ELLIPTIc Documentation

Release 1.0.0

Guilherme Caminha

Apr 18, 2018

Contents:

ELLIPTIc, The ExtensibLe LIbrary for Physical simulaTIons, is a library / framework for creating reusable and extensible Domain Specific Languages (DSL) for scientific purposes.

CHAPTER 1

Tutorials

1.1 Simple DSL Tutorial

In this tutorial you will learn how to create a `DSL Contract` and a `DSL Implementation`.

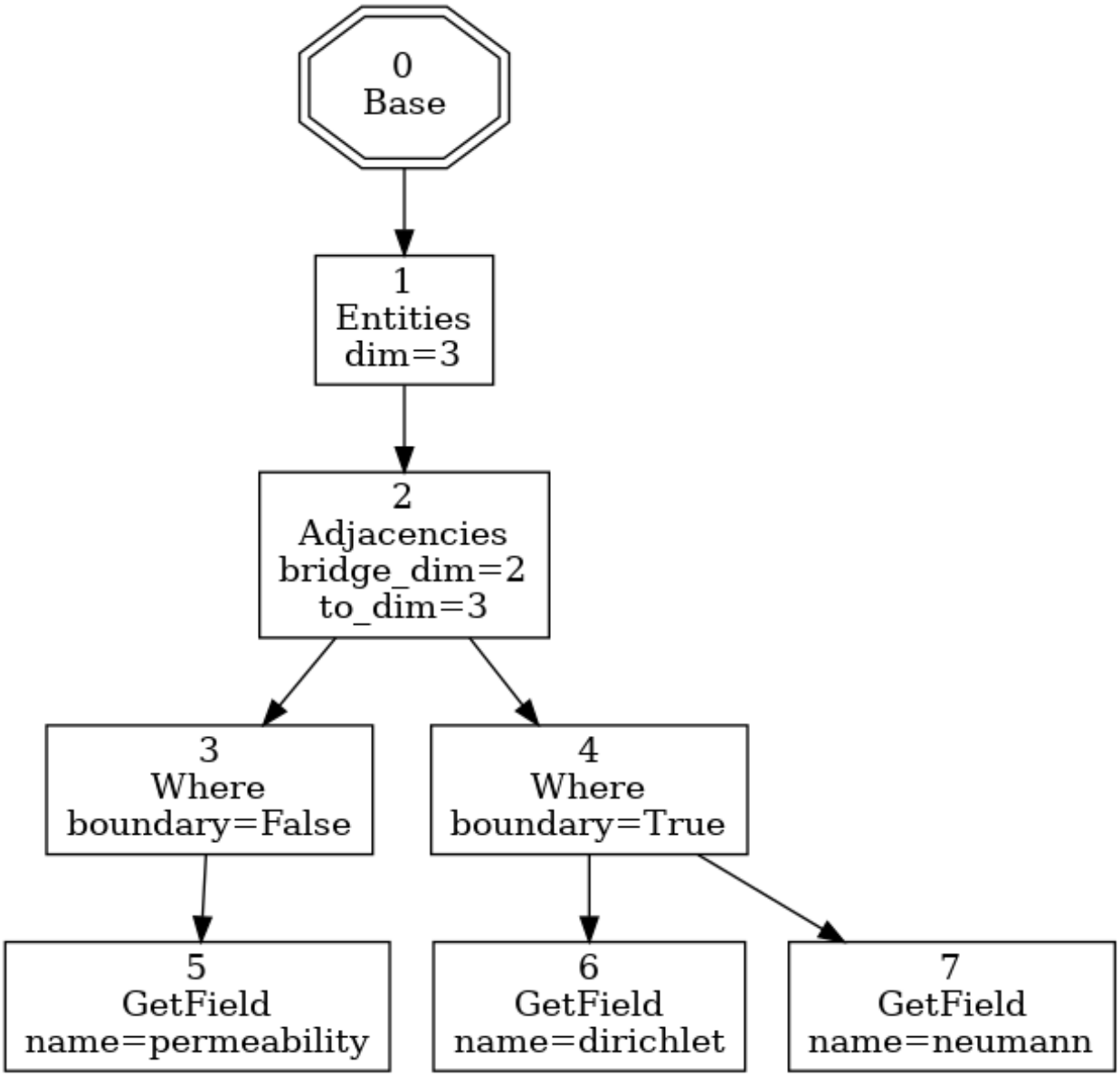
Using the created DSL will also be covered by this tutorial.

The full example can be found at the *examples/DSL_Example* folder.

1.1.1 Defining the DSL Contract

The first step is to define the DSL design by creating a class that inherits from `DSLContract`. This class will define the available operations for the DSL.

Each method in the DSL contract should call the method `append_tree()` with an `Expression` object. *Expression* objects are the nodes that form the DSL intermediate representation. The *append_tree* method will deal with modifying the DSL intermediate representation by adding the new *Expression* object accordingly. In other words, in the picture below, which shows an example of a DSL intermediate representation, each node is an *Expression* object representation.



To create an Expression object, you must pass a *Context Delegate* class (not an object!), plus two arguments representing the name of the node and its parameters, respectively.

For example, to create an *Expression* object that will render a node similar to the *Adjacencies* node shown above, you could instantiate it this way:

```
Expression(context_delegate, "Adjacencies", {"bridge_dim": 2, "to_dim": 3})
```

So, let's create a DSL contract for a simple vectorized processor. That DSL should be able to execute the following expression:

```
Range(start=0, count=100).ScalarSum(5).ScalarMult(2).Sum()
```

Where the *Range(start=0, count=100)* operation would generate 100 numbers starting from 0, the *ScalarSum(5)* operation would sum 5 to each generated number, *ScalarMult(2)* would multiply the result by 2, and *Sum()* would accumulate the results of the previous operations.

A pure python version of the above algorithm would be:

```
sum(((i+5)*2 for i in range(0, 100)))
```

This version is not as readable and expressive as the above ELLIPTIc expression, and it is also not as fast as the code that will be generated, as will be shown in the end of this tutorial.

Together with the DSL Contract, a *DSL Implementation* should be given. The DSL Implementation defines the expected *ContextDelegate* generating methods. Those methods should return a *Context Delegate* class.

We can begin by defining the abstract DSL Implementation:

```
class VectorImplementationBase(DSLImplementation):

    @abstractmethod
    def range_delegate(self, start, count):
        raise NotImplementedError

    @abstractmethod
    def scalar_mult_delegate(self, scalar):
        raise NotImplementedError

    @abstractmethod
    def scalar_sum_delegate(self, scalar):
        raise NotImplementedError

    @abstractmethod
    def sum_delegate(self):
        raise NotImplementedError
```

With the *@abstractmethod* decorators, we are telling Python ELLIPTIc that this class does not define a concrete DSL Implementation, but actually what a concrete DSL Implementation should have to conform to the DSL contract we will create. This way, it is possible to create the DSL Contract separated from the actual implementation, and therefore, to have several possible implementations to the same contract.

This characteristic allows for high decoupling between DSL contracts and DSL implementations. It is therefore possible to have an implementation for our DSL contract that would use, for example, a specialized third-party library to perform specific computations. An algorithm built with the DSL contract would not need to know the underlying implementation, and should yield the same results with any chosen implementation, given that the implementation is correct.

We can now create the DSL contract for our vectorized processor:

```
class VectorContract(DSLContract[VectorImplementationBase]):

    def Range(self, start, count):
        return self.append_tree(Expression(self.dsl_impl.range_delegate(start, count),
↪ "Range"))

    def ScalarMult(self, scalar):
        return self.append_tree(Expression(self.dsl_impl.scalar_mult_delegate(scalar),
↪ "ScalarMult"))

    def ScalarSum(self, scalar):
        return self.append_tree(Expression(self.dsl_impl.scalar_sum_delegate(scalar),
↪"ScalarSum"))

    def Sum(self):
        return self.append_tree(Expression(self.dsl_impl.sum_delegate(), "Sum"))
```

Here we are inheriting from *DSLContract*. The brackets in *DSLContract[VectorImplementationBase]* are telling ELLIPTIc that this DSL contract expects a DSL implementation that inherits from *VectorImplementationBase*.

Each method in this class is defining an operation for the DSL. As explained before, those methods must call *append_tree* with an *Expression* object. The *append_tree* method will create return an object of the *VectorContract* class, allowing for method chaining.

1.1.2 Creating the DSL Implementation

The next step is to define the DSL implementation. While the DSL contract creation step requires design planning to support important features for the language, this is the most involved step in the process of creating a DSL with ELLIPTIc. The DSL implementation defines how the Cython code will be generated.

First we must define the class that will inherit from *VectorImplementationBase*:

```
class VectorImplementation(VectorImplementationBase):
    ...
```

Let's begin by defining *base_delegate*, which should be responsible for creating and initializing variables that will be used:

```
def base_delegate(self):
    class BaseDelegate(ContextDelegate):

        def get_template_file(self):
            return 'base.pyx.etp'

        def template_kwargs(self):
            return {'declare_variables': self.context.context['declare_variable'],
                    'return_variable': self.context.get_value('return_variable')}

        def context_enter(self):
            pass

        def context_exit(self):
            pass

    return BaseDelegate
```

Notice that this method returns a *ContextDelegate* called *BaseDelegate*. Every context delegate must implement the methods shown above. The first method, *get_template_file*, tells ELLIPTIc where to look for the template file containing the Cython template code (we will get to that soon). The *template_kwargs* method tells ELLIPTIc which arguments should be passed to the template. *context_enter* and *context_exit* modify the context when the node is visited and left in the intermediate representation tree.

The template files are jinja2 templates. The base.pyx.etp template is shown below:

```
from libcpp.list cimport list as cpplist

def run():
    cdef cpplist[unsigned long int] arr

    {% for (var_type, var_name, initial_value) in declare_variables %}
    cdef {{ var_type }} {{ var_name }} = {{ initial_value }}
    {% endfor %}

    {{ child|indent }}
```

(continues on next page)

(continued from previous page)

```
    return {{ return_variable }}
```

Notice that this code has several template constructs such as *{{ var_type }}* and *% for (var_type, var_name, initial_value) in declare_variables %}*. This allows for high flexibility when designing the DSL implementation, as each node in the intermediate representation can communicate with each other through the `Context` object.

For example, the *BaseDelegate* class shown above will gather all variables that should be declared by accessing *self.context.context['declare_variable']*, and will also gather the variable that holds the value that should be returned by accessing *self.context.get_value('return_variable')*.

The *{{ child\|indent }}* is necessary to render the code corresponding to the operations that happen afterwards.

The context object is basically a dictionary of stacks. In other words, it defines a *stack_name -> stack* mapping.

Let's now define a more complicated delegate:

```
def range_delegate(self, start, count):
    start = str(start)
    count = str(count)

    class RangeDelegate(ContextDelegate):

        def get_template_file(self):
            return 'range.pyx.etp'

        def template_kwargs(self):
            return {'count': count,
                    'index': self.context.get_value('current_index_name'),
                    'variable': self.context.get_value('current_variable_name'),
                    'counter': self.context.get_value('current_counter_name')}

        def context_enter(self):
            var_type = 'unsigned long int'
            loop_name = 'range' + str(self.unique_id)

            self.context.put_value('declare_variable', (var_type,
                                                        loop_name + 'var',
                                                        '0'))
            self.context.put_value('current_variable_name', loop_name + 'var')

            self.context.put_value('declare_variable', (var_type,
                                                        loop_name + 'counter',
                                                        start))
            self.context.put_value('current_counter_name', loop_name + 'counter')

            self.context.put_value('declare_variable', (var_type,
                                                        loop_name + 'index',
                                                        '0'))
            self.context.put_value('current_index_name', loop_name + 'index')

        def context_exit(self):
            self.context.pop_value('current_variable_name')
            self.context.pop_value('current_counter_name')
            self.context.pop_value('current_index_name')

    return RangeDelegate
```

The range delegate defines several values in the context. Each of those values will be available to the next nodes in the intermediate representation tree. They will also be available to the range delegate itself when it is going to be rendered.

In this case, the *context_enter* method is putting several values in the *'declare_variable'* stack. This stack will be used by the base delegate to declare and initialize variables. Notice that the *context_enter* method is repeating the values it is putting in the *'declare_variable'* into other stacks. This is because since it is necessary for these values to still be available when the base template is rendered.

Therefore, the *'declare_variable'* stacked values are not removed when *context_exit* is called.

Since the *Range* operation consists of generating several values that will be processed, the corresponding template looks like:

```
while {{ index }} < {{ count }}:
    {{ variable }} = {{ counter }}

    {{ child|indent }}

    {{ counter }} += 1
    {{ index }} += 1
```

The remaining delegates are much simpler. For example, the delegate for scalar multiplication looks like:

```
def scalar_mult_delegate(self, scalar):
    scalar = str(scalar)

    class ScalarMulDelegate(ContextDelegate):
        def get_template_file(self):
            return 'scalarmult.pyx.etp'

        def template_kwargs(self):
            return {'scalar': scalar,
                    'variable': self.context.get_value('current_variable_name')}

        def context_enter(self):
            pass

        def context_exit(self):
            pass

    return ScalarMulDelegate
```

This operation will basically modify the current variable, whose name is defined in the context by the *'current_variable_name'* stack. The range delegate defines this stack and the variable name. The template for the scalar multiplication is:

```
{{ variable }} = {{ variable }} * {{ scalar }}

{{ child }}
```

The scalar sum delegate is very similar:

```
def scalar_sum_delegate(self, scalar: int) -> Type[ContextDelegate]:
    scalar = str(scalar)

    class ScalarSumDelegate(ContextDelegate):
        def get_template_file(self) -> str:
            return 'scalarsum.pyx.etp'
```

(continues on next page)

(continued from previous page)

```
        def template_kwargs(self) -> Dict[str, Any]:
            return {'scalar': scalar,
                    'variable': self.context.get_value('current_variable_name')}

        def context_enter(self) -> None:
            pass

        def context_exit(self) -> None:
            pass

    return ScalarSumDelegate
```

And its template is also similar:

```
{{ variable }} = {{ variable }} + {{ scalar }}

{{ child }}
```

The remaining delegate is the sum delegate. It will tell the base delegate to create an accumulation variable, and to return this variable as the result of the computation, using the *'declare_variable'* and *'return_variable'* stacks:

```
def sum_delegate(self):

    class SumDelegate(ContextDelegate):
        def get_template_file(self):
            return 'sum.pyx.etp'

        def template_kwargs(self):
            return {'variable': self.context.get_value('current_variable_name'),
                    'acc_variable': self.context.get_value('acc_variable_name')}

        def context_enter(self):
            self.context.put_value('declare_variable', ('int',
                                                        'acc' + str(self.unique_id),
                                                        '0'))
            self.context.put_value('acc_variable_name', 'acc' + str(self.unique_id))
            self.context.put_value('return_variable', 'acc' + str(self.unique_id))

        def context_exit(self):
            self.context.pop_value('acc_variable_name')

    return SumDelegate
```

The template code will simply accumulate the current variable value into the accumulating variable:

```
{{ acc_variable }} = {{ acc_variable }} + {{ variable }}

{{ child }}
```

1.1.3 Finishing the DSL Implementation

To finish the DSL implementation we must provide a `TemplateManager` and a *DSLMeta*. The template manager class is responsible for telling ELLIPTIc where to look for the template files, and the DSL Meta tells ELLIPTIc if the

DSL implementation has any dependencies, such as include files and libraries that should be linked during the Cython compilation.

In our case, we are using plain Cython, so the DSL Meta class will be simple. Our template manager will simply tell ELLIPTIc to look for templates in the *Templates* folder.

```
class VectorTemplateManager(TemplateManagerBase):

    def __init__(self) -> None:
        super().__init__(__package__, 'Templates')

class VectorMeta(DSLMeta):

    def include_dirs(self) -> List[str]:
        return []

    def libs(self) -> List[str]:
        return []
```

1.1.4 Running the Example

We can now create the DSL object. The *DSL class* takes a template manager, a contract and a DSL meta object as arguments to its constructor. You must also pass the DSL implementation when creating the contract object.

```
dsl = DSL(VectorTemplateManager(),
          VectorContract(VectorImplementation()),
          VectorMeta())
```

To use the DSL operations, your code must be within a context manager created by the *root* method from the DSL object. Let's use the DSL to solve the same problem we defined in the beginning of the tutorial:

```
with dsl.root() as root:
    ents = root.Range(start=0, count=100).ScalarSum(5).ScalarMult(2).Sum()
```

This might take some time to compile, but after the compilation step is done you can reuse the resulting compiled module freely:

```
print(dsl.get_built_module().run())
```

The *run()* function was defined in the base template.

Comparing the running times for the ELLIPTIc version and the pure Python version I obtained the following results when running on my machine:

```
# Execution time for the elliptic version:
t0 = time.time()
for i in range(0, 50000):
    dsl.get_built_module().run()
print(time.time() - t0)  # 0.11810016632080078

# Execution time for the pure
t0 = time.time()
for i in range(0, 50000):
    sum(((i + 5) * 2 for i in range(0, 100)))
print(time.time() - t0)  # 3.5026133060455322
```

Which indicates that the ELLIPTIc version is around 30x faster than the pure Python version.

The final generated Cython code is shown below:

```
from libcpp.list cimport list as cpplist

def run():
    cdef cpplist[unsigned long int] arr


    cdef unsigned long int range1var = 0

    cdef unsigned long int range1counter = 0

    cdef unsigned long int range1index = 0

    cdef int acc4 = 0


    while range1index < 100:
        range1var = range1counter

        range1var = range1var + 5

        range1var = range1var * 2

        acc4 = acc4 + range1var

        range1counter += 1
        range1index += 1

return acc4
```

CHAPTER 2

Packages

2.1 elliptic.Kernel package

2.1.1 elliptic.Kernel.Context Module

class elliptic.Kernel.Context.**Context**
Bases: object

Defines a context for code generation.

A Context is basically a dictionary of stacks. In other words, it defines a *stack_name* -> *stack* mapping. Each *stack* has is semantically defined by its *stack_name*.

Example

```
>>> context = Context()
>>> context.put_value('current_value', '100')
>>> context.put_value('current_value', '200')
>>> context.get_value('current_value')  # 200
>>> context.pop_value('current_value')
>>> context.get_value('current_value')  # 100
```

clear_values(*name*)
Clears the stack named *name*.

Parameters **name** (str) – Stack name.

Return type None

get_value(*name*)
Gets the front value of the stack named *name*.

Parameters **name** (str) – Stack name.

Return type str

pop_value(*name*)
Pops the front value of the stack named *name*.

Parameters **name** (str) – Stack name.

Return type None

put_value(*name*, *value*)
Pushes the value *value* to a stack named *name*.

Parameters

- **name** (str) – Stack name.
- **value** (Union[str, Iterable[str]]) – Value to be pushed into the stack.

Return type None

class elliptic.Kernel.Context.**ContextDelegate**(*context*, *unique_id*)
Bases: abc.ABC

Delegate class for getting the generated code template file and its kwargs for a given expression.

Also defines the context state changes when the corresponding expression node is visited and exited.

context
Context instance.

unique_id
A unique id that can be used to identify values that were created from this context delegate.

Parameters

- **context** (*Context*) – Context instance.
- **unique_id** (int) – A unique id.

context_enter()
Modifies the context state. Called when the expression node is visited.

Use this method to prepare the context for expressions that will be visited afterwards. It is preferable to keep most *context.put_value* calls in this method.

Return type None

context_exit()
Modifies the context state. Called when the expression node is exited.

Use this method to clear values from the context and prepare it for the expression nodes that were visited before. It is preferable to keep most *context.pop_value* calls in this method.

Return type None

get_template_file()
Returns the template file containing the generated code for the expression.

Return type str

template_kwargs()
Returns the arguments (a dictionary) that will be passed to the template.

Return type Dict[str, Any]

exception elliptic.Kernel.Context.**ContextException**
Bases: Exception

Exception raised when an error related to a Context operation occurs.

2.1.2 elliptic.Kernel.Contract module

class elliptic.Kernel.Contract.**DSLContract**(*dsl_impl*, *expr=None*)
Bases: typing.Generic

Defines the contract of the DSL.

The DSL contract is the set of operations that the DSL supports, plus a base operation. Each operation should create an *Expression* instance and call *append_tree* with the created expression.

Parameters

- **dsl_impl** (*~DSLImplementationSubclass*) – DSL Implementation object.
- **expr** (Optional[*Expression*]) – Current DSL operation. If set to None, will be set to the next operation done when calling *append_tree*.

dsl_impl
DSL Implementation object.

expr
Current DSL operation.

Base()
Base operation. Called by default from ELLIPTIc before any other operation.

Return type ~DSLContractSubclass

append_tree(*expr*)
Inserts the given expression to the expression tree and returns a new DSLContract bounded to the given expression.

Return type ~DSLContractSubclass

class elliptic.Kernel.Contract.**DSLImplementation**
Bases: abc.ABC

Abstract class that defines a DSL Implementation. Should be used to connect the corresponding *Context Delegates*.

base_delegate()
Returns the context delegate for the DSL base context handling.

The base context delegate could be used, for example, to declare variables and initialize dependencies for the generated code.

Return type Type[*ContextDelegate*]

2.1.3 elliptic.Kernel.DSL module

class elliptic.Kernel.DSL.**DSL**(*template_manager*, *dsl_contract*, *dsl_meta*)
Bases: object

Defines the interface for interacting with a DSL.

Parameters

- **template_manager** (*TemplateManagerBase*) – A *TemplateManagerBase* subinstance.

- **dsl_contract** (*DSLContract*[~DSLImplementationSubclass]) – A DSL Contract.
- **dsl_meta** (*DSLMeta*) – A *DSLMeta* instance.

get_built_module()
Returns the compiled module that holds the generated code.

Return type module

root()
Entry point for building expressions.

Should be used as a context manager, using the *with* statement.

Return type Iterator[*DSLContract*[~DSLImplementationSubclass]]

exception elliptic.Kernel.DSL.**DSLBuildError**
Bases: *elliptic.Kernel.DSL.DSLException*

Exception raised when an error related to a DSL build process happens.

exception elliptic.Kernel.DSL.**DSLException**
Bases: Exception

class elliptic.Kernel.DSL.**DSLMeta**
Bases: abc.ABC

Class which stores information regarding the DSL compilation.

include_dirs()
Returns the list of include directories that should be used when compiling.

Cypyler adds the numpy includes by default. Any extra include paths should be returned here.

Example

['/usr/local/include/moab']

Return type List[str]

libs()
Returns the list of libraries that should be linked against.

Example

['MOAB', 'Trilinos']

Return type List[str]

2.1.4 elliptic.Kernel.Expression module

class elliptic.Kernel.Expression.**EllipticNode**
Bases: anytree.node.nodemixin.NodeMixin

Base class for representing a node in the DSL tree.

export_tree(*filename*)
Exports a graphical representation of the DSL tree.

This method can be called from any tree node. The tree root will always be used.

Parameters **filename** (str) – Name for the exported image file.

Return type None

class elliptic.Kernel.Expression.**Expression**(*context_delegate*, *display_name=''*, *display_args=None*)

Bases: *elliptic.Kernel.Expression.EllipticNode*

Base class for building DSL expressions.

Parameters

- **context_delegate** (Type[*ContextDelegate*]) – A context delegate object responsible for generating some Cython code.
- **display_name** – The name that will be displayed when this Expression is rendered into a picture.
- **display_args** – The arguments that will be displayed below this Expression name in the rendered picture.

render(*template_manager*, *child*, *context_delegate*)

Render the expression generated code.

Parameters

- **template_manager** – A *TemplateManagerBase* instance.
- **child** (str) – The rendered template corresponding to this node's child.
- **context_delegate** (*ContextDelegate*) – The context delegate for the DSL.

Return type str

visit(*context*)

Context manager used when an expression node is visited in the DSL tree.

Calls the *context delegate context_enter()* and *context_exit()* methods.

Parameters **context** (*Context*) – A context object.

Return type Iterator[*ContextDelegate*]

2.1.5 elliptic.Kernel.TemplateManager module

class elliptic.Kernel.TemplateManager.**TemplateManagerBase**(*package*, *templates_folder*)

Bases: object

Class for abstracting interactions with the *jinja2* templating package.

Example

Creating a template manager that uses the *Templates* folder found inside the current package when searching for template files.

```
>>> class MyTemplateManager(TemplateManagerBase):
>>>     def __init__(self) -> None:
>>>         super().__init__(__package__, 'Templates')
```

Parameters

- **package** (str) – The path to a python package containing the templates folder.
- **templates_folder** (str) – The name of the templates folder.

get_template(*template_file*)
Returns the template object corresponding to the given template file.

Parameters **template_file** (str) – The template file.

Return type Template

render(*template_file*, ***kwargs*)
Returns the rendered template, passing in *kwargs* as template arguments.

Parameters

- **template_file** (str) – The template file.
- **kwargs** (str) – The arguments to be passed to the template.

Return type str

2.1.6 elliptic.Kernel.TreeBuilder module

class elliptic.Kernel.TreeBuilder.**TreeBuild**(*template_manager*, *libraries=None*, *include_dirs=None*)
Bases: object

Class responsible for processing a DSL tree and consolidating the generated Cython code.

Parameters

- **template_manager** (*TemplateManagerBase*) – A template manager object.
- **dsl_contract** – A dsl contract object.
- **libraries** (Optional[List[str]]) – A list containing any libraries that should be statically linked.
- **include_dirs** (Optional[List[str]]) – A list containing extra include directories. *Cypyler* adds numpy includes by default.

build(*root*)
Processes the DSL tree and returns the built Cython module.

Parameters **root** (*Expression*) – The DSL tree root.

Return type module

- search

Python Module Index

e

Index

T

U

V