Chapel Domain Documentation

Release 0.0.15

Chapel Team

2018-10-16

Contents

Chapel domain for Sphinx! Document Chapel modules and APIs using the Sphinx tool suite.

Package documentation is available on readthedocs.org.

CHAPTER 1

Installation

To install:

```
pip install sphinxcontrib-chapeldomain
```

To install from source on github:

```
git clone https://github.com/chapel-lang/sphinxcontrib-chapeldomain
cd sphinxcontrib-chapeldomain
python setup.py install
```

CHAPTER 2

Getting Started

See the *Chapel Domain Reference* for a description of directives and roles provided by this domain.

For tips on developing the chapel domain (as opposed to using the directives), see the *developer documentation*.

Contents:

- *genindex*
- *modindex*
- *search*

Chapel Domain Reference

The Chapel domain (named `chpl`) provides the directives outlined in this document.

The Chapel domain is strongly influenced by the Python domain that comes with Sphinx. In addition, this documentation takes cues from the documentation on Sphinx domain support, including Python domains.

Module Directives

`.. `**`chpl:module::`**` name`

This directive marks the beginning of the description of a module. It does not create content (like e.g. `chpl:class` does).

This directive will also cause an entry in the global module index.

The `synopsis` option should consist of one sentence describing the module's purpose. it is currently only used in the Global Module Index.

The `platform` option, if present, is a comma-separated list of the platforms on which the module is available. If it is available on all platforms, the option should be omitted.

The `deprecated` option can be given (with no value) to mark a module as deprecated. It will then be designated as such in various locations.

`.. `**`chpl:currentmodule::`**` name`

This directive tells Sphinx that the classes, functions, etc documented from here are in the given module (similar to `chpl:module`), but it will not create index entries, an entry in the Global Module Index, or a link target for `chpl:mod`. This is helpful in situations where documentation for a module's content is spread over multiple files or sections. One location needs to have `chpl:module` directive, and the others can have `chpl:currentmodule`.

Module and Class Contents

The following directives are provided for module and class contents.

`.. chpl:function:: signature`

Describes a module-level function. The signature should include the arguments as given in the Chapel definition. See *Chapel Signatures* for details.

For example:

```
.. chpl:function:: factorial(x: int) : int
```

For iterators, see `chpl:iterfunction`. For methods and method iterators, see `chpl:method` and `chpl:itermethod` respectively.

The description normally includes information about the arguments required, how they are used, side effects, return type, and return description.

This information can (in any `chpl` directive) optionally be given in a structured form, see *Info field lists*.

`.. chpl:iterfunction:: signature`

Describes a module-level iterator. The description should be similar to `chpl:function`.

For example, this would document a module with a `proc` and an `iter`:

```
.. chpl:module:: GMP
   :synopsis: multiple precision integer library

.. chpl:function:: proc factorial(n: int): BigNum

   Calculate and return ``n!``. Since this can result in very large
   numbers, the final result is returned as a :chpl:class:`BigNum`.

   :param n: nth factorial to calculate
   :type n: int
   :rtype: BigNum
   :returns: ``n!``
   :throw TimeoutError: if the the computation takes too long

.. chpl:iterfunction:: iter fibonacci(): BigNum

   Yield fibonacci numbers infinitely. It is up to caller to break
   iteration.

   Often called with :chpl:proc:`zip` to track current number. For
   example:

   .. code-block:: chapel

       for value, n in zip(fibonacci(), 1..) do
         writeln("fibonacci(", n, ") = ", value);

   :ytype: BigNum
   :yields: fibonacci numbers
```

`.. chpl:data:: signature`

Describes global data in a module including `const`, `var`, `param`, `config const`, etc. Class, record, and instance attributes are not documented using this environment.

`.. chpl:type:: signature`

Describes global type in module. Generic types for classes and records are not documented using this environment (see `chpl:attribute` for that).

.. chpl:enum:: `signature`

Describes enumerated type in module, including the constants. For example:

```
.. chpl:enum:: Color { Red, Yellow, Blue }

    Supported colors.

.. chpl:enum:: Weekdays { Sun=0, Mon, Tue, Wed, Thu, Fri, Sat }

    Days for the week. The values associated with the constants can be
    used with Date records.
```

.. chpl:class:: `signature`

Describe a class. The signature can optionally include parentheses with arguments which will be shown as the constructor arguments. See also *Chapel Signatures*.

Methods and attributes belonging to the class should be placed in this directive's body. If they are placed outside, the supplied name should contain the class name so that cross-references still work.

For example:

```
.. chpl:class:: Foo

    .. chpl:method:: bar()

or:

.. chpl:class:: Bar
.. chpl:method:: Bar.baz()
```

The first way is the preferred one.

.. chpl:record:: `signature`

Records work the same as `chpl:class`.

.. chpl:attribute:: `signature`

Describes an object data attribute. This can be a `param`, `const`, `var`, `type`, etc. The description should include information about the type of the data to be expected and whether it may be changed directly.

.. chpl:method:: `signature`

Describes an object instance method (for `chpl:class` or `chpl:record`). The description should include similar information to that described for `chpl:function`. See also *Chapel Signatures* and *Info field lists*.

.. chpl:itermethod:: `signature`

Describes an object instance iterator method (for `chpl:class` or `chpl:record`). The description should be similar to `chpl:iterfunction`.

Chapel Signatures

Signatures of functions, methods, classes, records, iterators, etc can be specified similar to how they would be written in Chapel.

Default values for optional arguments can be given. Signatures can also include their declarations, return types, and return intents. For example:

```
.. function:: inline proc foo()

.. iterfunction:: inline iter bar() ref

.. function:: proc baz(ref x) const

.. data:: config const n: int

.. type:: type T = domain(3, int, true)

.. attribute:: param MyMod.MyClass.communicative: bool = false

.. itermethod:: iter MyMod.MyClass.these(): string
```

Info field lists

Inside Chapel description directives, ReST field lists with these fields are recognized and formatted nicely:

- `param`, `parameter`, `arg`, `argument`: Description of a parameter.
- `type`: Type of a parameter. Creates a link if possible.
- `returns`, `return`: Description of the return value.
- `rtype`: Return type. Creates a link if possible.
- `yields`, `yield`: Description of the yield value, often used for iterators.
- `ytype`: Yield type. Creates a link if possible.

For `param`, `arg`, `type`, etc a field name must consist of one of the keywords and an argument. `returns`, `rtype`, `yields`, `ytype`, do not need an argument. See example:

```
.. chpl:module:: GMP
    :synopsis: multiple precision integer library

.. chpl:record:: BigNum

    multiple precision instances

    .. chpl:method:: proc add(a:BigNum, b:BigNum)

        Add two big ints, ``a`` and ``b``, and store the result in ``this``
        instance.

        :arg a: BigNum to be added
        :type a: BigNum

        :arg BigNum b: BigNum to be added

        :returns: nothing, result is stored in current instance

    .. chpl:itermethod:: iter these() ref

        Arbitrary iterator that returns individual digits of this instance.

        :ytype: reference
        :yields: reference to each individual digit of BigNum
```

The above will render like this:

record `BigNum`

multiple precision instances

proc `add``(`*a:BigNum*, *b:BigNum*`)`

Add two big ints, `a` and `b`, and store the result in `this` instance.

Arguments

- **a** : *BigNum* – BigNum to be added
- **b** : *BigNum* – BigNum to be added

Returns nothing, result is stored in current instance

iter `these``()` ref

Arbitrary iterator that returns individual digits of this instance.

Yield type reference

Yields reference to each individual digit of BigNum

Note that it is possible to combine the `arg` and `type` fields into a single `arg` field, like `:arg BigNum b:`. The same is true for `param` fields.

Cross-referencing Chapel objects

The following roles refer to objects in modules and are possibly hyperlinked if a matching identifier is found:

`:chpl:mod:`

Reference a module; a dotted name may be used. See *Cross-reference Contents* for details on dotted and non-dotted names.

`:chpl:proc:`
`:chpl:iter:`

Reference a Chapel function or iterator. The role text needs not include trailing parentheses to enhance readability.

These can also be used to reference a method or iterator on an object (class or record instance). The role text can include the type name and the method, in those cases. If it occurs within the description of a type, the type name can be omitted.

Dotted names may be used for any form.

`:chpl:data:`
`:chpl:const:`
`:chpl:var:`
`:chpl:param:`
`:chpl:type:`

Reference a module-level variable, constant, compiler param, or type.

`:chpl:class:`
`:chpl:record:`

Reference a class or record; a dotted name may be used.

`:chpl:attr:`

Reference a data attribute (const, var, param, generic type) of an object.

`:chpl:chplref:`

Special Chapel reference, which acts just like `:ref:`. Used to cross-reference the “Chapel Module Index”. For example:

```
* :chpl:chplref:`chplmodindex`
* :chpl:chplref:`The module index <chplmodindex>`

Or, with the default-domain or primary_domain set to chpl:

For example, see all modules in the :chplref:`chplmodindex`.
```

New in version 0.0.3.

> **Warning:** `chplmodindex` is a special name, like `modindex`, `genindex`, and `search`. Do not create documents named `chplmodindex`, as it will cause problems.

Cross-reference Contents

The name enclosed in this markup can include a module name and/or a class or record name. For example, `:chpl:proc:`writeln`` could refer to a function named `writeln` in the current module, or the built-in function of that name. In contrast, `:chpl:proc:`Foo.writeln`` clearly refers to the `writeln` function in the `Foo` module.

Normally, names in these roles are searched first without any further qualification, then with the current module name prepended, then with the current module and class name (if any) prepended. If you prefix the name with a dot, this order is reserved. For example, in the documentation of the `IO` module, `:chpl:proc:`writeln`` always refers to the built-in function, while `:chpl:proc:`.writeln`` refers to `IO.writeln`.

For example, here is a description with both a non-dotted and a dotted cross-reference:

```
.. module:: IO

.. class:: channel

    .. method:: read()

        Description...
        example 1 --> :chpl:proc:`writeln`
        example 2 --> :chpl:proc:`.writeln`
```

Example 1 will search for `writeln` cross-reference in this order:

1. `writeln`: built-in function
2. `IO.writeln`: writeln defined in IO module
3. `IO.channel.writeln`: writeln defined on IO.channel class

Example 2 will search for `writeln` cross-reference in the opposite order, because it is dotted:

1. `IO.channel.writeln`: writeln defined on IO.channel class
2. `IO.writeln`: writeln defined in IO module
3. `writeln`: built-in function

A similar heuristic is used to determine whether the name is an attribute of the currently documented class.

Also, if the name is prefixed with a dot, and no exact match is found, the target is taken as a suffix and all object names with that suffix are searched. For example, `:chpl:meth:`.channel.read`` references the `IO.channel.read()` function, even if the current module is not `IO`. Since this can get ambiguous, if there is more than one possible match, you will get a warning from Sphinx.

When ~ prefix is added to the cross-reference, the visible link will only display the leaf. For example, `:chpl:meth:`~IO.channel.read`` will display as `read` and still reference `IO.channel.read()` method.

Note that you can combine the ~ and . prefixes. `:chpl:meth:`~.channel.read`` will reference the `IO.channel.read()` method, but the visible link caption will only be `read`.

Sphinx Configuration

This section lists additional configuration values that are added to the "build configuration file", i.e. `conf.py`, when using the Chapel domain.

`chapeldomain_modindex_common_prefix`

A list of prefixes that are ignored for sorting the Chapel module index (e.g. if this is set to `['foo.']`, then `foo.bar` module is shown under `B`, instead of `F`). This is useful when documenting a project that consists of a single package. Currently only works for the HTML builder. Default is `[]`.

New in version 0.0.3.

Developer Documentation

This document describes the tools and practices used to develop the Chapel domain.

Overview

- Fork the repo on github.
- Create a well named branch.
- Run testing.
- Submit pull request.
- Email or @mention the team in the comments.

```
git clone <url_for_fork>
cd sphinxcontrib-chapeldomain/
git checkout -b <branch_name>
pip install -r requirements.txt -r test-requirements.txt
... develop ...
tox
```

Testing

Travis runs the tests automically and records code coverage in Coveralls. On a local workstation, tox can be used to run the tests in a similar fashion.

```
tox                # run unittests with py27, py34
tox -e flake8      # flake8 source code checker
tox -e coverage    # run code coverage analysis
tox -e docs        # verify the docs build
tox -e doc-test    # verify the acceptance tests build
```

Release

To release the latest sources on PyPI and tag the repo, update your working copy to the latest master, then use the `util/release.bash` script:

```
git checkout master
git pull https://github.com/chapel-lang/sphinxcontrib-chapeldomain master
./util/release.bash
```

The script does the following:

- Ensure the current branch is master.
- Ensure it is run inside a virtualenv.
- Discover the version, using `python setup.py --version`.
- Install the regular, docs, and test requirements.
- Install the package in develop mode.
- Run tox against several environments.
- Tag the tip of master with the version number, and push the tag to the remote.
- Clean the repo.
- Run the python package build and upload to PyPI.