>> fast_real('nan')
nan
>>> fast_real('nan', nan=0)
0
>>> fast_real('56.07', nan=0)
56.07
>>> fast_real('56.07 lb', raise_on_invalid=True)
Traceback (most recent call last):
  ...
ValueError: could not convert string to float: '56.07 lb'
>>> fast_real('invalid', key=len)
7
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> def py_fast_real(input, default=None, raise_on_invalid=False,
...                  key=None, nan=None, inf=None):
...     import math
...     try:
...         a = float(input)
...     except ValueError:
...         if raise_on_invalid:
...             raise
...         elif key is not None:
...             return key(input)
...         elif default is not None:
...             return default
...         else:
...             return input
...     else:
...         if nan is not None and math.isnan(a):
...             return nan
...         elif inf is not None and math.isinf(a):
...             return inf
...         else:
...             return int(a) if a.is_integer() else a
...
```

fast_float()

`fastnumbers.`**`fast_float`**(*x*, *default=None*, *raise_on_invalid=False*, *key=None*, *nan=None*, *inf=None*)

Quickly convert input to a *float*.

Any input that is valid for the built-in *float* function will be converted to a *float*. An input of a single numeric unicode character is also valid. The return value is guaranteed to be of type *str* or *float*.

If the given input is a string and cannot be converted to a *float* it will be returned as-is unless *default* or *raise_on_invalid* is given.

Parameters

- **input** (*{str, float, int, long}*) – The input you wish to convert to a *float*.
- **default** (*optional*) – This value will be returned instead of the input when the input cannot be converted. Has no effect if *raise_on_invalid* is *True*.
- **raise_on_invalid** (*bool, optional*) – If *True*, a *ValueError* will be raised if string input cannot be converted to a *float*. If *False*, the string will be returned as-is. The default is *False*.
- **key** (*callable, optional*) – If given and the *input* cannot be converted, the input will be passed to the callable object and its return value will be returned. The function must take one and only one required argument.
- **nan** (*optional*) – If the input value is NAN or can be parsed as NAN, return this value instead of NAN.
- **inf** (*optional*) – If the input value is INF or can be parsed as INF, return this value instead of INF.

Returns **out** – If the input could be converted to a *float* the return type will be *float*. Otherwise, the input *str* will be returned as-is (if *raise_on_invalid* is *False*) or whatever value is assigned to *default* if *default* is not *None*.

Return type {str, float}

Raises

- `TypeError` – If the input is not one of *str*, *float*, or *int* (or *long* on Python2).
- `ValueError` – If *raise_on_invalid* is *True*, this will be raised if the input string cannot be converted to a *float*.

See also:

isfloat(), *float()*

Examples

```
>>> from fastnumbers import fast_float
>>> fast_float('56')
56.0
>>> fast_float('56.0')
56.0
>>> fast_float('56.07')
56.07
>>> fast_float('56.07 lb')
'56.07 lb'
>>> fast_float(56.07)
56.07
>>> fast_float(56)
56.0
>>> fast_float('invalid', default=50)
50
>>> fast_float('invalid', 50)  # 'default' is first optional positional arg
50
>>> fast_float('nan')
```

```
nan
>>> fast_float('nan', nan=0.0)
0.0
>>> fast_float('56.07', nan=0.0)
56.07
>>> fast_float('56.07 lb', raise_on_invalid=True)
Traceback (most recent call last):
  ...
ValueError: could not convert string to float: '56.07 lb'
>>> fast_float('invalid', key=len)
7
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> def py_fast_float(input, default=None, raise_on_invalid=False,
...                   key=None, nan=None, inf=None):
...     try:
...         x = float(input)
...     except ValueError:
...         if raise_on_invalid:
...             raise
...         elif key is not None:
...             return key(input)
...         elif default is not None:
...             return default
...         else:
...             return input
...     else:
...         if nan is not None and math.isnan(x):
...             return nan
...         elif inf is not None and math.isinf(x):
...             return inf
...         else:
...             return x
...
```

fast_int()

`fastnumbers.`**`fast_int`**(*x*, *default=None*, *raise_on_invalid=False*, *key=None*, *base=10*)

Quickly convert input to an *int*.

Any input that is valid for the built-in *int* (or *long* on Python2) function will be converted to a *int* (or *long* on Python2). An input of a single digit unicode character is also valid. The return value is guaranteed to be of type *str* or *int* (or *long* on Python2).

If the given input is a string and cannot be converted to an *int* it will be returned as-is unless *default* or *raise_on_invalid* is given.

Parameters

- **input** (*{str, float, int, long}*) – The input you wish to convert to an *int*.
- **default** (*optional*) – This value will be returned instead of the input when the input cannot be converted. Has no effect if *raise_on_invalid* is *True*.

- **raise_on_invalid** (*bool, optional*) – If *True*, a *ValueError* will be raised if string input cannot be converted to an *int*. If *False*, the string will be returned as-is. The default is *False*.
- **key** (*callable, optional*) – If given and the *input* cannot be converted, the input will be passed to the callable object and its return value will be returned. The function must take one and only one required argument.
- **base** (*int, optional*) – Follows the rules of Python's built-in *int()*; see it's documentation for your Python version. If given, the input **must** be of type *str*.

Returns **out** – If the input could be converted to an *int*, the return type will be *int* (or *long* on Python2 if the integer is large enough). Otherwise, the input *str* will be returned as-is (if *raise_on_invalid* is *False*) or whatever value is assigned to *default* if *default* is not *None*.

Return type {str, *int*, long}

Raises

- TypeError – If the input is not one of *str*, *float*, or *int* (or *long* on Python2).
- ValueError – If *raise_on_invalid* is *True*, this will be raised if the input string cannot be converted to an *int*.

See also:

fast_forceint(), *isint()*, *int()*

Examples

```
>>> from fastnumbers import fast_int
>>> fast_int('56')
56
>>> fast_int('56.0')
'56.0'
>>> fast_int('56.07 lb')
'56.07 lb'
>>> fast_int(56.07)
56
>>> fast_int(56)
56
>>> fast_int('invalid', default=50)
50
>>> fast_int('invalid', 50)  # 'default' is first optional positional arg
50
>>> fast_int('56.07 lb', raise_on_invalid=True)
Traceback (most recent call last):
  ...
ValueError: could not convert string to int: '56.07 lb'
>>> fast_int('invalid', key=len)
7
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> def py_fast_int(input, default=None, raise_on_invalid=False, key=None):
...     try:
...         return int(input)
...     except ValueError:
...         if raise_on_invalid:
...             raise
...         elif key is not None:
...             return key(input)
...         elif default is not None:
...             return default
...         else:
...             return input
...
```

fast_forceint()

fastnumbers.**fast_forceint**(*x*, *default=None*, *raise_on_invalid=False*, *key=None*)

Quickly convert input to an *int*, truncating if is a *float*.

Any input that is valid for the built-in *int* (or *long* on Python2) function will be converted to a *int* (or *long* on Python2). An input of a single numeric unicode character is also valid. The return value is guaranteed to be of type *str* or *int* (or *long* on Python2).

In addition to the above, any input valid for the built-in *float* will be parsed and the truncated to the nearest integer; for example, '56.07' will be converted to *56*.

If the given input is a string and cannot be converted to an *int* it will be returned as-is unless *default* or *raise_on_invalid* is given.

Parameters

- **input** (*{str, float, int, long}*) – The input you wish to convert to an *int*.
- **default** (*optional*) – This value will be returned instead of the input when the input cannot be converted. Has no effect if *raise_on_invalid* is *True*
- **raise_on_invalid** (*bool, optional*) – If *True*, a *ValueError* will be raised if string input cannot be converted to an *int*. If *False*, the string will be returned as-is. The default is *False*.
- **key** (*callable, optional*) – If given and the *input* cannot be converted, the input will be passed to the callable object and its return value will be returned. The function must take one and only one required argument.

Returns **out** – If the input could be converted to an *int*, the return type will be *int* (or *long* on Python2 if the integer is large enough). Otherwise, the input *str* will be returned as-is (if *raise_on_invalid* is *False*) or whatever value is assigned to *default* if *default* is not *None*.

Return type {str, *int*, long}

Raises

- TypeError – If the input is not one of *str*, *float*, or *int* (or *long* on Python2).
- ValueError – If *raise_on_invalid* is *True*, this will be raised if the input string cannot be converted to an *int*.

See also:

fast_int(), *isintlike()*

Examples

```
>>> from fastnumbers import fast_forceint
>>> fast_forceint('56')
56
>>> fast_forceint('56.0')
56
>>> fast_forceint('56.07')
56
>>> fast_forceint('56.07 lb')
'56.07 lb'
>>> fast_forceint(56.07)
56
>>> fast_forceint(56)
56
>>> fast_forceint('invalid', default=50)
50
>>> fast_forceint('invalid', 50)  # 'default' is first optional positional arg
50
>>> fast_forceint('56.07 lb', raise_on_invalid=True)
Traceback (most recent call last):
  ...
ValueError: could not convert string to float: '56.07 lb'
>>> fast_forceint('invalid', key=len)
7
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> def py_fast_forceint(input, default=None, raise_on_invalid=False, key=None):
...     try:
...         return int(input)
...     except ValueError:
...         try:
...             return int(float(input))
...         except ValueError:
...             if raise_on_invalid:
...                 raise
...             elif key is not None:
...                 return key(input)
...             elif default is not None:
...                 return default
...             else:
...                 return input
...
```

The "Checking" Functions

These functions return a Boolean value that indicates if the input can return a certain number type or not.

isreal()

`fastnumbers.isreal`(*x*, *str_only=False*, *num_only=False*, *allow_inf=False*, *allow_nan=False*)

Quickly determine if a string is a real number.

Returns *True* if the input is valid input for the built-in *float* or *int* functions, or is a single valid numeric unicode character.

The input may be whitespace-padded.

Parameters

- **input** – The input you wish to test if it is a real number.
- **str_only** (*bool, optional*) – If *True*, then any non-*str* input will cause this function to return *False*. The default is *False*.
- **num_only** (*bool, optional*) – If *True*, then any *str* input will cause this function to return *False*. The default is *False*.
- **allow_inf** (*bool, optional*) – If *True*, then the strings 'inf' and 'infinity' will also return *True*. This check is case-insensitive, and the string may be signed (i.e. '+/-'). The default is *False*.
- **allow_nan** (*bool, optional*) – If *True*, then the string 'nan' will also return *True*. This check is case-insensitive, and the string may be signed (i.e. '+/-'). The default is *False*.

Returns **result** – Whether or not the input is a real number.

Return type bool

See also:

fast_real()

Examples

```
>>> from fastnumbers import isreal
>>> isreal('56')
True
>>> isreal('56.07')
True
>>> isreal('56.07', num_only=True)
False
>>> isreal('56.07 lb')
False
>>> isreal(56.07)
True
>>> isreal(56.07, str_only=True)
False
>>> isreal(56)
True
>>> isreal('nan')
False
>>> isreal('nan', allow_nan=True)
True
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> import re
>>> def py_isreal(input, str_only=False, num_only=False,
...               allow_nan=False, allow_inf=False):
...     if str_only and type(input) != str:
...         return False
...     if num_only and type(input) not in (float, int):
...         return False
...     try:
...         x = bool(re.match(r'[-+]?\d*\.?\d+(?:[eE][-+]?\d+)?$', input))
...     except TypeError:
...         return type(input) in (float, int)
...     else:
...         if x:
...             return True
...         elif allow_inf and input.lower().strip().lstrip('-+') in ('inf',
↪'infinity'):
...             return True
...         elif allow_nan and input.lower().strip().lstrip('-+') == 'nan':
...             return True
...         else:
...             return False
...
```

isfloat()

`fastnumbers.isfloat`(*x*, *str_only=False*, *num_only=False*, *allow_inf=False*, *allow_nan=False*)

Quickly determine if a string is a *float*.

Returns *True* if the input is valid input for the built-in *float* function, is already a valid *float*, or is a single valid numeric unicode character. It differs from *isreal* in that an *int* input will return *False*.

The input may be whitespace-padded.

Parameters

- **input** – The input you wish to test if it is a *float*.
- **str_only** (*bool, optional*) – If *True*, then any non-*str* input will cause this function to return *False*. The default is *False*.
- **num_only** (*bool, optional*) – If *True*, then any *str* input will cause this function to return *False*. The default is *False*.
- **allow_inf** (*bool, optional*) – If *True*, then the strings 'inf' and 'infinity' will also return *True*. This check is case-insensitive, and the string may be signed (i.e. '+/-'). The default is *False*.
- **allow_nan** (*bool, optional*) – If *True*, then the string 'nan' will also return *True*. This check is case-insensitive, and the string may be signed (i.e. '+/-'). The default is *False*.

Returns **result** – Whether or not the input is a *float*.

Return type bool

See also:

fast_float(), *isreal()*

Examples

```
>>> from fastnumbers import isfloat
>>> isfloat('56')
True
>>> isfloat('56.07')
True
>>> isreal('56.07', num_only=True)
False
>>> isfloat('56.07 lb')
False
>>> isfloat(56.07)
True
>>> isfloat(56.07, str_only=True)
False
>>> isfloat(56)
False
>>> isfloat('nan')
False
>>> isfloat('nan', allow_nan=True)
True
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> import re
>>> def py_isfloat(input, str_only=False, num_only=False,
...                allow_nan=False, allow_inf=False):
...     if str_only and type(input) != str:
...         return False
...     if num_only and type(input) != float:
...         return False
...     try:
...         x = bool(re.match(r'[-+]?\d*\.?\d+(?:[eE][-+]?\d+)?$', input))
...     except TypeError:
...         return type(input) == float
...     else:
...         if x:
...             return True
...         elif allow_inf and input.lower().strip().lstrip('-+') in ('inf',
↪'infinity'):
...             return True
...         elif allow_nan and input.lower().strip().lstrip('-+') == 'nan':
...             return True
...         else:
...             return False
```

isint()

`fastnumbers.isint`(*x*, *str_only=False*, *num_only=False*)

Quickly determine if a string is an *int*.

Returns *True* if the input is valid input for the built-in *int* function, is already a valid *int*, or is a single valid digit unicode character. It differs from *isintlike* in that a *float* input will return *False* and that *int*-like strings (i.e. '45.0') will return *False*.

The input may be whitespace-padded.

Parameters

- **input** – The input you wish to test if it is an *int*.
- **str_only** (*bool, optional*) – If *True*, then any non-*str* input will cause this function to return *False*. The default is *False*.
- **num_only** (*bool, optional*) – If *True*, then any *str* input will cause this function to return *False*. The default is *False*.

Returns **result** – Whether or not the input is an *int*.

Return type bool

See also:

fast_int(), *isintlike()*

Examples

```
>>> from fastnumbers import isint
>>> isint('56')
True
>>> isint('56', num_only=True)
False
>>> isint('56.07')
False
>>> isint('56.07 lb')
False
>>> isint(56.07)
False
>>> isint(56)
True
>>> isint(56, str_only=True)
False
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> import re
>>> def py_isint(input, str_only=False, num_only=False):
...     if str_only and type(input) != str:
...         return False
...     if num_only and type(input) != int:
...         return False
...     try:
...         return bool(re.match(r'[-+]?\d+$', input))
...     except TypeError:
...         return False
...
```

isintlike()

`fastnumbers.isintlike(x, str_only=False, num_only=False)`

Quickly determine if a string (or object) is an *int* or *int*-like.

Returns *True* if the input is valid input for the built-in *int* function, is already a valid *int* or *float*, or is a single valid numeric unicode character. It differs from *isintlike* in that *int*-like floats or strings (i.e. '45.0') will return *True*.

The input may be whitespace-padded.

Parameters **input** – The input you wish to test if it is a *int*-like.

Returns

- **result** (*bool*) – Whether or not the input is an *int*.
- **str_only** (*bool, optional*) – If *True*, then any non-*str* input will cause this function to return *False*. The default is *False*.
- **num_only** (*bool, optional*) – If *True*, then any *str* input will cause this function to return *False*. The default is *False*.

See also:

fast_forceint()

Examples

```
>>> from fastnumbers import isintlike
>>> isintlike('56')
True
>>> isintlike('56', num_only=True)
False
>>> isintlike('56.07')
False
>>> isintlike('56.0')
True
>>> isintlike('56.07 lb')
False
>>> isintlike(56.07)
False
>>> isintlike(56.0)
True
>>> isintlike(56.0, str_only=True)
False
>>> isintlike(56)
True
```

Notes

It is roughly equivalent to (but much faster than)

```
>>> import re
>>> def py_isintlike(input, str_only=False, num_only=False):
...     if str_only and type(input) != str:
...         return False
...     if num_only and type(input) not in (int, float):
```

```
...         return False
...     try:
...         if re.match(r'[-+]?\d+$', input):
...             return True
...         elif re.match(r'[-+]?\d*\.?\d+(?:[eE][-+]?\d+)?$', input):
...             return float(input).is_integer()
...         else:
...             return False
...     except TypeError:
...         if type(input) == float:
...             return input.is_integer()
...         elif type(input) == int:
...             return True
...         else:
...             return False
...
```

CHAPTER 4

Changelog

04-30-2017 v. 2.0.1

- Fixed bug in decimal digit limit on GCC.

04-30-2017 v. 2.0.0

- Dropped support for Python 2.6.
- Added support for Python 3.6 underscores.
- Added drop-in replacements for the built-in `int()` and `float()` functions.
- Incorporated unit tests from Python's testing library to ensure that any input that Python can handle will also be handled the same way by `fastnumbers`.
- Added Appveyor testing to ensure no surprises on Windows.
- Revamped documentation.
- Refactored internal mechanism for assessing overflow to be faster in the most common cases.

04-23-2016 v. 1.0.0

- "coerce" in `fast_real` now applies to any input, not just numeric; the default is now *True* instead of *False*.
- Now all ASCII whitespace characters are stripped by fastnumbers
- Typechecking is now more forgiving
- fastnumbers now checks for errors when converting between numeric types
- Fixed bug where very small numbers are not converted properly

- Testing now includes Python 2.6.
- Removed `safe_*` functions (which were deprecated since version 0.3.0)
- Fixed unicode handling on Windows.
- Fixed Python2.6 on Windows.

03-19-2016 v. 0.7.4

- Added the "coerce" option to fast_real.

03-08-2016 v. 0.7.3

- Newline is now considered to be whitespace (for consistency with the builtin float and int).

03-07-2016 v. 0.7.2

- Fixed overflow bug in exponential parts of floats.

02-29-2016 v. 0.7.1

- Fixed compilation bug with MSVC.
- Added "key" function to transform invalid input arguments.

01-18-2016 v. 0.7.0

- Broke all functions into smaller components, eliminating a lot of duplication.
- Sped up functions by eliminating an unnecessary string copy.
- Improved documentation.

11-01-2015 v. 0.6.2

- Fixed bug that caused a SystemError exception to be raised on Python 3.5 if a very large int was passed to the "fast" functions.

10-29-2015 v. 0.6.1

- Fixed segfault on Python 3.5 that seemed to be related to a change in the PyObject_CallMethod C function.
- Sped up unit testing.
- Added tox.ini.

10-27-2015 v. 0.6.0

- Fixed issue where giving a default of *None* would be ignored.
- Added the "nan" and "inf" options to `fast_real` and `fast_float`. These options allow alternate return values in the case of *nan* or *inf*, respectively.
- Improved documentation.
- Improved testing.

06-11-2015 v. 0.5.2

- Fixed compile error Visual Studio compilers.

06-04-2015 v. 0.5.1

- Solved rare segfault when parsing Unicode input.
- Made handling of Infinity and NaN for `fast_int` and `fast_forceint` consistent with the `built-in` int function.

05-12-2015 v. 0.5.0

- Made 'default' first optional argument instead of 'raise_on_invalid' for conversion functions.
- Added 'num_only' option for checker functions.

05-03-2015 v. 0.4.0

- Added support for conversion of single Unicode characters that represent numbers and digits.

04-23-2015 v. 0.3.0

- Updated the `fast_*` functions to check if an overflow loss of precision has occurred, and if so fall back on the more accurate number conversion method.
- In response to the above change, the `safe_*` functions are now deprecated, and internally now use the same code as the `fast_*` functions.
- Updated all unit testing to use the `hypothesis` module, which results in better test coverage.

09-03-2014 v. 0.2.0

- Added a 'default' option to the conversion functions.

08-12-2014 v. 0.1.4

- Fixed bug where '.' was incorrectly identified as a valid float/int and converted to 0. This bug only applied to the `fast_*` and `is*` functions.
- The method to catch corner-cases like '.', '+', 'e', etc. has been reworked to be more general... case-by-case patches should no longer be needed.

08-12-2014 v. 0.1.3

- Fixed bug where 'e' and 'E' were incorrectly identified as a valid float/int and converted to 0. This bug only applied to the `fast_*` and `is*` functions.

08-12-2014 v. 0.1.2

- Fixed bug where '+' and '-' were incorrectly identified as a valid float/int and converted to 0. This bug only applied to the `fast_*` and `is*` functions.
- Fixed bug where 'safe_forceint' did not handle 'nan' correctly.

08-11-2014 v. 0.1.1

- 'fastnumbers' now understands 'inf' and 'nan'.

08-10-2014 v. 0.1.0

- Initial release of 'fastnumbers'.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

F

I

R