taref_docs Documentation

Release 1.0

Thomas Aref

February 11, 2016

CONTENTS

Contents:

CHAPTER

ONE

WELCOME TO TAREF'S DOCUMENTATION!

Contents:

1.1 Getting Started

The taref package strives to make quick, easy-to-use, auto-display GUIs with the option to extend them to custom GUIs at a later date. To do this, taref is built on enaml, a programming language extension to python and framework for creating professional user interfaces, and Atom, a framework for creating memory efficient Python objects with enhanced features such as dynamic initialization, validation, and change notification for object attributes (similar in behavior to Enthought's Traits)

Basically, taref generates dynamic enaml templates from a minimal information Atom class with the option to later substitute these dynamic enaml templates

with enaml written specifically for the class.

For example, some code that makes use of taref's shower function might look like this:

```
from atom.api import Atom, Float, Unicode
from taref.core.shower import shower

class Test(Atom):
    a=Float()
    b=Unicode()

t=Test()
shower(t)
```

and these few lines of code are all that is needed to produce a simple GUI that shows a and b in our Test object t!

So what is happening? First, we are an Atom class. Atom class are very similar to python's regular classes. Something equivalent to our Test class above would be:

```
class Test(object):
    def __init__(self, a=0.0, b=""):
        self.a=a
        self.b=b
```

However, Atom provides some key advantages to using the above class for GUI making First, in the Atom class, the type of a is fixed to being a float so the GUI always knows how to display it. The members of Test are likewise fixed so that none are added dynamically later. Secondly, Atom can detect changes changes to it's members. To see this, we look at the following code:

```
class Test(Atom):
    a=Float()
    b=Unicode()

    def _observe_a(self, change):
        print change
```

Now every time variable a is changed, in the GUI or in code, it will print out that change. The final advantage of Atom is that metadata can be added to the variable. For example,

```
t.a=4.0
t.get_member("a").tag(label="My Float")
print t.a
print t.get_member("a").metadata
```

Combining this with the functionality with the shower function:

```
from atom.api import Atom, Float, Unicode
from taref.core.shower import shower

class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode()

    def _observe_a(self, change):
        print change

t=Test()
shower(t)
```

auto creates a GUI where a is now labelled "My Float" and every time a is changed it is printed.

There are a number of custom tags defined in taref, such as "label", to give easy access to some commonly used features. For example, suppose I wanted b to display as a multiline field rather than a single line field:

```
class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode().tag(spec="multiline")
```

In this case the spec tag allows quick access to a multiline field display. Now suppose I want full control over the window that Test objects reside in using the full power of enaml. I start an enaml file, "test_e.enaml" that looks like this:

```
from enaml.widgets.api import MainWindow, Field, Label, HGroup

enamldef TestWindow(MainWindow):
    attr test
    HGroup:
        Label:
            text << unicode(test.a)
        Field:
            text := b
```

In my python file, "test.py", I add the necessary pieces:

```
from atom.api import Atom, Float, Unicode, cached_property
from taref.core.shower import shower
from enaml import imports
with imports():
    from test_e import TestWindow
```

```
class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode()

    def _observe_a(self, change):
        print change

    @cached_property
    def view_window(self):
        return TestWindow(test=self)

t=Test()
shower(t)
```

and now I have replaced the default dynamic view of Test with a custom one, while still keeping it compatible with the rest of taref's framework!

CHAPTER

TWO

INTRODUCTION:

2.1 Other header

How does this work:

```
recognize code
print "yo"
```

Maybe

can I see this

CHAPTER

THREE

LET'S TRY LISTS:

- item 1
- item 2
- item 3

1. item 1
2. item 2
3. item 3

CHAPTER

FOUR

LET'S TRY TABLES:

italics, **bold**, or `monotype`.

Name	age
item1	12
item2	42
item3	3

```
Introduction:
==============

*****************
Other header
*****************

How does this work::

    recognize code
    print "yo"

Maybe


can I see this

.. _reftest:

Let's try lists:
================

*  item 1
* item 2
* item 3

#. item 1
#. item 2
#. item 3

Let's try tables:
=================
*italics*, **bold**, or ``monotype``.

========  ====
Name      age
========  ====
item1     12
```

```
item2     42
item3     3
========  ====

.. literalinclude:: introduction.rst

does a reference work? :ref:`reftest`

.. function:: enumerate(sequence[, start=0])

   Return an iterator that yields tuples of an index and an item of the
   *sequence*. (And so on.)

.. autofunction::taref.core.atom_extension.get_tag
```

does a reference work? *Let's try lists:*

enumerate(*sequence*[, *start=0*])

Return an iterator that yields tuples of an index and an item of the *sequence*. (And so on.)

CHAPTER

FIVE

GETTING STARTED

The taref package strives to make quick, easy-to-use, auto-display GUIs with the option to extend them to custom GUIs at a later date. To do this, taref is built on enaml, a programming language extension to python and framework for creating professional user interfaces, and Atom, a framework for creating memory efficient Python objects with enhanced features such as dynamic initialization, validation, and change notification for object attributes (similar in behavior to Enthought's Traits)

Basically, taref generates dynamic enaml templates from a minimal information Atom class with the option to later substitute these dynamic enaml templates

with enaml written specifically for the class.

For example, some code that makes use of taref's shower function might look like this:

```
from atom.api import Atom, Float, Unicode
from taref.core.shower import shower

class Test(Atom):
    a=Float()
    b=Unicode()

t=Test()
shower(t)
```

and these few lines of code are all that is needed to produce a simple GUI that shows a and b in our Test object t!

So what is happening? First, we are an Atom class. Atom class are very similar to python's regular classes. Something equivalent to our Test class above would be:

```
class Test(object):
    def __init__(self, a=0.0, b=""):
        self.a=a
        self.b=b
```

However, Atom provides some key advantages to using the above class for GUI making First, in the Atom class, the type of a is fixed to being a float so the GUI always knows how to display it. The members of Test are likewise fixed so that none are added dynamically later. Secondly, Atom can detect changes changes to it's members. To see this, we look at the following code:

```
class Test(Atom):
    a=Float()
    b=Unicode()

    def _observe_a(self, change):
        print change
```

Now every time variable a is changed, in the GUI or in code, it will print out that change. The final advantage of Atom is that metadata can be added to the variable. For example,

```
t.a=4.0
t.get_member("a").tag(label="My Float")
print t.a
print t.get_member("a").metadata
```

Combining this with the functionality with the shower function:

```
from atom.api import Atom, Float, Unicode
from taref.core.shower import shower

class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode()

    def _observe_a(self, change):
        print change

t=Test()
shower(t)
```

auto creates a GUI where a is now labelled "My Float" and every time a is changed it is printed.

There are a number of custom tags defined in taref, such as "label", to give easy access to some commonly used features. For example, suppose I wanted b to display as a multiline field rather than a single line field:

```
class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode().tag(spec="multiline")
```

In this case the spec tag allows quick access to a multiline field display. Now suppose I want full control over the window that Test objects reside in using the full power of enaml. I start an enaml file, "test_e.enaml" that looks like this:

```
from enaml.widgets.api import MainWindow, Field, Label, HGroup

enamldef TestWindow(MainWindow):
    attr test
    HGroup:
        Label:
            text << unicode(test.a)
        Field:
            text := b
```

In my python file, "test.py", I add the necessary pieces:

```
from atom.api import Atom, Float, Unicode, cached_property
from taref.core.shower import shower
from enaml import imports
with imports():
    from test_e import TestWindow

class Test(Atom):
    a=Float().tag(label="My Float")
    b=Unicode()

    def _observe_a(self, change):
        print change
```

```
    @cached_property
    def view_window(self):
        return TestWindow(test=self)

t=Test()
shower(t)
```

and now I have replaced the default dynamic view of Test with a custom one, while still keeping it compatible with the rest of taref's framework!

CHAPTER

SIX

TAREF

Contents:

6.1 core

Contents:

6.1.1 atom_extension

Created on Fri Jan 22 19:12:36 2016

@author: thomasaref

A collection of utility functions that extend Atom's functionality, used heavily by taref other modules. To maintain compatibility with Atom classes, these are defined as standalone functions rather than extending the class. Runtime also seem slightly better as standalone functions than as an extended class

taref.core.atom_extension.**call_func**(*obj*, *name*, ***kwargs*)
: calls a func using keyword assignments. If name corresponds to a Property, calls the get func. otherwise, if name_mangled func "_get_"+name exists, calls that. Finally calls just the name if these are not the case

taref.core.atom_extension.**get_all_main_params**(*obj*)
: all members in all_params that are not tagged as sub. Convenience function for more easily custom defining main_params in child classes

taref.core.atom_extension.**get_all_params**(*obj*)
: all members that are not tagged as private, i.e. not in reserved_names and will behave as agents. order of magnitude faster when combine with private_property

taref.core.atom_extension.**get_all_tags**(*obj*, *key*, *key_value=None*, *none_value=None*, *search_list=None*)
: returns a list of names of parameters with a certain key_value Shortcut retrieve members with particular metadata. There are several variants based on inputs.

  - With only obj and key specified, returns all member names who have that key
  - with key_value specified, returns all member names that have that key set to key_value
  - with key_value and none_value specified equal, returns all member names that have that key set to key_value or do not have the tag
  - specifying search list limits the members searched
  - Finally, if key_value is none, returns those members not matching none_value

taref.core.atom_extension.**get_inv**(*obj*, *name*, *value*)
returns the inverse mapped value (meant for an Enum)

taref.core.atom_extension.**get_main_params**(*obj*)
returns main_params if it exists and all possible main params if it does not

taref.core.atom_extension.**get_map**(*obj*, *name*, *value=None*, *reset=False*)
gets the mapped value specified by the property mapping and returns the attribute value if it doesn't exist gets the map of an Enum defined in the property name_mapping or tag mapping. value can be used to get the map for another value besides the Enum's current one.

taref.core.atom_extension.**get_property_names**(*obj*)
returns property names that are in all_params

taref.core.atom_extension.**get_property_values**(*obj*)
returns property values that are in all_params

taref.core.atom_extension.**get_reserved_names**(*obj*)
reserved names not to perform standard logging and display operations on, i.e. members that are tagged as private and will behave as usual Atom members

taref.core.atom_extension.**get_run_params**(*f*, *skip_first=True*)
returns names of parameters a function will call, skips first parameter if skip_first is True

taref.core.atom_extension.**get_tag**(*obj*, *name*, *key*, *none_value=None*)
Shortcut to retrieve metadata from an Atom member which also returns a none_value if the metadata does not exist. This is an easy way to get a tag on a particular member and provide a default if it isn't there.

taref.core.atom_extension.**get_type**(*obj*, *name*)
returns type of member with given name, with possible override via tag typer

class taref.core.atom_extension.**instancemethod**(*obj*, *name=None*)
disposable decorator object for instancemethods defined outside of Atom class

taref.core.atom_extension.**log_func**(*func*, *pname=None*)
logging decorator for Callables that logs call if tag log!=False

taref.core.atom_extension.**lowhigh_check**(*obj*, *name*, *value*)
can specify low and high tags to keep float or int within a range.

taref.core.atom_extension.**private_property**(*fget*)
A decorator which converts a function into a cached Property tagged as private. Improves performance greatly over property!

taref.core.atom_extension.**reset_properties**(*obj*)
resets all properties that are in all_params

taref.core.atom_extension.**set_all_tags**(*obj*, ***kwargs*)
Shortcut to use Atom's tag functionality to set metadata on members not marked private, i.e. all_params. This is an easy way to set the same tag on all params

taref.core.atom_extension.**set_attr**(*self*, *name*, *value*, ***kwargs*)
utility function for setting tags while setting value

taref.core.atom_extension.**set_log**(*obj*, *name*, *value*)
called when parameter of given name is set to value i.e. instr.parameter=value. Customized messages for different types. Also saves data

taref.core.atom_extension.**set_tag**(*obj*, *name*, ***kwargs*)
sets the tag of a member using Atom's built in tag functionality

taref.core.atom_extension.**set_value_map**(*obj*, *name*, *value*)
checks floats and ints for low/high limits and automaps an Enum when setting. Not working for List?

class taref.core.atom_extension.**tag_Callable**(***kwargs*)
disposable decorator class that returns a Callable tagged with kwargs

class taref.core.atom_extension.**tag_Property**(*cached=True*, ***kwargs*)
disposable decorator class that returns a cached Property tagged with kwargs

6.1.2 shower

Created on Mon Aug 24 12:38:54 2015

@author: thomasaref

taref.core.shower.**shower**(**agents*, ***kwargs*)
A powerful showing function for any Atom object(s) specified in agents. Checks if an object has a view_window and otherwise uses a default window for the object.

Checks kwargs for particular keywords:

- start_it: boolean representing whether to go through first time setup prior to starting app
- app: defaults to existing QtApplication instance and will default to a new instance if none exists

chief_cls: if not included defaults to the first agent and defaults to Backbone if no agents are passed. show_log: shows the log_window of chief_cls if it has one, defaults to not showing show_ipy: shows the interactive_window of chief_cls if it has one, defaults to not showing show_code: shows the code_window of chief_cls if it has one, defaults to not showing

shower also provides a chief_window (generally for controlling which agents are visible) which defaults to Backbone's chief_window if chief_cls does not have one. attributes of chief_window can be modified with the remaining kwargs

6.1.3 backbone

Created on Tue Jul 7 21:52:51 2015

@author: thomasaref

class taref.core.backbone.**Backbone**(**args*, ***kwargs*)
Class combining primary functions for viewer operation. Extends __init__ to allow extra setup. extends __setattr__ to perform low/high check on params

all_main_params
A Member which behaves similar to a Python property.

all_params
A Member which behaves similar to a Python property.

app = None

call_func(*name*, ***kwargs*)
calls a func using keyword assignments. If name corresponds to a Property, calls the get func. otherwise, if name_mangled func "_get_"+name exists, calls that. Finally calls just the name if these are not the case

chief_window = <taref.core.agent_e.BasicView object>

code_window = <taref.core.interactive_e.CodeWindow object>

extra_setup(*param*, *typer*)
Performs extra setup during initialization where param is name of parameter and typer is it's Atom type. Can be customized in child classes. default extra setup handles units, auto tags low and high for Ranges, and makes Callables into instancemethods

instancemethod(*func*)
decorator for adding instancemethods defined outside of class

interactive_window = <taref.core.interactive_e.InteractiveWindow object>

log_window = <taref.core.log_e.LogWindow object>

main_params
A Member which behaves similar to a Python property.

property_dict
A Member which behaves similar to a Python property.

property_names
A Member which behaves similar to a Python property.

property_values
A Member which behaves similar to a Python property.

reserved_names
A Member which behaves similar to a Python property.

unit_dict = {'c': 0.01, '%': 0.01, 'nm': 1e-09, 'G': 1000000000.0, 'mm': 0.001, 'M': 1000000.0, 'k': 1000.0, 'm': 0.001,

view_window
A Member which behaves similar to a Python property.

6.2 physics

Contents:

6.2.1 fundamentals

Created on Thu Jan 29 21:05:03 2015

@author: thomasaref

Gathers all useful constants and functions in one location. Also initiates a default log_file.

taref.physics.fundamentals.**sinc**(*X*)
sinc function which doesn't autoinclude pi

taref.physics.fundamentals.**sinc_sq**(*X*)
sinc squared which doesn't autoinclude pi

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

t