6.5 SymPy Docstrings Style Guide

6.5.1 General Guidelines

To contribute to SymPy's docstrings, please read these guidelines in full.

A documentation string (docstring) is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

Example

Here is a basic docstring:

```
def fdiff(self, argindex=1):
    """
    Returns the first derivative of a Heaviside Function.

    Examples
    ========

    >>> from sympy import Heaviside, diff
    >>> from sympy.abc import x

    >>> Heaviside(x).fdiff()
    DiracDelta(x)

    >>> Heaviside(x**2 - 1).fdiff()
    DiracDelta(x**2 - 1)

    >>> diff(Heaviside(x)).fdiff()
    DiracDelta(x, 1)

    """
```

Every public function, class, method, and module should have a docstring that describes what it does. Documentation that is specific to a function or class in the module should be in the docstring of that function or class. The module level docstring should discuss the purpose and scope of the module, and give a high-level example of how to use the functions or classes in the module. A module docstring is the docstring at the very top of the file, for example, the docstring for solvers.ode.

A public function is one that is intended to be used by the end- user, or the public. Documentation is important for public functions because they will be seen and used by many people.

A private function, on the other hand, is one that is only intended to be used in the code in SymPy itself. Although it is less important to document private functions, it also helps to have docstrings on private functions to help other SymPy developers understand how to use the function.

It may not always be clear what is a public function and what is a private function. If a function begins with an underscore, it is private, and if a function is included in `__init__.py` it is public, but the converse is not always true, so sometimes you have to decide based on context. In general, if you are unsure, having documentation on a function is better than not having documentation, regardless if it is public or private.

Docstrings should contain information aimed at users of the function. Comments specific to the code or other notes that would only distract users should go in comments in the code, not in docstrings.

Every docstring should have examples that show how the function works. Examples are the most important part of a docstring. A single example showing input and output to a function can be more helpful than a paragraph of descriptive text.

Remember that the primary consumers of docstrings are other human beings, not machines, so it is important to describe what the function does in plain English. Likewise, examples of how to use the function should be designed for human readers, not just for the doctest machinery.

Keep in mind that while Sphinx is the primary way users consume docstrings, and therefore the first platform to keep in mind while writing docstrings (especially for public functions), it is not the only way users consume docstrings. You can also view docstrings using `help()` or `?` in IPython. When using `help()`, for instance, it will show you all of the docstrings on private methods. Additionally, anyone reading the source code directly will see every docstring.

All public functions, classes, and methods and their corresponding docstrings should be imported into the Sphinx docs, instructions on which can be found at the end of this guide.

6.5.2 Formatting

Docstrings are are written in reStructuredText format extended by Sphinx. Here is a concise guide to Quick reStructuredText. More in-depth information about using reStructuredText can be found in the Sphinx Documentation.

In order for Sphinx to render docstrings nicely in the HTML documentation, some formatting guidelines should be followed when writing docstrings:

- Always use """triple double quotes""" around docstrings. Use r"""raw triple double quotes""" if you use any backslashes in your docstrings.
- Include a blank line before the docstring's closing quotes.
- Lines should not be longer than 80 characters.
- Always write class-level docstrings under the class definition line, as that is better to read in the source code.
- The various methods on the class can be mentioned in the docstring or examples if they are important, but details on them should go in the docstring for the method itself.
- Be aware that :: creates code blocks, which are rarely used in the docstrings. Any code example with example Python should be put in a doctest. Always check that the final version as rendered by Sphinx looks correct in the HTML.
- In order to make section underlining work nicely in docstrings, numpydoc Sphinx extension is used.
- Always double check that you have formatted your docstring correctly:

1. Make sure that your docstring is imported into Sphinx.
2. Build the Sphinx docs (`cd doc; make html`).
3. Make sure that Sphinx doesn't output any errors.
4. Open the page in `_build/html` and make sure that it is formatted correctly.

6.5.3 Sections

In SymPy's docstrings, it is preferred that function, class, and method docstrings consist of the following sections in this order:

1. Single-Sentence Summary
2. Explanation
3. Examples
4. Parameters
5. See Also
6. References

The Single-Sentence Summary and Examples sections are **required** for every docstring. Docstrings will not pass review if these sections are not included.

Do not change the names of these supported sections, for example, the heading "Examples" as a plural should be used even if there is only one example.

SymPy will continue to support all of the section headings listed in the NumPy Docstring Guide.

Headings should be underlined with the same length in equals signs.

If a section is not required and that information for the function in question is unnecessary, do not use it. Unnecessary sections and cluttered docstrings can make a function harder to understand. Aim for the minimal amount of information required to understand the function.

1. Single-Sentence Summary

This section is **required** for every docstring. A docstring will not pass review if it is not included. No heading is necessary for this section.

This section consists of a one-line sentence ending in a period that describes the function, class, or method's effect.

Deprecation warnings should go directly after the Single-Sentence Summary, so as to notify users right away. Deprecation warnings should be written as a `deprecated` in the Sphinx directive:

```
.. deprecated:: 1.1

   The ``simplify_this`` function is deprecated. Use :func:`simplify`
   instead. See its documentation for more information.
```

See *Documenting a deprecation* (page 3022) for more details.

2. Explanation Section

This section is encouraged. If you choose to include an Explanation section in your docstring, it should be labeled with the heading "Explanation" underlined with the same length in equals signs.

```
Explanation
===========
```

This section consists of a more elaborate description of what the function, class, or method does when the concise Single-Sentence Summary will not suffice. This section should be used to clarify functionality in several sentences or paragraphs.

3. Examples Section

This section is **required** for every docstring. A docstring will not pass review if it is not included. It should be labeled with the heading "Examples" (even if there is only one example) underlined with the same length in equals signs.

```
Examples
========
```

This section consists of examples that show how the function works, called doctests. Doctests should be complicated enough that they fully demonstrate the API and functionality of the function, but simple enough that a user can understand them without too much thought. The perfect doctest tells the user exactly what they need to know about the function without reading any other part of the docstring.

There should always be a blank line before the doctest. When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them.

Do not think of doctests as tests. Think of them as examples that happen to be tested. They should demonstrate the API of the function to the user (i.e., what the input parameters look like, what the output looks like, and what it does). If you only want to test something, add a test to the relevant `test_*.py file`.

You can use the `./bin/coverage_doctest.py` script to test the doctest coverage of a file or module. Run the doctests with `./bin/doctest`.

You should only skip the testing of an example if it is impossible to test it. If necessary, testing of an example can be skipped by adding a special comment.

Example

```
>>> import random
>>> random.random()
0.6868680200532414
```

If an example is longer than 80 characters, it should be line wrapped. Examples should be line wrapped so that they are still valid Python code, using `...` continuation as in a Python prompt. For example, from the ODE module documentation:

Example

```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

Here `dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x), f(x), hint='1st_exact')` is too long, so we line break it after a comma so that it is readable, and put `...` on the continuation lines. If this is not done correctly, the doctests will fail.

The output of a command can also be line wrapped. No `...` should be used in this case. The doctester automatically accepts output that is line wrapped.

Example

```
>>> list(range(30))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

In a doctest, write imports like `sympy import ...` instead of `import sympy` or `from sympy import *`. To define symbols, use `from sympy.abc import x`, unless the name is not in `sympy.abc` (for instance, if it has assumptions), in which case use `symbols` like `x, y = symbols('x y')`.

In general, you should run `./bin/doctest` to make sure your examples run correctly, and fix them if they do not.

4. Parameters Section

This section is encouraged. If you choose to include a Parameters section in your docstring, it should be labeled with the heading "Parameters" underlined with the same length in equals signs.

```
Parameters
==========
```

If you have parameters listed in parentheses after a function, class, or method name, you must include a parameters section.

This section consists of descriptions of the function arguments, keywords, and their respective types.

Enclose variables in double backticks. The colon must be preceded by a space, or omitted if the type is absent. For the parameter types, be as precise as possible. If it is not necessary to specify a keyword argument, use `optional`. Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description.

When a parameter can only assume one of a fixed set of values, those values can be listed in braces, with the default appearing first. When two or more input parameters have exactly the same type, shape, and description, they can be combined.

If the Parameters section is not formatted correctly, the documentation build will render incorrectly.

If you wish to include a Returns section, write it as its own section with its own heading.

Example

Here is an example of a correctly formatted Parameters section:

```
def opt_cse(exprs, order='canonical'):
    """
    Find optimization opportunities in Adds, Muls, Pows and negative
    coefficient Muls.

    Parameters
    ==========

    exprs : list of sympy expressions
        The expressions to optimize.
    order : string, 'none' or 'canonical'
        The order by which Mul and Add arguments are processed. For large
        expressions where speed is a concern, use the setting order='none'.

    """
```

5. See Also Section

This section is encouraged. If you choose to include a See Also section in your docstring, it should be labeled with the heading "See Also" underlined with the same length in equals signs.

```
See Also
========
```

This section consists of a listing of related functions, classes, and methods. The related items can be described with a concise fragment (not a full sentence) if desired, but this is not required. If the description spans more than one line, subsequent lines must be indented.

The See Also section should only be used to reference other SymPy objects. Anything that is a link should be embedded as a hyperlink in the text of the docstring instead; see the References section for details.

Do not reference classes with `class:Classname`, `class:\`Classname\``, or `:class:\`Classname\``, but rather only by their class name.

Examples

Here is a correctly formatted See Also section with concise descriptions:

```
class erf(Function):
    r"""
    The Gauss error function.

    See Also
    ========

    erfc: Complementary error function.
    erfi: Imaginary error function.
```

(continues on next page)

(continued from previous page)

```
    erf2: Two-argument error function.
    erfinv: Inverse error function.
    erfcinv: Inverse Complementary error function.
    erf2inv: Inverse two-argument error function.

    """
```

Here is a correctly formatted See Also section with just a list of names:

```
class besselj(BesselBase):
    r"""
    Bessel function of the first kind.

    See Also
    ========

    bessely, besseli, besselk

    """
```

6. References Section

This section is encouraged. If you choose to include a References section in your docstring, it should be labeled with the heading "References" underlined with the same length in equals signs.

```
References
==========
```

This section consists of a list of references cited anywhere in the previous sections. Any reference to other SymPy objects should go in the See Also section instead.

The References section should include online resources, paper citations, and/or any other printed resource giving general information about the function. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

For online resources, only link to freely accessible and stable online resources such as Wikipedia, Wolfram MathWorld, and the NIST Digital Library of Mathematical Functions (DLMF), which are unlikely to suffer from hyperlink rot.

References for papers should include, in this order: reference citation, author name, title of work, journal or publication, year published, page number.

If there is a DOI (Digital Object Identifier), include it in the citation and make sure it is a clickable hyperlink.

Examples

Here is a References section that cites a printed resource:

```
References
==========

```

(continues on next page)

(continued from previous page)

```
.. [1] [Kozen89] D. Kozen, S. Landau, Polynomial Decomposition Algorithms,
       Journal of Symbolic Computation 7 (1989), pp. 445-456
```

Here is a References section that cites printed and online resources:

```
References
==========

.. [1] Abramowitz, Milton; Stegun, Irene A., "Chapter 9," Handbook of
       Mathematical Functions with Formulas, Graphs, and Mathematical
       Tables, eds. (1965)
.. [2] Luke, Y. L., The Special Functions and Their Approximations,
       Volume 1, (1969)
.. [3] https://en.wikipedia.org/wiki/Bessel_function
.. [4] http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/
```

6.5.4 Sample Docstring

Here is an example of a correctly formatted docstring:

```
class gamma(Function):
    r"""
    The gamma function

    .. math::
       \Gamma(x) := \int^{\infty}_{0} t^{x-1} e^{-t} \mathrm{d}t.

    Explanation
    ===========

    The ``gamma`` function implements the function which passes through the
    values of the factorial function (i.e., $\Gamma(n) = (n - 1)!$), when n
    is an integer. More generally, $\Gamma(z)$ is defined in the whole
    complex plane except at the negative integers where there are simple
    poles.

    Examples
    ========

    >>> from sympy import S, I, pi, oo, gamma
    >>> from sympy.abc import x

    Several special values are known:

    >>> gamma(1)
    1
    >>> gamma(4)
    6
    >>> gamma(S(3)/2)
    sqrt(pi)/2
```

(continues on next page)

(continued from previous page)

```
    The ``gamma`` function obeys the mirror symmetry:

    >>> from sympy import conjugate
    >>> conjugate(gamma(x))
    gamma(conjugate(x))

    Differentiation with respect to $x$ is supported:

    >>> from sympy import diff
    >>> diff(gamma(x), x)
    gamma(x)*polygamma(0, x)

    Series expansion is also supported:

    >>> from sympy import series
    >>> series(gamma(x), x, 0, 3)
    1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-
↪EulerGamma*pi**2/12 +
    polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)

    We can numerically evaluate the ``gamma`` function to arbitrary
    precision on the whole complex plane:

    >>> gamma(pi).evalf(40)
    2.288037795340032417959588909060233922890
    >>> gamma(1+I).evalf(20)
    0.49801566811835604271 - 0.15494982830181068512*I

    See Also
    ========

    lowergamma: Lower incomplete gamma function.
    uppergamma: Upper incomplete gamma function.
    polygamma: Polygamma function.
    loggamma: Log Gamma function.
    digamma: Digamma function.
    trigamma: Trigamma function.
    beta: Euler Beta function.

    References
    ==========

    .. [1] https://en.wikipedia.org/wiki/Gamma_function
    .. [2] http://dlmf.nist.gov/5
    .. [3] http://mathworld.wolfram.com/GammaFunction.html
    .. [4] http://functions.wolfram.com/GammaBetaErf/Gamma/

    """
```

6.5.5 Docstrings for Classes that are Mathematical Functions

SymPy is unusual in that it also has classes that are mathematical functions. The docstrings for classes that are mathematical functions should include details specific to this kind of class, as noted below:

- The Explanation section should include a mathematical definition of the function. This should use LaTeX math. Use $$ for *inline math* (page 3013) and .. math:: for display math, which should be used for the main definition. The variable names in the formulas should match the names of the parameters, and the LaTeX formatting should match the LaTeX pretty printing used by SymPy. As relevant, the mathematical definitions should mention their domain of definition, especially if the domain is different from the complex numbers.
- If there are multiple conventions in the literature for a function, make sure to clearly specify which convention SymPy uses.
- The Explanation section may also include some important mathematical facts about the function. These can alternately be demonstrated in the Examples section. Mathematical discussions should not be too long, as users can check the references for more details.
- The docstring does not need to discuss every implementation detail such as at which operations are defined on the function or at which points it evaluates in the "eval" method. Important or illuminating instances of these can be shown in the Examples section.
- The docstring should go on the class level (right under the line that has "class"). The "eval" method should not have a docstring.
- Private methods on the class, that is, any method that starts with an underscore, do not need to be documented. They can still be documented if you like, but note that these docstrings are not pulled into the Sphinx documentation, so they will only be seen by developers who are reading the code, so if there is anything very important that you want to mention here, it should go in the class-level docstring as well.

6.5.6 Best Practices for Writing Docstrings

When writing docstrings, please follow all of the same formatting, style, and tone preferences as when writing narrative documentation. For guidelines, see *Best Practices for Writing Documentation* (page 3013), Formatting, Style, and Tone.

6.5.7 Importing Docstrings into the Sphinx Documentation

Here are excerpts from the `doc/src/modules/geometry` directory that imports the relevant docstrings from geometry module into documentation:

```
Utils
=====

.. module:: sympy.geometry.util

.. autofunction:: intersection

.. autofunction:: convex_hull

```

(continues on next page)

(continued from previous page)

```
.. autofunction:: are_similar

Points
======

.. module:: sympy.geometry.point

.. autoclass:: Point
   :members:

Lines
=====

.. module:: sympy.geometry.line

.. autoclass:: LinearEntity
   :members:

.. autoclass:: Line
   :members:

.. autoclass:: Ray
   :members:

.. autoclass:: Segment
   :members:

Curves
======

.. module:: sympy.geometry.curve

.. autoclass:: Curve
   :members:

Ellipses
========

.. module:: sympy.geometry.ellipse

.. autoclass:: Ellipse
   :members:

.. autoclass:: Circle
   :members:

Polygons
========

.. module:: sympy.geometry.polygon

.. autoclass:: Polygon
```

(continues on next page)

(continued from previous page)

```
  :members:

.. autoclass:: RegularPolygon
   :members:

.. autoclass:: Triangle
   :members:
```

First namespace is set to particular submodule (file) with `.. module::` directive, then docstrings are imported with `.. autoclass::` or `.. autofunction::` relative to that submodule (file). Other methods are either cumbersome to use (using full paths for all objects) or break something (importing relative to main module using `.. module:: sympy.geometry` breaks viewcode Sphinx extension). All files in `doc/src/modules/` should use this format.

6.5.8 Cross-Referencing

Any text that references another SymPy function should be formatted so that a cross-reference link to that function's documentation is created automatically. This is done using the RST cross-reference syntax. There are two different kinds of objects that have conventions here:

1. Objects that are included in `from sympy import *`, for example, `sympy.acos`.

For these, use `:obj:`~.acos()``. The `~` makes it so that the text in the rendered HTML only shows `acos` instead of the fully qualified name `sympy.functions.elementary.trigonometric.acos`. (This will encourage importing names from the global `sympy` namespace instead of a specific submodule.) The `.` makes it so that the function name is found automatically. (If Sphinx gives a warning that there are multiple names found, replace the `.` with the full name. For example, `:obj:`~sympy.solvers.solvers.solve()`.) Adding a trailing pair of parentheses is a convention for indicating the name is a function, method, or class.

You may also use a more specific type indicator instead of `obj` (see https://www.sphinx-doc.org/en/master/usage/restructuredtext/domains.html#cross-referencing-python-objects). However, `obj` will always work, and sometimes SymPy names are not the type you might expect them to be. For example, mathematical function objects such as `sin` are not actually a Python function, rather they are a Python class, therefore `:func:`~.sin`` will not work.

2. Objects that are not included in `from sympy import *`, for example, `sympy.physics.vector.dynamicsymbols`.

This can be public API objects from submodules that are not included in the main `sympy/__init__.py`, such as the physics submodule, or private API objects that are not necessarily intended to be used by end-users (but should still be documented). In this case, you must show the fully qualified name, so do not use the `~.` syntax. For example, `:obj:`sympy.physics.vector.dynamicsymbols()``.

You may also write custom text that links to the documentation for something using the following syntax `:obj:`custom text<object>``. For example, `:obj:`the sine function <.sin>`` produces the text "the sine function" that links to the documentation for `sin`. Note that the ~ character should not be used here.

Note that references in the *See Also* (page 3002) section of the docstrings do not require the `:obj:` syntax.

If the resulting cross reference is written incorrectly, Sphinx will error when building the docs with an error like:

```
WARNING: py:obj reference target not found: expand
```

Here are some troubleshooting tips to fix the errors:

- Make sure you have used the correct syntax, as described above.
- Make sure you spelled the function name correctly.
- Check if the function you are trying to cross-reference is actually included in the Sphinx documentation. If it is not, Sphinx will not be able to create a reference for it. In that case, you should add it to the appropriate RST file as described in the *Docstring Guidelines* (page 2997).
- If the function or object is not included in `from sympy import *`, you will need to use the fully qualified name, like `sympy.submodule.submodule.function` instead of just `function`.
- A fully qualified name must include the full submodule for a function all the way down to the file. For example, `sympy.physics.vector.ReferenceFrame` will not work (even though you can access it that way in code). It has to be `sympy.physics.vector.frame.ReferenceFrame`.
- If the thing you are referring to does not actually have somewhere to link to, do not use the `:obj:` syntax. Instead, mark it as code using double backticks. Examples of things that cannot be linked to are Python built in functions like `int` or `NotImplementedError`, functions from other modules outside of SymPy like `matplotlib.plot`, and variable or parameter names that are specific to the text at hand. In general, if the object cannot be accessed as `sympy.something.something.object`, it cannot be cross- referenced and you should not use the `:obj:` syntax.
- If you are using are using one of the type specific identifiers like `:func:`, be sure that the type for it is correct. `:func:` only refers to Python functions. For classes, you need to use `:class:`, and for methods on a class you need to use `:method:`. In general, it is recommended to use `:obj:`, as this will work for any type of object.
- If you cannot get the cross-referencing syntax to work, go ahead and submit the pull request as is and ask the reviewers for help.

You may also see errors like:

```
WARNING: more than one target found for cross-reference 'subs()':
sympy.core.basic.Basic.subs, sympy.matrices.common.MatrixCommon.subs,
sympy.physics.vector.vector.Vector.subs,
sympy.physics.vector.dyadic.Dyadic.subs
```

for instance, from using `:obj:`~.subs``. This means that the `.` is not sufficient to find the function, because there are multiple names in SymPy named `subs`. In this case, you need to use the fully qualified name. You can still use `~` to make it shortened in the final text, like `:obj:`~sympy.core.basic.Basic.subs``.

The line numbers for warnings in Python files are relative to the top of the docstring, not the file itself. The line numbers are often not completely correct, so you will generally have to search the docstring for the part that the warning is referring to. This is due to a bug in Sphinx.

6.6 Documentation Style Guide

6.6.1 General Guidelines

Documentation is one of the most highly valued aspects of an open source project. Documentation teaches users and contributors how to use a project, how to contribute, and the standards of conduct within an open source community. But according to GitHub's Open Source Survey, incomplete or confusing documentation is the most commonly encountered problem in open source. This style guide aims to change that.

The purpose of this style guide is to provide the SymPy community with a set of style and formatting guidelines that can be utilized and followed when writing SymPy documentation. Adhering to the guidelines offered in this style guide will bring greater consistency and clarity to SymPy's documentation, supporting its mission to become a full-featured, open source computer algebra system (CAS).

The SymPy documentation found at docs.sympy.org is generated from docstrings in the source code and dedicated narrative documentation files in the doc/src directory. Both are written in reStructuredText format extended by Sphinx.

The documentation contained in the doc/src directory and the docstrings embedded in the Python source code are processed by Sphinx and various Sphinx extensions. This means that the documentation source format is specified by the documentation processing tools. The SymPy Documentation Style Guide provides both the essential elements for writing SymPy documentation as well as any deviations in style we specify relative to these documentation processing tools. The following lists the processing tools:

- reStructuredText: Narrative documentation files and documentation strings embedded in Python code follow the reStructuredText format. Advanced features not described in this document can be found at http://docutils.sourceforge.net/rst.html.
- Sphinx: Sphinx includes additional default features for the reStructuredText specification that are described at: http://www.sphinx-doc.org/.
- Sphinx extensions included with Sphinx that we enable:
  - `sphinx.ext.autodoc`: Processes Python source code files for the associated documentation strings to automatically generate pages containing the Application Programming Interface (API). See section on calling autodoc directives in this document to get started. More information is at: https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html.
  - `sphinx.ext.graphviz`: Provides a directive for adding Graphviz graphics. See https://www.sphinx-doc.org/en/master/usage/extensions/graphviz.html for more info.
  - `sphinx.ext.mathjax`: Causes math written in LaTeX to display using MathJax in the HTML version of the documentation. More information is at: https://www.sphinx-doc.org/en/master/usage/extensions/math.html#module-sphinx.ext.mathjax. *No bearing on documentation source format.*
  - `sphinx.ext.linkcode`: Causes links to source code to direct to the related files on Github. More information is at: https://www.sphinx-doc.org/en/master/usage/extensions/linkcode.html. *No bearing on documentation source format.*
- Sphinx extensions that are not included with Sphinx that we enable:

- `numpydoc`: Processes docstrings written in the "numpydoc" format, see https://numpydoc.readthedocs.io. We recommend the subset of numpydoc formatting features in this document. (Note that we currently use an older modified fork of numpydoc, which is included in the SymPy source code.)
- `sphinx_math_dollar`: Allows math to be delimited with dollar signs instead of reStructuredText directives (e.g., `$a^2$` instead of `:math:`a^2``). See https://www.sympy.org/sphinx-math-dollar/ for more info.
- `matplotlib.sphinxext.plot_directive`: Provides directives for included matplotlib generated figures in reStructuredText. See https://matplotlib.org/devel/plot_directive.html for more info.

Everything supported by the above processing tools is available for use in the SymPy documentation, but this style guide supersedes any recommendations made in the above documents. Note that we do not follow PEP 257 or the www.python.org documentation recommendations.

If you are contributing to SymPy for the first time, please read our Introduction to Contributing page as well as this guide.

6.6.2 Types of Documentation

There are four main locations where SymPy's documentation can be found:

SymPy Website https://sympy.org

The SymPy website's primary function is to advertise the software to users and developers. It also serves as an initial location to point viewers to other relevant resources on the web. The SymPy website has basic information on SymPy and how to obtain it, as well as examples to advertise it to users, but it does not have technical documentation. The source files are located in the SymPy webpage directory. Appropriate items for the website are:

- General descriptions of what SymPy and the SymPy community are
- Explanations/demonstrations of major software features
- Listings of other major software that uses SymPy
- Getting started info for users (download and install instructions)
- Getting started info for developers
- Where users can get help and support on using SymPy
- News about SymPy

SymPy Documentation https://docs.sympy.org

This is the main place where users go to learn how to use SymPy. It contains a tutorial for SymPy as well as technical documentation for all of the modules. The source files are hosted in the main SymPy repository in the doc directory at and are built using the Sphinx site generator and uploaded to the docs.sympy.org site automatically. There are two primary types of pages that are generated from different source files in the docs directory:

- Narrative Pages: reStructuredText files that correspond to manually written documentation pages not present in the Python source code. Examples are the tutorial RST files. In general, if your documentation is not API documentation it belongs in a narrative page.

- API Documentation Pages: reStructuredText files that contain directives that generate the Application Programming Interface documentation. These are automatically generated from the SymPy Python source code.

SymPy Source Code https://github.com/sympy/sympy

Most functions and classes have documentation written inside it in the form of a docstring, which explains the function and includes examples called doctests. The purpose of these docstrings are to explain the API of that class or function. The doctests examples are tested as part of the test suite, so that we know that they always produce the output that they say that they do. Here is an example docstring. Most docstrings are also automatically included in the Sphinx documentation above, so that they appear on the SymPy Documentation website. Here is that *same docstring* (page 383) on the SymPy website. The docstrings are formatted in a specific way so that Sphinx can render them correctly for the docs website. The SymPy sources all contain sparse technical documentation in the form of source code comments, although this does not generally constitute anything substantial and is not displayed on the documentation website.

SymPy Wiki https://github.com/sympy/sympy/wiki

The SymPy Wiki can be edited by anyone without review. It contains various types of documentation, including:

- High-level developer documentation (for example: https://github.com/sympy/sympy/wiki/Args-Invariant)
- Guides for new contributors (for example: https://github.com/sympy/sympy/wiki/Introduction-to-contributing)
- Development policies (for example: https://github.com/sympy/sympy/wiki/Python-version-support-policy)
- Release notes (for example: https://github.com/sympy/sympy/wiki/Release-Notes-for-1.5)
- Various pages that different contributors have added

6.6.3 Narrative Documentation Guidelines

Extensive documentation, or documentation that is not centered around an API reference, should be written as a narrative document in the Sphinx docs (located in the doc/src directory). The narrative documents do not reside in the Python source files, but as standalone restructured files in the doc/src directory. SymPy's narrative documentation is defined as the collective documents, tutorials, and guides that teach users how to use SymPy. Reference documentation should go in the docstrings and be pulled into the RST with autodoc. The RST itself should only have narrative style documentation that is not a reference for a single specific function.

6.6.4 Documentation using Markdown

Narrative documentation can be written using either Restructured Text (`.rst`) or Markdown (`.md`). Markdown documentation uses MyST. See this guide for more information on how to write documents in MArkdown. Markdown is only supported for narrative documentation. Docstrings should continue to use RST syntax. Any part of this style guide that is not specific to RST syntax should still apply to Markdown documents.

6.6.5 Best Practices for Writing Documentation

Please follow these formatting, style, and tone preferences when writing documentation.

Formatting Preferences

In order for math and code to render correctly on the SymPy website, please follow these formatting guidelines.

Math

Text that is surrounded by dollar signs `$ _ $` will be rendered as LaTeX math. Any text that is meant to appear as LaTeX math should be written as `$math$`. In the HTML version of the docs, MathJax will render the math.

Example

```
The Bessel $J$ function of order $\nu$ is defined to be the function
satisfying Bessel's differential equation.
```

LaTeX Recommendations

- If a docstring has any LaTeX, be sure to make it "raw." See the *Docstring Formatting* (page 2998) section for details.
- If you are not sure how to render something, you can use the SymPy *latex()* (page 2170) function. But be sure to strip out the unimportant parts (the bullet points below).
- Avoid unnecessary `\left` and `\right` (but be sure to use them when they are required).
- Avoid unnecessary `{}`. (For example, write `x^2` instead of `x^{2}`.)
- Use whitespace in a way that makes the equation easiest to read.
- Always check the final rendering to make sure it looks the way you expect it to.
- The HTML documentation build will not fail if there is invalid math, but rather it will show as an error on the page. However, the PDF build, which is run on GitHub Actions on pull requests, will fail. If the LaTeX PDF build fails on CI, there is likely an issue with LaTeX math somewhere.

Examples

Correct:

```
\int \sin(x)\,dx
```

Incorrect:

```
\int \sin{\left( x\right)}\, dx
```

For more in-depth resources on how to write math in LaTeX, see:

- https://math.meta.stackexchange.com/questions/5020/mathjax-basic-tutorial-and-quick-reference
- https://en.wikibooks.org/wiki/LaTeX/Mathematics
- https://www.overleaf.com/learn/latex/Mathematical_expressions

Code

Text that should be printed verbatim, such as code, should be surrounded by a set of double backticks `like this`.

Example

```
To use this class, define the ``_rewrite()`` and ``_expand()`` methods.
```

Sometimes a variable will be the same in both math and code, and can even appear in the same paragraph, making it difficult to know if it should be formatted as math or code. If the sentence in question is discussing mathematics, then LaTeX should be used, but if the sentence is discussing the SymPy implementation specifically, then code should be used.

In general, the rule of thumb is to consider if the variable in question were something that rendered differently in code and in math. For example, the Greek letter α would be written as `alpha` in code and `$\alpha$` in LaTeX. The reason being that `$\alpha$` cannot be used in contexts referring to Python code because it is not valid Python, and conversely `alpha` would be incorrect in a math context because it does not render as the Greek letter (α).

Example

```
class loggamma(Function):
    r"""
    The ``loggamma`` function implements the logarithm of the gamma
    function (i.e, $\log\Gamma(x)$).

    """
```

Variables listed in the parameters after the function name should, in written text, be italicized using Sphinx emphasis with asterisks like `*this*`.

Example

```
def stirling(n, k, d=None, kind=2, signed=False):
    """
    ...

    The first kind of Stirling number counts the number of permutations of
    *n* distinct items that have *k* cycles; the second kind counts the
    ways in which *n* distinct items can be partitioned into *k* parts.
    If *d* is given, the "reduced Stirling number of the second kind" is
```

(continues on next page)

(continued from previous page)

```
    returned: $S^{d}(n, k) = S(n - d + 1, k - d + 1)$ with $n \ge k \ge d$.
    This counts the ways to partition $n$ consecutive integers into $k$
    groups with no pairwise difference less than $d$.

    """
```

Note that in the above example, the first instances of *n* and *k* are referring to the input parameters of the function `stirling`. Because they are Python variables but also parameters listed by themselves, they are formatted as parameters in italics. The last instances of n and k are talking about mathematical expressions, so they are formatted as math.

If a variable is code, but is also a parameter written by itself, the parameter formatting takes precedence and it should be rendered in italics. However, if a parameter appears in a larger code expression it should be within double backticks to be rendered as code. If a variable is only code and not a parameter as well, it should be in double backticks to be rendered as code.

Please note that references to other functions in SymPy are handled differently from parameters or code. If something is referencing another function in SymPy, the cross-reference reStructuredText syntax should be used. See the section on *Cross-Referencing* (page 3008) for more information.

Headings

Section headings in reStructuredText files are created by underlining (and optionally overlining) the section title with a punctuation character at least as long as the text.

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, for SymPy's documentation, here is a suggested convention:

=== with overline: title (top level heading)

=== heading 1

--- heading 2

^^^ heading 3

~~~ heading 4

""" heading 5

Style Preferences

Spelling and Punctuation

All narrative writing in SymPy follows American spelling and punctuation standards. For example, "color" is preferred over "colour" and commas should be placed inside of quotation marks.

Examples
~~~

```
If the ``line_color`` aesthetic is a function of arity 1, then the coloring
is a function of the x value of a point.

The term "unrestricted necklace," or "bracelet," is used to imply an object
that can be turned over or a sequence that can be reversed.
```

If there is any ambiguity about the spelling of a word, such as in the case of a function named after a person, refer to the spelling of the actual SymPy function.

For example, Chebyshev polynomials are named after Pafnuty Lvovich Tchebychev, whose name is sometimes transliterated from Russian to be spelled with a "T," but in SymPy it should always be spelled "Chebyshev" to refer to the SymPy function.

Example

```
class chebyshevt(OrthogonalPolynomial):
    r"""
    Chebyshev polynomial of the first kind, $T_n(x)$
    ...

    """
```

Capitalization

Title case capitalization is preferred in all SymPy headings.

Example

```
What is Symbolic Computation?
-----------------------------
```

Tone Preferences

Across SymPy documentation, please write in:

- The present tense (e.g., In the following section, we are going to learn…)
- The first-person inclusive plural (e.g., We did this the long way, but now we can try it the short way…)
- Use the generic pronoun "you" instead of "one." Or use "the reader" or "the user." (e.g., You can access this function by… The user can then access this function by…)
- Use the gender-neutral pronoun "they" instead of "he" or "she." (e.g., A good docstring tells the user exactly what they need to know.)

Avoid extraneous or belittling words such as "obviously," "easily," "simply," "just," or "straightforward."

Avoid unwelcoming or judgement-based phrases like "That is wrong." Instead use friendly and inclusive language like "A common mistake is…"

Avoid extraneous phrases like, "we just have to do one more thing."

6.7 Making a Contribution

For in-depth instructions on how to contribute to SymPy's code base including coding conventions, creating your environment, picking an issue to fix, and opening a pull request, please read our full Development Workflow guide.

6.8 Deprecation Policy

This page outlines SymPy's policy on doing deprecations, and describes the steps developers should take to properly deprecate code.

A list of all currently active deprecations in SymPy can be found at *List of active deprecations* (page 163).

6.8.1 What is a deprecation?

A deprecation is a way to make backwards incompatible changes in a way that allows users to update their code. Deprecated code continues to work as it used to, but whenever someone uses it, it prints a warning to the screen indicating that it will be removed in a future version of SymPy, and indicating what the user should be using instead.

This allows users a chance to update their code without it completely breaking. It also gives SymPy an opportunity to give users an informative message on how to update their code, rather than making their code simply error or start giving wrong answers.

6.8.2 Try to avoid backwards incompatible changes in the first place

Backwards incompatible API changes should not be made lightly. Any backwards compatibility break means that users will need to fix their code. Whenever you want to make a breaking change, you should consider whether this is worth the pain for users. Users who have to update their code to match new APIs with every SymPy release will become frustrated with the library, and may go seek a more stable alternative. Consider whether the behavior you want can be done in a way that is compatible with existing APIs. New APIs do not necessarily need to completely supplant old ones. It is sometimes possible for old APIs to exist alongside newer, better designed ones without removing them. For example, the newer *solveset* (page 858) API was designed to be a superior replacement for the older *solve* (page 836) API. But the older `solve()` function remains intact and is still supported.

It is important to try to be cognizant of API design whenever adding new functionality. Try to consider what a function may do in the future, and design the API in a way that it can do so without having to make a breaking change. For example, if you add a property to an object `A.attr`, it is impossible to later convert that property into a method `A.attr()` so that it can take arguments, except by doing so in a backwards incompatible way. If you are unsure about your API design for a new functionality, one option is to mark the new functionality as explicitly private or as experimental.

With that being said, it may be decided that the API of SymPy must change in some incompatible way. Some reasons APIs are changed can include:

- The existing API is confusing.

- There is unnecessary redundancy in the API.
- The existing API limits what is possible.

Because one of the core use-cases of SymPy is to be usable as a library, we take API breakage very seriously. Whenever an API breakage is necessary, the following steps should be taken:

- Discuss the API change with the community. Be sure that the improved API is indeed better, and worth a breakage. It is important to get API right so that we will not need to break the API again to "fix" it a second time.
- If it is possible, deprecate the old API. The technical steps for doing this are described *below* (page 3020).
- Document the change so that users know how to update their code. The documentation that should added is described *below* (page 3022).

6.8.3 When does a change require deprecation?

When considering whether a change requires a deprecation, two things must be considered:

- Is the change backwards incompatible?
- Is the behavior changing public API?

A change is backwards incompatible if user code making use of it would stop working after the change.

What counts as "public API" needs to be considered on a case-by-case basis. The exact rules for what does and doesn't constitute public API for SymPy are still not yet fully codified. Cleaning up the distinction between public and private APIs, as well as the categorization in the reference documentation is currently an open issue for SymPy.

Here are some thing that constitute public API. *Note: these are just general guidelines. This list is not exhaustive, and there are always exceptions to the rules.*

Public API

- Function names.
- Keyword argument names.
- Keyword argument default values.
- Positional argument order.
- Submodule names.
- The mathematical conventions used to define a function.

And here are some things that generally aren't public API, and therefore don't require deprecations to change (again, this list is only a general set of guidelines).

Not Public API

- The precise form of an expression. In general, functions may be changed to return a different but mathematically equivalent form of the same expression. This includes a function returning a value which it was not able to compute previously.

- Functions and methods that are private, i.e., for internal use only. Such things should generally be prefixed with an underscore _, although this convention is not currently universally adhered to in the SymPy codebase.
- Anything explicitly marked as "experimental".
- Changes to behavior that were mathematically incorrect previously (in general, bug fixes are not considered breaking changes, because despite the saying, bugs in SymPy are not features).
- Anything that was added before the most recent release. Code that has not yet made it into a release does not need to be deprecated. If you are going to change the API of new code, it is best to do it before a release is made so that no deprecations are necessary for future releases.

Note: both public and private API functions are included in the *reference documentation* (page 185), and many functions are not included there which should be, or are not documented at all which should be, so this should not be used to determine whether something public or not.

If you're unsure, there is no harm in deprecating something even if it might not actually be "public API".

6.8.4 The purpose of deprecation

Deprecation has several purposes:

- To allow existing code to continue to work for a while, giving people a chance to upgrade SymPy without fixing all deprecation issues immediately.
- To warn users that their code will break in a future version.
- To inform users how to fix their code so that it will continue work in future versions.

All deprecation warnings should be something that users can remove by updating their code. Deprecation warnings that fire unconditionally, even when using the "correct" newer APIs, should be avoided.

This also means all deprecated code must have a completely functioning replacement. If there is no way for users to update their code, then this means API in question is not yet read to be deprecated. The deprecation warning should inform users of a way to change their code so that it works in the same version of SymPy, as well as all future versions, and, if possible, previous versions of SymPy as well. See *below* (page 3022).

Deprecations should always

1. Allow users to continue to use the existing APIs unchanged during the deprecation period (with a warning, which can be *silenced* (page 164) with `warnings.filterwarnings`).
2. Allow users to always fix their code so that it stops giving the warning.
3. After users fix their code, it should continue to work after the deprecated code is removed.

The third point is important. We do not want the "new" method to itself cause another API break when the deprecation period is over. Doing this would completely defeat the purpose of doing a deprecation.

When it is not technically possible to deprecate

In some cases, this is not technically possible to make a deprecation that follows the above three rules. API changes of this nature should be considered the most heavily, as they will break people's code immediately without warning. Consideration into how easy it will be for users to support multiple versions of SymPy, one with the change and one without, should also be taken into account.

If you decide that the change is nonetheless worth making, there are two options:

- Make the non-deprecatable change immediately, with no warnings. This will break user code.
- Warn that the code will change in the future. There won't be a way for users to fix their code until a version is released with the breaking change, but they will at least be aware that changes are coming.

Which of the two to make should decided on a case-by-case basis.

6.8.5 How long should deprecations last?

Deprecations should remain intact for **at least 1 year** after the first major release is made with the deprecation. This is only a minimum period: deprecations are allowed to remain intact for longer than this. If a change is especially hard for users to migrate, the deprecation period should be lengthened. The period may also be lengthened for deprecated features that do not impose a significant maintenance burden to keep around.

The deprecation period policy is time-based rather than release-based for a few reasons. Firstly, SymPy does not have a regular release schedule. Sometimes multiple releases will be made in a year, and some years only a single release will be made. Being time-based assures that users have sufficient opportunity to update their code regardless of how often releases happen to be made.

Secondly, SymPy does not make use of a rigorous versioning scheme like semantic versioning. The API surface of SymPy and number of contributions are both large enough that virtually every major release has some deprecations and backwards incompatible changes made in some submodule. Encoding this into the version number would be virtually impossible. The development team also does not backport changes to prior major releases, except in extreme cases. Thus a time-based deprecation scheme is more accurate to SymPy's releasing model than a version-based one would be.

Finally, a time-based scheme removes any temptation to "fudge" a deprecation period down by releasing early. The best way for the developers to accelerate the removal of deprecated functionality is to make a release containing the deprecation as early as possible.

6.8.6 How to deprecate code

Checklist

Here is a checklist for doing a deprecation. See below for details on each step.

Discuss the backwards incompatible change with the community. Ensure the change is really worth making as per the discussion above.

Remove all instance of the deprecated code from everywhere in the codebase (including doctest examples).

Add *sympy_deprecation_warning()* (page 2067) to the code.

Write a descriptive message for the *sympy_deprecation_warning()* (page 2067). Make sure the message explains both what is deprecated and what to replace it with. The message may be a multiline string and contain examples.

Set `deprecated_since_version` to the version in `sympy/release.py` (without the `.dev`).

Set `active_deprecations_target` to the target used in the `active-deprecations.md` file.

Make sure `stacklevel` is set to the right value so that the deprecation warning shows the user line of code.

Visually confirm the deprecation warning looks good in the console.

Add a `.. deprecated:: <version>` note to the top of the relevant docstring(s).

Add a section to the `doc/src/explanation/active-deprecations.md` file.

Add a cross-reference target `(deprecation-xyz)=` before the section header (this is the same reference used by `active_deprecations_target` above).

Explain what is deprecated and what to replace it with.

Explain *why* the given thing is deprecated.

Add a test using *warns_deprecated_sympy()* (page 2029) that tests that the deprecation warning is issued properly. This test should be the only place in the code that actually uses the deprecated functionality.

Run the test suite to ensure the above test works and that no other code uses the deprecated code, which will cause the tests to fail.

In your PR, add a `BREAKING CHANGE` entry to the release notes for the deprecation.

Once the PR is merged, manually add the change to the "Backwards compatibility breaks and deprecations" section of the release notes on the wiki.

Adding the deprecation to the code

All deprecations should use *sympy.utilities.exceptions.sympy_deprecation_warning()* (page 2067). If an entire function or method is deprecated, you can use the *sympy.utilities.decorator.deprecated()* (page 2057) decorator. The `deprecated_since_version` and `active_deprecations_target` flags are required. Do not use the `SymPyDeprecationWarning` class directly to issue a deprecation warning. Please see the docstring of *sympy_deprecation_warning()* (page 2067) for more information. See *below* (page 3022) for an example.

Add a test for the deprecated behavior. Use the *sympy.testing.pytest.warns_deprecated_sympy()* (page 2029) context manager.

```
from sympy.testing.pytest import warns_deprecated_sympy

with warns_deprecated_sympy():
    <deprecated behavior>
```

Note: `warns_deprecated_sympy` is only intended to be used internally by the SymPy test suite. Users of SymPy should use the warnings module directly to filter SymPy deprecation warnings. See *Silencing SymPy Deprecation Warnings* (page 164).

This has two purposes: to test that the warning is emitted correctly, and to test that the deprecated behavior still actually functions.

If you want to test multiple things and assert that each emits a warning then use separate with blocks for each:

```
with warns_deprecated_sympy():
    <deprecated behavior1>
with warns_deprecated_sympy():
    <deprecated behavior2>
```

This should be the only part of the codebase and test suite that uses the deprecated behavior. Everything else should be changed to use the new, non-deprecated behavior. The SymPy test suite is configured to fail if a `SymPyDeprecationWarning` is issued anywhere except in a `warns_deprecated_sympy()` block. You should not use this function or a `warnings.filterwarnings(SymPyDeprecationWarning)` anywhere except in the test for the deprecation. This includes the documentation examples. The documentation for a deprecated function should just have a note pointing to the non-deprecated alternative. If you want to show a deprecated function in a doctest use `# doctest: +SKIP`. The only exception to this rule is that you may use `ignore_warnings(SymPyDeprecationWarning)` to prevent the exact same warning from triggering twice, i.e., if a deprecated function calls another function that issues the same or a similar warning.

If it is not possible to remove the deprecated behavior somewhere, that is a sign that it is not ready to be deprecated yet. Consider that users may not be able to replace the deprecated behavior for exact same reason.

Documenting a deprecation

All deprecations should be documented. Every deprecation needs to be documented in three primary places:

- The *sympy_deprecation_warning()* (page 2067) warning text. This text is allowed to be long enough to describe the deprecation, but it should not be more than one paragraph. The primary purpose of the warning text should be *to inform users how to update their code*. The warning text should *not* discuss why a feature was deprecated or unnecessary internal technical details. This discussion can go in the other sections mentioned below. Do not include information in the message that is already part of the metadata provided to the keyword arguments to `sympy_deprecation_warning()`, like the version number or a link to the active deprecations document. Remember that the warning text will be shown in plain-text, so do not use RST or Markdown markup in the text. Code blocks should be clearly delineated with newlines so that they are easy to read. All text in the warning message should be wrapped to 80 characters, except for code examples that cannot be wrapped.

  Always include full context of what is deprecated in the message. For example, write "the abc keyword to func() is deprecated" instead of just "the abc keyword is deprecated". That way if a user has a larger line of code that is using the deprecated functionality, it will be easier for them to see exactly which part is causing the warning.

- A deprecation note in the relevant docstring(s). This should use the `deprecated` Sphinx directive. This uses the syntax `.. deprecated:: <version>`. If the entire function is deprecated, this should be placed at the top of the docstring, right below the first line. Otherwise, if only part of a function is deprecated (e.g., a single keyword argument), it should be placed near the part of the docstring that discusses that feature, e.g., in the parameters list.

The text in the deprecation should be short (no more than a paragraph), explaining what is deprecated and what users should use instead. If you want, you may use the same text here as in the *sympy_deprecation_warning()* (page 2067). Be sure to use RST formatting, including cross-references to the new function if relevant, and a cross-reference to the longer description in the `active-deprecations.md` document (see *below* (page 3024)).

If the documentation for the feature is otherwise the same as the replaced feature (i.e., the deprecation is just a renaming of a function or argument), you may replace the rest of the documentation with a note like "see the documentation for <new feature>". Otherwise, the documentation for the deprecated functionality should be left intact.

Here are some (imaginary) examples:

```
@deprecated("""\
The simplify_this(expr) function is deprecated. Use simplify(expr)
instead.""", deprecated_since_version="1.1",
active_deprecations_target='simplify-this-deprecation')
def simplify_this(expr):
    """
    Simplify ``expr``.

    .. deprecated:: 1.1

       The ``simplify_this`` function is deprecated. Use :func:`simplify`
       instead. See its documentation for more information. See
       :ref:`simplify-this-deprecation` for details.

    """
    return simplify(expr)
```

```
def is_this_zero(x, y=0):
    """
    Determine if x = 0.

    Parameters
    ==========

    x : Expr
      The expression to check.

    y : Expr, optional
      If provided, check if x = y.

      .. deprecated:: 1.1

         The ``y`` argument to ``is_this_zero`` is deprecated. Use
         ``is_this_zero(x - y)`` instead. See
         :ref:`is-this-zero-y-deprecated` for more details.

    """
    if y != 0:
        sympy_deprecation_warning("""\
The y argument to is_zero() is deprecated. Use is_zero(x - y) instead.""
```

(continues on next page)

(continued from previous page)

```
↪",
            deprecated_since_version="1.1",
            active_deprecations_target='is-this-zero-y-deprecation')
    return simplify(x - y) == 0
```

- A longer description of the deprecation should be added to *the page listing all currently active deprecations* (page 163) in the documentation (in `doc/src/explanation/active-deprecations.md`).

  This page is where you can go into more detail about the technical details of a deprecation. Here you should also list *why* a feature was deprecated. You may link to relevant issues, pull requests, and mailing list discussions about the deprecation, but these discussion should be summarized so that users can get the basic idea of why the deprecation without having to read through pages of old discussions. You may also give longer examples here that would not fit in the *sympy_deprecation_warning()* (page 2067) message or `.. deprecated::` text.

  Every deprecation should have a cross-reference target (using `(target-name)=` above the section header) so that the `.. deprecated::` note in the relevant docstring can refer to it. This target should also be passed to the `active_deprecations_target` option of *sympy_deprecation_warning()* (page 2067) or *@deprecated* (page 2057). This will automatically put a link to the page in the documentation in the warning message. The target name should include the word "deprecation" or "deprecated" (target names are global in Sphinx, so the target name needs to be unique across the entire documentation).

  The section header name should be the thing that is deprecated, and should be a level 3 header under the corresponding version (typically it should be added to the top of the file).

  If multiple deprecations are related to one another, they can all share a single section on this page.

  If the deprecated function is not included in the top-level `sympy/__init__.py` be sure to clearly indicate which submodule the object is referring to. If you refer to anything that is documented in the Sphinx module reference, cross-reference it, like `{func}`~.func_name`.

  Note that examples here are helpful, but you generally should not use doctests to show the deprecated functionality, as this will itself raise the deprecation warning and fail the doctest. Instead you may use `# doctest: +SKIP`, or just show the example as a code block instead of a doctest.

  Here are examples corresponding to the (imaginary) examples above:

```
(simplify-this-deprecation)=
### `simplify_this()`

The `sympy.simplify.simplify_this()` function is deprecated. It has been
replaced with the {func}`~.simplify` function. Code using `simplify_
↪this()`
can be fixed by replacing `simplfiy_this(expr)` with `simplify(expr)`.␣
↪The
behavior of the two functions is otherwise identical.

This change was made because `simplify` is a much more Pythonic name than
`simplify_this`.
```

```
(is-this-zero-y-deprecation)=
### `is_this_zero()` second argument
The second argument to {func}`~.is_this_zero()` is deprecated. Previously
`is_this_zero(x, y)` would check if x = y. However, this was removed␣
↪because
it is trivially equivalent to `is_this_zero(x - y)`. Furthermore,␣
↪allowing
to check $x=y$ in addition to just $x=0$ is is confusing given the␣
↪function
is named "is this zero".

In particular, replace

```py
is_this_zero(expr1, expr2)
```

with

```py
is_this_zero(expr1 - expr2)
```
```

In addition to the above examples, there are dozens of examples of existing deprecations which can be found by searching for `sympy_deprecation_warning` in the SymPy codebase

Release notes entry

In the pull request, document the breaking change in the release notes section with `BREAKING CHANGE`.

Once the PR is merged, you should also add it to the "Backwards compatibility breaks and deprecations" section of the release notes for the upcoming release. This needs to be done manually, in addition to the change from the bot. See https://github.com/sympy/sympy/wiki/Writing-Release-Notes#backwards-compatibility-breaks-and-deprecations

Whenever a deprecated functionality is removed entirely after its deprecation period, this also needs to be marked as a `BREAKING CHANGE` and added to the "Backwards compatibility breaks and deprecations" section of the release notes.

BIBLIOGRAPHY

[commutative] https://en.wikipedia.org/wiki/Commutative_property
[infinite] https://en.wikipedia.org/wiki/Infinity
[antihermitian] https://en.wikipedia.org/wiki/Skew-Hermitian_matrix
[complex] https://en.wikipedia.org/wiki/Complex_number
[algebraic] https://en.wikipedia.org/wiki/Algebraic_number
[transcendental] https://en.wikipedia.org/wiki/Transcendental_number
[extended_real] https://en.wikipedia.org/wiki/Extended_real_number_line
[real] https://en.wikipedia.org/wiki/Real_number
[imaginary] https://en.wikipedia.org/wiki/Imaginary_number
[rational] https://en.wikipedia.org/wiki/Rational_number
[irrational] https://en.wikipedia.org/wiki/Irrational_number
[integer] https://en.wikipedia.org/wiki/Integer
[parity] https://en.wikipedia.org/wiki/Parity_(mathematics)
[prime] https://en.wikipedia.org/wiki/Prime_number
[composite] https://en.wikipedia.org/wiki/Composite_number
[zero] https://en.wikipedia.org/wiki/0
[positive] https://en.wikipedia.org/wiki/Positive_real_numbers
[negative] https://en.wikipedia.org/wiki/Negative_number
[R5] https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)
[R6] https://en.wikipedia.org/wiki/Sexy_prime
[R3] https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)
[R4] https://en.wikipedia.org/wiki/Sexy_prime
[R7] https://en.wikipedia.org/wiki/Finite
[R8] https://en.wikipedia.org/wiki/Symmetric_matrix
[R9] https://en.wikipedia.org/wiki/Invertible_matrix
[R10] https://en.wikipedia.org/wiki/Orthogonal_matrix
[R11] https://en.wikipedia.org/wiki/Unitary_matrix

[R12] https://en.wikipedia.org/wiki/Positive-definite_matrix

[R13] http://mathworld.wolfram.com/UpperTriangularMatrix.html

[R14] http://mathworld.wolfram.com/LowerTriangularMatrix.html

[R15] https://en.wikipedia.org/wiki/Diagonal_matrix

[R16] https://en.wikipedia.org/wiki/Square_matrix

[R17] http://mathworld.wolfram.com/SingularMatrix.html

[R18] https://en.wikipedia.org/wiki/Normal_matrix

[R19] https://en.wikipedia.org/wiki/Triangular_matrix

[R20] https://en.wikipedia.org/wiki/Integer

[R21] https://en.wikipedia.org/wiki/Rational_number

[R22] https://en.wikipedia.org/wiki/Irrational_number

[R23] https://en.wikipedia.org/wiki/Real_number

[R24] http://mathworld.wolfram.com/HermitianOperator.html

[R25] https://en.wikipedia.org/wiki/Complex_number

[R26] https://en.wikipedia.org/wiki/Imaginary_number

[R27] http://mathworld.wolfram.com/HermitianOperator.html

[R28] https://en.wikipedia.org/wiki/Algebraic_number

[R29] https://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation

[R30] https://en.wikipedia.org/wiki/Mathematical_singularity

[R31] Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0

[R32] https://en.wikipedia.org/wiki/Convex_function

[R33] http://www.ifp.illinois.edu/~angelia/L3_convfunc.pdf

[R34] https://en.wikipedia.org/wiki/Logarithmically_convex_function

[R35] https://en.wikipedia.org/wiki/Logarithmically_concave_function

[R36] https://en.wikipedia.org/wiki/Concave_function

[R50] https://en.wikipedia.org/wiki/Partition_%28number_theory%29

[R68] Skiena, S. ‘Permutations.’ 1.1 in Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 3-16, 1990.

[R69] Knuth, D. E. The Art of Computer Programming, Vol. 4: Combinatorial Algorithms, 1st ed. Reading, MA: Addison-Wesley, 2011.

[R70] Wendy Myrvold and Frank Ruskey. 2001. Ranking and unranking permutations in linear time. Inf. Process. Lett. 79, 6 (September 2001), 281-284. DOI=10.1016/S0020-0190(01)00141-7

[R71] D. L. Kreher, D. R. Stinson ‘Combinatorial Algorithms’ CRC Press, 1999

[R72] Graham, R. L.; Knuth, D. E.; and Patashnik, O. Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Reading, MA: Addison-Wesley, 1994.

[R73] https://en.wikipedia.org/wiki/Permutation#Product_and_inverse

[R74] https://en.wikipedia.org/wiki/Lehmer_code

[R75] https://en.wikipedia.org/wiki/Commutator

[R76] http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm

[R77] https://en.wikipedia.org/wiki/Flavius_Josephus

[R78] https://en.wikipedia.org/wiki/Josephus_problem

[R79] http://www.wou.edu/~burtonl/josephus.html

[R80] https://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties

[R51] Holt, D., Eick, B., O'Brien, E. "Handbook of Computational Group Theory"

[R52] Seress, A. "Permutation Group Algorithms"

[R53] https://en.wikipedia.org/wiki/Schreier_vector

[R54] https://en.wikipedia.org/wiki/Nielsen_transformation#Product_replacement_algorithm

[R55] Frank Celler, Charles R.Leedham-Green, Scott H.Murray, Alice C.Niemeyer, and E.A.O'Brien. "Generating Random Elements of a Finite Group"

[R56] https://en.wikipedia.org/wiki/Block_%28permutation_group_theory%29

[R57] http://www.algorithmist.com/index.php/Union_Find

[R58] https://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups

[R59] https://en.wikipedia.org/wiki/Center_%28group_theory%29

[R60] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[R66] http://www.algorithmist.com/index.php/Union_Find

[R62] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[R68] http://www.algorithmist.com/index.php/Union_Find

[R64] The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis

[R65] 1978: John S. Rose: A Course on Group Theory, Introduction to Finite Group Theory: 1.4

[R81] http://mathworld.wolfram.com/PolyhedralGroup.html

[R82] http://mathworld.wolfram.com/LabeledTree.html

[R83] https://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html

[R39] Nijenhuis,A. and Wilf,H.S.(1978). Combinatorial Algorithms. Academic Press.

[R40] Knuth, D. (2011). The Art of Computer Programming, Vol 4 Addison Wesley

[R41] Knuth, D. (2011). The Art of Computer Programming, Vol 4, Addison Wesley

[R42] http://statweb.stanford.edu/~susan/courses/s208/node12.html

[R46] https://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations

[R47] https://en.wikipedia.org/wiki/Dihedral_group

[R48] Armstrong, M. “Groups and Symmetry”

[R49] http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups

[R43] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.

[R44] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.

[R45] Pakianathan, J., Shankar, K., *Nilpotent Numbers*, The American Mathematical Monthly, 107(7), 631-634.

[R84] Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

[R85] Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

[R86] Holt, D., Eick, B., O’Brien, E.”Handbook of computational group theory”

[CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. Math. Comp., 27:463– 490, 1973.

[Ho05] Derek F. Holt, Handbook of Computational Group Theory. In the series ‘Discrete Mathematics and its Applications’, Chapman & Hall/CRC 2005, xvi + 514 p.

[Hav91] George Havas, Coset enumeration strategies. In Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC’91), Bonn 1991, pages 191–199. ACM Press, 1991.

[Ho05] Derek F. Holt, Handbook of Computational Group Theory. In the series ‘Discrete Mathematics and its Applications’, Chapman & Hall/CRC 2005, xvi + 514 p.

[R238] https://en.wikipedia.org/wiki/Complex_conjugation

[R239] https://en.wikipedia.org/wiki/Trigonometric_functions

[R240] http://dlmf.nist.gov/4.14

[R241] http://functions.wolfram.com/ElementaryFunctions/Sin

[R242] http://mathworld.wolfram.com/TrigonometryAngles.html

[R243] https://en.wikipedia.org/wiki/Trigonometric_functions

[R244] http://dlmf.nist.gov/4.14

[R245] http://functions.wolfram.com/ElementaryFunctions/Cos

[R246] https://en.wikipedia.org/wiki/Trigonometric_functions

[R247] http://dlmf.nist.gov/4.14

[R248] http://functions.wolfram.com/ElementaryFunctions/Tan

[R249] https://en.wikipedia.org/wiki/Trigonometric_functions

[R250] http://dlmf.nist.gov/4.14

[R251] http://functions.wolfram.com/ElementaryFunctions/Cot

[R252] https://en.wikipedia.org/wiki/Trigonometric_functions

[R253] http://dlmf.nist.gov/4.14

[R254] http://functions.wolfram.com/ElementaryFunctions/Sec

[R255] https://en.wikipedia.org/wiki/Trigonometric_functions
[R256] http://dlmf.nist.gov/4.14
[R257] http://functions.wolfram.com/ElementaryFunctions/Csc
[R258] https://en.wikipedia.org/wiki/Sinc_function
[R259] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R260] http://dlmf.nist.gov/4.23
[R261] http://functions.wolfram.com/ElementaryFunctions/ArcSin
[R262] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R263] http://dlmf.nist.gov/4.23
[R264] http://functions.wolfram.com/ElementaryFunctions/ArcCos
[R265] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R266] http://dlmf.nist.gov/4.23
[R267] http://functions.wolfram.com/ElementaryFunctions/ArcTan
[R268] http://dlmf.nist.gov/4.23
[R269] http://functions.wolfram.com/ElementaryFunctions/ArcCot
[R270] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R271] http://dlmf.nist.gov/4.23
[R272] http://functions.wolfram.com/ElementaryFunctions/ArcSec
[R273] http://reference.wolfram.com/language/ref/ArcSec.html
[R274] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R275] http://dlmf.nist.gov/4.23
[R276] http://functions.wolfram.com/ElementaryFunctions/ArcCsc
[R277] https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
[R278] https://en.wikipedia.org/wiki/Atan2
[R279] http://functions.wolfram.com/ElementaryFunctions/ArcTan2
[R280] https://en.wikipedia.org/wiki/Hyperbolic_function
[R281] http://dlmf.nist.gov/4.37
[R282] http://functions.wolfram.com/ElementaryFunctions/ArcSech/
[R283] https://en.wikipedia.org/wiki/Hyperbolic_function
[R284] http://dlmf.nist.gov/4.37
[R285] http://functions.wolfram.com/ElementaryFunctions/ArcCsch/
[R286] “Concrete mathematics” by Graham, pp. 87
[R287] http://mathworld.wolfram.com/CeilingFunction.html
[R288] “Concrete mathematics” by Graham, pp. 87
[R289] http://mathworld.wolfram.com/FloorFunction.html
[R290] https://en.wikipedia.org/wiki/Fractional_part

[R291] http://mathworld.wolfram.com/FractionalPart.html
[R292] https://en.wikipedia.org/wiki/Lambert_W_function
[R293] https://en.wikipedia.org/wiki/Directed_complete_partial_order
[R294] https://en.wikipedia.org/wiki/Lattice_%28order%29
[R295] https://en.wikipedia.org/wiki/Square_root
[R296] https://en.wikipedia.org/wiki/Real_root
[R297] https://en.wikipedia.org/wiki/Root_of_unity
[R298] https://en.wikipedia.org/wiki/Principal_value
[R299] http://mathworld.wolfram.com/CubeRoot.html
[R300] https://en.wikipedia.org/wiki/Square_root
[R301] https://en.wikipedia.org/wiki/Principal_value
[R302] https://en.wikipedia.org/wiki/Cube_root
[R303] https://en.wikipedia.org/wiki/Principal_value
[R193] https://en.wikipedia.org/wiki/Bell_number
[R194] http://mathworld.wolfram.com/BellNumber.html
[R195] http://mathworld.wolfram.com/BellPolynomial.html
[R196] https://en.wikipedia.org/wiki/Bernoulli_number
[R197] https://en.wikipedia.org/wiki/Bernoulli_polynomial
[R198] http://mathworld.wolfram.com/BernoulliNumber.html
[R199] http://mathworld.wolfram.com/BernoulliPolynomial.html
[R200] https://www.johndcook.com/blog/binomial_coefficients/
[R201] https://en.wikipedia.org/wiki/Catalan_number
[R202] http://mathworld.wolfram.com/CatalanNumber.html
[R203] http://functions.wolfram.com/GammaBetaErf/CatalanNumber/
[R204] http://geometer.org/mathcircles/catalan.pdf
[R205] https://en.wikipedia.org/wiki/Euler_numbers
[R206] http://mathworld.wolfram.com/EulerNumber.html
[R207] https://en.wikipedia.org/wiki/Alternating_permutation
[R208] http://mathworld.wolfram.com/AlternatingPermutation.html
[R209] http://dlmf.nist.gov/24.2#ii
[R210] https://en.wikipedia.org/wiki/Subfactorial
[R211] http://mathworld.wolfram.com/Subfactorial.html
[R212] https://en.wikipedia.org/wiki/Double_factorial
[R213] http://mathworld.wolfram.com/FallingFactorial.html
[R214] Peter Paule, “Greatest Factorial Factorization and Symbolic Summation”, Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

[R216] https://en.wikipedia.org/wiki/Fibonacci_number

[R217] http://mathworld.wolfram.com/FibonacciNumber.html

[R218] https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers#Tribonacci_numbers

[R219] http://mathworld.wolfram.com/TribonacciNumber.html

[R220] https://oeis.org/A000073

[R221] https://en.wikipedia.org/wiki/Harmonic_number

[R222] http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/

[R223] http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/

[R224] https://en.wikipedia.org/wiki/Lucas_number

[R225] http://mathworld.wolfram.com/LucasNumber.html

[R226] https://en.wikipedia.org/wiki/Genocchi_number

[R227] http://mathworld.wolfram.com/GenocchiNumber.html

[R228] https://en.wikipedia.org/wiki/Partition_(number_theory%29

[R229] https://en.wikipedia.org/wiki/Pentagonal_number_theorem

[R230] https://en.wikipedia.org/wiki/Pochhammer_symbol

[R231] Peter Paule, “Greatest Factorial Factorization and Symbolic Summation”, Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

[R232] https://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind

[R233] https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind

[R234] https://en.wikipedia.org/wiki/Combination

[R235] http://tinyurl.com/cep849r

[R236] https://en.wikipedia.org/wiki/Permutation

[R237] http://undergraduate.csse.uwa.edu.au/units/CITS7209/partition.pdf

[R304] http://mathworld.wolfram.com/DeltaFunction.html

[R305] http://mathworld.wolfram.com/HeavisideStepFunction.html

[R306] http://dlmf.nist.gov/1.16#iv

[R307] https://en.wikipedia.org/wiki/Singularity_function

[R308] https://en.wikipedia.org/wiki/Gamma_function

[R309] http://dlmf.nist.gov/5

[R310] http://mathworld.wolfram.com/GammaFunction.html

[R311] http://functions.wolfram.com/GammaBetaErf/Gamma/

[R312] https://en.wikipedia.org/wiki/Gamma_function

[R313] http://dlmf.nist.gov/5

[R314] http://mathworld.wolfram.com/LogGammaFunction.html

[R315] http://functions.wolfram.com/GammaBetaErf/LogGamma/

[R316] https://en.wikipedia.org/wiki/Polygamma_function

[R317] http://mathworld.wolfram.com/PolygammaFunction.html

[R318] http://functions.wolfram.com/GammaBetaErf/PolyGamma/

[R319] http://functions.wolfram.com/GammaBetaErf/PolyGamma2/

[R320] https://en.wikipedia.org/wiki/Digamma_function

[R321] http://mathworld.wolfram.com/DigammaFunction.html

[R322] http://functions.wolfram.com/GammaBetaErf/PolyGamma2/

[R323] https://en.wikipedia.org/wiki/Trigamma_function

[R324] http://mathworld.wolfram.com/TrigammaFunction.html

[R325] http://functions.wolfram.com/GammaBetaErf/PolyGamma2/

[R326] https://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_incomplete_Gamma_function

[R327] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R328] http://dlmf.nist.gov/8

[R329] http://functions.wolfram.com/GammaBetaErf/Gamma2/

[R330] http://functions.wolfram.com/GammaBetaErf/Gamma3/

[R331] https://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions

[R332] https://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_incomplete_Gamma_function

[R333] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R334] http://dlmf.nist.gov/8

[R335] http://functions.wolfram.com/GammaBetaErf/Gamma2/

[R336] http://functions.wolfram.com/GammaBetaErf/Gamma3/

[R337] https://en.wikipedia.org/wiki/Multivariate_gamma_function

[R338] https://en.wikipedia.org/wiki/Beta_function

[R339] http://mathworld.wolfram.com/BetaFunction.html

[R340] http://dlmf.nist.gov/5.12

[R341] https://en.wikipedia.org/wiki/Error_function

[R342] http://dlmf.nist.gov/7

[R343] http://mathworld.wolfram.com/Erf.html

[R344] http://functions.wolfram.com/GammaBetaErf/Erf

[R345] https://en.wikipedia.org/wiki/Error_function

[R346] http://dlmf.nist.gov/7

[R347] http://mathworld.wolfram.com/Erfc.html

[R348] http://functions.wolfram.com/GammaBetaErf/Erfc

[R349] https://en.wikipedia.org/wiki/Error_function

[R350] http://mathworld.wolfram.com/Erfi.html

[R351] http://functions.wolfram.com/GammaBetaErf/Erfi

[R352] http://functions.wolfram.com/GammaBetaErf/Erf2/

[R353] https://en.wikipedia.org/wiki/Error_function#Inverse_functions

[R354] http://functions.wolfram.com/GammaBetaErf/InverseErf/

[R355] https://en.wikipedia.org/wiki/Error_function#Inverse_functions

[R356] http://functions.wolfram.com/GammaBetaErf/InverseErfc/

[R357] http://functions.wolfram.com/GammaBetaErf/InverseErf2/

[R358] https://en.wikipedia.org/wiki/Fresnel_integral

[R359] http://dlmf.nist.gov/7

[R360] http://mathworld.wolfram.com/FresnelIntegrals.html

[R361] http://functions.wolfram.com/GammaBetaErf/FresnelS

[R362] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley

[R363] https://en.wikipedia.org/wiki/Fresnel_integral

[R364] http://dlmf.nist.gov/7

[R365] http://mathworld.wolfram.com/FresnelIntegrals.html

[R366] http://functions.wolfram.com/GammaBetaErf/FresnelC

[R367] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley

[R368] http://dlmf.nist.gov/6.6

[R369] https://en.wikipedia.org/wiki/Exponential_integral

[R370] Abramowitz & Stegun, section 5: http://people.math.sfu.ca/~cbm/aands/page_228.htm

[R371] http://dlmf.nist.gov/8.19

[R372] http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/

[R373] https://en.wikipedia.org/wiki/Exponential_integral

[R374] https://en.wikipedia.org/wiki/Logarithmic_integral

[R375] http://mathworld.wolfram.com/LogarithmicIntegral.html

[R376] http://dlmf.nist.gov/6

[R377] http://mathworld.wolfram.com/SoldnersConstant.html

[R378] https://en.wikipedia.org/wiki/Logarithmic_integral

[R379] http://mathworld.wolfram.com/LogarithmicIntegral.html

[R380] http://dlmf.nist.gov/6

[R381] https://en.wikipedia.org/wiki/Trigonometric_integral

[R382] https://en.wikipedia.org/wiki/Trigonometric_integral

[R383] https://en.wikipedia.org/wiki/Trigonometric_integral

[R384] https://en.wikipedia.org/wiki/Trigonometric_integral

[R385] Abramowitz, Milton; Stegun, Irene A., eds. (1965), “Chapter 9”, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R386] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1

[R387] https://en.wikipedia.org/wiki/Bessel_function

[R388] http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/

[R389] http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/

[R390] http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/

[R391] http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/

[R392] http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/

[R393] http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/

[R394] http://dlmf.nist.gov/10.47

[R395] http://dlmf.nist.gov/10.47

[R396] https://en.wikipedia.org/wiki/Marcum_Q-function

[R397] http://mathworld.wolfram.com/MarcumQ-Function.html

[R398] https://en.wikipedia.org/wiki/Airy_function

[R399] http://dlmf.nist.gov/9

[R400] http://www.encyclopediaofmath.org/index.php/Airy_functions

[R401] http://mathworld.wolfram.com/AiryFunctions.html

[R402] https://en.wikipedia.org/wiki/Airy_function

[R403] http://dlmf.nist.gov/9

[R404] http://www.encyclopediaofmath.org/index.php/Airy_functions

[R405] http://mathworld.wolfram.com/AiryFunctions.html

[R406] https://en.wikipedia.org/wiki/Airy_function

[R407] http://dlmf.nist.gov/9

[R408] http://www.encyclopediaofmath.org/index.php/Airy_functions

[R409] http://mathworld.wolfram.com/AiryFunctions.html

[R410] https://en.wikipedia.org/wiki/Airy_function

[R411] http://dlmf.nist.gov/9

[R412] http://www.encyclopediaofmath.org/index.php/Airy_functions

[R413] http://mathworld.wolfram.com/AiryFunctions.html

[R414] https://en.wikipedia.org/wiki/B-spline

[R415] http://dlmf.nist.gov/25.11

[R416] https://en.wikipedia.org/wiki/Hurwitz_zeta_function

[R417] https://en.wikipedia.org/wiki/Dirichlet_eta_function

[R418] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.