## class.def: .

comm:
int: Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified
on the first line, separated by a colon.

## class.def: .function_with_types_in_docstring

comm:
```
Example function with types documented in the docstring.

`PEP 484`_ type annotations are supported. If attribute, parameter, and
return types are annotated according to `PEP 484`_, they do not need to be
included in the docstring:

Args:
    param1 (int): The first parameter.
    param2 (str): The second parameter.

Returns:
    bool: The return value. True for success, False otherwise.

.. _PEP 484:
    https://www.python.org/dev/peps/pep-0484/
```

## class.def: .function_with_pep484_type_annotations

comm:
```
Example function with PEP 484 type annotations.

Args:
    param1: The first parameter.
    param2: The second parameter.

Returns:
    The return value. True for success, False otherwise.
```

## class.def: .module_level_function

comm:
```
This is an example of a module level function.

Function parameters should be documented in the ``Args`` section. The name
of each parameter is required. The type and description of each parameter
is optional, but should be included if not obvious.

If \*args or \*\*kwargs are accepted,
they should be listed as ``*args`` and ``**kwargs``.
```

```
The format for a parameter is::

    name (type): description
        The description may span multiple lines. Following
        lines should be indented. The "(type)" is optional.

        Multiple paragraphs are supported in parameter
        descriptions.

Args:
    param1 (int): The first parameter.
    param2 (:obj:`str`, optional): The second parameter. Defaults to None.
        Second line of description should be indented.
    *args: Variable length argument list.
    **kwargs: Arbitrary keyword arguments.

Returns:
    bool: True if successful, False otherwise.

    The return type is optional and may be specified at the beginning of
    the ``Returns`` section followed by a colon.

    The ``Returns`` section may span multiple lines and paragraphs.
    Following lines should be indented to match the first line.

    The ``Returns`` section supports any reStructuredText formatting,
    including literal blocks::

        {
            'param1': param1,
            'param2': param2
        }

Raises:
    AttributeError: The ``Raises`` section is a list of all exceptions
        that are relevant to the interface.
    ValueError: If `param2` is equal to `param1`.
```

## class.def: .example_generator

comm:

```
Generators have a ``Yields`` section instead of a ``Returns`` section.

Args:
    n (int): The upper limit of the range to generate, from 0 to `n` - 1.

Yields:
    int: The next number in the range of 0 to `n` - 1.
```

Examples:
    Examples should be written in doctest format, and should illustrate how
    to use the function.

    >>> print([i for i in example_generator(4)])
    [0, 1, 2, 3]

## class.def: ExampleError(Exception).

comm:

Exceptions are documented in the same way as classes.

The __init__ method may be documented in either the class level
docstring, or as a docstring on the __init__ method itself.

Either form is acceptable, but the two should not be mixed. Choose one
convention to document the __init__ method and be consistent with it.

Note:
    Do not include the `self` parameter in the ``Args`` section.

Args:
    msg (str): Human readable string describing the exception.
    code (:obj:`int`, optional): Error code.

Attributes:
    msg (str): Human readable string describing the exception.
    code (int): Exception error code.

## class.def: ExampleClass(object).

comm:

The summary line for a class docstring should fit on one line.

If the class has public attributes, they may be documented here
in an ``Attributes`` section and follow the same formatting as a
function's ``Args`` section. Alternatively, attributes may be documented
inline with the attribute's declaration (see __init__ method below).

Properties created with the ``@property`` decorator should be documented
in the property's getter method.

Attributes:
    attr1 (str): Description of `attr1`.
    attr2 (:obj:`int`, optional): Description of `attr2`.

## class.def: ExampleClass(object).__init__

comm:

    str: Docstring *after* attribute, with type specified.

## class.def: ExampleClass(object).readonly_property

comm:

```
str: Properties should be documented in their getter method.
```

## class.def: ExampleClass(object).readwrite_property

comm:

```
:obj:`list` of :obj:`str`: Properties with both a getter and setter
should only be documented in their getter method.

If the setter method contains notable behavior, it should be
mentioned here.
```

## class.def: ExampleClass(object).example_method

comm:

```
Class methods are similar to regular functions.

Note:
    Do not include the `self` parameter in the ``Args`` section.

Args:
    param1: The first parameter.
    param2: The second parameter.

Returns:
    True if successful, False otherwise.
```

## class.def: ExampleClass(object).__special__

comm:

```
By default special members with docstrings are not included.

Special members are any methods or attributes that start with and
end with a double underscore. Any special member with a docstring
will be included in the output, if
``napoleon_include_special_with_doc`` is set to True.

This behavior can be enabled by changing the following setting in
Sphinx's conf.py::

    napoleon_include_special_with_doc = True
```

## class.def: ExampleClass(object)._private

comm:

```
By default private members are not included.

Private members are any methods or attributes that start with an
underscore and are *not* special. By default they are not included
in the output.
```

This behavior can be changed such that private members *are* included
by changing the following setting in Sphinx's conf.py::

```
    napoleon_include_private_with_doc = True
```