SphinxFortran Documentation

Release 1.0

Stephane Raynaud

October 26, 2015

Contents

CHAPTER 1

Purpose

This package provides two Sphinx (http://sphinx.pocoo.org/) extensions to the Fortran (90) language:

- `sphinxfortran.fortran_domain`: Sphinx domain for fortran.
- `sphinxfortran.fortran_autodoc`: Auto-documenting fortran code.

CHAPTER 2

License

This package has the same license as VACUMM (http://www.ifremer.fr/vacumm) from which it originates: CeciLL-A (http://www.cecill.info/licences/Licence_CeCILL_V2.1-en.html), which is compatible with the GPL.

CHAPTER 3

Prerequisites

The `sphinx` and `numpy` packages.

CHAPTER 4

Installation

Using `pip`:

```
pip install sphinx-fortran
```

From sources:

```
git clone https://github.com/VACUMM/sphinx-fortran.git
cd sphinx-fortran
python setup.py install
```

You can download sources also from the forge at IFREMER: https://forge.ifremer.fr/frs/?group_id=93

CHAPTER 5

Quick start

1. Add this extension to your sphinx `conf.py`.
2. List you fortran source files in the variable `fortran_src` of your `conf.py`.
3. Generate their documentation in rst files using directives like:

```
.. f:automodule:: mymodule
```

CHAPTER 6

Bugs and requests

Please go to this GitHub page: https://github.com/VACUMM/sphinx-fortran/issues

CHAPTER 7

Authors

Stephane Raynaud (stephane.raynaud(at)gmail.com)

Thanks: Thomas Gastine

CHAPTER 8

Documentation

Website: http://sphinx-fortran.readthedocs.org

Contents:

8.1 User manual

8.1.1 Fortran extension to `sphinx.domains`

The module *sphinxfortran.fortran_domain* is an extension to Sphinx adding the FORTRAN "domain" (see the sphinx documentation on syntactic domains), and can therefore document FORTRAN codes.

Configure Sphinx

Just add *sphinxfortran.fortran_domain* to the list defined by the `extension` variable in the file `conf.py` sphinx configuration file (assuming the `vacumm` python package is available to you).

Syntax

This extension provides "guidelines" to declare (describe and reference) fortran entities (program, module, type, function, subroutine and variable), as well as "roles" to refer to the declared entities.

Directives

All directives are prefixed by `f:` to refer to the fortran domain.

Note: Thereafter, the brackets `[]` denote an optional argument.

All directives accept content to describe the entity, which will interpreted by sphinx. This description can also take advantage of *docfields* to describe the arguments of functions, subroutines and programs.

`.. f:program:: progname`
Description of a FORTRAN program. This directive accepts *docfields*.

Example:

```
.. f:program:: main

    This is the main program.
```

.. **f:module::** modname
This creates a reference to a module and produces no output. It accepts the `:synopsis:` option to briefly describe the module in the modules index. It also defines the current module, like *f:currentmodule*.

Example:

```
.. f:module:: monmodule
    :synopsis: Statistics module
```

.. **f:currentmodule::** [modname]
This directive produces no output, it makes of `modname` the current module: functions, subroutines, types and variables described in the following will be considered as belonging to this module. If `modname` is empty or equal to `None`, there is no more current module.

Example:

```
.. f:currentmodule:: mymodule

.. f:variable:: float myvar

.. f:currentmodule::
```

.. **f:type::** [~][modname][/]typename
This directive describes a derived type in a module. It accepts the special docfield *:f ...:* to describe the fields of the module.

Example:

```
.. f:type:: mymod/mytype

    :f integer var1: Variable 1
    :f float var2: Variable 2
```

.. **f:variable::** [type] [~][modname][/]varname[(shape)]
This directive describes a variable of a module. It accepts the following options:

```
- ``:type:``: Type of the variable (``float``, ``mytype``, etc).
  Si présent, un lien est créé vers ce type.
  The type can also be specified before the variable name.
- ``:shape:``: Shape in the form ``nx,ny``,
  which can also be declared after the name (in brackets).
  A reference to all variables found is created.
- ``:attrs:``: Additional Attributes (``in``, ``parameter``, etc).
```

Example:

```
.. f:function:: float myvar
    :shape: nx,ny
    :attrs: in

    Description of my variable.
```

.. **f:function::** [type] [~][modname][/]funcname(signature)
This directive describes a function belonging or not to a module. It accepts the option *:type:* and uses *docfields* to describe his arguments, his calls and modules used.

Example:

```
    .. f:function:: myfunc(a [,b])

        This is my primary function.

        :p float a: My first argument.
        :o integer b [optional]: My second argument.
        :from: :f:subr:`mysub`
```

.. f:subroutine:: [~][modname][/]subrname[(signature)]

This directive describes a subroutine like the directive *f:function*.

Example:

```
    .. f:subroutine:: mysub(a)

        Description.

        :param a: My parameter.
        :to: :f:func:`myfunc`
```

The roles

The roles allow in rst language to refer to entities (program, function, etc.). They are used with a syntax like:

```
:role:`cible`
:role:`nom <cible>` # avec nom alternatif
```

Several have been defined with respect to fortran guidelines presented above.

:f:prog:
Reference to a program declared with *f:program*.

:f:mod::
Reference to a module declared with *f:module*.

:f:type::
Reference to a derived type declared with *f:type*.

:f:var::
Reference to a variable declared with *f:variable*.

:f:func:
Reference to a fonction or a subroutine declared respectively with *f:function* and *f:subroutine*.

:f:subr::
Alias for *f:func*.

It is possible to make reference to derived types, variables, functions and subroutines belonging to a module, with the typical following syntax:

```
monmodule/myfunction()
```

If a local function and the module have the same name, it is possible to use the following syntax if the current module is the same as the function module:

```
/myfunction()
```

If the / is omitted, the reference will focus on the local function and not that of the current module.

If the module is specified in the reference, it is possible not to display the name by prefixing "~":

```
:f:func:`~mymodule/myfunction`
```

The *docfields*

The docfields are rst tags of type field list, which are interpreted in the content of certain directives to describe the settings, options and other special fields.

The fortran domain allows a use pretty close to that implemented for the python domain.

There are two families of fortran *docfields*: one for functions and subroutines, and one for derived types.

Fonctions et subroutines family

For this family, the *docfields* are used to describe the mandatory and optional arguments, the modules used, the programs, functions and subroutines that call the current entity, and the functions and subroutines called by this entity. Some of them have aliases.

- `param` (or `p`, `parameter`, `a`, `arg`, `argument`): Mandatory argument.

```
:param myvar: Description.
```

  It is possible to specify the type, the size and special attributes in the declaration following the example below:

  param mytype myvar(nx,ny) [in] Description.

- `type` (or `paramtype`, `ptype`): Parameter type (eg: float). Reference is made to this parameter if present.

```
:type: float
```

- `shape` (ort `pshape`): Shape of the parameter (dimensions are separated by commas).

  shape nx, ny

- `attrs` (or `pattrs`, `attr`): Special Attributes (separated by commas).

```
:attr: in/out
```

- `option` (or `o`, `optional`, `keyword`): Declaration of an optional parameter with a similar syntax to required parameters (`param`).
- `otype` (or `optparamtype`): Its type.
- `oshape` : Its shape.
- `oattrs` (or `oattrs`): Its attributes.
- `return` (or `r`, `returns`): Variable returned by the function.
- `rtype` (or `returntype`): Its type.
- `rshape`: Its shape.
- `rattrs` (or ``rattrs`): Its attributes.
- `calledfrom` (or `from`): Functions, subroutines or programs calling the current routine.
- `callto` (or `to`): Fonctions ou subroutines called by the current routine.

The **docfiels** are merged into one list for those mandatory, and another one for those optional, and their associated **docfiels** (type, dimensions et attributs) are deleted and inserted in the parameter declaration.

Note: In addition to these *docfiels* that are intepreted, you can add other of your choice. For example:

```
:actions: This function performs

    #) Une initialisation.
    #) Un calcul.
    #) Un plot.

:p float myvar [in]: Variable à tracer.
:use: Fait appel au module :f:mod:`mymod`.
```

Derived types family

These *docfields* describe the fields of derived types. They are inserted into the header of a *f:type* directive.

- `ftype` (or `f`, typef, typefield): Fields of a derived type with a syntax similar to that of required parameters or routines (`param`).
- `ftype` (or `fieldtype`): Its type.
- `fshape`: Its shape.
- `fattrs` ` (or `fattrs`): Its attributes.

Example

```
**Programme**

.. f:program:: make_stats

    Programm for computing statistics calcul des statistiques

**Module** :f:mod:`mymodule`

.. f:module:: mymodule
    :synopsis: Main statistical module

.. f:variable:: nx
    :type: integer
    :attrs: parameter=5

    Zonal dimension.

.. f:variable:: sst(nx)
    :type: float

    SST du modèle.

.. f:type:: obs

    Type that describes observations.

    :f x(nx): zonal axis.
    :ftype x: float
    :f float sst(nx): SST

.. f:subroutine:: stats(data, b, [c, d])
```

```
    Description of the routine.

    :param obs data: Data to analyse.
    :p integer a(nx,5) [in]: Also mandatory.
    :o float c [optional]: Optional.
    :o d: Also optional.
    :from: :f:prog:`make_stats`.
    :to: :f:func:`rms`

**Module** :f:mod:`special_stats`

.. f:module:: special_stats

.. f:function:: rms(mod, obs, unbiased)

    Compute RMS errors.

    :p float mod(nx) [in]: Model outputs.
    :p float obs(nx) [in]: Observations.
    :p logical mask(\:) [in]: Mask.
    :o logical unbiased [default=.true.]: Biased?
    :r rms(nx): Computed RMS.
    :rtype rms: float
    :from: :f:func:`mymodule/stats` :f:func:`~mymodule/stats` :f:subr:`stats` :f:func:`stats`

.. f:currentmodule::
```

Which gives:

Programme

program **make_stats**
Programm for computing statistics calcul des statistiques

Module *mymodule*

mymodule/**nx** *[integer,parameter=5]*
Zonal dimension.

mymodule/**sst** (*nx*) *[float]*
SST du modèle.

type mymodule/**obs**
Type that describes observations.

Type fields

- **% x** (*nx*) *[float]* :: zonal axis.
- **% sst** (*nx*) *[float]* :: SST

subroutine mymodule/**stats** (*data*, *b*[, *c*, *d*])
Description of the routine.

Parameters

- **data** *[obs]* :: Data to analyse.
- **a** (*nx*,5) *[integer,in]* :: Also mandatory.

Options

- **c** *[float,optional]* :: Optional.

- **d** :: Also optional.

Called from *make_stats*.

Call to *rms()*

Module *special_stats*

function special_stats/**rms** (*mod*, *obs*, *unbiased*)
Compute RMS errors.

Parameters

- **mod** (*nx*) *[float,in]* :: Model outputs.
- **obs** (*nx*) *[float,in]* :: Observations.
- **mask** (*) *[logical,in]* :: Mask.

Options **unbiased** *[logical,default=.true.]* :: Biased?

Return **rms** (*nx*) *[float]* :: Computed RMS.

Called from *mymodule/stats() stats() stats() stats()*

Note: Declared modules are listed in their index, and other fortran entities are also accesible from the main index.

8.1.2 Auto-documenting fortran codes

Sphinx extension *sphinxfortran.fortran_autodoc* provides directives for semi-automatically documenting (F90+) fortran codes. It helps describing et referencing programs, modules, derived types, functions, subroutines and variables in documentatin generated by sphinx.

Note: You need modules numpy et *sphinxfortran.fortran_domain* tu use this extension.

How it works

The process of auto-documentation is the following:

1. The first step consists in **analyzing the code** included in a list of fortran files.
   (a) The module numpy.f2py.crackfortran first indexes all fortran entities (modules, functions, calling arguments, etc).
   (b) Then all comments associated to identified entities are extracted to get complementary information.
2. The second step **auto-documments on demand** an entity indexed during the first step, using the sphinx extension *fortran_domain*.

Usage

Configure Sphinx

You can configure sphinx by editing the file conf.py (see documentation).

You must first **load the extension**:

- *sphinxfortran.fortran_domain*: manual documentation of fortran code.

- *sphinxfortran.fortran_autodoc*: auto-documentation.

Just add the name of the two modules to the list of the configuration variable extension.

Then, you must specify the **list of fortran source files** in the configuration variable *fortran_src*.

Here are the available configuration variables.

fortran_src
This variable must be set with file pattern, like `"*.f90`, or a list of them. It is also possible to specify a directory name; in this case, all files than have an extension matching those define by the config variable *fortran_ext* are used.

Note: All paths are relative to the sphinx configuration directory (where the `conf.py` is).

---

fortran_ext
List of possible extensions in the case of a directory listing (default: `['f90', 'f95']`).

fortran_encoding
Character encoding of fortran files (default : `"utf8"`).

Note: It is strongly recommanded to encode your sources with a set of universal character as UTF-8.

---

fortran_subsection_type
Section type for the documentation of modules and files. Choice:

- `"rubric"` (default) : use directive `rubric` (lightweight title in bold).
- `"title"` : uses a conventional title (text with underlining, whose character is defined by u *fortran_title_underline*).

fortran_title_underline
Character used for underlining (default `"-"`) if `fortran_subsection_type = "title"`.

fortran_indent
Indentation string or length (default `4`). If it is an integer, indicates the number of spaces.

Inserting an auto-documentation

The insertion of an auto-documentation can be chosen with the following diectives.

`.. f:autoprogram:: progname`
Document a program.

`.. f:autofunction:: [modname/]funcname`
Document a function.

`.. f:autosubroutine:: [modname/]subrname`
Document a subroutine.

`.. f:autotype:: [modname/]typename`
Document a derived type.

`.. f:autovariable:: [modname/]varname`
Document a module variable.

`.. f:autovariable:: modname`
Document a module. This directive accepts options `:subsection_type:` and `:title_underline:`.

`.. f:autosrcfile:: pathname`
Document programs, functions and subroutines of a source file. This directive accepts options `::search_mode:` and `:objtype:` (see *filter_by_srcfile()*). Example:

```
.. f:autosrcfile::  myfile.f90
    :search_mode: basename
    :objtype: function subroutine
```

> **Warning:** Untested directive!

Optimize the process

To optimize the process of documentation, it is recommended to follow some rules when commenting FORTRAN codes: these comments provide a way to better decribe fortran entities, and are interpreted in rst language.

Header comments

The comments in the **modules** headers, up to the first line of code, are are systématically used. Example:

```
module mymod

! This is my **super** module and its description

integer :: var

end module mymod
```

In the case of **programs**, **functions**, **subroutines** and **types**, comments are used if they start immediately after the declaration line. Examples:

```
subroutine mysub(a)
! Description
end subroutine mysub

type mytype
    ! Description
    integer :: var
end type mytype
```

Inline comments

These comments are in a line of code. They are used to declare **fields of derived types**, **module variables** et **arguments of functions and subroutines**. Example:

```
type mytype
    integer :: myvar &, ! Description1
        &       myvar2   ! Description2
end type mytype

subroutine mysub(a, b)
    ! Description mysub
     integer, intent(in) :: a ! Description a
     real, intent(out) :: b   ! Description b
end subroutine mysub
```

Warning: There must have only one declaration of variable or field a description comment is specified.

8.2 Library

8.2.1 sphinxfortran.fortran_domain – Fortran domain

A fortran domain for sphinx

class sphinxfortran.fortran_domain.**FortranCallField**(*name*, *names=()*, *label=None*, *rolename=None*)

Bases: *sphinxfortran.fortran_domain.FortranField*

is_grouped = True

make_field(*types*, *domain*, *items*, ***kwargs*)

class sphinxfortran.fortran_domain.**FortranCompleteField**(*name*, *names=()*, *typenames=()*, *label=None*, *rolename=None*, *typerolename=None*, *shapenames=None*, *attrnames=None*, *prefix=None*, *strong=True*, *can_collapse=False*)

Bases: *sphinxfortran.fortran_domain.FortranField*, sphinx.util.docfields.GroupedField

A doc field that is grouped and has type information for the arguments. It always has an argument. The argument can be linked using the given *rolename*, the type using the given *typerolename*.

Two uses are possible: either parameter and type description are given separately, using a field from *names* and one from *typenames*, respectively, or both are given using a field from *names*, see the example.

Example:

```
:param foo: description of parameter foo
:type foo:  SomeClass

-- or --

:param SomeClass foo: description of parameter foo
```

is_typed = 2

make_field(*types*, *domain*, *items*, *shapes=None*, *attrs=None*, *modname=None*, *typename=None*)

class sphinxfortran.fortran_domain.**FortranCurrentModule**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

This directive is just to tell Sphinx that we're documenting stuff in module foo, but links to module foo won't lead here.

final_argument_whitespace = False

has_content = False

option_spec = {}

optional_arguments = 1

required_arguments = 0

run()

class sphinxfortran.fortran_domain.**FortranDocFieldTransformer**(*directive*, *modname=None*, *typename=None*)

Bases: sphinx.util.docfields.DocFieldTransformer

Transforms field lists in "doc field" syntax into better-looking equivalents, using the field type definitions given on a domain.

preprocess_fieldtypes(*types*)

scan_fieldarg(*fieldname*)

Extract type, name, shape and attributes from a field name.

Some possible syntaxes

- `p name`
- `p type name(shape) [attr1,attr2]`
- `p type name`
- `p name [attr1, attr2]`

Returns `name, shape, type, list of attributes.` if no shape is specified, it is set to `None`,

transform(*node*)

Transform a single field list *node*.

class sphinxfortran.fortran_domain.**FortranDomain**(*env*)

Bases: sphinx.domains.Domain

Fortran language domain.

clear_doc(*docname*)

directives = {'function': <class 'sphinxfortran.fortran_domain.FortranWithSig'>, 'currentmodule': <class 'sphinxfor

find_obj(*env*, *modname*, *name*, *role*, *searchorder=0*)

Find a Fortran object for "name", perhaps using the given module and/or typename.

Params

- **searchorder**, optional: Start using relative search

get_objects()

indices = [<class 'sphinxfortran.fortran_domain.FortranModuleIndex'>]

initial_data = {'objects': {}, 'modules': {}}

label = 'Fortran'

name = 'f'

object_types = {'function': <sphinx.domains.ObjType object at 0x7f4b9adfaa90>, 'subroutine': <sphinx.domains.Obj

resolve_xref(*env*, *fromdocname*, *builder*, *type*, *target*, *node*, *contnode*)

roles = {'subr': <sphinxfortran.fortran_domain.FortranXRefRole object at 0x7f4b9adfad50>, 'var': <sphinxfortran.for

class sphinxfortran.fortran_domain.**FortranField**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

Directive to describe a change/addition/deprecation in a specific version.

final_argument_whitespace = True

has_content = True

option_spec = {'shape': <function parse_shape at 0x7f4b9ae71398>, 'type': <function unchanged at 0x7f4b9de485f0>,

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_domain.**FortranModule**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

Directive to mark description of a new module.

final_argument_whitespace = False

has_content = False

option_spec = {'noindex': <function flag at 0x7f4b9de48500>, 'platform': <function <lambda> at 0x7f4b9adff140>, 'sy

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_domain.**FortranModuleIndex**(*domain*)

Bases: sphinx.domains.Index

Index subclass to provide the Fortran module index.

generate(*docnames=None*)

localname = iu'Fortran Module Index'

name = 'modindex'

shortname = iu'fortran modules'

class sphinxfortran.fortran_domain.**FortranObject**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: sphinx.directives.ObjectDescription

Description of a general Fortran object.

_parens = ''

add_shape_and_attrs(*signode*, *modname*, *ftype*, *shape*, *attrs*)

add_target_and_index(*name*, *sig*, *signode*)

after_content()

before_content()

doc_field_types = [<sphinxfortran.fortran_domain.FortranCompleteField object at 0x7f4b9adfa210>, <sphinxfortra

get_index_text(*modname*, *name*)

get_signature_prefix(*sig*)
May return a prefix to put before the object name in the signature.

handle_signature(*sig*, *signode*)
Transform a Fortran signature into RST nodes. Returns (fully qualified name of the thing, classname if any).

If inside a class, the current class name is handled intelligently: * it is stripped from the displayed name if present * it is added to the full name (return value) if not present

needs_arglist()
May return true if an empty argument list is to be generated even if the document contains none.

option_spec = {'noindex': <function flag at 0x7f4b9de48500>, 'shape': <function parse_shape at 0x7f4b9ae71398>, 'ty

stopwords = set(['integer', 'float', 'character', 'long', 'double'])

class sphinxfortran.fortran_domain.**FortranProgram**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: *sphinxfortran.fortran_domain.FortranSpecial*, *sphinxfortran.fortran_domain.WithFortr*
sphinxfortran.fortran_domain.FortranObject

class sphinxfortran.fortran_domain.**FortranSpecial**

get_signature_prefix(*sig*)
May return a prefix to put before the object name in the signature.

class sphinxfortran.fortran_domain.**FortranType**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: *sphinxfortran.fortran_domain.FortranSpecial*, *sphinxfortran.fortran_domain.WithFortr*
sphinxfortran.fortran_domain.FortranObject

before_content()

class sphinxfortran.fortran_domain.**FortranTypeField**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: *sphinxfortran.fortran_domain.FortranObject*

before_content()

class sphinxfortran.fortran_domain.**FortranWithSig**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: *sphinxfortran.fortran_domain.FortranSpecial*, *sphinxfortran.fortran_domain.WithFortr*
sphinxfortran.fortran_domain.FortranObject

Description of a function of subroutine

_parens = '()'

get_signature_prefix(*sig*)
May return a prefix to put before the object name in the signature.

needs_arglist()

class sphinxfortran.fortran_domain.**FortranXRefRole**(*fix_parens=False*, *lowercase=False*, *nodeclass=None*, *innernodeclass=None*, *warn_dangling=False*)
Bases: sphinx.roles.XRefRole

process_link(*env*, *refnode*, *has_explicit_title*, *title*, *target*)

class sphinxfortran.fortran_domain.**WithFortranDocFieldTransformer**

run()
Same as sphinx.directives.ObjectDescription() but using *FortranDocFieldTransformer*

sphinxfortran.fortran_domain.**_pseudo_parse_arglist**(*signode*, *arglist*)
"Parse" a list of arguments separated by commas.

Arguments can have "optional" annotations given by enclosing them in brackets. Currently, this will split at any comma, even if it's inside a string literal (e.g. default argument value).

sphinxfortran.fortran_domain.**add_shape**(*node*, *shape*, *modname=None*, *nodefmt=<class 'docutils.nodes.Text'>*)
Format a shape expression for a node

sphinxfortran.fortran_domain.**convert_arithm**(*node*, *expr*, *modname=None*, *nodefmt=<class 'docutils.nodes.Text'>*)
Format an arithmetic expression for a node

sphinxfortran.fortran_domain.**parse_shape**(*shape*)

sphinxfortran.fortran_domain.**re_fieldname_match**()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

sphinxfortran.fortran_domain.**setup**(*app*)

8.2.2 sphinxfortran.fortran_autodoc – Fortran auto-documenter

Sphinx extension for autodocumenting fortran codes.

class sphinxfortran.fortran_autodoc.**F90toRst**(*ffiles*, *ic='t'*, *ulc='-'*, *vl=0*, *encoding='utf8'*, *sst='rubric'*)
Bases: object

Fortran 90 parser and restructeredtext formatter

Parameters

- **ffiles**: Fortran files (glob expression allowed) or dir (or list of)

Options

- **ic**: Indentation char.
- **ulc**: Underline char for titles.
- **sst**: Subsection type.
- **vl**: Verbose level (0=quiet).

_fmt_fvardesc = '%(vtype)s%(vdim)s %(vattr)s%(vdesc)s'

_fmt_vardesc = ':%(role)s %(vtype)s %(vname)s%(vdim)s%(vattr)s: %(vdesc)s'

_fmt_vattr = ' [%(vattr)s]'

_re_comment_match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

_re_space_prefix_match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

_re_unended_match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

_re_unstarted_match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

build_callfrom_index()
For each function, index which function call it

build_index()
Register modules, functions, types and module variables for quick access

Index constituents are:

modules
Dictionary where each key is a module name, and each value is the cracked block.

routines
Module specific functions and subroutines

types
Module specific types

variables
Module specific variables

filter_by_srcfile(*sfile*, *mode=None*, *objtype=None*, ***kwargs*)
Search for subblocks according to origin file

Params

- **sfile**: Source file name.
- **mode**, optional: Mode for searching for sfile. If `"strict"`, exact match is needed, else only basename.
- **objtype**, optional: Restrict search to one or a list of object types (i.e. `"function"`, `"program"`, etc).

format_argattr(*block*)
Filter and format the attributes (optional, in/out/inout, etc) of a variable

Parameters

- *block*: a variable block

format_argdim(*block*)
Format the dimension of a variable

Parameters

- *block*: a variable block

format_argfield(*blockvar*, *role=None*, *block=None*)
Format the description of a variable

Parameters

- *block*: a variable block

format_argtype(*block*)

format_declaration(*dectype*, *name*, *description=None*, *indent=0*, *bullet=None*, *options=None*)
Create an simple rst declaration

Example

```
>>> print format_declaration('var', 'myvar', 'my description', indent=1, bullet='-')
- .. f:var:: myvar
```

my description

format_description(*block*, *indent=0*)
Format the description of an object

format_funcref(*fname*, *current_module=None*, *aliasof=None*, *module=None*)
Format the reference link to a module function

Formatting may vary depending on if function is local and is an alias.

Example

```
>>> print obj.format_type('myfunc')
:f:func:`~mymodule.myfunc`
```

format_function(*block*, *indent=0*)
Format the description of a function, a subroutine or a program

format_lines(*lines*, *indent=0*, *bullet=None*, *nlc='\n'*, *strip=False*)
Convert a list of lines to text

format_module(*block*, *indent=0*)
Recursively format a module and its declarations

format_options(*options*, *indent=0*)
Format directive options

format_quickaccess(*module*, *indent=<function indent>*)
Format an abstract of all types, variables and routines of a module

format_routine(*block*, *indent=0*)
Format the description of a function, a subroutine or a program

format_routines(*block*, *indent=0*)
Format the list of all subroutines and functions

format_rubric(*text*, *indent=0*)
Create a simple rst rubric with indentation

Parameters

- *text*: text of the rubric

Example

```
>>> print o.format_rubric('My title', '-')
.. rubric:: My rubric
```

format_signature(*block*)

format_srcfile(*srcfile*, *indent=0*, *objtype=None*, *search_mode='basename'*, ***kwargs*)
Format all declaration of a file, except modules

format_subroutine(*block*, *indent=0*)
Format the description of a function, a subroutine or a program

format_subsection(*text*, *indent=<function indent>*, ***kwargs*)
Format a subsection for describing list of subroutines, types, etc

format_title(*text*, *ulc=None*, *indent=0*)
Create a simple rst titlec with indentation

Parameters

- *text*: text of the title

Options

- *ulc*: underline character (default to attribute `ucl`)

Example

```
>>> print o.format_title('My title', '-')
My title
--------
```

format_type(*block*, *indent=0*, *bullet=True*)
Format the description of a module type

Parameters

- *block*: block of the type

format_types(*block*, *indent=0*)
Format the description of all fortran types

format_use(*block*, *indent=0*, *short=False*)
Format use statement

Parameters

- *block*: a module block

format_var(*block*, *indent=0*, *bullet=True*)
Format the description of a module type

Parameters

- *block*: block of the variable

format_variables(*block*, *indent=0*)
Format the description of all variables (global or module)

get_blocklist(*choice*, *module*)
Get the list of types, variables or function of a module

get_blocksrc(*block*, *src=None*, *istart=0*, *getidx=False*, *stopmatch=None*, *exclude=None*)
Extract an identified block of source code

Parameters

- *block*: Cracked block

Options

- *src*: List of source line including the block
- *istart*: Start searching from this line number

Return `None` or a list of lines

get_comment(*src*, *iline=1*, *aslist=False*, *stripped=False*, *getilast=False*, *rightafter=True*)
Search for and return the comment starting after `iline` in `src`

Params

- **src**: A list of lines.
- **iline**, optional: Index of first line to read.
- **aslist**, optional: Return the comment as a list.
- **stripped**, optional: Strip each line of comment.
- **getilast**, optional: Get also index of last line of comment.
- **rightafter**, optional: Suppose the comment right after the signature line. If True, it prevents from reading a comment that is not a description of the routine.

Return

- `scomment`: string or list
- OR `scomment,ilast`: if `getilast is True`

get_ic()
Get the indentation character

get_module(*block*)
Get the name of the current module

get_src(*block*)
Get the source lines of the file including this block

get_sst()
Get the subsection type

get_synopsis(*block*, *nmax=2*)
Get the first `nmax` non empty lines of the function, type or module comment as 1 line.

If the header has more than `nmax` lines, the first one is taken and appended of '...'. If description if empty, it returns an empty string.

get_ulc()
Get the underline character for title inside module description

get_varopts(*block*)
Get options for variable declaration as a dict

ic
Indentation character

indent(*n*)
Get a proper indentation

join_src(*src*)
Join unended lines that does not finish with a comment

scan()
Scan

scan_container(*block*, *insrc=None*)
Scan a block of program, routines or type

set_ic(*ic*)
Set the indentation character

set_sst (*sst*)
Set the subsection type

set_ulc (*ulc*)
Set the underline character for title inside module description

sst
Subsection type (“title” or “rubric”)

strip_blocksrc (*block*, *exc*, *src=None*)
Strip blocks from source lines

Parameters

- *block*:
- *exc* list of block type to remove

Options

- *src*: list of source lines

Example

```
>>> obj.strip_blocksrc(lines, 'type')
>>> obj.strip_blocksrc(lines, ['function', 'type']
```

ulc
Underline character for title inside module description

exception sphinxfortran.fortran_autodoc.**F90toRstException**
Bases: exceptions.Exception

class sphinxfortran.fortran_autodoc.**FortranAutoFunctionDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: *sphinxfortran.fortran_autodoc.FortranAutoObjectDirective*

_objtype = ‘function’

_warning = ‘Wrong function name: %s’

class sphinxfortran.fortran_autodoc.**FortranAutoModuleDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: docutils.parsers.rst.Directive

has_content = True

option_spec = {‘title_underline’: <function unchanged at 0x7f4b9de485f0>, ‘indent’: <function fmt_indent at 0x7f4b9

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_autodoc.**FortranAutoObjectDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

Generic directive for fortran object auto-documentation

Redefine *_warning* and *_objtype* attribute when subcassling.

_warning
Warning message when object is not found, like:

```
>>> _warning = 'Wrong function or subroutine name: %s'
```

_objtype
Type of fortran object. If "toto" is set as object type, then *F90toRst* must have attribute `totos` containg index of all related fortran objects, and method `format_totos()` for formatting the object.

_objtype = 'routine'

_warning = 'Wrong routine name: %s'

has_content = False

option_spec = {}

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_autodoc.**FortranAutoProgramDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

has_content = False

option_spec = {}

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_autodoc.**FortranAutoSrcfileDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

has_content = False

option_spec = {'objtype': <function unchanged at 0x7f4b9de485f0>, 'search_mode': <function unchanged at 0x7f4b9d

optional_arguments = 0

required_arguments = 1

run()

class sphinxfortran.fortran_autodoc.**FortranAutoSubroutineDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: *sphinxfortran.fortran_autodoc.FortranAutoObjectDirective*

_objtype = 'subroutine'

_warning = 'Wrong subroutine name: %s'

class sphinxfortran.fortran_autodoc.**FortranAutoTypeDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: *sphinxfortran.fortran_autodoc.FortranAutoObjectDirective*

_objtype = 'type'

_warning = 'Wrong type name: %s'

class sphinxfortran.fortran_autodoc.**FortranAutoVariableDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: *sphinxfortran.fortran_autodoc.FortranAutoObjectDirective*

_objtype = 'variable'

_warning = 'Wrong variable name: %s'

sphinxfortran.fortran_autodoc.**fmt_indent**(*string*)

sphinxfortran.fortran_autodoc.**fortran_parse**(*app*)

sphinxfortran.fortran_autodoc.**list_files**(*fortran_src*, *exts=['f', 'f90', 'f95']*, *absolute=True*)

Get the list of fortran files

sphinxfortran.fortran_autodoc.**setup**(*app*)

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

m

s

S

Symbols

A

B

C

D

F

G

P

R

S

T

U

V

W