

---

# nunavut Documentation

*Release 2.1.1*

**OpenCyphal Development Team**

**Jun 14, 2023**



---

## Contents

---

<b>1</b>	<b>nunavut</b>	<b>3</b>
<b>2</b>	<b>Software Language Generation Guide</b>	<b>23</b>
2.1	C++ (experimental) . . . . .	23
<b>3</b>	<b>Template Language Guide</b>	<b>25</b>
3.1	Environment . . . . .	25
3.2	Template Mapping and Use . . . . .	48
3.3	Built-in Template Guide . . . . .	50
<b>4</b>	<b>nnvg</b>	<b>51</b>
4.1	Usage . . . . .	51
<b>5</b>	<b>Contributor Notes</b>	<b>57</b>
5.1	Tools . . . . .	57
5.2	Running The Tests . . . . .	58
5.3	Building The Docs . . . . .	60
5.4	Coverage and Linting Reports . . . . .	60
5.5	Nunavut Verification Suite . . . . .	61
<b>6</b>	<b>Licenses</b>	<b>67</b>
6.1	Licence . . . . .	67
<b>7</b>	<b>Nunavut: DSDL transpiler</b>	<b>69</b>
7.1	Installation . . . . .	70
7.2	Examples . . . . .	70
7.3	Bundled third-party software . . . . .	71
7.4	Documentation . . . . .	71
	<b>Python Module Index</b>	<b>73</b>
	<b>Index</b>	<b>75</b>



While this python package is normally used as a command-line program we provide documentation here for developers of the command-line and anyone that wants to integrate nunavut directly into their own Python package.



Code generator built on top of pydsdl.

Nunavut uses pydsdl to generate text files using templates. While these text files are often source code this module could also be used to generate documentation or data interchange formats like JSON or XML.

The input to the nunavut library is a list of templates and a list of `pydsdl.pydsdl.CompositeType` objects. The latter is typically obtained by calling pydsdl:

```
from pydsdl import read_namespace
compound_types = read_namespace(root_namespace, include_paths)
```

Next a `nunavut.LanguageContext` is needed which is used to configure all Nunavut objects for a specific target language

```
from nunavut import LanguageContextBuilder
# Here we are going to generate C headers.
language_context = LanguageContextBuilder().set_target_language("c").create()
```

`nunavut.AbstractGenerator` objects require a `nunavut.Namespace` tree which can be built from the pydsdl type map using `nunavut.build_namespace_tree()`:

```
from nunavut import build_namespace_tree
root_namespace = build_namespace_tree(compound_types,
                                     root_ns_folder,
                                     out_dir,
                                     language_context)
```

Putting this all together, the typical use of this library looks something like this:

```
from pydsdl import read_namespace
from nunavut import build_namespace_tree
```

(continues on next page)

(continued from previous page)

```

from nunavut.lang import LanguageContextBuilder
from nunavut.jinja import DSDLCodeGenerator

# parse the dsdl
compound_types = read_namespace(root_namespace, include_paths)

# select a target language
language_context = LanguageContextBuilder().set_target_language("c").create()

# build the namespace tree
root_namespace = build_namespace_tree(compound_types,
                                     root_ns_folder,
                                     out_dir,
                                     language_context)

# give the root namespace to the generator and...
generator = DSDLCodeGenerator(root_namespace)

# generate all the code!
generator.generate_all()

```

**class** `nunavut.AbstractGenerator` (*namespace: nunavut.\_namespace.Namespace, generate\_namespace\_types: nunavut.\_utilities.YesNoDefault = <YesNoDefault.DEFAULT: 2>*)

Bases: `object`

Abstract base class for classes that generate source file output from a given pydsdl parser result.

**Parameters**

- **namespace** (`nunavut.Namespace`) – The top-level namespace to generates types at and from.
- **generate\_namespace\_types** (`YesNoDefault`) – Set to YES to force generation files for namespaces and NO to suppress. DEFAULT will generate namespace files based on the language preference.

**namespace**

The root `nunavut.Namespace` for this generator.

**generate\_namespace\_types**

If true then the generator is set to emit files for `nunavut.Namespace` in addition to the pydsdl datatypes. If false then only files for pydsdl datatypes will be generated.

**get\_templates** (*omit\_serialization\_support: bool = False*) → `Iterable[pathlib.Path]`

Enumerate all templates found in the templates path. :param bool omit\_serialization\_support: If True then templates needed only for serialization will be omitted. :return: A list of paths to all templates found by this Generator object.

**generate\_all** (*is\_dryrun: bool = False, allow\_overwrite: bool = True, omit\_serialization\_support: bool = False*) → `Iterable[pathlib.Path]`

Generates all output for a given `nunavut.Namespace` and using the templates found by this object.

**Parameters**

- **is\_dryrun** (`bool`) – If True then no output files will actually be written but all other operations will be performed.



- **allow\_overwrite** (*bool*) – If True then the generator will attempt to overwrite any existing files it encounters. If False then the generator will raise an error if the output file exists and the generation is not a dry-run.
- **omit\_serialization\_support** (*bool*) – If True then the generator will emit only types without additional serialization and deserialization support and logic.

**Returns** 0 for success. Non-zero for errors.

**Raises** `PermissionError` if `allow_overwrite` is False and the file exists.

```
nunavut.build_namespace_tree (types:      List[pydsdl._serializable._composite.CompositeType],
                             root_namespace_dir: str, output_dir: str, lan-
                             guage_context: nunavut.lang.LanguageContext) →
                             nunavut._namespace.Namespace
```

Generates a `nunavut.Namespace` tree.

Given a list of pydsdl types, this method returns a root `nunavut.Namespace`. The root `nunavut.Namespace` is the top of a tree where each node contains references to nested `nunavut.Namespace` and to any `pydsdl.CompositeType` instances contained within the namespace.

#### Parameters

- **types** (*list*) – A list of pydsdl types.
- **root\_namespace\_dir** (*str*) – A path to the folder which is the root namespace.
- **output\_dir** (*str*) – The base directory under which all generated files will be created.
- **language\_context** (`nunavut.LanguageContext`) – The language context to use when building `nunavut.Namespace` objects.

**Returns** The root `nunavut.Namespace`.

```
class nunavut.CodeGenerator (namespace:      nunavut._namespace.Namespace, generate_namespace_types:
                             nunavut._utilities.YesNoDefault =
                             <YesNoDefault.DEFAULT: 2>, templates_dir: Union[pathlib.Path,
                             List[pathlib.Path], None] = None, followlinks: bool = False,
                             trim_blocks: bool = False, lstrip_blocks: bool = False, ad-
                             ditional_filters: Optional[Dict[str, Callable]] = None, ad-
                             ditional_tests: Optional[Dict[str, Callable]] = None, addi-
                             tional_globals: Optional[Dict[str, Any]] = None, post_processors:
                             Optional[List[nunavut._postprocessors.PostProcessor]] =
                             None, builtin_template_path: str = 'templates', pack-
                             age_name_for_templates: Optional[str] = None, search_policy:
                             nunavut._utilities.ResourceSearchPolicy = <ResourceSearchPol-
                             icy.FIND_ALL: 0>)
```

Bases: `nunavut._generators.AbstractGenerator`

Abstract base class for all Generators that build source code using Jinja templates.

#### Parameters

- **namespace** (`nunavut.Namespace`) – The top-level namespace to generates code at and from.
- **generate\_namespace\_types** (`YesNoDefault`) – Set to YES to emit files for namespaces. NO will suppress namespace file generation and DEFAULT will use the language's preference.
- **templates\_dir** (`typing.Optional[typing.Union[pathlib.Path, typing.List[pathlib.Path]]]`) – Directories containing jinja templates. These will be available along with any built-in templates provided by the target language. The

templates at these paths will take precedence masking any built-in templates where the names are the same. See `jinja2.ChoiceLoader` for rules on the lookup hierarchy.

- **followlinks** (*bool*) – If True then symbolic links will be followed when searching for templates.
- **trim\_blocks** (*bool*) – If this is set to True the first newline after a block is removed (block, not variable tag!).
- **lstrip\_blocks** (*bool*) – If this is set to True leading spaces and tabs are stripped from the start of a line to a block. Defaults to False.
- **typing.Callable] additional\_filters** (*typing.Dict[str, typing.Optional[jinja.filters.Filter]]*) – typing.Optional jinja filters to add to the global environment using the key as the filter name and the callable as the filter.
- **typing.Callable] additional\_tests** (*typing.Dict[str, typing.Optional[jinja.tests.Test]]*) – typing.Optional jinja tests to add to the global environment using the key as the test name and the callable as the test.
- **typing.Any] additional\_globals** (*typing.Dict[str, typing.Any]*) – typing.Optional objects to add to the template environment globals collection.
- **post\_processors** (*typing.Optional[typing.List[nunavut.postprocessors.PostProcessor]]*) – A list of `nunavut.postprocessors.PostProcessor`
- **builtin\_template\_path** – If provided overrides the folder name under which built-in templates are loaded from within a target language’s package (i.e. ignored if no target language is specified). For example, if the target language is `c` and this parameter was set to `foo` then built-in templates would be loaded from `nunavut.lang.c.foo`.
- **package\_name\_for\_templates** – If provided then the generator will look for templates within the given package name if a template is not found under a given filesystem path first (see `templates_dir` parameter).

**Raises** `RuntimeError` – If any additional filter or test attempts to replace a built-in or otherwise already defined filter or test.

**get\_templates** (*omit\_serialization\_support: bool = False*) → `Iterable[pathlib.Path]`

Enumerate all templates found in the templates path. `TEMPLATE_SUFFIX` as the suffix for the filename.

**Returns** A list of paths to all templates found by this Generator object.

**class** `nunavut.DSDLCodeGenerator` (*namespace: nunavut.\_namespace.Namespace, \*\*kwargs*)

Bases: `nunavut.jinja.CodeGenerator`

`CodeGenerator` implementation that generates code for a given set of DSDL types.

**static filter\_yamlfy** (*value: Any*) → `str`

Filter to, optionally, emit a dump of the `dSDL` input as a `yaml` document. Available as `yamlfy` in all template environments.

Example:

```
/*
{{ T | yamlfy }}
*/
```

Result Example (truncated for brevity):

```

/*
!!python/object:pydsdl.StructureType
_attributes:
- !!python/object:pydsdl.Field
_serializable: !!python/object:pydsdl.UnsignedIntegerType
  _bit_length: 16
  _cast_mode: &id001 !!python/object/apply:pydsdl.CastMode
  - 0
_name: value
*/

```

**Parameters** `value` – The input value to parse as yaml.

**Returns** If a yaml parser is available, a pretty dump of the given value as yaml. If a yaml parser is not available then an empty string is returned.

**filter\_type\_to\_template** (*value: Any*) → str

Template for type resolution as a filter. Available as `type_to_template` in all template environments.

Example:

```

%- for attribute in T.attributes %
  {%* include attribute.data_type | type_to_template %}
  {%- if not loop.last %}|{% endif %}
%- endfor %

```

**Parameters** `value` – The input value to change into a template include path.

**Returns** A path to a template named for the type with `TEMPLATE_SUFFIX`

**filter\_type\_to\_include\_path** (*value: Any, resolve: bool = False*) → str

Emits an include path to the output target for a given type.

Example:

```

# include "{{ T.my_type | type_to_include_path }}"

```

Result Example:

```

# include "foo/bar/my_type.h"

```

**Parameters**

- **value** (*typing.Any*) – The type to emit an include for.
- **resolve** (*bool*) – If True the path returned will be absolute else the path will be relative to the folder of the root namespace.

**Returns** A string path to output file for the type.

**static filter\_tynename** (*value: Any*) → str

Filters a given token as its type name. Available as `tynename` in all template environments.

This example supposes that `T.some_value == "some string"`

Example:

```

{{ T.some_value | tynename }}

```

Result Example:

```
str
```

**Parameters** `value` – The input value to filter into a type name.

**Returns** The `__name__` of the python type.

**static filter\_alignment\_prefix** (*offset: pydsdl.\_bit\_length\_set.\_bit\_length\_set.BitLengthSet*) → str

Provides a string prefix based on a given `pydsdl.BitLengthSet`.

```
# Given
B = pydsdl.BitLengthSet(32)

# and
template = '{{ B | alignment_prefix }}'

# then ('str' is stropped to 'str_' before the version is suffixed)
rendered = 'aligned'
```

```
# Given
B = pydsdl.BitLengthSet(32)
B += 1

# and
template = '{{ B | alignment_prefix }}'

# then ('str' is stropped to 'str_' before the version is suffixed)
rendered = 'unaligned'
```

**Parameters** `offset` (*pydsdl.BitLengthSet*) – A bit length set to test for alignment.

**Returns** ‘aligned’ or ‘unaligned’ based on the state of the `offset` argument.

**static filter\_bit\_length\_set** (*values: Union[Iterable[int], int, None]*) → *pydsdl.\_bit\_length\_set.\_bit\_length\_set.BitLengthSet*

Convert an integer or a list of integers into a `pydsdl.BitLengthSet`.

**static filter\_remove\_blank\_lines** (*text: str*) → str

Remove blank lines from the supplied string. Lines that contain only whitespace characters are also considered blank.

456

789’) == ‘123 456 789’

**static filter\_bits2bytes\_ceil** (*n\_bits: int*) → int

Implements `int(ceil(x/8)) | x >= 0`.

**static is\_None** (*value: Any*) → bool

Tests if a value is `None`

**static is\_saturated** (*t: pydsdl.\_serializable.\_primitive.PrimitiveType*) → bool

Tests if a type is a saturated type or not.

**static is\_service\_request** (*instance: pydsdl.\_expression.\_any.Any*) → bool

Tests if a type is request type of a service type.

**static is\_service\_response** (*instance: pydSDL.\_expression.\_any.Any*) → bool  
 Tests if a type is response type of a service type.

**static is\_deprecated** (*instance: pydSDL.\_expression.\_any.Any*) → bool  
 Tests if a type is marked as deprecated

**generate\_all** (*is\_dryrun: bool = False, allow\_overwrite: bool = True, omit\_serialization\_support: bool = False*) → Iterable[pathlib.Path]  
 Generates all output for a given `nunavut.Namespace` and using the templates found by this object.

**Parameters**

- **is\_dryrun** (*bool*) – If True then no output files will actually be written but all other operations will be performed.
- **allow\_overwrite** (*bool*) – If True then the generator will attempt to overwrite any existing files it encounters. If False then the generator will raise an error if the output file exists and the generation is not a dry-run.
- **omit\_serialization\_support** (*bool*) – If True then the generator will emit only types without additional serialization and deserialization support and logic.

**Returns** 0 for success. Non-zero for errors.

**Raises** PermissionError if `allow_overwrite` is False and the file exists.

`nunavut.generate_types` (*language\_key: str, root\_namespace\_dir: pathlib.Path, out\_dir: pathlib.Path, omit\_serialization\_support: bool = True, is\_dryrun: bool = False, allow\_overwrite: bool = True, lookup\_directories: Optional[Iterable[str]] = None, allow\_unregulated\_fixed\_port\_id: bool = False, language\_options: Mapping[str, Any] = {}, include\_experimental\_languages: bool = False*) → None

Helper method that uses default settings and built-in templates to generate types for a given language. This method is the most direct way to generate code using Nunavut.

**Parameters**

- **language\_key** (*str*) – The name of the language to generate source for. See the *Template Language Guide* for details on available language support.
- **root\_namespace\_dir** (*pathlib.Path*) – The path to the root of the DSDL types to generate code for.
- **out\_dir** (*pathlib.Path*) – The path to generate code at and under.
- **omit\_serialization\_support** (*bool*) – If True then logic used to serialize and deserialize data is omitted.
- **is\_dryrun** (*bool*) – If True then nothing is generated but all other activity is performed and any errors that would have occurred are reported.
- **allow\_overwrite** (*bool*) – If True then generated files are allowed to overwrite existing files under the `out_dir` path.
- **lookup\_directories** (*typing.Optional[typing.Iterable[str]]*) – Additional directories to search for dependent types referenced by the types provided under the `root_namespace_dir`. Types will not be generated for these unless they are used by a type in the root namespace.
- **allow\_unregulated\_fixed\_port\_id** (*bool*) – If True then errors will become warning when using fixed port identifiers for unregulated datatypes.
- **typing.Any]] language\_options** (*typing.Optional[typing.Mapping[str, Any]]*) – Opaque arguments passed through to the language objects. The

supported arguments and valid values are different depending on the language specified by the *language\_key* parameter.

- **include\_experimental\_languages** (*bool*) – If true then experimental languages will also be available.

**class** `nunavut.LanguageConfig`

Bases: `object`

Configuration storage encapsulating parsers and other configuration format details. For any configuration type used the concept of “sections” must be maintained which requires that the top-level configuration be structured as key/value pairs with the keys using the form “`nunavut.lang.[language name]`”. For example, yaml configuration must have a top-level structure like this:

```
example_yaml = '''
  nunavut.lang.a:
    key_one: value_one
    key_two: value_two
  nunavut.lang.b:
    key_one: value_one
    key_two: value_two
  nunavut.lang.c:
    key_one: value_one
    key_two: value_two
'''
```

---

**Note:** The “language name” part of the section identifier must not start with a number and can contain only alphanumeric characters. That is, the section identifier must match this pattern:

```
nunavut.lang.[a-zA-Z]{1}w*
```

---

The values of the section data can be anything:

```
example_yaml = '''
  nunavut.lang.d:
    key_one:
      - is
      - a
      - list:
          where: index2
          is: a_dictionary
'''
```

```
SECTION_NAME_PATTERN = re.compile('^nunavut\\.lang\\. ([a-zA-Z]{1}\\w*)$')
```

Required pattern for section name identifiers.

**update** (*configuration: Any*) → None

Add configuration data to this configuration from a string.

Unlike `add_section`, this method will update section data with existing keys.

For example, the first update is to an empty configuration so it will act as a simple insert operation:

...but updating this data is now possible where sections can be added and updated:

**add\_section** (*section\_name: str*) → None

Add a section to the configuration.

Sections are top-level containers that contain key/value pairs of configuration of a single language type.

**Parameters `section_name`** – The name of the language section. This must adhere to the `SECTION_NAME_PATTERN` pattern.

**`get_config_value`** (*section\_name: str, key: str, default\_value: Optional[str] = None*) → str  
Get an optional language property from the language configuration.

:param section\_name : The name of the section to get the value from. :param str key : The config value to retrieve. :param default\_value: The value to return if the key was not in the configuration. If provided this method will not raise. :type default\_value : typing.Optional[str] :return: Either the value from the config or the default\_value if provided. :rtype: str :raises: KeyError if the section or the key in the section does not exist and a default\_value was not provided.

**`get_config_value_as_bool`** (*section\_name: str, key: str, default\_value: bool = False*) → bool  
Get an optional language property from the language configuration returning a boolean. The rules for boolean conversion are as follows:

```
# "Any string" = True
config.set('nunavut.lang.cpp', 'v', 'Any string')
assert config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "true" = True
config.set('nunavut.lang.cpp', 'v', 'true')
assert config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "TrUe" = True
config.set('nunavut.lang.cpp', 'v', 'TrUe')
assert config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "1" = True
config.set('nunavut.lang.cpp', 'v', '1')
assert config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "false" = False
config.set('nunavut.lang.cpp', 'v', 'false')
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "FaLse" = False
config.set('nunavut.lang.cpp', 'v', 'FaLse')
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "0" = False
config.set('nunavut.lang.cpp', 'v', '0')
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# "" = False
config.set('nunavut.lang.cpp', 'v', '')
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'v')

# False if not defined
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'not_a_key')

# True if not defined but default_value is True
assert not config.get_config_value_as_bool('nunavut.lang.cpp', 'not_a_key')
```

:param section\_name : The name of the section to get the value from. :param str key : The config value to retrieve. :param bool default\_value : The value to use if no value existed. :return : The config value as either True or False. :rtype : bool

**get\_config\_value\_as\_dict** (*section\_name: str, key: str, default\_value: Optional[Dict[KT, VT]] = None*) → Dict[str, Any]

Get a language property parsing it as a map with string keys.

Example:

:param str section\_name : The name of the section to get the key from. :param str key : The config value to retrieve. :param default\_value : The value to return if the key was not in the configuration. If provided this method will not raise a KeyError nor a TypeError. :type default\_value : typing.Optional[typing.Mapping[str, typing.Any]] :return : Either the value from the config or the default\_value if provided. :rtype : typing.Mapping[str, typing.Any] :raises : KeyError if the key does not exist and a default\_value was not provided. :raises : TypeError if the value exists but is not a dict and a default\_value was not provided.

**get\_config\_value\_as\_list** (*section\_name: str, key: str, default\_value: Optional[List[T]] = None*) → List[Any]

Get a language property parsing it as a map with string keys.

Example:

:param str section\_name : The name of the section to get the key from. :param str key : The config value to retrieve. :param default\_value : The value to return if the key was not in the configuration. If provided this method will not raise a KeyError nor a TypeError. :type default\_value : typing.Optional[typing.List[typing.Any]] :return : Either the value from the config or the default\_value if provided. :rtype : typing.List[typing.Any] :raises : KeyError if the key does not exist and a default\_value was not provided. :raises : TypeError if the value exists but is not a dict and a default\_value was not provided.

**class** nunavut.**Language** (*language\_module\_name: str, config: nunavut.lang.\_config.LanguageConfig, \*\*kwargs*)

Bases: `object`

Facilities for generating source code for a specific language. Concrete Language classes must be implemented by the language support package below lang and should be instantiated using `nunavut.lang.LanguageClassLoader`.

### Parameters

- **module\_name** (*str*) – The name of the `nunavut.lang` module that contains the concrete language type and its resources.
- **config** (`LanguageConfig`) – All configuration as defined by the `properties.yaml` schema.
- **kwargs** –

**Opaque arguments passed through to the target** `nunavut.lang.Language` object.  
See all “WKLA” constants on this class for well-known keyword arguments.

**classmethod** **default\_filter\_id\_for\_target** (*instance: Any*) → str

The default transformation of any object into a string.

**Parameters** **instance** (*any*) – Any object or data that either has a name property or can be converted to a string.

**Returns** Either `str(instance.name)` if the instance has a name property or just `str(instance)`

### extension

The extension to use for files generated in this language.

### namespace\_output\_stem

The name of a namespace file for this language.



**name**

The name of the language used by the `nunavut.lang` module.

**support\_namespace**

The hierarchical namespace used by the support software. The property is a dot separated string when specified in configuration. This property returns that value split into namespace components with the first identifier being the first index in the array, etc.

**enable\_stropping**

Whether or not to strop identifiers for this language.

**has\_standard\_namespace\_files**

Whether or not the language defines special namespace files as part of its core standard (e.g. python's `__init__`).

**stable\_support**

Whether support for this language is designated 'stable', and not experimental.

**named\_types**

Get a map of named types to the type name to emit for this language.

**named\_values**

Get a map of named values to the token to emit for this language.

**get\_support\_module** () → Tuple[str, Tuple[int, int, int], Optional[module]]

Returns the module object for the language support files. :return: A tuple of module name, x.y.z module version, and the module object itself.

**get\_includes** (*dep\_types: nunavut.\_dependencies.Dependencies*) → List[str]

Get a list of include paths that are specific to this language and the options set for it. :param Dependencies dep\_types: A description of the dependencies includes are needed for. :return: A list of include file paths. The list may be empty if no includes were needed.

**filter\_id** (*instance: Any, id\_type: str = 'any'*) → str

Produces a valid identifier in the language for a given object. The encoding may not be reversible.

**Parameters**

- **instance** (*any*) – Any object or data that either has a name property or can be converted to a string.
- **id\_type** (*str*) – A type of identifier. This is different for each language. For example, for C this value can be 'typedef', 'macro', 'function', or 'enum'. Use 'any' to apply stropping rules for all identifier types to the instance.

**Returns** A token that is a valid identifier in the language, is not a reserved keyword, and is transformed in a deterministic manner based on the provided instance.

**filter\_short\_reference\_name** (*t: pydsdl.\_serializable.\_composite.CompositeType, stropping: nunavut.\_utilities.YesNoDefault = <YesNoDefault.DEFAULT: 2>, id\_type: str = 'any'*) → str

Provides a string that is a shorted version of the full reference name omitting any namespace parts of the type.

**Parameters**

- **t** (*pydsdl.CompositeType*) – The DSDL type to get the reference name for.
- **stropping** (*YesNoDefault*) – If DEFAULT then the stropping value configured for the target language is used else this overrides that value.

- **id\_type** (*str*) – A type of identifier. This is different for each language. For example, for C this value can be ‘typedef’, ‘macro’, ‘function’, or ‘enum’. Use ‘any’ to apply stripping rules for all identifier types to the instance.

**get\_config\_value** (*key: str, default\_value: Optional[str] = None*) → *str*  
 Get an optional language property from the language configuration.

**Parameters**

- **key** (*str*) – The config value to retrieve.
- **default\_value** (*typing.Optional[str]*) – The value to return if the key was not in the configuration. If provided this method will not raise.

**Returns** Either the value from the config or the default\_value if provided.

**Return type** *str*

**Raises** *KeyError* if the section or the key in the section does not exist and a default\_value was not provided.

**get\_config\_value\_as\_bool** (*key: str, default\_value: bool = False*) → *bool*  
 Get an optional language property from the language configuration returning a boolean.

**Parameters**

- **key** (*str*) – The config value to retrieve.
- **default\_value** (*bool*) – The value to use if no value existed.

**Returns** The config value as either True or False.

**Return type** *bool*

**get\_config\_value\_as\_dict** (*key: str, default\_value: Optional[Dict[KT, VT]] = None*) → *Dict[str, Any]*  
 Get a language property parsing it as a map with string keys.

**Parameters**

- **key** (*str*) – The config value to retrieve.
- **default\_value** (*typing.Optional[typing.Mapping[str, typing.Any]]*) – The value to return if the key was not in the configuration. If provided this method will not raise a *KeyError* nor a *TypeError*.

**Returns** Either the value from the config or the default\_value if provided.

**Return type** *typing.Mapping[str, typing.Any]*

**Raises** *KeyError* if the key does not exist and a default\_value was not provided.

**Raises** *TypeError* if the value exists but is not a dict and a default\_value was not provided.

**get\_config\_value\_as\_list** (*key: str, default\_value: Optional[List[T]] = None*) → *List[Any]*  
 Get a language property parsing it as a map with string keys.

**Parameters**

- **key** (*str*) – The config value to retrieve.
- **default\_value** (*typing.Optional[typing.List[typing.Any]]*) – The value to return if the key was not in the configuration. If provided this method will not raise a *KeyError* nor a *TypeError*.

**Returns** Either the value from the config or the default\_value if provided.

**Return type** *typing.List[typing.Any]*

**Raises** `KeyError` if the key does not exist and a `default_value` was not provided.

**Raises** `TypeError` if the value exists but is not a dict and a `default_value` was not provided.

**get\_support\_files** (*resource\_type: nunavut.\_utilities.ResourceType = <ResourceType.ANY: 0>*)  
 → Generator[pathlib.Path, None, None]

Iterates over supporting files embedded within the Nunavut distribution.

**Parameters** `resource_type` – The type of support resources to enumerate.

**get\_option** (*option\_key: str, default\_value: Union[Mapping[str, Any], str, None] = None*) →  
 Union[Mapping[str, Any], str, None]

Get a language option for this language.

```
# Values can come from defaults...
assert lang_cpp.get_option('target_endianness') == 'little'

# ... or can come from a sane default.
assert lang_cpp.get_option('foobar', 'sane_default') == 'sane_default'
```

**Returns** Either the value provided to the `nunavut.lang.Language` instance, the value from `properties.yaml` or the `default_value`.

**get\_templates\_package\_name** () → str

The name of the nunavut python package containing filters, types, and configuration for this language.

**get\_globals** () → Mapping[str, Any]

Get all values for this language that should be available in a global context.

**Returns** A mapping of global names to global values.

**get\_options** () → Mapping[str, Any]

Get all language options for this Language.

**Returns** A mapping of option names to option values.

**class** `nunavut.LanguageContext` (*language\_configuration: nunavut.lang.\_config.LanguageConfig,*  
*target\_language: nunavut.lang.\_language.Language,*  
*supported\_language\_builder: Callable[[], Dict[str,*  
*nunavut.lang.\_language.Language]]*)

Bases: `object`

Context object containing the current target language and all supported `nunavut.lang.Language` objects.

**Parameters**

- **language\_configuration** – The configuration for all languages as defined by the `properties.yaml` schema.
- **target\_language** – The target language.
- **supported\_language\_builder** – factory closure that will create `nunavut.lang.Language` objects for all supported languages when `LanguageContext.get_target_languages()` is first called.

**get\_language** (*key\_or\_module\_name: str*) → `nunavut.lang._language.Language`

Get a `nunavut.lang.Language` object for a given language identifier.

**Parameters** `key_or_module_name` (*str*) – Either one of the Nunavut mnemonics for a supported language or the `__name__` of one of the `nunavut.lang.[language]` python modules.

**Returns** A `nunavut.lang.Language` object cached by this context.

**Return type** nunavut.lang.Language

**get\_target\_language** () → nunavut.lang.\_language.Language

Returns the target language for code generation.

**filter\_id\_for\_target** (*instance: Any, id\_type: str = 'any'*) → str

A filter that will transform a given string or pydsdl identifier into a valid identifier in the target language.

**Parameters**

- **instance** (*any*) – Any object or data that either has a name property or can be converted to a string.
- **id\_type** (*str*) – A type of identifier. This is different for each language. Use ‘any’ to apply stropping rules for all identifier types to the instance.

**Returns** A token that is a valid identifier in the target language, is not a reserved keyword, and is transformed in a deterministic manner based on the provided instance.

**get\_supported\_languages** () → Dict[str, nunavut.lang.\_language.Language]

Returns a collection of available language support objects.

**class** nunavut.LanguageContextBuilder (*include\_experimental\_languages: bool = False*)

Bases: object

Used to instantiate new *LanguageContext* objects.

The simplest invocation will always work by using the *LanguageContextBuilder.DEFAULT\_TARGET\_LANGUAGE* constant:

```
from nunavut.lang import LanguageContextBuilder

default_language_context = LanguageContextBuilder().create()

assert LanguageContextBuilder.DEFAULT_TARGET_LANGUAGE == default_language_context.
↳get_target_language().name
```

Typically a target language is specified at minimum. Also see constants on *nunavut.lang.Language* for well-known options that the builder can override:

```
from nunavut.lang import LanguageContextBuilder

customized_language_context = (
    LanguageContextBuilder()
    .set_target_language("c")
    .set_target_language_configuration_override(Language.WKCV_DEFINITION_FILE_
↳EXTENSION, ".h")
    .create()
)

assert customized_language_context.get_target_language().extension == ".h"
```

**Parameters include\_experimental\_languages** – If set then languages that are not fully supported will be allowed otherwise any experimental languages will be missing and errors will be raised as if the language specified was unknown.

**DEFAULT\_TARGET\_LANGUAGE = 'c'**

The target language used for new contexts if none is specified.

**get\_supported\_language\_names** () → Iterable[str]

Get a list of target languages supported by Nunavut.

**Returns** An iterable of strings which are languages with special support within Nunavut templates.

**set\_target\_language\_configuration\_override** (key: str, value: Any) → nunavut.lang.LanguageContextBuilder

Stores a key and value to override in the configuration for a language target when a LanguageContext is crated. These overrides are always set under the language section of the target language.

```
builder = LanguageContextBuilder().set_target_language("c")

default_c_file_extension = builder.config.get_config_value(
    LanguageClassLoader.to_language_module_name("c"),
    Language.WKCV_DEFINITION_FILE_EXTENSION)

assert default_c_file_extension == ".h"
```

We can now try to override the file extension for a future “C” target language object:

```
builder.set_target_language_configuration_override(Language.WKCV_DEFINITION_
FILE_EXTENSION, ".foo")
```

... but that value will not be overridden until you create the target language:

```
default_c_file_extension = builder.config.get_config_value(
    LanguageClassLoader.to_language_module_name("c"),
    Language.WKCV_DEFINITION_FILE_EXTENSION)

assert default_c_file_extension == ".h"

_ = builder.create()

overridden_c_file_extension = builder.config.get_config_value(
    LanguageClassLoader.to_language_module_name("c"),
    Language.WKCV_DEFINITION_FILE_EXTENSION)

assert overridden_c_file_extension == ".foo"
```

Note that the config is scoped by the builder but is then inherited by the language objects created by the builder:

```
one = (
    LanguageContextBuilder()
        .set_target_language("c")
        .set_target_language_configuration_override("foo", 1)
)

two = (
    LanguageContextBuilder()
        .set_target_language("c")
        .set_target_language_configuration_override("foo", 2)
)

# Here we see that the second override of "foo" does not affect the first_
because they
```

(continues on next page)

(continued from previous page)

```
# are in different builders.

assert (
    one.create().get_target_language().get_config_value("foo")
    !=
    two.create().get_target_language().get_config_value("foo")
)
```

**set\_target\_language\_extension** (*target\_language\_extension: Optional[str]*) → `nunavut.lang.LanguageContextBuilder`

Helper method for setting the target language file extension (since this is a common override).

Calling this method is the same as doing:

```
LanguageContextBuilder().set_target_language_configuration_override(
    Language.NKCV_DEFINITION_FILE_EXTENSION,
    ".h")
```

**set\_target\_language** (*target\_language: Optional[str]*) → `nunavut.lang.LanguageContextBuilder`

Set the language name to target. This can be either the name of the language, as defined by Nunavut, or it can be the language package name.

```
assert LanguageContextBuilder().set_target_language("c").create().get_target_
↳ language().name == "c"

assert (
    LanguageContextBuilder()
        .set_target_language(LanguageClassLoader.co_language_module_name("c"))
        .create()
        .get_target_language().name == "c"
)
```

Also note that, if the language name is None, the default name will be assigned internally:

```
target_language = LanguageContextBuilder().set_target_language(None).create().
↳ get_target_language()

assert target_language.name == LanguageContextBuilder.DEFAULT_TARGET_LANGUAGE
```

**set\_additional\_config\_files** (*additional\_config\_files: List[pathlib.Path]*) → `nunavut.lang.LanguageContextBuilder`

A list of paths to additional yaml files to load as configuration. These will override any values found in the `nunavut.lang.properties.yaml` file and files appearing later in this list will override value found in earlier entries.

```
target_language_w_overrides = (
    LanguageContextBuilder()
        .set_target_language("c")
        .set_additional_config_files([overrides_file])
        .create()
        .get_target_language()
)

target_language_no_overrides = (
    LanguageContextBuilder()
        .set_target_language("c")
)
```

(continues on next page)

(continued from previous page)

```

        .create()
        .get_target_language()
    )

assert target_language_w_overrides.extension == ".foo"
assert target_language_no_overrides.extension == ".h"

```

Overrides are applied as unions. For example, given this override data:

```

overrides_data = '''
    nunavut.lang.c:
        extension: .foo
        non-standard: bar
'''

```

... the standard “extension” property will be overridden and the “non-standard” property will be added.

```

target_language_w_overrides = (
    LanguageContextBuilder()
        .set_target_language("c")
        .set_additional_config_files([second_overrides_file])
        .create()
        .get_target_language()
)

assert ".foo" == target_language_w_overrides.extension
assert "bar" == target_language_w_overrides.get_config_value("non-standard")

```

**create** () → `nunavut.lang.LanguageContext`

Applies all pending configuration overrides to the internal `LanguageConfig` object and instantiates a `LanguageContext` object.

**class** `nunavut.Namespace` (`full_namespace: str`, `root_namespace_dir: pathlib.Path`, `base_output_path: pathlib.PurePath`, `language_context: nunavut.lang.LanguageContext`)

Bases: `pydsdl._expression._any.Any`

K-ary tree (where K is the largest set of data types in a single dsdl namespace) where the nodes represent dsdl namespaces and the children are the datatypes and other nested namespaces (with datatypes always being leaf nodes). This structure extends `pydsdl.Any` and is a `pydsdl.pydsdl.CompositeType` via duck typing.

### Parameters

- **full\_namespace** (`str`) – The full, dot-separated name of the namespace. This is expected to be a unique identifier.
- **root\_namespace\_dir** (`pathlib.Path`) – The directory representing the dsdl namespace and containing the namespaces’s datatypes and nested namespaces.
- **base\_output\_path** (`pathlib.PurePath`) – The base path under which all namespaces and datatypes should be generated.
- **language\_context** (`LanguageContext`) – The generated software language context the namespace is within.

### output\_folder

The folder where this namespace’s output file and datatypes are generated.

**get\_support\_output\_folder** () → `pathlib.PurePath`

The folder under which support artifacts are generated.

**get\_language\_context** () → nunavut.lang.LanguageContext  
 The generated software language context the namespace is within.

**get\_root\_namespace** () → nunavut.\_namespace.Namespace  
 Traverses the namespace tree up to the root and returns the root node.

**Returns** The root namespace object.

**get\_nested\_namespaces** () → Iterator[nunavut.\_namespace.Namespace]  
 Get an iterator over all the nested namespaces within this namespace. This is a shallow iterator that only provides directly nested namespaces.

**get\_nested\_types** () → ItemsView[pydsdl.\_serializable.\_composite.CompositeType, pathlib.Path]  
 Get a view of a tuple relating datatypes in this namespace to the path for the type’s generated output. This is a shallow view including only the types directly within this namespace.

**get\_all\_datatypes** () → Generator[Tuple[pydsdl.\_serializable.\_composite.CompositeType, pathlib.Path], None, None]  
 Generates tuples relating datatypes at and below this namespace to the path for each type’s generated output.

**get\_all\_namespaces** () → Generator[Tuple[nunavut.\_namespace.Namespace, pathlib.Path], None, None]  
 Generates tuples relating nested namespaces at and below this namespace to the path for each namespace’s generated output.

**get\_all\_types** () → Generator[Tuple[pydsdl.\_expression.\_any.Any, pathlib.Path], None, None]  
 Generates tuples relating datatypes and nested namespaces at and below this namespace to the path for each type’s generated output.

**find\_output\_path\_for\_type** (*any\_type: pydsdl.\_expression.\_any.Any*) → pathlib.Path  
 Searches the entire namespace tree to find a mapping of the type to an output file path.

**Parameters** *any\_type* (*Any*) – Either a Namespace or pydsdl.CompositeType to find the output path for.

**Returns** The path where a file will be generated for a given type.

**Raises** **KeyError** – If the type was not found in this namespace tree.

**class** nunavut.SupportGenerator (*namespace: nunavut.\_namespace.Namespace, \*\*kwargs*)  
 Bases: nunavut.jinja.CodeGenerator

Generates output files by copying them from within the Nunavut package itself for non templates but uses jinja to generate headers from templates with the language environment provided but no T (DSDL type) global set. This generator always copies files from those returned by the `file_iterator` to locations under `nunavut.Namespace.get_support_output_folder()`

**get\_templates** (*omit\_serialization\_support: bool = False*) → Iterable[pathlib.Path]  
 Enumerate all templates found in the templates path. `TEMPLATE_SUFFIX` as the suffix for the filename.

**Returns** A list of paths to all templates found by this Generator object.

**generate\_all** (*is\_dryrun: bool = False, allow\_overwrite: bool = True, omit\_serialization\_support: bool = False*) → Iterable[pathlib.Path]  
 Generates all output for a given `nunavut.Namespace` and using the templates found by this object.

**Parameters**

- **is\_dryrun** (*bool*) – If True then no output files will actually be written but all other operations will be performed.



- **allow\_overwrite** (*bool*) – If True then the generator will attempt to overwrite any existing files it encounters. If False then the generator will raise an error if the output file exists and the generation is not a dry-run.
- **omit\_serialization\_support** (*bool*) – If True then the generator will emit only types without additional serialization and deserialization support and logic.

**Returns** 0 for success. Non-zero for errors.

**Raises** `PermissionError` if `allow_overwrite` is False and the file exists.

**exception** `nunavut.UnsupportedLanguageError`

Bases: `ValueError`

Error type raised if an unsupported language type is used.

**class** `nunavut.YesNoDefault`

Bases: `enum.Enum`

Ternary type for decisions that allow a default behavior to be requested that can be different based on other contexts. For example:

```
def should_we_order_pizza(answer: YesNoDefault) -> bool:
    if answer == YesNoDefault.YES or (
        answer == YesNoDefault.DEFAULT and
        datetime.today().isoweekday() == 5):
        # if yes or if we are taking the default action which is to
        # order pizza on Friday, and today is Friday, then we order pizza
        return True
    else:
        return False
```

**exception** `nunavut.InternalError`

Bases: `RuntimeError`

Internal, opaque error within Nunavut.

This exception is a “should never happen” exception. If caught you’ve probably hit a bug. This is the only exception type within the library that can be use where no unit tests are covering the error (i.e. pragma: no cover branches).



---

## Software Language Generation Guide

---

---

**Note:** This is a placeholder for documentation this project owes you, the user, for how to integrate nnvg with build systems and how to tune and optimize source code generation for each supported language.

---

### 2.1 C++ (experimental)

See *Template Language Guide* until this section is more complete.

#### 2.1.1 Using a Different Variable-Length Array Type

For now this tip is important for people using the experimental C++ support. To use `std::vector` instead of the minimal build-in `variable_length_array` type create a properties override yaml file and pass it to nnvg.

##### vector.yaml

```
nunavut.lang.cpp:
  options:
    variable_array_type_include: <vector>
    variable_array_type_template: std::vector<{TYPE}>
```

##### nnvg command

```
nnvg --configuration=vector.yaml \
  -l cpp \
  --experimental-languages \
  -I path/to/public_regulated_data_types/uavcan \
  /path/to/my_types
```



For now we have only a jinja generator for code generation. As such this guide will only discuss using nunavut with jinja templates. There are no immediate plans to support any other template syntax.

### 3.1 Environment

---

**Note:** See `nunavut.jinja` and the language support modules within this one for detailed documentation on available filters and tests provided by nunavut.

The  [Jinja templates documentation](#) is indispensable as nunavut embeds a full-featured version of jinja 2.

---

Each template has in its global environment the following:

#### 3.1.1 nunavut

A global `nunavut` is available in the global namespace with the following properties:

##### **version**

A `pep 440` compatible version number for the version of Nunavut that the template is running within.

##### **support**

Meta-data about built-in support for serialization.

##### **omit**

`bool` that is `True` if serialization support was switched off for this template.

## namespace

An array of identifiers under which Nunavut support files and types are namespaced. The use of this value by built-in templates and generators is language dependant.

### 3.1.2 T

The T global contains the type for the given template. For example:

```
{{T.full_name}}
{%- for attr in T.attributes %}
    {{ attr.data_type }}
{%- endfor %}
```

### 3.1.3 now\_utc

The time UTC as a Python `datetime.datetime` object. This is the system time right before the file generation step for the current template began. This will be the same time for included templates and parent templates.

### 3.1.4 In

Options and other language-specific facilities for all supported languages. This namespace provides access to other languages that are *not* the target language for the current template. This allows the development of mixed language templates.

### 3.1.5 Filters and Tests

In addition to the built-in Jinja filters and tests (again, see the [Jinja templates documentation](#) for details) pydsdl adds several global tests and filters to the template environment. See `nunavut.jinja` for full documentation on these. For example:

```
# typename filter returns the name of the value's type.
{{ field | typename }}
```

Also, for every pydsdl type there is a test automatically appended to the global environment. This means you can do:

```
{% if field is IntegerType %}
    // stuff for integer fields
{% endif %}
```

Finally, for every pydsdl object that ends in “Type” or “Field” a lower-case name omitting these suffixes is made available. For example:

```
template = '{% if some_type is integer %}This is a pydsdl.IntegerType{% endif %}'
```

### 3.1.6 Named Types

Some language provide named types to allow templates to use a type without making concrete decisions about the headers and conventions in use. For example, when using C it is common to use `size_t` as an unsigned, integer length type. To avoid hard-coding this type a C template can use the named type:

```
{| typename_unsigned_length |} array_len;
```

## Named Types by Language

Type name	Language(s)	Use
type-name_unsigned_length	C, C++	An unsigned integer type suitable for expressing the length of any valid type on the local system.
typename_byte	C	An unsigned integer type used to represent a single byte (8-bits).

### 3.1.7 Named Values

Some languages can use different values to represent certain data like null references or boolean values. Named values allow templates to insert a token appropriate for the language and configurable by the generator in use. For example:

```
MyType* p = {| valuetoken_null |};
```

## Named Values by Language

Value name	Language(s)
valuetoken_true	C
valuetoken_false	C
valuetoken_null	C

### 3.1.8 Language Options

The target language for a template contributes options to the template globals. These options can be invented by users of the Nunavut library but a built-in set of defaults exists.

All language options are made available as globals within the *options* namespace. For example, a language option “target\_arch” would be available as the “options.target\_arch” global in templates.

For options that do not come with built-in defaults you’ll need to test if the option is available before you use it. For example:

```
# This will throw an exception
template = '{% if options.foo %}bar{% endif %}'
```

Use the built-in test `defined` to avoid these exceptions:

```
# Avoid the exception
template = '{% if options.foo is defined and options.foo %}bar{% endif %}'
```

## Language Options with Built-in Defaults

The following options have built-in defaults for certain languages. These options will always be defined in templates targeting their languages.

## options.target\_endianness

This option is currently defined for C and C++; the possible values are as follows:

- `any` — generate endianness-agnostic code that is compatible with big-endian and little-endian machines alike.
- `big` — generate code optimized for big-endian platforms only. Implementations may treat this option like `any` when no such optimizations are possible.
- `little` — generate code optimized for little-endian platforms only. Little-endian optimizations are made possible by the fact that DSDL is a little-endian format.

```
template = '{{ options.target_endianness }}'

# then
rendered = 'any'
```

## 3.1.9 Filters

### Common Filters

`nunavut.jinja.DSDLCodeGenerator.filter_yamlfy` (*value: Any*) → str

Filter to, optionally, emit a dump of the dsdl input as a yaml document. Available as `yamlfy` in all template environments.

Example:

```
/*
{{ T | yamlfy }}
*/
```

Result Example (truncated for brevity):

```
/*
!!python/object:pydsdl.StructureType
_attributes:
- !!python/object:pydsdl.Field
_serializable: !!python/object:pydsdl.UnsignedIntegerType
  _bit_length: 16
  _cast_mode: &id001 !!python/object/apply:pydsdl.CastMode
  - 0
_name: value
*/
```

**Parameters** `value` – The input value to parse as yaml.

**Returns** If a yaml parser is available, a pretty dump of the given value as yaml. If a yaml parser is not available then an empty string is returned.

`nunavut.jinja.DSDLCodeGenerator.filter_type_to_template` (*self, value: Any*) → str

Template for type resolution as a filter. Available as `type_to_template` in all template environments.

Example:

```
{%- for attribute in T.attributes %}
  {% include attribute.data_type | type_to_template %}
```

(continues on next page)



(continued from previous page)

```

    %- if not loop.last %},{% endif %
{%- endfor %}

```

**Parameters** **value** – The input value to change into a template include path.

**Returns** A path to a template named for the type with `TEMPLATE_SUFFIX`

`nunavut.jinja.DSDLCodeGenerator.filter_type_to_include_path` (*self*, *value*: Any, *resolve*: bool = False) → str

Emits an include path to the output target for a given type.

Example:

```
# include "{{ T.my_type | type_to_include_path }}"
```

Result Example:

```
# include "foo/bar/my_type.h"
```

**Parameters**

- **value** (*typing.Any*) – The type to emit an include for.
- **resolve** (*bool*) – If True the path returned will be absolute else the path will be relative to the folder of the root namespace.

**Returns** A string path to output file for the type.

`nunavut.jinja.DSDLCodeGenerator.filter_typename` (*value*: Any) → str  
Filters a given token as its type name. Available as `typename` in all template environments.

This example supposes that `T.some_value == "some string"`

Example:

```
{{ T.some_value | typename }}
```

Result Example:

```
str
```

**Parameters** **value** – The input value to filter into a type name.

**Returns** The `__name__` of the python type.

`nunavut.jinja.DSDLCodeGenerator.filter_alignment_prefix` (*offset*: `pydsdl._bit_length_set._bit_length_set.BitLengthSet`) → str

Provides a string prefix based on a given `pydsdl.BitLengthSet`.

```

# Given
B = pydsdl.BitLengthSet(32)

# and
template = '{{ B | alignment_prefix }}'

# then ('str' is stripped to 'str_' before the version is suffixed)
rendered = 'aligned'

```

```
# Given
B = pydsdl.BitLengthSet(32)
B += 1

# and
template = '{{ B | alignment_prefix }}'

# then ('str' is stripped to 'str_' before the version is suffixed)
rendered = 'unaligned'
```

**Parameters** `offset` (`pydsdl.BitLengthSet`) – A bit length set to test for alignment.

**Returns** ‘aligned’ or ‘unaligned’ based on the state of the `offset` argument.

`nunavut.jinja.DSDLCodeGenerator.filter_bit_length_set` (*values: Union[Iterable[int], int, None]*) → `pydsdl._bit_length_set._bit_length_set.BitLengthSet`

Convert an integer or a list of integers into a `pydsdl.BitLengthSet`.

`nunavut.jinja.DSDLCodeGenerator.filter_remove_blank_lines` (*text: str*) → `str`

Remove blank lines from the supplied string. Lines that contain only whitespace characters are also considered blank.

456

789’) == ‘123 456 789’

`nunavut.jinja.DSDLCodeGenerator.filter_bits2bytes_ceil` (*n\_bits: int*) → `int`  
Implements `int(ceil(x/8)) | x >= 0`.

## Common Tests

`nunavut.jinja.DSDLCodeGenerator.is_None` (*value: Any*) → `bool`  
Tests if a value is None

`nunavut.jinja.DSDLCodeGenerator.is_saturated` (*t: pydsdl.\_serializable.\_primitive.PrimitiveType*) → `bool`  
Tests if a type is a saturated type or not.

`nunavut.jinja.DSDLCodeGenerator.is_service_request` (*instance: pydsdl.\_expression.\_any.Any*) → `bool`  
Tests if a type is request type of a service type.

`nunavut.jinja.DSDLCodeGenerator.is_service_response` (*instance: pydsdl.\_expression.\_any.Any*) → `bool`  
Tests if a type is response type of a service type.

`nunavut.jinja.DSDLCodeGenerator.is_deprecated` (*instance: pydsdl.\_expression.\_any.Any*) → `bool`  
Tests if a type is marked as deprecated

## C Filters

`nunavut.lang.c.filter_id` (*language: nunavut.lang.c.Language, instance: Any, id\_type: str = ‘any’*) → `str`  
Filter that produces a valid C identifier for a given object. The encoding may not be reversible.

```
# Given
I = 'I c'

# and
template = '{{ I | id }}'

# then
rendered = 'I_zX2764_c'
```

```
# Given
I = 'if'

# and
template = '{{ I | id }}'

# then
rendered = '_if'
```

```
# Given
I = '_Reserved'

# and
template = '{{ I | id }}'

# then
rendered = '_reserved'
```

```
# Given
I = 'EMACRO_TOKEN'

# and
template = '{{ I | id("macro") }}'

# then
rendered = '_eMACRO_TOKEN'
```

### Parameters

- **instance** (*any*) – Any object or data that either has a name property or can be converted to a string.
- **id\_type** (*str*) – A type of identifier. For C this value can be ‘typedef’, ‘macro’, ‘function’, or ‘enum’. use ‘any’ to apply stropping rules for all identifier types to the instance.

**Returns** A token that is a valid identifier for C, is not a reserved keyword, and is transformed in a deterministic manner based on the provided instance.

`nunavut.lang.c.filter_macrofy` (*language: nunavut.lang.c.Language, value: str*) → str

Filter to transform an input into a valid C preprocessor identifier token.

```
# Given
template = '#ifndef {{ "my full name" | macrofy }}'

# then
rendered = '#ifndef MY_FULL_NAME'
```

Note that individual tokens are not stropped so the appearance of an identifier in the

SCREAMING\_SNAKE\_CASE output may be different than the token as it appears on its own. For example:

```
# "register" is reserved so it will be stopped if it appears as an
# identifier...
template = '''#ifndef {{ "namespaced.Type.register" | macrofy }}
{{ "register" | id }}
'''

# ...but it will not be stopped within the macro.
rendered = '''#ifndef NAMESPACE_TYPE_REGISTER
_register
'''
```

If stopping is enabled, however, the entire token generated by this filter will be stopped:

```
# Given
template = '#ifndef {{ "_starts_with_underscore" | macrofy }}'

# then
rendered = '#ifndef _STARTS_WITH_UNDERSCORE'
```

And again with stopping disabled:

```
# Given
template = '#ifndef {{ "_starts_with_underscore" | macrofy }}'

# then with stopping disabled
rendered = '#ifndef _STARTS_WITH_UNDERSCORE'
```

**Parameters** `value` (*str*) – The value to transform.

**Returns** A valid C preprocessor identifier token.

`nunavut.lang.c.filter_type_from_primitive` (*language: nunavut.lang.c.Language, value: pydsdl.\_serializable.\_primitive.PrimitiveType*)

Filter to transform a pydsdl `PrimitiveType` into a valid C type.

```
# Given
template = '{{ unsigned_int_32_type | type_from_primitive }}'

# then
rendered = 'uint32_t'
```

```
# Given
template = '{{ int_64_type | type_from_primitive }}'

# then
rendered = 'int64_t'
```

**Parameters** `value` (*str*) – The dsdl primitive to transform.

**Returns** A valid C99 type name.

**Raises** `RuntimeError` – If the primitive cannot be represented as a standard C type.

`nunavut.lang.c.filter_to_snake_case` (*value: str*) → *str*

Filter to transform a string into a snake-case token.

```
# Given
template = '{{ "scotec.mcu.Timer" | to_snake_case }} a();'

# then
rendered = 'scotec_mcu_timer a();'
```

```
# Given
template = '{{ "scotec.mcu.TimerHelper" | to_snake_case }} b();'

# then
rendered = 'scotec_mcu_timer_helper b();'
```

```
# and Given
template = '{{ "SCOTEC_MCU_TimerHelper" | to_snake_case }} b();'

# then
rendered = 'scotec_mcu_timer_helper b();'
```

**Parameters** **value** (*str*) – The string to transform into C snake-case.

**Returns** A valid C99 token using the snake-case convention.

`nunavut.lang.c.filter_to_screaming_snake_case` (*value: str*) → *str*  
Filter to transform a string into a SCREAMING\_SNAKE\_CASE token.

```
# Given
template = '{{ "scotec.mcu.Timer" | to_screaming_snake_case }} a();'

# then
rendered = 'SCOTEC_MCU_TIMER a();'
```

`nunavut.lang.c.filter_to_template_unique_name` (*\_: Any, base\_token: str*) → *str*  
Filter that takes a base token and forms a name that is very likely to be unique within the template the filter is invoked. This name is also very likely to be a valid C identifier.

---

**Important:** The exact tokens generated may change between major or minor versions of this library. The only guarantee provided is that the tokens will be stable for the same version of this library given the same input.

Also note that name uniqueness is only likely within a given template. Between templates there is no guarantee of uniqueness and, since this library does not lex generated source, there is no guarantee that the generated name does not conflict with a name generated by another means.

---

```
# Given
template = '{{ "foo" | to_template_unique_name }},{{ "Foo" | to_template_unique_
↪name }},'
template += '{{ "f00" | to_template_unique_name }}'

# then
rendered = '_foo0_,_fool_,_f000_'
```

```
# Given
template = '{{ "i like coffee" | to_template_unique_name }}'

# then
rendered = '_i like coffee0_'
```

**Parameters** `base_token` (*str*) – A token to include in the base name.

**Returns** A name that is likely to be valid C identifier and is likely to be unique within the file generated by the current template.

`nunavut.lang.c.filter_short_reference_name` (*language: nunavut.lang.c.Language, t: pydsdl.\_serializable.\_composite.CompositeType*)  
→ *str*

Provides a string that is a shorted version of the full reference name.

```
# Given a type with illegal C characters
my_type.short_name = '_Foo'
my_type.version.major = 1
my_type.version.minor = 2

# and
template = '{{ my_type | short_reference_name }}'

# then, with stropping enabled
rendered = '_foo_1_2'
```

With stropping disabled:

```
rendered = '_Foo_1_2'
```

**Parameters** `t` (*pydsdl.CompositeType*) – The DSDL type to get the reference name for.

`nunavut.lang.c.filter_includes` (*language: nunavut.lang.c.Language, env: nunavut.jinja.jinja2.environment.Environment, t: pydsdl.\_serializable.\_composite.CompositeType, sort: bool = True*) → *List[str]*

Returns a list of all include paths for a given type.

```
# Listing the includes for a union with only integer types:
template = '''{% for include in my_type | includes -%}
{{include}}
{% endfor %}'''

# stdint.h will normally be generated
rendered = '''<stdint.h>
<stdlib.h>
'''
```

```
# You can suppress std includes by setting use_standard_types to False under
# nunavut.lang.c
rendered = ''
```

### Parameters

- `t` (*pydsdl.CompositeType*) – The type to scan for dependencies.
- `sort` (*bool*) – If true the returned list will be sorted.

**Returns** a list of include headers needed for a given type.

`nunavut.lang.c.filter_to_static_assertion_value` (*obj: Any*) → *int*

Tries to convert a Python object into a value compatible with static comparisons in C. This allows stable comparison of static values in headers to promote consistency and version compatibility in generated code.

Will raise a `ValueError` if the object provided does not (yet) have an available conversion in this function.

Currently supported types are string:

```
# given
template = '{{ "Any" | to_static_assertion_value }}'

# then
rendered = '1556001108'
```

int:

```
# given
template = '{{ 123 | to_static_assertion_value }}'

# then
rendered = '123'
```

and bool:

```
# given
template = '{{ True | to_static_assertion_value }}'

# then
rendered = '1'
```

`nunavut.lang.c.filter_constant_value` (*language*: `nunavut.lang.c.Language`, *constant*: `pydsdl._serializable._attribute.Constant`) → str

Renders the specified constant as a literal. This is a shorthand for `filter_literal()`.

```
# given
template = '{{ my_true_constant | constant_value }}'

# then
rendered = 'true'
```

Language configuration can control the output of some constant tokens. For example, to use non-standard true and false values in c:

```
# given
template = '{{ my_true_constant | constant_value }}'

# then, if true = 'NUNAVUT_TRUE' in the named_values for nunavut.lang.c
rendered = 'NUNAVUT_TRUE'
```

Floating point values are converted as fractions to ensure no python-specific transformations are applied:

```
# given a float value using a fraction of 355/113
template = '{{ almost_pi | constant_value }}'

# ...the rendered value with include that fraction as a division statement.
rendered = '((float) (355.0 / 113.0))'
```

`nunavut.lang.c.filter_literal` (*language*: `nunavut.lang.c.Language`, *value*: `Union[fractions.Fraction, bool, int]`, *ty*: `pydsdl._expression._any.Any`, *cast\_format*: `Optional[str]` = `None`) → str

Renders the specified value of the specified type as a literal.

`nunavut.lang.c.filter_full_reference_name` (*language*: `nunavut.lang.c.Language`, *t*: `pydsdl._serializable._composite.CompositeType`)  
 → str

Provides a string that is the full namespace, typename, major, and minor version for a given composite type.

```
# Given a type with illegal characters for C++
my_obj.full_name = 'any.int.2Foo'
my_obj.full_namespace = 'any.int'
my_obj.version.major = 1
my_obj.version.minor = 2

# and
template = '{{ my_obj | full_reference_name }}'

# then, with stropping enabled
rendered = 'any_int_2Foo_1_2'
```

**Parameters** *t* (`pydsdl.CompositeType`) – The DSDL type to get the fully-resolved reference name for.

`nunavut.lang.c.filter_to_standard_bit_length` (*t*: `pydsdl._serializable._primitive.PrimitiveType`)  
 → int

Returns the nearest standard bit length of a type as an int.

```
# Given
i = pydsdl.UnsignedIntegerType(7, pydsdl.PrimitiveType.CastMode.TRUNCATED)

# and
template = '{{ I | to_standard_bit_length }}'

# then
rendered = '8'
```

## C Tests

`nunavut.lang.c.is_zero_cost_primitive` (*language*: `nunavut.lang.c.Language`, *t*: `pydsdl._serializable._primitive.PrimitiveType`)  
 → bool

Assuming that the target platform is IEEE754-conformant detects whether the native in-memory representation of a value of the supplied primitive type is the same as its on-the-wire representation defined by the DSDL Specification.

For instance; all little-endian, IEEE754-conformant platforms have compatible in-memory representations of int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, float64. Values of other primitive types typically require some transformations (e.g., float16).

It follows that arrays, certain composite types, and some other entities composed of zero-cost composites are also zero-cost types, but such non-trivial conjectures are not recognized by this function.

Raises a `TypeError` if the argument is not a value of type `pydsdl.PrimitiveType`.

```
# Given
i7 = pydsdl.SignedIntegerType(7, pydsdl.PrimitiveType.CastMode.SATURATED)
u32 = pydsdl.UnsignedIntegerType(32, pydsdl.PrimitiveType.CastMode.TRUNCATED)
f16 = pydsdl.FloatType(16, pydsdl.PrimitiveType.CastMode.TRUNCATED)
f32 = pydsdl.FloatType(32, pydsdl.PrimitiveType.CastMode.SATURATED)
b1 = pydsdl.BooleanType(pydsdl.PrimitiveType.CastMode.SATURATED)
```

(continues on next page)



(continued from previous page)

```
# and
template = (
  '{{ i7 is zero_cost_primitive }} '
  '{{ u32 is zero_cost_primitive }} '
  '{{ f16 is zero_cost_primitive }} '
  '{{ f32 is zero_cost_primitive }} '
  '{{ b1 is zero_cost_primitive }} '
)

# then
rendered = 'False True False True False'
```

## C++ Filters

`nunavut.lang.cpp.filter_id` (*language: nunavut.lang.cpp.Language, instance: Any, id\_type: str = 'any'*) → str

Filter that produces a valid C and/or C++ identifier for a given object. The encoding may not be reversible.

```
# Given
I = 'I like c++'

# and
template = '{{ I | id }}'

# then
rendered = 'I_like_czX002BzX002B'
```

```
# Given
I = 'if'

# and
template = '{{ I | id }}'

# then
rendered = '_if'
```

```
# Given
I = 'I really like coffee'

# and
template = '{{ I | id }}'

# then
rendered = 'I_really_like_coffee'
```

**Parameters** *instance* (*any*) – Any object or data that either has a name property or can be converted to a string.

**Returns** A token that is a valid identifier for C and C++, is not a reserved keyword, and is transformed in a deterministic manner based on the provided instance.

`nunavut.lang.cpp.filter_open_namespace` (*language: nunavut.lang.cpp.Language, full\_namespace: str, bracket\_on\_next\_line: bool = True, linesep: str = '\n'*) → str

Emits c++ opening namespace syntax parsed from a pydsdl “full\_namespace”, dot-separated value.

```
# Given
T.full_namespace = 'uavcan.foo'

# and
template = '{{ T.full_namespace | open_namespace }}'

# then
rendered = '''namespace uavcan
{
namespace foo
{'''
```

### Parameters

- **full\_namespace** (*str*) – A dot-separated namespace string.
- **bracket\_on\_next\_line** (*bool*) – If True (the default) then the opening brackets are placed on a newline after the namespace keyword.
- **linesep** (*str*) – The line-separator to use when emitting new lines. By default this is \n.

**Returns** C++ namespace declarations with opening brackets.

nunavut.lang.cpp.**filter\_close\_namespace** (*language: nunavut.lang.cpp.Language, full\_namespace: str, omit\_comments: bool = False, linesep: str = '\n'*) → *str*

Emits c++ closing namespace syntax parsed from a pydsdl “full\_namespace”, dot-separated value.

```
# Given
T.full_namespace = 'uavcan.foo'

# and
template = '{{ T.full_namespace | close_namespace }}'

# then
rendered = '''} // namespace foo
} // namespace uavcan'''
```

### Parameters

- **full\_namespace** (*str*) – A dot-separated namespace string.
- **omit\_comments** (*bool*) – If True then the comments following the closing bracket are omitted.
- **linesep** (*str*) – The line-separator to use when emitting new lines. By default this is \n

**Returns** C++ namespace declarations with opening brackets.

nunavut.lang.cpp.**filter\_full\_reference\_name** (*language: nunavut.lang.cpp.Language, t: pydsdl.\_serializable.\_composite.CompositeType*) → *str*

Provides a string that is the full namespace, typename, major, and minor version for a given composite type.

```
# Given a type with illegal characters for C++
my_obj.full_name = 'any.int.2Foo'
my_obj.version.major = 1
my_obj.version.minor = 2
```

(continues on next page)

(continued from previous page)

```
# and
template = '{{ my_obj | full_reference_name }}'

# then, with stropping enabled
rendered = 'any::_int::_2Foo_1_2'
```

**Parameters** *t* (*pydsdl.CompositeType*) – The DSDL type to get the fully-resolved reference name for.

`nunavut.lang.cpp.filter_short_reference_name` (*language: nunavut.lang.cpp.Language, t: pydsdl.\_serializable.\_composite.CompositeType* → *str*)

Provides a string that is a shorted version of the full reference name. This type is unique only within its namespace.

```
# Given a type with illegal C++ characters
my_type.short_name = '2Foo'
my_type.version.major = 1
my_type.version.minor = 2

# and
template = '{{ my_type | short_reference_name }}'

# then, with stropping enabled
rendered = '_2Foo_1_2'
```

```
# Given a type with legal C++ characters
my_type.short_name = 'Struct_'
my_type.version.major = 0
my_type.version.minor = 1

# and
template = '{{ my_type | short_reference_name }}'

# then, with stropping enabled
rendered = 'Struct__0_1'
```

```
# Given a service type
my_service_type.short_name = 'Struct_'
my_service_type.version.major = 0
my_service_type.version.minor = 1

# and
template = '''
{{ my_service_type | short_reference_name }}
{{ my_service_type.request_type | short_reference_name }}
{{ my_service_type.response_type | short_reference_name }}
'''

# then, with stropping enabled
rendered = '''
Struct_
Request_0_1
Response_0_1
'''
```

**Parameters** *t* (*pydsdl.CompositeType*) – The DSDL type to get the reference name for.

`nunavut.lang.cpp.filter_includes` (*language:* *nunavut.lang.cpp.Language*, *env:*  
*nunavut.jinja.jinja2.environment.Environment*, *t:*  
*pydsdl.\_serializable.\_composite.CompositeType*, *sort:*  
*bool = True*) → List[str]

Returns a list of all include paths for a given type.

**Parameters**

- *t* (*pydsdl.CompositeType*) – The type to scan for dependencies.
- *sort* (*bool*) – If true the returned list will be sorted.

**Returns** a list of include headers needed for a given type.

```
# Listing the includes for a union with only integer types:
template = "{% for include in my_type | includes -%}{{include}}{%- endfor %}"

# cstdint will normally be generated. limits is always generated.
rendered = "<cstdint><limits>"
```

```
# You can suppress std includes by setting use_standard_types to False under
# nunavut.lang.cpp
rendered = "<limits>"
```

`nunavut.lang.cpp.filter_destructor_name` (*language:* *nunavut.lang.cpp.Language*, *instance:*  
*pydsdl.\_expression.\_any.Any*) → str

Returns a token that is the local destructor name. For example:

```
# Given a pydsdl.FixedLengthArrayType "my_type":
my_type.short_name = 'Foo'
my_type.capacity = 2

# and
template = 'ptr->{{ my_type | destructor_name }}'

# then
rendered = 'ptr->~array<std::uint8_t,2>'
```

**Parameters** *t* (*pydsdl.CompositeType*) – The type to generate a destructor template for.

**Returns** A destructor name token.

`nunavut.lang.cpp.filter_declaration` (*language:* *nunavut.lang.cpp.Language*, *instance:*  
*pydsdl.\_expression.\_any.Any*) → str

Emit a declaration statement for the given instance.

`nunavut.lang.cpp.filter_to_namespace_qualifier` (*namespace\_list:* List[str]) → str

Converts a list of namespace names into a qualifier string. For example:

```
my_namespace = ['foo', 'bar']
template = '{{ my_namespace | to_namespace_qualifier }}myType()'
expected = 'foo::bar::myType()'
```

This filter gracefully handles empty namespace lists:

```
my_namespace = []
template = '{{ my_namespace | to_namespace_qualifier }}myType()'
expected = 'myType()'
```

`nunavut.lang.cpp.filter_type_from_primitive` (*language*: `nunavut.lang.cpp.Language`,  
*value*: `pydsdl._serializable._primitive.PrimitiveType`)  
 $\rightarrow$  `str`  
 Filter to transform a pydsdl `PrimitiveType` into a valid C++ type.

```
# Given
template = '{{ unsigned_int_32_type | type_from_primitive }}'

# then
rendered = 'std::uint32_t'
```

Also note that this is sensitive to the `use_standard_types` configuration in the language properties:

```
# rendered will be different if use_standard_types is False
rendered = 'unsigned long'
```

**Parameters** `value` (*str*) – The dsdl primitive to transform.

**Returns** A valid C++ type name.

**Raises** `TemplateRuntimeError` – If the primitive cannot be represented as a standard C++ type.

`nunavut.lang.cpp.filter_to_template_unique_name` (*base\_token*: *str*)  $\rightarrow$  `str`

Filter that takes a base token and forms a name that is very likely to be unique within the template the filter is invoked. This name is also very likely to be a valid C++ identifier.

---

**Important:** The exact tokens generated may change between major or minor versions of this library. The only guarantee provided is that the tokens will be stable for the same version of this library given the same input.

Also note that name uniqueness is only likely within a given template. Between templates there is no guarantee of uniqueness and, since this library does not lex generated source, there is no guarantee that the generated name does not conflict with a name generated by another means.

---

```
# Given
template = '{{ "foo" | to_template_unique_name }},{{ "Foo" | to_template_unique_
↵name }}, '
template += '{{ "f00" | to_template_unique_name }}'

# then
rendered = '_foo0_,_foo1_,_f000_'
```

```
# Given
template = '{{ "i like coffee" | to_template_unique_name }}'

# then
rendered = '_i like coffee0_'
```

**Parameters** `base_token` (*str*) – A token to include in the base name.

**Returns** A name that is likely to be valid C++ identifier and is likely to be unique within the file generated by the current template.

`nunavut.lang.cpp.filter_as_boolean_value` (*value*: *bool*)  $\rightarrow$  `str`

Filter a boolean expression to produce a valid C++ “true” or “false” token.

```
assert "true" == filter_as_boolean_value(True)
assert "false" == filter_as_boolean_value(False)
```

`nunavut.lang.cpp.filter_indent_if_not` (*language: nunavut.lang.cpp.Language, text: str, depth: int = 1*) → str

Emit indent characters as configured for the language but only as needed. This is different from the built-in indent filter in that it may add or remove spaces based on the existing indent.

```
# Given a string with an existing indent of 4 spaces...
template = '{{ "    int a = 1;" | indent_if_not }}'

# then if cpp.indent == 4 we expect no change.
rendered = '    int a = 1;'
```

```
# If the indent is only 3 spaces...
template = '{{ "    int a = 1;" | indent_if_not }}'

# then if cpp.indent == 4 we expect 4 spaces (i.e. not 7 spaces)
rendered = '    int a = 1;'
```

```
# We can also specify multiple indents...
template = '{{ "int a = 1;" | indent_if_not(2) }}'

rendered = '        int a = 1;'
```

```
# ...or no indent
template = '{{ "    int a = 1;" | indent_if_not(0) }}'

rendered = 'int a = 1;'
```

```
# Finally, note that blank lines are not indented.
template = '''
    {%- set block_text -%}
    int a = 1;
    {# empty line #}
    int b = 2;
    {%- endset -%}}{{ block_text | indent_if_not(1) }}'''

rendered = '    int a = 1;'
rendered += '\n'
rendered += '\n' # Nothing but spaces so this is stripped
rendered += '    int b = 2;'
```

### Parameters

- **text** – The text to indent.
- **depth** – The number of indents. For example, if depth is 2 and the indent for this language is 4 spaces then the text will be indented by 8 spaces.

`nunavut.lang.cpp.filter_minimum_required_capacity_bits` (*t: pydsdl.\_serializable.\_serializable.SerializableType*) → int

Returns the minimum number of bits required to store the deserialized value of a pydsdl SerializableType. This capacity may be too small for some instances of the value (e.g. variable length arrays).

```
# Given
template = '{{ unsigned_int_32_type | minimum_required_capacity_bits }}'

# then
rendered = '32'
```

**Parameters** `t` (`pydsdl.SerializableType`) – The dsdl type.

**Returns** The minimum, required bits needed to store some values of the given type.

`nunavut.lang.cpp.filter_block_comment` (*language: nunavut.lang.cpp.Language, text: str, style: str, indent: int = 0, line\_length: int = 100*) → str  
Reformats text as a block comment using Python's `textwrap.TextWrapper.wrap()` function.

#### Parameters

- **text** – The text to emit as a block comment.
- **style** – Dictates the style of comments (see return documentation for valid style names).
- **indent** – The number of spaces to indent the comments by (tab indent is not supported. Sorry).
- **line\_length** – The soft maximum width to wrap text at. Some violations may occur where long words are used.

**Returns** `str` A comment block. Comment styles supported are:

#### javadoc

```
# Given a type with the following docstring
text = 'This is a bunch of documentation.'

# and
template = '''
    {{ text | block_comment('javadoc', 4, 24) }}
    void some_method();
'''

# the output will be
rendered = '''
    /**
     * This is a bunch
     * of documentation.
     */
    void some_method();
'''
```

#### cpp-doxygen

```
# that same template using the cpp style of doxygen...
template = '''
    {{ text | block_comment('cpp-doxygen', 4, 24) }}
    void some_method();
'''

# ...will be
rendered = '''
    ///
    /// This is a bunch
'''
```

(continues on next page)

(continued from previous page)

```

    /// of
    /// documentation.
    ///
    void some_method();
    '''

```

**cpp**

```

# also supported is cpp style...
template = '''
    {{ text | block_comment('cpp', 4, 24) }}
    void some_method();
    '''

rendered = '''
    // This is a bunch of
    // documentation.
    void some_method();
    '''

```

**c**

```

# c style...
template = '''
    {{ text | block_comment('c', 4, 24) }}
    void some_method();
    '''

rendered = '''
    /*
     * This is a bunch
     * of documentation.
     */
    void some_method();
    '''

```

**qt**

```

# and Qt style...
template = '''
    {{ text | block_comment('qt', 4, 24) }}
    void some_method();
    '''

rendered = '''
    /*!
     * This is a bunch
     * of documentation.
     */
    void some_method();
    '''

```

**C++ Use Queries**

`nunavut.lang.cpp.uses_std_variant` (*language: nunavut.lang.cpp.Language*) → bool  
 Uses query for std variant.



If the language options contain an `std` entry for C++ and the specified standard includes the `std::variant` type added to the language at C++17 then this value is true. The logic included in this filter can be stated as “options has key `std` and the value for `options.std` evaluates to C++ version 17 or greater” but the implementation is able to parse out actual compiler flags like `gnu++20` and is aware of any overrides to suppress use of the standard variant type even if available.

Example:

```
template = '''
    {%- ifuses "std_variant" -%}
        #include <variant>
    {%- else -%}
        #include "user_variant.h"
    {%- endifuses -%}
'''
```

## Python Filters

`nunavut.lang.py.filter_id` (*language: nunavut.lang.py.Language, instance: Any, id\_type: str = 'any'*) → str

Filter that produces a valid Python identifier for a given object. The encoding may not be reversible.

```
# Given
I = 'I like python'

# and
template = '{{ I | id }}'

# then
rendered = 'I_like_python'
```

```
# Given
I = '&because'

# and
template = '{{ I | id }}'

# then
rendered = 'zX0026because'
```

```
# Given
I = 'if'

# and
template = '{{ I | id }}'

# then
rendered = 'if_'
```

**Parameters** `instance` (*any*) – Any object or data that either has a `name` property or can be converted to a string.

**Returns** A token that is a valid Python identifier, is not a reserved keyword, and is transformed in a deterministic manner based on the provided instance.

`nunavut.lang.py.filter_to_template_unique_name` (*context: nunavut.\_templates.SupportsTemplateContext, base\_token: str*) → str

Filter that takes a base token and forms a name that is very likely to be unique within the template the filter is invoked. This name is also very likely to be a valid Python identifier.

**Important:** The exact tokens generated may change between major or minor versions of this library. The only guarantee provided is that the tokens will be stable for the same version of this library given the same input.

Also note that name uniqueness is only likely within a given template. Between templates there is no guarantee of uniqueness and, since this library does not lex generated source, there is no guarantee that the generated name does not conflict with a name generated by another means.

```
# Given
template = '{{ "f" | to_template_unique_name }},{{ "f" | to_template_unique_name_
↵}},'
template += '{{ "f" | to_template_unique_name }},{{ "bar" | to_template_unique_
↵name }}'

# then
rendered = '_f0_,_f1_,_f2_,_bar0_'
```

```
# Given
template = '{{ "i like coffee" | to_template_unique_name }}'

# then
rendered = '_i like coffee0_'
```

**Parameters** `base_token` (*str*) – A token to include in the base name.

**Returns** A name that is likely to be valid python identifier and is likely to be unique within the file generated by the current template.

`nunavut.lang.py.filter_full_reference_name` (*language: nunavut.lang.py.Language, t: pydsdl.\_serializable.\_composite.CompositeType*) → str

Provides a string that is the full namespace, typename, major, and minor version for a given composite type.

```
# Given
full_name = 'any.str.2Foo'
major = 1
minor = 2

# and
template = '{{ my_obj | full_reference_name }}'

# then
rendered = 'any_.str_.zX0032Foo_1_2'
```

**Parameters** `t` (*pydsdl.CompositeType*) – The DSDL type to get the fully-resolved reference name for.

`nunavut.lang.py.filter_short_reference_name` (*language: nunavut.lang.py.Language, t: pydsdl.\_serializable.\_composite.CompositeType*) → str

Provides a string that is a shorted version of the full reference name. This type is unique only within its namespace.

```
# Given
short_name = '2Foo'
major = 1
minor = 2

# and
template = '{{ my_obj | short_reference_name }}'

# then
rendered = 'zX0032Foo_1_2'
```

**Parameters** `t` (`pydsdl.CompositeType`) – The DSDL type to get the reference name for.

`nunavut.lang.py.filter_imports` (`language: nunavut.lang.py.Language, t: pydsdl._serializable._composite.CompositeType, sort: bool = True`) → `List[str]`

Returns a list of all modules that must be imported to use a given type.

**Parameters**

- `t` (`pydsdl.CompositeType`) – The type to scan for dependencies.
- `sort` (`bool`) – If true the returned list will be sorted.

**Returns** a list of python module names the provided type depends on.

`nunavut.lang.py.filter_longest_id_length` (`language: nunavut.lang.py.Language, attributes: List[pydsdl._serializable._attribute.Attribute]`) → `int`

Return the length of the longest identifier in a list of `pydsdl.Attribute` objects.

```
# Given
I = ['one.str.int.any', 'three.str.int.any']

# and
template = '{{ I | longest_id_length }}'

# then
rendered = '32'
```

### HTML Filters

`nunavut.lang.html.filter_extent` (`instance: pydsdl_expression._any.Any`) → `int`

`nunavut.lang.html.filter_max_bit_length` (`instance: pydsdl_expression._any.Any`) → `int`

`nunavut.lang.html.filter_tag_id` (`instance: pydsdl_expression._any.Any`) → `str`

`nunavut.lang.html.filter_url_from_type` (`instance: pydsdl_expression._any.Any`) → `str`

`nunavut.lang.html.filter_make_unique` (`_: Any, base_token: str`) → `str`

Filter that takes a base token and forms a name that is very likely to be unique within the template the filter is invoked.

---

**Important:** The exact tokens generated may change between major or minor versions of this library. The only guarantee provided is that the tokens will be stable for the same version of this library given the same input.

Also note that name uniqueness is only likely within a given template. Between templates there is no guarantee of uniqueness and, since this library does not lex generated source, there is no guarantee that the generated name does not conflict with a name generated by another means.

```
# Given
template = '{{ "foo" | make_unique }},{ "Foo" | make_unique }},'
template += '{{ "f00" | make_unique }}'

# then
rendered = 'foo0,foo1,f000'
```

```
# Given
template = '{{ "coffee > tea" | make_unique }}'

# then
rendered = 'coffee &gt; tea0'
```

**Parameters** `base_token` (*str*) – A token to include in the base name.

**Returns** A name that is likely to be unique within the file generated by the current template.

`nunavut.lang.html.filter_namespace_doc` (*ns: nunavut.\_namespace.Namespace*) → *str*

`nunavut.lang.html.filter_display_type` (*instance: pydsdl.\_expression.\_any.Any*) → *str*

`nunavut.lang.html.filter_natural_sort_namespace` (*instance: List[pydsdl.\_expression.\_any.Any]*) → *List[pydsdl.\_expression.\_any.Any]*

Namespaces come in plain lists; sort by name only.

`nunavut.lang.html.filter_natural_sort_type` (*instance: pydsdl.\_expression.\_any.Any*) → *List[pydsdl.\_expression.\_any.Any]*

Types come in tuples (type, path). Sort by type name.

## 3.2 Template Mapping and Use

Templates are resolved as `templates/path/[dsdl_typename].j2` This means you must, typically, start with four templates under the `templates` directory given to the `Generator` instance

```
ServiceType.j2
StructureType.j2
DelimitedType.j2
UnionType.j2
```

**Note:** You can chose to use a single template `Any.j2` but this may lead to more complex templates with many more control statements. By providing discreet templates named for top-level data types and using `jinja` template inheritance and includes your templates will be smaller and easier to maintain.

To share common formatting for these templates use [Jinja template inheritance](#). For example, given a template `common_header.j2`:

```
/*
 * Cyphal data structure definition for nunavut.
```

(continues on next page)

(continued from previous page)

```

*
* Auto-generated, do not edit.
*
* Source file: {{T.source_file_path.as_posix()}}
* Generated at: {{now_utc}}
* Template: {{ self._TemplateReference__context.name }}
* deprecated: {{T.deprecated}}
* fixed_port_id: {{T.fixed_port_id}}
* full_name: {{T.full_name}}
* full_namespace: {{T.full_namespace}}
*/

#ifdef {{T.full_name | ln.c.macrofy}}
#define {{T.full_name | ln.c.macrofy}}

{%- block contents %} {% endblock -%

#endif // {{T.full_name | ln.c.macrofy}}

/*
{{ T | yamify }}
*/

```

... your three top-level templates would each start out with something like this:

```

{% extends "common_header.j2" %}
{% block contents %}
// generate stuff here
{% endblock %}

```

### 3.2.1 Resolving Types to Templates

You can apply the same logic used by the top level generator to recursively include templates by type if this seems useful for your project. Simply use the `nunavut.jinja.Generator.filter_type_to_template()` filter:

```

{%- for attribute in T.attributes %}
    {%* include attribute.data_type | type_to_template %}
{%- endfor %}

```

### 3.2.2 Namespace Templates

If the `generate_namespace_types` parameter of `Generator` is `YES` then the generator will always invoke a template for the root namespace and all nested namespaces regardless of language. `NO` suppresses this behavior and `DEFAULT` will choose the behavior based on the target language. For example:

```

root_namespace = build_namespace_tree(compound_types,
                                     root_ns_folder,
                                     out_dir,
                                     language_context)

generator = Generator(root_namespace, YesNoDefault.DEFAULT)

```

Would generate python `__init__.py` files to define each namespace as a python module but would not generate any additional headers for C++.

The `Generator` will use the same template name resolution logic as used for `pydsdl` data types. For namespaces this will resolve first to a template named `Namespace.j2` and then, if not found, `Any.j2`.

### 3.2.3 Internals

Nunavut reserves all global identifiers that start with `_nv_` as private internal globals.

## 3.3 Built-in Template Guide

This section will contain more information as the project matures about the build-in language support for generating code. Nunavut is both a framework that allows users to write their own `dsdl` transformation templates but also works, out-of-the-box, as a transpiler for C, and C++. More languages may be added in the future.

### 3.3.1 C++

---

**Note:** C++ support is currently experimental. You can only use this by setting the `--experimental-languages` flag when invoking `nnvg`.

---

### 3.3.2 C

---

**Note:** Documentation is provided in the generated source.

---

#### Manual Override of Array Capacity

By default, the C structures generated will utilize C arrays sized by the maximum size of a variable-length array. To override this behavior you can pre-define the `_ARRAY_CAPACITY_` for individual fields. For example:

```
#include <stdio.h>

#define reg_drone_service_battery_Status_0_2_cell_voltages_ARRAY_CAPACITY_ 6
#include "inc/UAVCAN/reg/drone/service/battery/Status_0_2.h"

int main(int argc, char *argv[])
{
    reg_drone_service_battery_Status_0_2 msg;
    printf("Size of reg_drone_service_battery_Status_0_2 %li\n", sizeof(msg));
}
```

## 4.1 Usage

Generate code from Cyphal DSDL using pydsdl and jinja2

```
usage: nnvg [-h] [--lookup-dir LOOKUP_DIR] [--verbose] [--version]
            [--outdir OUTDIR] [--templates TEMPLATES]
            [--target-language TARGET_LANGUAGE] [--experimental-languages]
            [--output-extension OUTPUT_EXTENSION] [--dry-run] [--list-outputs]
            [--generate-support {always,never,as-needed,only}] [--list-inputs]
            [--generate-namespace-types] [--omit-serialization-support]
            [--namespace-output-stem NAMESPACE_OUTPUT_STEM] [--no-overwrite]
            [--file-mode FILE_MODE] [--trim-blocks] [--lstrip-blocks]
            [--allow-unregulated-fixed-port-id]
            [--pp-max-emptylines PP_MAX_EMPTYLINES]
            [--pp-trim-trailing-whitespace] [-pp-rp PP_RUN_PROGRAM]
            [-pp-rpa PP_RUN_PROGRAM_ARG]
            [--target-endianness {any,big,little}]
            [--omit-float-serialization-support]
            [--enable-serialization-asserts]
            [--enable-override-variable-array-capacity]
            [--language-standard LANGUAGE_STANDARD]
            [--configuration [CONFIGURATION [CONFIGURATION ...]]]
            [--list-configuration]
            [root_namespace]
```

### 4.1.1 Positional Arguments

**root\_namespace**    A source directory with DSDL definitions.  
Default: “.”

## 4.1.2 Named Arguments

- lookup-dir, -I** List of other namespace directories containing data type definitions that are referred to from the target root namespace. For example, if you are reading a vendor-specific namespace, the list of lookup directories should always include a path to the standard root namespace “uavcan”, otherwise the types defined in the vendor-specific namespace won’t be able to use data types from the standard namespace.
- Additional directories can also be specified through an environment variable `DSDL_INCLUDE_PATH` where the path entries are separated by colons “:” on posix systems and “;” on Windows.
- Default: []
- verbose, -v** verbosity level (-v, -vv)
- version** show program’s version number and exit
- outdir, -O** output directory
- Default: “nunavut\_out”
- templates** Paths to a directory containing templates to use when generating code.
- Templates found under these paths will override the built-in templates for a given language.
- target-language, -l** Language support to install into the templates.
- If provided then the output extension (-e) can be inferred otherwise the output extension must be provided.
- experimental-languages, -Xlang** Activate languages with unstable, experimental support.
- By default, target languages where support is not finalized are not enabled when running nunavut, to make it clear that the code output may change in a non-backwards-compatible way in future versions, or that it might not even work yet.
- Default: False
- output-extension, -e** The extension to use for generated files.
- dry-run, -d** If True then no files will be generated.
- Default: False
- list-outputs** Emit a semicolon-separated list of files. (implies -dry-run) Emits files that would be generated if invoked without -dry-run. This command is useful for integrating with CMake and other build systems that need a list of targets to determine if a rebuild is necessary.
- Default: False
- generate-support** Possible choices: always, never, as-needed, only
- Change the criteria used to enable or disable support code generation.
- as-needed (default) - generate support code if serialization is enabled. always - always generate support code. never - never generate support code. only - only generate support code.
- Default: “as-needed”



- list-inputs** Emit a semicolon-separated list of files. (implies `--dry-run`) A list of files that are resolved given input arguments like templates. This command is useful for integrating with CMake and other build systems that need a list of inputs to determine if a rebuild is necessary.
- Default: False
- generate-namespace-types** If enabled this script will generate source for namespaces. All namespaces including and under the root namespace will be treated as a pseudo-type and the appropriate template will be used. The generator will first look for a template with the stem "Namespace" and will then use the "Any" template if that is available. The name of the output file will be the default value for the `--namespace-output-stem` argument and can be changed using that argument.
- Default: False
- omit-serialization-support, -pod** If provided then the types generated will be POD datatypes with no additional logic. By default types generated include serialization routines and additional support libraries, headers, or methods.
- Default: False
- namespace-output-stem** The name of the file generated when `--generate-namespace-types` is provided.
- no-overwrite** By default, generated files will be silently overwritten by subsequent invocations of the generator. If this argument is specified an error will be raised instead preventing overwrites.
- Default: False
- file-mode** The file-mode each generated file is set to after it is created. Note that this value is interpreted using python auto base detection. Because of this, to provide an octal value, you'll need to prefix your literal with '0o' (e.g. `--file-mode 0o664`).
- Default: 292
- trim-blocks** If this is set to True the first newline after a block in a template is removed (block, not variable tag!).
- Default: False
- lstrip-blocks** If this is set to True leading spaces and tabs are stripped from the start of a line to a block in templates.
- Default: False
- allow-unregulated-fixed-port-id** Do not reject unregulated fixed port identifiers. This is a dangerous feature that must not be used unless you understand the risks. The background information is provided in the Cyphal specification.
- Default: False
- pp-max-emptylines** If provided this will suppress generation of additional consecutive empty lines beyond the limit set by this argument.
- Note that this will insert a line post-processor which may reduce performance. Consider using a code formatter on the generated output to enforce whitespace rules instead.
- pp-trim-trailing-whitespace** Enables a line post-processor that will elide all whitespace at the end of each line.

Note that this will insert a line post-processor which may reduce performance. Consider using a code formatter on the generated output to enforce whitespace rules instead.

Default: False

**-pp-rp, --pp-run-program** Runs a program after each file is generated but before the file is set to read-only.

example

```
# invokes clang-format with the "in-place" argument on
↳ each file after it is
# generated.

nsvg --outdir include --templates c_jinja -s .h -pp-rp
↳ clang-format -pp-rpa== dddl
```

**-pp-rpa, --pp-run-program-arg** Additional arguments to provide to the program specified by `-pp-run-program`. The last argument will always be the path to the generated file.

### 4.1.3 language options

Options passed through to templates as *options* on the target language.

Note that these arguments are passed though without validation, have no effect on the Nunavut library, and may or may not be appropriate based on the target language and generator templates in use.

**--target-endianness** Possible choices: any, big, little

Specify the endianness of the target hardware. This allows serialization logic to be optimized for different CPU architectures.

**--omit-float-serialization-support** Instruct support header generators to omit support for floating point operations in serialization routines. This will result in errors if floating point types are used, however; if you are working on a platform without IEEE754 support and do not use floating point types in your message definitions this option will avoid dead code or compiler errors in generated serialization logic.

Default: False

**--enable-serialization-asserts** Instruct support header generators to generate language-specific assert statements as part of serialization routines. By default the serialization logic generated may make assumptions based on documented requirements for calling logic that could expose a system to undefined behavior. The alternative, for languages that do not support exception handling, is to use assertions designed to halt a program rather than execute undefined logic.

Default: False

**--enable-override-variable-array-capacity** Instruct support header generators to add the possibility to override max capacity of a variable length array in serialization routines. This option will disable serialization buffer checks and add conditional compilation statements which violates MISRA.

Default: False

**--language-standard, -std** For language generators that support different standards of their core language this option can be used to optimize the output. For example, C templates may generate slightly different code for the the c99 standard then for c11. For

available support in Nunavut see the documentation for built-in templates (<https://nunavut.readthedocs.io/en/latest/docs/templates.html#built-in-template-guide>).

**--configuration, -c** There is a set of built-in configuration for Nunvut that provides default values for known languages as documented [in the template guide](#). This argument lets you specify override configuration yamls.

**--list-configuration, -lc** Lists all configuration values resolved for the given arguments.

Default: False

#### Example Usage:

```
# This would include j2 templates for a folder named 'c_jinja'  
# and generate .h files into a directory named 'include' using  
# dsdl root namespaces found under a folder named 'dsdl'.  
  
nnvg --outdir include --templates c_jinja -e .h dsdl
```



---

## Contributor Notes

---

Thanks for contributing. This page contains all the details about getting your dev environment setup.

---

**Note:** This is documentation for contributors developing nunavut. If you are a user of this software you can ignore everything here.

- To ask questions about nunavut or Cyphal in general please see the [OpenCyphal forum](#).
  - See [nunavut on read the docs](#) for the full set of nunavut documentation.
  - See the [OpenCyphal website](#) for documentation on the Cyphal protocol.
- 

**Warning:** When committing to main you **must** bump at least the patch number in `src/nunavut/_version.py` or the build will fail on the upload step.

## 5.1 Tools

### 5.1.1 tox -e local

I highly recommend using the local tox environment when doing python development. It'll save you hours of lost productivity the first time it keeps you from pulling in an unexpected dependency from your global python environment. You can install tox from brew on osx or apt-get on GNU/Linux. I'd recommend the following environment for vscode:

```
git submodule update --init --recursive
tox -e local
source .tox/local/bin/activate
```

## 5.1.2 cmake

Our language generation verification suite uses CMake to build and run unit tests. If you are working with a native language see *Nunavut Verification Suite* for details on manually running these builds and tests.

## 5.1.3 Visual Studio Code

To use vscode you'll need:

1. vscode
2. install vscode command line (*Shell Command: Install*)
3. tox
4. cmake (and an available GCC or Clang toolchain, or Docker to use our toolchain-as-container)

Do:

```
cd path/to/nunavut
git submodule update --init --recursive
tox -s local
source .tox/local/bin/activate
code .
```

Then install recommended extensions.

## 5.2 Running The Tests

To run the full suite of `tox` tests locally you'll need docker. Once you have docker installed and running do:

```
git submodule update --init --recursive
docker pull ghcr.io/opencyphal/toxic:tx22.4.1
docker run --rm -v $PWD:/repo ghcr.io/opencyphal/toxic:tx22.4.1 tox
```

To run a limited suite using only locally available interpreters directly on your host machine, skip the docker invocations and use `tox -s`.

To run the language verification build you'll need to use a different docker container:

```
docker pull ghcr.io/opencyphal/toolshed:ts20.4.1
docker run --rm -it -v $PWD:/workspace ghcr.io/opencyphal/toolshed:ts20.4.1
cd /workspace
./github/verify.py -l c
./github/verify.py -l cpp
```

The `verify.py` script is a simple commandline generator for our cmake scripts. Use help for details:

```
./github/verify.py --help
```

If you get a “denied” response from ghcr your ip might be getting rate-limited. While these are public containers you'll have to login to get around any rate-limiting for your local site. See [the github docs](<https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry>) for how to setup a docker client login.

### 5.2.1 Files Generated by the Tests

Given that Nunavut is a file generator our tests do have to write files. Normally these files are temporary and therefore automatically deleted after the test completes. If you want to keep the files so you can debug an issue provide a “keep-generated” argument.

#### example

```
pytest -k test_namespace_stropping --keep-generated
```

You will see each test’s output under “build/(test name)”.

**Warning:** Don’t use this option when running tests in parallel. You will get errors.

### 5.2.2 Sybil Doctest

This project makes extensive use of Sybil doctests. These take the form of docstrings with a structure like thus:

```
.. invisible-code-block: python
    from nunavut.lang.c import filter_to_snake_case
.. code-block:: python
    # an input like this:
    input = "scotec.mcu.Timer"
    # should yield:
    filter_to_snake_case(input)
    >>> scotec_mcu_timer
```

The invisible code block is executed but not displayed in the generated documentation and, conversely, code-block is both rendered using proper syntax formatting in the documentation and executed. REPL works the same as it does for doctest but assert is also a valid way to ensure the example is correct especially if used in a trailing invisible-code-block:

```
.. invisible-code-block: python
    assert 'scotec_mcu_timer' == filter_to_snake_case(input)
```

These tests are run as part of the regular pytest build. You can see the Sybil setup in the `conf_test.py` found under the `src` directory but otherwise shouldn’t need to worry about it. The simple rule is; if the docstring ends up in the rendered documentation then your code-block tests will be executed as unit tests.

### 5.2.3 import file mismatch

If you get an error like the following:

```
_____ ERROR collecting test/gentest_dsd1/test_dsd1.py _____
→
import file mismatch:
imported module 'test_dsd1' has this __file__ attribute:
/mnt/Workspace/nunavut/test/gentest_dsd1/test_dsd1.py
```

(continues on next page)

(continued from previous page)

```
which is not the same as the test file we want to collect:
/repo/test/gentest_dsd/test_dsd1.py
HINT: remove __pycache__ / .pyc files and/or use a unique basename for your test file.
↳ modules
```

Then you are probably a wonderful developer that is running the unit-tests locally. Pytest's cache is interfering with your docker test run. To work around this simply delete the pycache files. For example:

```
#!/usr/bin/env bash
clean_dirs="src test"

for clean_dir in $clean_dirs
do
    find $clean_dir -name __pycache__ | xargs rm -rf
    find $clean_dir -name \.coverage\* | xargs rm -f
done
```

Note that we also delete the .coverage intermediates since they may contain different paths between the container and the host build.

Alternatively just nuke everything temporary using git clean:

```
git clean -X -d -f
```

## 5.3 Building The Docs

We rely on [read the docs](#) to build our documentation from github but we also verify this build as part of our tox build. This means you can view a local copy after completing a full, successful test run (See [Running The Tests](#)) or do `docker run --rm -t -v $PWD:/repo ghcr.io/opencyphal/toxic:tx22.4.1 /bin/sh -c "tox -e docs"` to build the docs target. You can open the index.html under `.tox/docs/tmp/index.html` or run a local web-server:

```
python3 -m http.server --directory .tox/docs/tmp &
open http://localhost:8000/docs/index.html
```

Of course, you can just use [Visual Studio Code](#) to build and preview the docs using `> reStructuredText: Open Preview`.

### 5.3.1 apidoc

We manually generate the api doc using `sphinx-apidoc`. To regenerate use `tox -e gen-apidoc`.

**Warning:** `tox -e gen-apidoc` will start by deleting the docs/api directory.

## 5.4 Coverage and Linting Reports

We publish the results of our coverage data to [sonarcloud](#) and the tox build will fail for any mypy or black errors but you can view additional reports locally under the `.tox` dir.



### 5.4.1 Coverage

We generate a local html coverage report. You can open the index.html under `.tox/report/tmp` or run a local web-server:

```
python -m http.server --directory .tox/report/tmp &
open http://localhost:8000/index.html
```

### 5.4.2 Mypy

At the end of the mypy run we generate the following summaries:

- `.tox/mypy/tmp/mypy-report-lib/index.txt`
- `.tox/mypy/tmp/mypy-report-script/index.txt`

## 5.5 Nunavut Verification Suite

Nunavut has built-in support for several languages. Included with this is a suite of tests using typical test frameworks and language compilers, interpreters, and/or virtual machines. While each release of Nunavut is gated on automatic and successful completion of these tests this guide is provided to give system integrators information on how to customize these verifications to use other compilers, interpreters, and/or virtual machines.

### 5.5.1 CMake scripts

Our language generation verification suite uses CMake to build and run unit tests. Instructions for reproducing the CI automation execution steps are below. This section will tell you how to manually build and run individual unit tests as you develop them.

TLDR:

```
git submodule update --init --recursive
export NUNAVUT_VERIFICATION_LANG=:
cd verification
mkdir "build_${NUNAVUT_VERIFICATION_LANG}"
cd "build_${NUNAVUT_VERIFICATION_LANG}"
cmake ..
cmake --build . --target help
```

Try running a test which will first compile the test. For example, in the C language build

```
cmake --build . --target run_test_serialization
```

To run the C++ test use the same steps shown in the TLDR above but set `NUNAVUT_VERIFICATION_LANG` to “cpp” first.

In the list of targets that the `cmake --build . --target help` command lists the targets that build tests will be prefixed with `test_` and the pseudo-target that also executes the test will be prefixed with `run_test_`. You should avoid the `_with_lcov` when you are manually building tests.

To obtain coverage information for the verification suite (not the Python code), build the `cov_all` target and inspect the output under the `coverage` directory.

### cmake build options

The following options are supported when configuring your build. These can be specified by using `-D` arguments to `cmake`. For example

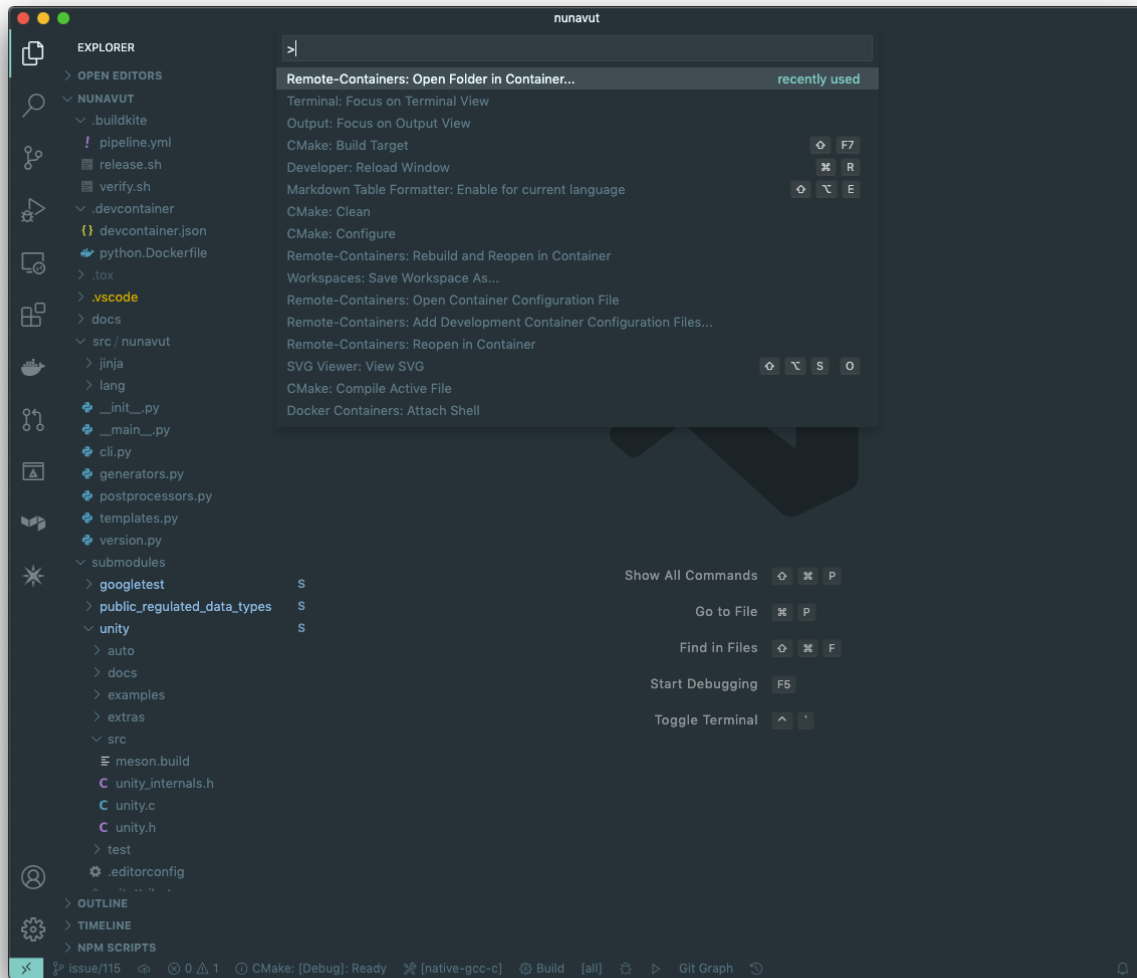
```
cmake -DNUNAVUT_VERIFICATION_LANG=c -DNUNAVUT_VERIFICATION_TARGET_ENDIANNESS=any ..
```

Option	Type	Default	Values	Description
CMAKE_BUILD_TYPE	string	release	Debug, Release, MinSizeRel	Compiler optimizations are set based on the CMake build type.
NUNAVUT_VERIFICATION_LANG	string	c, cpp	Specifies the language for source	code generated by nnvg.
NUNAVUT_VERIFICATION_TARGET_ENDIANNESS	string	any	little, big, any	Modifies generated serialization code and support code to support various CPU architectures. Other than endianness, Nunavut serialization and support code should be generic.
NUNAVUT_VERIFICATION_TARGET_PLATFORM	string	(unset)	native32, native64	The target platform to compile for. In future releases we hope to support ppc (Big), AVR8, RISCV, ARM.
NUNAVUT_VERIFICATION_SER_ASSERT	bool	ON	ON, OFF	Enable or disable asserts in generated serialization and support code.
NUNAVUT_VERIFICATION_SER_FP_DISABLE	bool	OFF	ON, OFF	Enable to omit floating-point serialization routines.
NUNAVUT_VERIFICATION_LANG_STANDARD	string	(unset)	c++17, c99 (etc)	override value for the -std compiler flag of the target language
<b>5.5. Nunavut Verification Suite</b>				

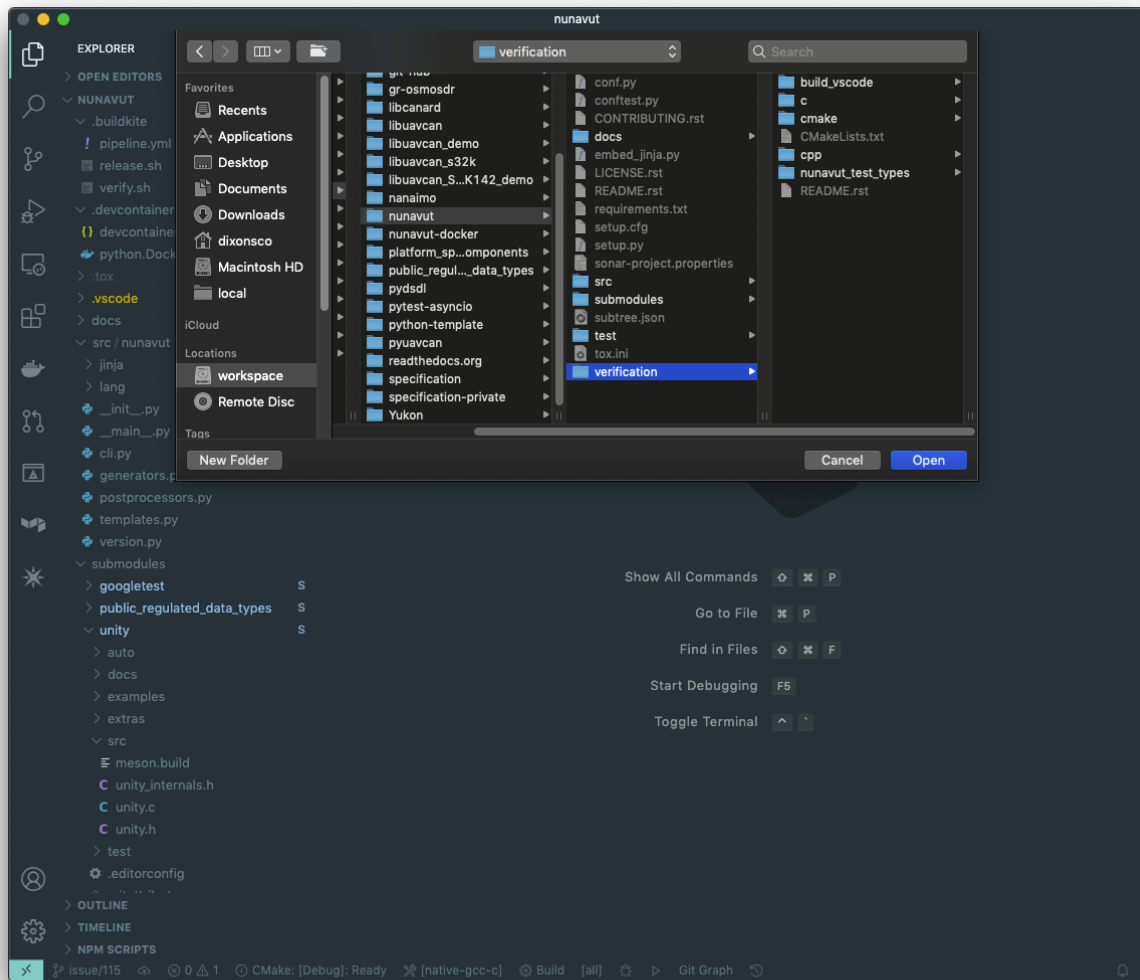
\* Because this option has no default, a value must be provided by the user.

## 5.5.2 VSCode Remote Container Development of Verification Tests

To write and debug verification tests using [VSCode Remote Containers](#) you'll need to use the “Open Folder in Container...” option:



Open the “verification” folder:



We play a little trick here where we dump you back into the Nunvut repo root when you reopen in the container. This lets you also work with the Python source. If you “reopen locally” while in this state, however, you’ll find yourself back in the verification folder which can be a little disorienting. Write to Microsoft asking them to allow multiple images in the .devcontainer json and we can get rid of this ugly hack. Sorry.



## 6.1 Licence

### 6.1.1 nunavut (MIT)

The MIT License (MIT)

Copyright (c) 2014 Pavel Kirienko Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 6.1.2 Jinja2 and MarkupSafe (BSD 3 clause)

Copyright (c) 2009 by the Jinja Team, see AUTHORS for more details.

Some rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



---

Nunavut: DSDL transpiler

---

tox build (main)	
static analysis	
unit test code coverage	
Python versions supported	
latest released version	
documentation	
license	
community/support	

Nunavut is a source-to-source compiler (transpiler) that automatically converts [OpenCyphal](#) DSDL definitions into source code in a specified target programming language. It is constructed as a template engine that exposes a [PyDSDL](#) abstract syntax tree to [Jinja2](#) templates allowing authors to generate code, schemas, metadata, documentation, etc.

Fig. 1: Nunavut DSDL transcompilation pipeline.

Nunavut ships with built-in support for some programming languages, and it can be used to generate code for other languages if custom templates (and some glue logic) are provided. Currently, the following languages are supported out of the box:

- **C11** (generates header-only libraries)
- **HTML** (generates documentation pages) (experimental support)

The following languages are currently on the roadmap:

- **Python** (already supported in [Pycyphal](#), pending [transplantation into Nunavut](#))
- **C++ 14 and newer** (generates header-only libraries; [work-in-progress](#))

Nunavut is named after the [Canadian territory](#). We chose the name because it is a beautiful word to say and read.

## 7.1 Installation

Nunavut depends on PyDSDL.

Install from PIP:

```
pip install -U nunavut
```

## 7.2 Examples

The examples do not replace the documentation, please do endeavor to read it.

### 7.2.1 Generate C headers using the command-line tool

This example assumes that the public regulated namespace directories `reg` and `uavcan` reside under `public_regulated_data_types/`. Nunavut is invoked to generate code for the former.

```
nnvg --target-language c --target-endianness=little --enable-serialization-asserts_
↪public_regulated_data_types/reg --lookup-dir public_regulated_data_types/uavcan
```

### 7.2.2 Generate HTML documentation pages using the command-line tool

See above assumptions. The below commands generate documentation for the `reg` namespace. Note that we have to generate documentation for the `uavcan` namespace as well, because there are types in `reg` that will link to `uavcan` documentation sections.

```
nnvg --experimental-languages --target-language html public_regulated_data_types/reg -
↪-lookup-dir public_regulated_data_types/uavcan
nnvg --experimental-languages --target-language html public_regulated_data_types/
↪uavcan
```

### 7.2.3 Use custom templates

Partial example: generating a C struct

```
/*
 * Cyphal data structure definition
 *
 * Auto-generated, do not edit.
 *
 * Source file: {{T.source_file_path.as_posix()}}
 */

#ifndef {{T.full_name | ln.c.macrofy}}
#define {{T.full_name | ln.c.macrofy}}

{%- for constant in T.constants %}
#define {{ T | ln.c.macrofy }}_{{ constant.name | ln.c.macrofy }} {{ constant |_
↪constant_value }}
{%- endfor %}
```

(continues on next page)

(continued from previous page)

```

typedef struct
{
    /*
     * Note that we're not handling union types properly in this simplified_
     * example. Unions take a bit more logic to generate correctly.
     */
    {%- for field in T.fields_except_padding %}
    {{ field.data_type | declaration }} {{ field | id }}
    {%- if field.data_type is ArrayType -%}
        [{{ field.data_type.capacity }}]
    {%- endif -%}
    {%- if field is VariableLengthArrayType %}
    {{ typename_unsigned_length }} {{ field | id }}_length;
    {%- endif -%}
    {%- endfor %}
}

) {{ T | full_reference_name }};

#endif // {{ T.full_name | ln.c.macrofy }}

```

## 7.2.4 More examples

Where to find more examples to get started:

1. See built-in templates under `nunavut.lang.LANGUAGE.templates`.
2. API usage examples can be found in the [Pycyphal](#) library.

## 7.3 Bundled third-party software

Nunavut embeds the following third-party software libraries into its source (i.e. these are not dependencies and do not need to be installed):

- [Jinja2](#) by Armin Ronacher and contributors, BSD 3-clause license.
- [markupsafe](#) by Armin Ronacher and contributors, BSD 3-clause license (needed for Jinja).

## 7.4 Documentation

The documentation for Nunavut is hosted on [readthedocs.io](#):

- [nunavut](#) - The python library provided by this project.
- [nnyg](#) - Command-line script for using `nunavut` directly or as part of a build system.
- [nunavut template guide](#) - Documentation for authors of nunavut templates.
- [nunavut contributors guide](#) - Documentation for contributors to the Nunavut project.
- [nunavut licenses](#) - Licenses and copyrights

Nunavut is part of the [OpenCyphal](#) project:

- [OpenCyphal website](#)
- [OpenCyphal forum](#)

**n**

nunavut, 3



**A**

AbstractGenerator (*class in nunavut*), 4  
 add\_section() (*nunavut.LanguageConfig method*),  
 10

**B**

build\_namespace\_tree() (*in module nunavut*), 5

**C**

CodeGenerator (*class in nunavut*), 5  
 create() (*nunavut.LanguageContextBuilder method*),  
 19

**D**

default\_filter\_id\_for\_target() (*nunavut.Language class method*), 12  
 DEFAULT\_TARGET\_LANGUAGE (*nunavut.LanguageContextBuilder attribute*),  
 16  
 DSDLCodeGenerator (*class in nunavut*), 6

**E**

enable\_stropping (*nunavut.Language attribute*), 13  
 extension (*nunavut.Language attribute*), 12

**F**

filter\_alignment\_prefix() (*nunavut.DSDLCodeGenerator static method*),  
 8  
 filter\_bit\_length\_set() (*nunavut.DSDLCodeGenerator static method*),  
 8  
 filter\_bits2bytes\_ceil() (*nunavut.DSDLCodeGenerator static method*),  
 8  
 filter\_id() (*nunavut.Language method*), 13  
 filter\_id\_for\_target() (*nunavut.LanguageContext method*), 16

filter\_remove\_blank\_lines() (*nunavut.DSDLCodeGenerator static method*),  
 8  
 filter\_short\_reference\_name() (*nunavut.Language method*), 13  
 filter\_type\_to\_include\_path() (*nunavut.DSDLCodeGenerator method*),  
 7  
 filter\_type\_to\_template() (*nunavut.DSDLCodeGenerator method*),  
 7  
 filter\_typename() (*nunavut.DSDLCodeGenerator static method*), 7  
 filter\_yamlfy() (*nunavut.DSDLCodeGenerator static method*), 6  
 find\_output\_path\_for\_type() (*nunavut.Namespace method*), 20

**G**

generate\_all() (*nunavut.AbstractGenerator method*), 4  
 generate\_all() (*nunavut.DSDLCodeGenerator method*), 9  
 generate\_all() (*nunavut.SupportGenerator method*), 20  
 generate\_namespace\_types (*nunavut.AbstractGenerator attribute*), 4  
 generate\_types() (*in module nunavut*), 9  
 get\_all\_datatypes() (*nunavut.Namespace method*), 20  
 get\_all\_namespaces() (*nunavut.Namespace method*), 20  
 get\_all\_types() (*nunavut.Namespace method*), 20  
 get\_config\_value() (*nunavut.Language method*),  
 14  
 get\_config\_value() (*nunavut.LanguageConfig method*), 11  
 get\_config\_value\_as\_bool() (*nunavut.Language method*), 14

[get\\_config\\_value\\_as\\_bool\(\)](#) (*nunavut.LanguageConfig* method), 11  
[get\\_config\\_value\\_as\\_dict\(\)](#) (*nunavut.Language* method), 14  
[get\\_config\\_value\\_as\\_dict\(\)](#) (*nunavut.LanguageConfig* method), 11  
[get\\_config\\_value\\_as\\_list\(\)](#) (*nunavut.Language* method), 14  
[get\\_config\\_value\\_as\\_list\(\)](#) (*nunavut.LanguageConfig* method), 12  
[get\\_globals\(\)](#) (*nunavut.Language* method), 15  
[get\\_includes\(\)](#) (*nunavut.Language* method), 13  
[get\\_language\(\)](#) (*nunavut.LanguageContext* method), 15  
[get\\_language\\_context\(\)](#) (*nunavut.Namespace* method), 19  
[get\\_nested\\_namespaces\(\)](#) (*nunavut.Namespace* method), 20  
[get\\_nested\\_types\(\)](#) (*nunavut.Namespace* method), 20  
[get\\_option\(\)](#) (*nunavut.Language* method), 15  
[get\\_options\(\)](#) (*nunavut.Language* method), 15  
[get\\_root\\_namespace\(\)](#) (*nunavut.Namespace* method), 20  
[get\\_support\\_files\(\)](#) (*nunavut.Language* method), 15  
[get\\_support\\_module\(\)](#) (*nunavut.Language* method), 13  
[get\\_support\\_output\\_folder\(\)](#) (*nunavut.Namespace* method), 19  
[get\\_supported\\_language\\_names\(\)](#) (*nunavut.LanguageContextBuilder* method), 16  
[get\\_supported\\_languages\(\)](#) (*nunavut.LanguageContext* method), 16  
[get\\_target\\_language\(\)](#) (*nunavut.LanguageContext* method), 16  
[get\\_templates\(\)](#) (*nunavut.AbstractGenerator* method), 4  
[get\\_templates\(\)](#) (*nunavut.CodeGenerator* method), 6  
[get\\_templates\(\)](#) (*nunavut.SupportGenerator* method), 20  
[get\\_templates\\_package\\_name\(\)](#) (*nunavut.Language* method), 15

## H

[has\\_standard\\_namespace\\_files](#) (*nunavut.Language* attribute), 13

## I

[InternalError](#), 21  
[is\\_deprecated\(\)](#) (*nunavut.DSDLCodeGenerator* static method), 9

[is\\_None\(\)](#) (*nunavut.DSDLCodeGenerator* static method), 8  
[is\\_saturated\(\)](#) (*nunavut.DSDLCodeGenerator* static method), 8  
[is\\_service\\_request\(\)](#) (*nunavut.DSDLCodeGenerator* static method), 8  
[is\\_service\\_response\(\)](#) (*nunavut.DSDLCodeGenerator* static method), 8

## L

[Language](#) (class in *nunavut*), 12  
[LanguageConfig](#) (class in *nunavut*), 10  
[LanguageContext](#) (class in *nunavut*), 15  
[LanguageContextBuilder](#) (class in *nunavut*), 16

## N

[name](#) (*nunavut.Language* attribute), 12  
[named\\_types](#) (*nunavut.Language* attribute), 13  
[named\\_values](#) (*nunavut.Language* attribute), 13  
[Namespace](#) (class in *nunavut*), 19  
[namespace](#) (*nunavut.AbstractGenerator* attribute), 4  
[namespace\\_output\\_stem](#) (*nunavut.Language* attribute), 12  
[nunavut](#) (module), 3

## O

[output\\_folder](#) (*nunavut.Namespace* attribute), 19

## S

[SECTION\\_NAME\\_PATTERN](#) (*nunavut.LanguageConfig* attribute), 10  
[set\\_additional\\_config\\_files\(\)](#) (*nunavut.LanguageContextBuilder* method), 18  
[set\\_target\\_language\(\)](#) (*nunavut.LanguageContextBuilder* method), 18  
[set\\_target\\_language\\_configuration\\_override\(\)](#) (*nunavut.LanguageContextBuilder* method), 17  
[set\\_target\\_language\\_extension\(\)](#) (*nunavut.LanguageContextBuilder* method), 18  
[stable\\_support](#) (*nunavut.Language* attribute), 13  
[support\\_namespace](#) (*nunavut.Language* attribute), 13  
[SupportGenerator](#) (class in *nunavut*), 20

## U

[UnsupportedLanguageError](#), 21  
[update\(\)](#) (*nunavut.LanguageConfig* method), 10

## Y

[YesNoDefault](#) (class in *nunavut*), 21