atom Documentation

Release 0.6.0

Nucleic Development Team

Nov 04, 2020

Contents

Atom is a framework for creating memory efficient Python objects with enhanced features such as dynamic initialization, validation, and change notification for object attributes. It provides the default model binding behaviour for the Enaml UI framework.

CHAPTER 1

Getting Started

Getting started with atom is easy. The following sections will cover the basics of atom and should cover the needs of most users. In particular it will try to shed lights on the following points:

- how to define a new Atom object
- how type validation works
- how the observer pattern works

1.1 Installing Atom

Atom is supported on Python 2.7, and 3.4+. Installing it is a straight-forward process. There are three approaches to choose from.

1.1.1 The easy way: Pre-compiled packages

The easiest way to install atom is through pre-compiled packages. Atom is distributed pre-compiled in two-forms.

Conda packages

If you use the Anaconda Python distribution platform (or Miniconda, its lighter-weight companion), the latest release of Atom can be installed using conda from the default channel or the conda-forge channel:

```
$ conda install atom

$ conda install atom -c conda-forge
```

Wheels

If you don't use Anaconda, you can install Atom pre-compiled, through PIP, for most common platforms:

```
$ pip install atom
```

1.1.2 Compiling it yourself: The Hard Way

Building atom from scratch requires Python and a C++ compiler. On Unix platform getting a C++ compiler properly configured is generally straighforward. On Windows, starting with Python 3.6 the free version of the Microsoft toolchain should work out of the box. Installing atom is then as simple as:

```
$ python setup.py install
```

Note: For MacOSX users on OSX Mojave, one needs to set MACOSX_DEPLOYMENT_TARGET to higher than 10.9 to force teh compiler to use the new C++ stdlib:

```
$ export MACOSX_DEPLOYMENT_TARGET=10.10
```

1.1.3 Supported Platforms

Atom is known to run on Windows, OSX, and Linux; and compiles cleanly with MSVC, Clang, GCC, and MinGW. If you encounter a bug, please report it on the Issue Tracker.

1.2 Anatomy

Since atom is designed to allow to define compact objects, the best way to illustrate how it works is to study a class definition making use of it. This example will serve to introduce key concepts that will be explained in more details in the following sections.

```
from atom.api import Atom, Value, Int, List, set_default, observe

class CompactObject(Atom):
    """Compact object generating notifications.

    """
    untyped_value = Value()

    int_value = Int(10)

    list_value = List().tag(pref=True)

    def _post_setattr_int_value(self, old, new):
        self.untyped_value = (old, new)

    def _observe_int_value(self, change):
        print(change)

    @observe('list_value')
```

(continues on next page)

(continued from previous page)

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


class NewCompactObject(Atom):
    """Subclass with different default values.

    """
    list_value = default_value([1, 2])

    def _default_int_value(self):
        return 1
```

First note that contrary to a number of projects, atom does not export any objects in the top level atom package. To access the publicly available names you should import from *atom.api*.

```
from atom.api import Atom, Value, Int, List, default_value, observe
```

Here we import several things:

- *Atom*: This is the base class to use for all object relying on Atom. It provides the some basic methods that will be described later on or in the API documentation. (This class inherits from a more basic class CAtom)
- *Value*, *Int*, *List*: Those are members. One can think of them as advanced properties (ie they are descriptors). They define the attributes that are available on the instances of the class. They also provide type validation.
- *observe*: This is a decorator. As we will see later it can be used to call the decorated method when a member value 'change'.
- *set_default*: Members can have a default value and this object is used to alter it when subclassing an Atom object.

Now that the imports are hopefully clear (or at least clearer), let's move to the beginning of the first class definition.

```
class CompactObject(Atom):
    """Compact object generating notifications.

    """
    untyped_value = Value()

    int_value = Int(10)

    list_value = List().tag(pref=True)
```

Here we define a class and add it three members. Those three members will be the attributes that can be manipulated on the class instances. In particular it means the following will crash while it would work for a usual object:

```
obj = CompactObject()
obj.non_defined = 0
```

This may be surprising since on usual Python object one can define new attributes on instances. This limitation is the price to pay for the compacity of Atom objects.

Note: This limitation should rarely be an issue and if it is one can get dynamic attributes back by adding teh following line to the class definition:

```
__slots__ = ('__dict__',)
```

Ok, so each member will be one instance attribute. Now let's look at them in more details. Our first member is a simple *Value*, this member actaully does not perform any type validation and can be used when the attributes can really store anything. Our second member is an *Int*. This member will validate that the assigned value is actually an integer and the default value is 10 instead of 0. Finally, we have *List* which obviously can only be a list. In addition, we tagged the member. Tags are actually metadata attached to the descriptors. They have no built-in use in atom but they can be used to filter on an instance members when filtering them. Refer to the *metadata.py* example for an illustration.

Note: All the available members are described in details in *Introducing the members*

Coming back to the class definition, we reached the methods definition.

```
def _post_setattr_int_value(self, old, new):
    self.untyped_value = (old, new)

def _observe_int_value(self, change):
    print(change)

@observe('list_value')
def notify_change(self, change):
    print(change)
```

Here we define three methods of which none are meant to be called directly by the user code but will be called by the framework at appropriate time.

- `_post_setattr_int_value`: This function will be called right after setting the value of `int_value` as its name indicates. It will get both the value of the member before the setting operation (old) and the value that was just set (set).
- `_observe_int_value`: This function will be called each the value of `int_value` changes (not necessarily through a setattr operation). It is passed of dictionary containing a bunch of information about the actual modification. We will describe the content of this dictionary in details in *Notifications and observers*.
- `notify_changes`: Because this function is decorated with the observe decorator, it will be called each time `list_value`. Note however that changes to the container through `append` for example will not be caught.

Note: Prefixed methods (_post_setattr, _observe, ...) are discussed in more details in *Customimizing members: specially named methods*.

Note: Here we have only seen observer definition from within a class. It IS possible to define observers on instances and this will be discussed in *Notifications and observers*.

Now we can look at the second class definition and discuss a bit more default values.

```
class NewCompactObject(Atom):
    """Subclass with different default values.

    """
    list_value = set_default([1, 2])
```

(continues on next page)

(continued from previous page)

```
    def _default_int_value(self):
        return 1
```

In this subclass, we simply alter the default values of two of the members, and we do that in two ways:

- using *set_default* which indicates to the framework that it should create a copy of the member existing of the base class and change the default value.
- using a specially named method starting with `_default_` followed by the member name.

To clarify, what this does we can look at what happens after we create instances of each of our classes.

```
obj1 = CompactObject()
print(obj1.int_value)
print(obj1.list_value)

obj2 = NewCompactObject()
print(obj2.int_value)
print(obj2.list_value)
```

The output of this block will be:

- `10`: which match the specified default value in the class definition
- `[]`: which corresponds to the absence of a specific default value for a list.
- `1`: which corresponds to the value returned by the method used to compute the default value.
- `[1, 2]` which corresponds to the default value we specified using *set_default*.

Note: First note that even though we did not define an `__init__` methods, we can pass as keyword argument any of the member of the class in which case the argument will be used to set the value of the corresponding member.

```
obj1 = CompactObject(untyped_value='e')
```

Note: Atom objects can be frozen using `unfreeze()` at any time of their lifetime to forbid further modifications. At a later time the object can unfrozen using *notify()*.

1.2.1 Conclusion

This brief introduction should have given some basics concerning Atom working. The next three sections will cover in more details three points introduced here: the members, notifications and in particular observers specific to an instance, and finally the specially named methods used to alter default member behaviors.

1.3 Introducing the members

As we have seen in the introduction, members are used in the class definition of an atom object to define the fields that will exist on each instance of that class. As such members are central to atom.

The following sections will shed some lights on the different members that come with atom and also how they work which will come handy when we will discuss how you can customize the behaviors of members later in this guide.

1.3.1 Member working

From a technical point of view, members are descriptors like property and they can do different things when you try to access or set the attribute.

Member reading

Let's first look at what happen when you access an attribute:

```
class Custom(Atom):

    value = Int()

obj = Custom()
obj.value
obj.value
```

Since we did not pass a value for `value` when instantiating our class (we did not do `obj=Custom(value=1)`), when we first access `value` it does not have any value. As a consequence the framework will fetch the default value. As we have seen in the introduction, the default value can be specified in several ways, either as argument to the member, or using `set_default` or even by using a specially named method (more on that in *Customimizing members: specially named methods*).

Once the framework has fetched the default value it will *validate* it. In particular here, we are going to check that we did get an integer for example. The details of the validation will obviously depend on the member.

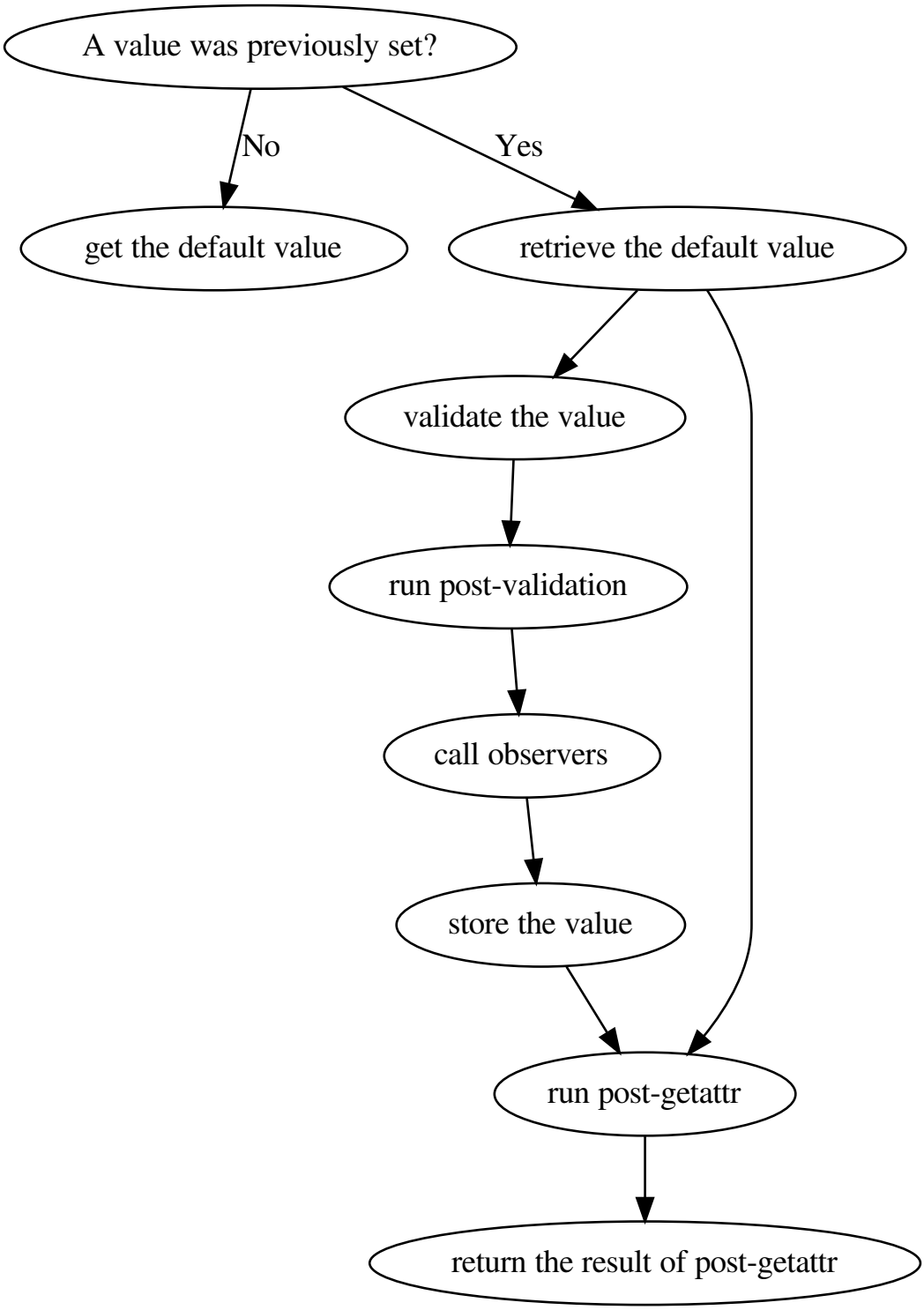
If the value is valid, next a post-validation method will be called that can some do further processing. By default this is a no-op and we will see in *Customimizing members: specially named methods* how this can be customized.

With this process complete, the state of our object has changed since we created the value stored in that instance. This corresponds to a *create* that will be sent to the observers if any is registered.

The observer called, the value can now be stored (so that we don't go through this again) and is now ready to be returned to the sure, the *get* step is complete. However before doing that we will actually perform a *post-gettatr* step. Once again this is a no-op by default but can be customized.

On further accesses, since the value exists, we will go directly retrieve the value and perform the *post-getattr*, and no notification will be generated.

To summarize:


A value was previously set?
No
Yes
get the default value
retrieve the default value
validate the value
run post-validation
call observers
store the value
run post-getattr
return the result of post-getattr

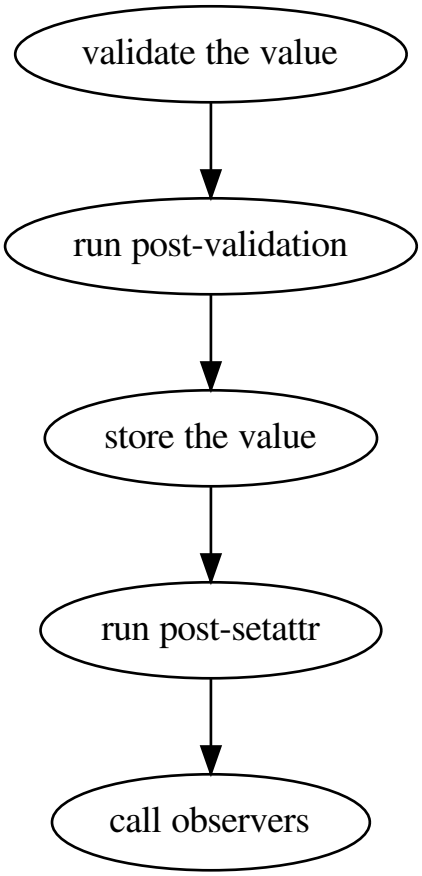
Member writing

Setting a value follows a very similar pattern. First the value is of course validated (and post-validated). It is then actually stored (*set*).

Next as for the *get* and *validate* operation, a *post-setattr* step is run. As for the other *post* by default this won't do anything.

Finally is any observer is attached, the observers are notified.

To summarize:



1.3.2 Members introduction

Now that the behavior of members is a bit less enigmatic let's introduce the members that comes with atom.

Members for simple values

Atom provides the following members for basic scalars types:

- *Value*: a member that can receives any value, no validation is performed
- *Int*: an integer value. One can choose if it is allowed to cast the assigned values (float to int), the default is true.
- *Float*: a floating point value. One can choose if it is allowed to cast the assigned values (int to float, ...), the default is true.
- *Bytes*, *Str*: bytes and unicode strings. One can choose if it is allowed to cast the assigned values (str to bytes, ...), the default is false.
- *Enum*: a value that can only take a finite set of values. Note that this is unrelated to the enum module.

Containers and type validation

Atom also provides members for three basic containers tuple, list and dictionaries: *Tuple*, *List*, *Dict*. In each case, you can specify the type of the values (key and value for dict), using members, as follows:

```
class MyAtom(Atom):

    t = Tuple(Int())
    l = List(Float())
    d = Dict(Str(), Int())
```

Alternatively, you can pass simple Python types. In this cases they will be wrapped in an *Instance* member that will be introduced in the next section.

```
class MyAtom(Atom):

    t = Tuple(int)
    l = List(float)
    d = Dict(str, int)
```

Note: Note that you cannot (by default) enforce a specific number of items in a tuple.

Note: In order to enforce type validation of container, atom has to use custom subclass. As a consequence, when assigning to a member, the original container is copied. This copy on assignment behavior can cause some surprises if you modifiy the original container after assigning it.

One additional important point, atom does not track the content of the container is not tracked. As a consequence, in place modification of the container do not trigger any notifications. One workaround can be to copy the container, modify it and re-assign it. Another option for lists is to use a `ContainerList` member, which uses a special list subclass sending notifications when the list is modified.

Enforcing custom types

Sticking to simple types can quickly be limitating and this is why atom provides member to enforce that the value is simply of a certain type or a subclass:

- *Typed*: the value must of the specified type or a subtypes. Only one type can be specified.
- *Instance*: the value must pass `isinstance(value, types)`. Using *Instance* once can specify a tuple of types.

- *Subclass*: the value must be a class and a subclass of the specified type.

In some cases, the type is not accessible when the member is instantiated (because it will be created later in the same file for example), atom also provides *ForwardTyped*, *ForwardInstance*, *ForwardSubclass*. Those three members rather than taking a type or a tuple of type as argument, accept a callable taking no argument and returning the type(s) to use for validation.

```
class Leaf(Atom):

    node = ForwardTyped(lambda : Node)

class Node(Atom):

    parent = ForwardTyped(lambda : Node)

    leaves = List(Typed(Leaf))
```

In some cases, the same information may be convienently represented either by a custom class or something simpler, like a tuple. One example of such a use case is a color: a color can be easily represented by the four components (red, green, blue, alpha) but in a library may be represented by a custom class. Atom provides the *Coerced* member to allow to enforce a particular type while also allowing seamless conversion from alternative representations.

The conversion can occur in two ways as illustrated below:

- by calling the specified types on the provided value
- by calling an alternative coercer function provided to the member

```
class Color(object):

    def __init__(self, components):
        self.red, self.green. self.blue, self.alpha = components

def dict_to_color(color_dict):
    components = []
    for c in ('red', 'green', 'blue', 'alpha')
        components.append(color_dict[c])
    return Color(components)

class MyAtom(Atom):

    color = Coerced(Color)

    color2 = Coerced(Color, coercer=dict_to_color)
```

Memory less members

Atom also provides two members that do not remember the value they are provided, but that can be used to fire notifications:

- *Event*: this is a member to which each time a value is assigned to, a notification is fired. Additionally one can specify the type of value that are accepted. An alternative way to fire the notification is to call the object you get when accessing the member.
- *Signal*: this member is similar to Qt signal. One cannot be assigned to it, however one can call it on instances, and when called the notifier will be called **with the arguments and keyword arguments passed to the signal**. Note that this is at odds with the general behavior of observers described in *Notifications and observers*.

  The example below illustrates how those members work:

```
class MyAtom(Atom):

    s = Signal()

    e = Event()

    @observe('s', 'e')
    def print_value(self, change):
        print(change)

obj = MyAtom()
obj.e = 2
obj.e(1)
obj.s(2)
obj.emit(1)
```

Delegator

This last member is a bit special. It does not do anything by itself but can be used to copy the behaviors of another member. In addition, any observer attached to the delegator will also be attached to the delegate member.

Property

The `Property` member is a special case and it will be discussed in details in *The Property member*.

1.4 Notifications and observers

One key feature of the framework in addition to small memory footprint and type validation is the implementation of the observer pattern. In *Anatomy*, we introduced the notion of static observers. Here we will discuss them in more details along with dynamic observers. We will also describe in depth the possible signature for notification handlers and the arguments they receive upon invocation.

Note: The point at which notifications are fired has been discussed in *Introducing the members*.

1.4.1 Static and dynamic observers

An observer is a callable that is called each time a member changes. For most members it will be:

- when the member get value for the first time either through an assignment or a first access when the default value is used. We will refer to this as a 'create' event.
- whenever a different value is assigned to the member. We will refer to this as an 'update' event.
- when the value of a member is deleted. We will refer to this as a delete event.

Note: The `ContainerList` member is a special case simce it can emit notifications when elements are added or removed from the list. This will be refrred to as 'container' events.

The distinction between static and dynamic observers comes from the moment at which the binding of the observer to the member is defined. In the case of static observers, this is done at the time of the class definition and hence affects all instances of the class. On the contrary, dynamic observers are bound to a specific instance at a later time.

The next two sections will focus on how to manage static and dynamic observers binding, while the following sections will focus on the signature of the handlers and the content of the notification dictionary passed to the handlers in different situations.

Static observers

Static observers can be bound to a member in three ways:

- declaring a method matching the anme of the member to observe but whose name starts with `_observe_`
- using the *observe* decorator on method. The decorator can take an arbitrary number of arguments which allows to tie the same observer to multiple members. In addition, *observe* accept as argument a dot separated name to automatically observe a member of an atom object stored in a member. Note that this mechanism is limited to a single depth (hence a single dot in the name).
- finally one can manage manually static observer using the following methods defined on the base class of all members: + *add_static_observer* which takes a single callable as arguemnt + *remove_static_observer* which takes a single callable as arguemnt

Dynamic observers

Dynamic observers are managed using the *observe* and *unobserve* methods of the *Atom* class. To observe one needs to pass the name of the member to observe and the callback function. When unobserving, you can either pass just the member name to remove all observers at once or a name and a callback to remove specific observer.

Note: Two specific members have an additional way to manage observers:

- *Event*: expose the methods `bind()` and `unbind()` which takes as single argument the callback to bind.
- *Signal*: similarly *Signal* exposes `connect()` and `disconnect()` which match Qt signals.

1.4.2 Notification handlers

Now that we discussed all kind of observers and how to manage them, it is more than time to discuss the expected signatures of callback and what information the callback is passed when called.

For observers connected to all members except *Signal*, the callback should accept a single argument which is usually called *change*. This argument is a dictionary with `str` as keys which are described below:

- **`'type'`: A string describing the event that triggered the notification:**
  - 'created': when accessing or assigning to a member that has no previous value.
  - 'update': when assigning a new value to a member with a previous value.
  - 'delete': when deleting a member (using `del` or `delattr`)
  - 'container': when doing inplace modification on a the of `ContainerList`.
- `'object'`: This is the *Atom* instance that triggered the notification.
- `'name'`: Name of the member from which the notification originate.
- `'value'`: New value of the member (or old value of the member in the case of a delete event).

- `'oldvalue'`: Old value of the member in the case of an update.

In the case of `'container'` events emitted by `ContainerList` the change disctionary can contains additional information (note that `'value'` and `'oldvalue'` are present):

- `'operation'`: a str describing the operation that took place (append, extend, __setitem__, insert, __delitem__, pop, remove, reverse, sort, __imul__, __iadd__)
- `'item'`: the item that was modified if the modification affected a single item.
- `'items'`: the items that were modified if the modification affected multiple items.

Note: As mentionned previously, *`Signal`* emits notifications in a different format. When calling (emitting) the signal, it will pass whatever arguments and keyword arguments it was passed as is to the observers as illustrated below.

```
class MyAtom(Atom):

    s = Signal()

def print_pair(name, value):
    print(name, value

a = MyAtom()
a.s.connect(print_pair)
a.s('a', 1)
```

1.4.3 Suppressing notifications

If for any reason you need to prevent notifications to be progated you can use the *`suppress_notifications`* context manager. Inside this context manager, notifications will not be propagated.

1.5 Customimizing members: specially named methods

Atom offers multiple ways to customize the inner working of members. The easiest one is to use specially named methods on your class definition. Since this covers most of the use cases, it is the only one that is covered here more details and advanced methods can be found in *Member customization: advanced techniques*.

The customization method should use the name of the member to customize with one of the following prefixes depending on the operation to customize:

- `_default_`: to define default values.
- `_observe_`: to define a static observer.
- `_validate_`: to define a custom validation algorithm.
- `_post_getattr_`: to customize the post-getattr step.
- `_post_setattr_`: to customize the post-setattr step.
- `_post_validate_`: to customize the post-setattr step.

1.5.1 Default values

A default value handler should take no argument and return the default value for the member.

```
class MyAtom(Atom):

    v = Value()

    def _default_v(self):
        return [{}, 1, 'a']
```

1.5.2 Static observers

A static observer is basically an observer so it should take the change dictionary as argument (save for *Signal*).

```
class MyAtom(Atom):

    v = Value()

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

1.5.3 Validation

A validation handler should accept both the old value of the member and the new value to validate. It should return a valid value.

```
class MyAtom(Atom):

    v = Value()

    def _validate_v(self, old, new):
        if old and not isinstance(new, type(old)):
            raise TypeError()
        return new
```

1.5.4 Post-operation methods

Post-gettatr should take a single argument, ie the value that was retrieve during the *get* step, and return whatever value it decides to.

```
class MyAtom(Atom):

    v = Value()

    def _post_getattr_v(self, value):
        print('v was accessed')
        return value
```

Post-setattr and post-validate both take the old and the new value of the member as input, and post-validate should return a valid value.

```
class MyAtom(Atom):

    v = Value()

    def _post_setattr_v(self, old, new):
        print('v was set')
        return value

class MyAtom(Atom):

    v = Value()

    def _post_validate_v(self, old, new):
        print('v was validated')
        return value
```

CHAPTER 2

Advanced features of atom

The discussion provided in the previous section of the docs should cover most usecases. However atom does offer some advanced features that advanced users can take advantage of. Those will be described in the following sections

2.1 The Property member

The *Property* member looks a lot like a normal Python property which makes it quite different from other members. In particular because there is no way from atom to know when the value returned by a *Property* changes, **no notifications are emitted by default when getting or setting it**.

2.1.1 Defining a Property member

Defining a *Property* and the getter, setter and deleter associated to it can be done in several equivalent manners illustrated below:

```
from atom.api import Atom, Property, Value

def get_v(owner):
    return owner.v

def set_v(owner, value):
    owner.v = value

def del_v(owner):
    del owner.v

class MyAtom(Atom):

    v = Value()

    p1 = Property(get_v, set_v, del_v)
```

(continues on next page)

(continued from previous page)

```
    p2 = Property()

    def _get_p2(self):
        return get_v(self)

    def _set_p2(self, value):
        set_v(self, value

    def _del_p2(self):
        del_v(self)

    p3 = Property()

    p3.getter
    def _get(self):
        return get_v(self)

    p3.setter
    def _set(self, value):
        set_v(self, value)

    p3.deleter
    def _del(self):
        del_v(self)
```

2.1.2 Cached properties

For **read-only** properties, atom offers the option to cache the value returned by the getter. This can be convenient if the getter performs an expensive operation. The cache can be reset at a later time by deleting the property or by calling the *atom.property.Property.reset()* method of the member as illustrated below:

```
from atom.api import Atom, Property, cached_property

class MyAtom(Atom):

    cp1 = Property(cached=True)

    def _get_cp1(self):
        print('Called cp1')

    @cached_property
    def cp2(self):
        print('Called cp2')

a = MyAtom()

a.cp1
a.cp1
del a.cp1
a.cp1

a.cp2
a.cp2
MyAtom.cp2.reset(a)
```

(continues on next page)

(continued from previous page)

```
a.cp2
```

Running this code will print "Called cp1/2" only twice each.

2.1.3 Notifications from a Property

As mentionned in the introduction, *Property* does not fire notifications upon get/setattr. However it will always fire notifications upon deletion/reset. To manually fire notifications from a property, please refer to *Manual notifications*.

2.2 Atom and weak references

Because atom objects are slotted by default and do not have an instance an instance dictionary, they do not support weak references by defaults.

Depending on the context in which you need weak references, you have two options:

- if you need weak references to interact with an external library (such as pyqt), you will need to enable the standard weakref mechanism.
- if you use weak references only internally and memory is a concern (Python standard weak references have a not so small overhead), you can use an alternative mechanism provided by atom.

2.2.1 Enabling default weak references

In order to use the standard weak references of Python, you simply need to add the proper slot to your object as illustrated below:

```
from atom.api import Atom

MyWeakRefAtom(Atom):

    __slots__ = ('__weakref__',)
```

2.2.2 Using atom builtin weak references: `atomref`

To create a weak reference to atom object using the builtin mechanism, you simply have to create an instance of *atomref* using the object to reference as argument.

In order to access the object referenced by the *atomref*, you simply need to call it which will return the object. If the referenced object is not alive anymore, *atomref* will return None.

```
import gc
from atom.api import Atom, atomref

class MyAtom(Atom):
    pass

obj = MyAtom()
ref = atomref(obj)

assert obj is ref()
```

(continues on next page)

(continued from previous page)

```
del obj
gc.collect()
assert ref() is None
```

2.3 Atom's containers

Atom uses custom containers to implement type validation and notifications. It implements two list subclasses and one dictionary subclass to this effect.

Note: Currently the typed validated dictionary is not a subclass of the Python builtin dictionary type. This can cause some unexpected issues in particular when assigning the value stored in a `Dict` member to another `Dict` which will fail. To circumvent this issue one should call `dict` on the content of the first member. This is a known problem and will be fixed in a future version of atom.

Usually, users should not instantiate those containers manually, in particular because they need a reference to both the member and the instance to which they are tied.

Atom provides however an alternative mapping type which uses less memory than a regular dictionary in particular when the mapping contain only few objects: `sortedmap`.

2.3.1 `sortedmap`

`sortedmap` can be imported from `atom.datastructures.api`. Contrary to a regular Python dictionary, `sortedmap` does not requires the keys to be hashable, however they should be sortable. `sortedmap` will fall back on the Python 2 behavior to order any Python object based on the class name and the object id if two objects cannot be compared otherwise.

In terms of memory efficiency, here is a quick comparison: +————-+————-+————+ | | dict | sortedmap | +==========+==========+==========+ | empty | 240 | 72 | +————-+————-+————+ | 1 key | 240 | 88 | +————-+————-+————+ | 2 key | 240 | 104 | +————-+————-+————+ | 100 key | 4704 | 2120 | +————-+————-+————+

`sortedmap` is not meant to replace dictionaries but can be valuable when a large number of small containers is necessary.

2.4 Member customization: advanced techniques

In the basics, we covered how to customize the behavior of a member by using prefixed method in a class definition. However this is not the only way to customize members. The following section will first describe how a member determine the action to take at each step of the getting, setting process, before giving more details about the behaviors taht can be used to build custom members.

2.4.1 Members inner working

For each step of the process described of *Introducing the members*, Members check the mode identifying the action it should take. Based on this flag, it will either call a builtin function or the appropriate user provided function or method.

One can inspect the mode set on the member by accesing the matching attribute as described in the table below. This attribute is a tuple containing two items. The first item is the flag value matching the member behavior, the second item depend on the value of the flag.

The behavior of the member can be modified by calling the matching `set_` method, to set the flag to a new and provide the additional item matching the flag value.

For all steps of the getattr, setattr process, you can invoke them separatly by calling the mathcing `do_` method. Among them, *`do_full_validate()`* is special in that it will run both the validate and post_validate steps on the provided value.

The following table summarizes the different steps along with the flags and the aforementionned attributes and methods:

Steps	Mode	Mode (getter/setter)	Manual running
getattr	*`GetAttr`*	*`getattr_mode()`* / *`set_getattr_mode()`*	*`do_getattr()`*
post_getattr	*`PostGetAttr`*	*`post_getattr_mode()`* / *`set_post_getattr_mode()`*	*`do_post_getattr()`*
setattr	*`SetAttr`*	*`setattr_mode()`* / *`set_setattr_mode()`*	*`do_setattr()`*
post_setattr	*`PostSetAttr`*	*`post_setattr_mode()`* / *`set_post_setattr_mode()`*	*`do_post_setattr()`*
validate	*`Validate`*	*`validate_mode()`* / *`set_validate_mode()`*	*`do_validate()`*
post_validate	*`PostValidate`*	*`post_validate_mode()`* / *`set_post_validate_mode()`*	*`do_post_validate()`*
default_value	*`DefaultValue`*	*`default_value_mode()`* / *`set_default_value_mode()`*	*`do_default_value()`*

Note: Delattr works on the same model but the flag is not exposed as part of the public API. You can still access the mode using *`delattr_mode()`*

2.4.2 Behaviors for custom members

In order to create custom members, you can either subclass *`Member`* and set the modes in the `__init__` method, or set the modes after instantiating the member. The modes that can be used in conjuction with custom callable or methods are listed below and expected signature of the callable or the method can be directly inferred from the mode. When specifying a method, the second item of the mode should be the name of the method. In the following, **Object** always refers to an *`Atom`* subclass instance and **Name** to the member name.

- ***`GetAttr`*:**
  - CallObject_Object
  - CallObject_ObjectName
  - ObjectMethod
  - ObjectMethod_Name
  - MemberMethod_Object
- ***`PostGetAttr`*:**
  - ObjectMethod_Value
  - ObjectMethod_NameValue
  - MemberMethod_ObjectValue

- ***SetAttr***:
  - CallObject_ObjectValue
  - CallObject_ObjectNameValue
  - ObjectMethod_Value
  - ObjectMethod_NameValue
  - MemberMethod_ObjectValue
- ***PostSetAttr***:
  - ObjectMethod_OldNew
  - ObjectMethod_NameOldNew
  - MemberMethod_ObjectOldNew
- ***DefaultValue***:
  - CallObject
  - CallObject_Object
  - CallObject_ObjectName
  - ObjectMethod
  - ObjectMethod_Name
  - MemberMethod_Object
- ***Validate***:
  - ObjectMethod_OldNew
  - ObjectMethod_NameOldNew
  - MemberMethod_ObjectOldNew
- ***PostValidate***:
  - ObjectMethod_OldNew
  - ObjectMethod_NameOldNew
  - MemberMethod_ObjectOldNew

Note: It is recommended to avoid customizing specialized members that may make some assumptions regarding the values of the other modes.

2.5 Manual notifications

Atom object usually fire notifications at the proper times. However, in some cases (Property member, manual handling of container change), it may be desirable to manually fire a notification.

This is possible but require some care.

First, when manually notifying, you are responsible for building the change dictionary that will be passed to the handlers. You may refer to *Notifications and observers* for a description of the content of this dictionary for normal notifications.

Second, because atom handle separately the static observers and the dynamic observers, you will to be sure to call both kinds. To notify the static observers, you should call the *notify()* method, while to notify dynamic obsevers you need to call the *freeze()* method on the instance.

CHAPTER 3

Atom examples

3.1 Tutorial Examples

3.1.1 Hello World Example

Hello world example: how to write an atom class.

Tip: To see this example in action, download it from `hello_world` and run:

```
$ python hello_world.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Hello world example: how to write an atom class.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, Str


class Hello(Atom):
```

(continues on next page)

(continued from previous page)

```
    message = Str('Hello')


if __name__ == "__main__":
    hello = Hello()
    print(hello.message)
    hello.message = 'Goodbye'
    print(hello.message)
```

3.1.2 Person Example

Simple class using atom and static observers.

Tip: To see this example in action, download it from `person` and run:

```
$ python person.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Simple class using atom and static observers.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, Str, Range, Bool, observe


class Person(Atom):
    """ A simple class representing a person object.

    """
    last_name = Str()

    first_name = Str()

    age = Range(low=0)

    debug = Bool(False)

    @observe('age')
    def debug_print(self, change):
        """ Prints out a debug message whenever the person's age changes.
```

(continues on next page)

(continued from previous page)

```
        """
        if self.debug:
            templ = "{first} {last} is {age} years old."
            s = templ.format(
                first=self.first_name, last=self.last_name, age=self.age,
            )
            print(s)


if __name__ == '__main__':
    john = Person(first_name='John', last_name='Doe', age=42)
    john.debug = True
    john.age = 43
```

3.1.3 Employee Example

Simple example of a class hierarchy built on atom.

Tip: To see this example in action, download it from `employee` and run:

```
$ python employee.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Simple example of a class hierarchy built on atom.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

import datetime

from atom.api import (
    Atom, Str, Range, Bool, Value, Int, Tuple, Typed, observe
)


class Person(Atom):
    """ A simple class representing a person object.

    """
    last_name = Str()
```

(continues on next page)

(continued from previous page)

```
    first_name = Str()

    age = Range(low=0)

    dob = Value(datetime.date(1970, 1, 1))

    debug = Bool(False)

    @observe('age')
    def debug_print(self, change):
        """ Prints out a debug message whenever the person's age changes.

        """
        if self.debug:
            templ = "{first} {last} is {age} years old."
            s = templ.format(
                first=self.first_name, last=self.last_name, age=self.age,
            )
            print(s)


class Employer(Person):
    """ An employer is a person who runs a company.

    """
    # The name of the company
    company_name = Str()


class Employee(Person):
    """ An employee is person with a boss and a phone number.

    """
    # The employee's boss
    boss = Typed(Employer)

    # The employee's phone number as a tuple of 3 ints
    phone = Tuple(Int())

    # This method will be called automatically by atom when the
    # employee's phone number changes
    def _observe_phone(self, val):
        if val['type'] == 'update':
            msg = 'received new phone number for %s: %s'
            print(msg % (self.first_name, val['value']))


if __name__ == '__main__':
    # Create an employee with a boss
    boss_john = Employer(
        first_name='John', last_name='Paw', company_name="Packrat's Cats",
    )
    employee_mary = Employee(
        first_name='Mary', last_name='Sue', boss=boss_john,
        phone=(555, 555, 5555),
    )
```

(continues on next page)

(continued from previous page)

```
    employee_mary.phone = (100, 100, 100)
    employee_mary.age = 40  # no debug message
    employee_mary.debug = True
    employee_mary.age = 50
```

3.2 API Examples

3.2.1 Coersion Example

Demonstration of the basic use of the Coerced member.

Tip: To see this example in action, download it from `coersion` and run:

```
$ python coersion.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the basic use of the Coerced member.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, Coerced


class Demo(Atom):

    cint = Coerced(int)
    cfloat = Coerced(float)
    cstr = Coerced(str)


if __name__ == '__main__':
    demo = Demo()

    print('CInt Demo')
    demo.cint = '1'
    print(demo.cint)
    demo.cint = 51.5
    print(demo.cint)

    print('\nCFloat Demo')
```

(continues on next page)

(continued from previous page)

```
    demo.cfloat = '1.5'
    print(demo.cfloat)
    demo.cfloat = 100
    print(demo.cfloat)

    print('\nCStr Demo')
    demo.cstr = 100
    print(demo.cstr)
    demo.cstr = Demo
    print(demo.cstr)
```

3.2.2 Composition Example

Demonstrate the use of Compostion of Atom objects.

1. If the class has not been declared, use a ForwardTyped - Note the use of lambda, because "Person" is not yet defined
2. A Typed object can be instantiated three ways: - Provide args, kwargs, or a factory in the definition - Provide a _default_* static constructor - Provide a pre-created object in the constructor

Tip: To see this example in action, download it from `composition` and run:

```
$ python composition.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
""" Demonstrate the use of Compostion of Atom objects.

1. If the class has not been declared, use a ForwardTyped
   - Note the use of lambda, because "Person" is not yet defined

2. A Typed object can be instantiated three ways:
   - Provide args, kwargs, or a factory in the definition
   - Provide a _default_* static constructor
   - Provide a pre-created object in the constructor

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, Typed, ForwardTyped, Str


class Dog(Atom):
```

(continues on next page)

(continued from previous page)

```
    name = Str()

    # note the use of lambda, because Person has not been defined
    owner = ForwardTyped(lambda: Person)


class Person(Atom):

    name = Str()

    # uses static constructor
    fido = Typed(Dog)

    # uses kwargs provided in the definition
    fluffy = Typed(Dog, kwargs=dict(name='Fluffy'))

    # uses an object provided in Person constructor
    new_dog = Typed(Dog)

    def _default_fido(self):
        return Dog(name='Fido', owner=self)


if __name__ == '__main__':
    bob = Person(name='Bob Smith')

    print('Fido')
    print('name: {0}'.format(bob.fido.name))
    print('owner: {0}'.format(bob.fido.owner.name))

    print('\nFluffy')
    print('name: {0}'.format(bob.fluffy.name))
    print('original owner: {0}'.format(repr(bob.fluffy.owner)))  # none
    bob.fluffy.owner = bob
    print('new owner: {0}'.format(bob.fluffy.owner.name))

    print('\nNew Dog')
    new_dog = Dog(name='Scruffy', owner=bob)
    bob.new_dog = new_dog
    print('name: {0}'.format(bob.new_dog.name))
    print('owner: {0}'.format(bob.new_dog.owner.name))
```

3.2.3 Containers Example

Demonstration of the member handling containers.

Tip: To see this example in action, download it from `containers` and run:

```
$ python containers.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the member handling containers.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, List, ContainerList, Tuple, Dict


class Data(Atom):

    dlist = List(default=[1, 2, 3])

    dcont_list = ContainerList(default=[1, 2, 3])

    dtuple = Tuple(default=(5, 4, 3))

    ddict = Dict(default=dict(foo=1, bar='a'))

    def _observe_dcont_list(self, change):
        print('container list change: {0}'.format(change['value']))


if __name__ == '__main__':
    data = Data()
    print(data.dlist)
    print(data.dcont_list)
    data.dcont_list.append(1)
    data.dcont_list.pop(0)
    print(data.dtuple)
    print(data.ddict)
```

3.2.4 Default Value Example

Demonstrate all the ways to initialize a value

1. Pass the value directly
2. Assign the default value explicitly
3. Provide the value during initialization of the object
4. Provide factory callable that returns a value
5. Use a _default_* static method

Tip: To see this example in action, download it from `default_value` and run:

```
$ python default_value.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
""" Demonstrate all the ways to initialize a value

1. Pass the value directly
2. Assign the default value explicitly
3. Provide the value during initialization of the object
4. Provide factory callable that returns a value
5. Use a _default_* static method

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

import sys

from atom.api import Atom, Str, Int


def get_mother():
    return 'Maude ' + get_last_name()


def get_last_name():
    """ Return a last name based on the system byteorder.

    """
    return sys.byteorder.capitalize()


class Person(Atom):
    """ A simple class representing a person object.

    """
    first_name = Str('Bob')

    age = Int(default=40)

    address = Str()

    mother = Str(factory=get_mother)

    last_name = Str()

    def _default_last_name(self):
```

(continues on next page)

(continued from previous page)

```
        return get_last_name()


if __name__ == '__main__':
    bob = Person(address='101 Main')
    print((bob.first_name, bob.last_name, bob.age))
    print(bob.mother)
```

3.2.5 Metadata Example

Example demonstrating the use of metadata to filter members.

Tip: To see this example in action, download it from `metadata` and run:

```
$ python metadata.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2018, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Example demonstrating the use of metadata to filter members.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

import sys

from atom.api import Atom, Str, Int


def get_last_name():
    """ Return a last name based on the system byteorder.

    """
    return sys.byteorder.capitalize()


class Person(Atom):
    """ A simple class representing a person object.

    """
    first_name = Str('Bob').tag(pref=True)

    age = Int(default=40).tag(pref=False)
```

(continues on next page)

(continued from previous page)

```
    last_name = Str()

    def _default_last_name(self):
        return get_last_name()


if __name__ == '__main__':
    bob = Person()

    for name, member in bob.members().items():
        if member.metadata and 'pref' in member.metadata:
            print(name, member.metadata['pref'])
```

3.2.6 Numeric Example

Demonstration of the member handling numeric values.

Tip: To see this example in action, download it from `numeric` and run:

```
$ python numeric.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the member handling numeric values.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

import sys

from atom.api import Atom, Int, Float, Bool


class Data(Atom):

    ival = Int(1)

    lval = Int(sys.maxsize + 1)

    fval = Float(1.5e6)

    bval = Bool(False)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    data = Data()
    print(data.ival)
    print(data.lval)
    print(data.fval)
    print(data.bval)
```

3.2.7 Observe Example

Demonstration of the use of static and dynamic observers.

Tip: To see this example in action, download it from `observe` and run:

```
$ python observe.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the use of static and dynamic observers.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)

from atom.api import Atom, Str, Range, Typed, observe


class Dog(Atom):

    name = Str()


class Person(Atom):
    """ A simple class representing a person object.

    """
    name = Str()

    age = Range(low=0)

    dog = Typed(Dog, ())

    def _observe_age(self, change):
        print('Age changed: {0}'.format(change['value']))
```

(continues on next page)

(continued from previous page)

```
    @observe('name')
    def any_name_i_want(self, change):
        print('Name changed: {0}'.format(change['value']))

    @observe('dog.name')
    def another_random_name(self, change):
        print('Dog name changed: {0}'.format(change['value']))


def main():
    bob = Person(name='Robert Paulson', age=40)
    bob.name = 'Bobby Paulson'
    bob.age = 50
    bob.dog.name = 'Scruffy'

    def watcher_func(change):
        print('Watcher func change: {0}'.format(change['value']))

    bob.observe('age', watcher_func)
    bob.age = 51
    bob.unobserve('age', watcher_func)
    bob.age = 52  # No call to watcher func


if __name__ == '__main__':
    main()
```

3.2.8 Property Example

Demonstration of the basics of the Property member.

Tip: To see this example in action, download it from `property` and run:

```
$ python property.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the basics of the Property member.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)
```

(continues on next page)

(continued from previous page)

```
from atom.api import Atom, Str, Property, Int


class Person(Atom):
    """ A simple class representing a person object.

    """
    first_name = Str()

    age = Property()
    _age = Int(40)

    def _get_age(self):
        return self._age

    def _set_age(self, age):
        if age < 100 and age > 0:
            self._age = age


if __name__ == '__main__':
    bob = Person(first_name='Bob')
    print(bob.age)
    bob.age = -10
    print(bob.age)
    bob.age = 20
    print(bob.age)
```

3.2.9 Range Example

Demonstration of the ranges members.

Tip: To see this example in action, download it from `range` and run:

```
$ python range.py
```

Example Atom Code

```
#------------------------------------------------------------------------------
# Copyright (c) 2013-2017, Nucleic Development Team.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
#------------------------------------------------------------------------------
"""Demonstration of the ranges members.

"""
from __future__ import (division, unicode_literals, print_function,
                        absolute_import)
```

(continues on next page)

(continued from previous page)

```
from atom.api import Atom, Range, FloatRange


class Experiment(Atom):

    coef = FloatRange(-1.0, 1.0, 0.0)

    gain = Range(0, 100, 10)


if __name__ == '__main__':
    exp = Experiment()

    print(exp.coef)
    exp.coef = 0.5
    print(exp.coef)

    print(exp.gain)
    exp.gain = 99
    print(exp.gain)
```

CHAPTER 4

Developer notes

These notes are meant to help developers and contributors with regards to some details of the implementation and coding style of the project.

4.1 Python codebase

The Python codebase currently targets Python 3.6+, strives to follow PEP 8 and uses Numpy style docstring.

4.2 Python C++ bindings

The bindings are hand-written and relies on cppy (https://github.com/nucleic/cppy). Atom tries to use a reasonably modern C API and to support sub-interpreter, this has a couple of consequences:

- static variables use is limited to cases that cannot lead to state leakage between multiple sub-interpreters. Note that this is currently not heavily tested and may require some improvements.
- all the non exported symbol are enclosed in anonymous namespaces
- atom does not use static types and only dynamical types (note that the type slots and related structures are stored in a static variable)
- modules use the multi-phases initialization mechanism as defined in PEP 489 – Multi-phase extension module initialization

CHAPTER 5

atom package

Atom does not export any name at the root of the package (*__init__.py* is empty). To access the most commonly used component import the *api.py* module found in each package.

5.1 Subpackages

5.1.1 atom.datastructures package

Submodules

sortedmap extension module

5.2 Submodules

5.2.1 atom.api module

Module exporting the public interface to atom.

5.2.2 atom.atom module

class atom.atom.**Atom**

Bases: *atom.catom.CAtom*

The base class for defining atom objects.

Atom objects are special Python objects which never allocate an instance dictionary unless one is explicitly requested. The storage for an atom is instead computed from the *Member* objects declared on the class. Memory is reserved for these members with no over allocation.

This restriction make atom objects a bit less flexible than normal Python objects, but they are between 3x-10x more memory efficient than normal objects depending on the number of attributes.

classmethod members()
Get the members dictionary for the type.

Returns result – The dictionary of members defined on the class. User code should not modify the contents of the dict.

Return type dict

suppress_notifications()
Disable member notifications within in a context.

Returns result – A context manager which disables atom notifications for the duration of the context. When the context exits, the state is restored to its previous value.

Return type contextmanager

class atom.atom.**AtomMeta**
Bases: type

The metaclass for classes derived from Atom.

This metaclass computes the memory layout of the members in a given class so that the CAtom class can allocate exactly enough space for the object data slots when it instantiates an object.

All classes deriving from Atom will be automatically slotted, which will prevent the creation of an instance dictionary and also the ability of an Atom to be weakly referenceable. If that behavior is required, then a subclasss should declare the appropriate slots.

class atom.atom.**ExtendedObserver**(*funcname*, *attr*)
Bases: object

A callable object used to implement extended observers.

attr

funcname

class atom.atom.**ObserveHandler**(*pairs*)
Bases: object

An object used to temporarily store observe decorator state.

clone()
Create a clone of the sentinel.

func

funcname

pairs

atom.atom.**observe**(**names*)
A decorator which can be used to observe members on a class.

Parameters *names – The str names of the attributes to observe on the object. These must be of the form 'foo' or 'foo.bar'.

class atom.atom.**set_default**(*value*)
Bases: object

An object used to set the default value of a base class member.

clone()
Create a clone of the sentinel.

name

value

5.2.3 atom.catom module

catom extension module

class atom.catom.**CAtom**
Bases: object

freeze()
Freeze the atom to prevent further modifications to its attributes.

get_member()
Get the named member for the atom.

has_observer()
Get whether the atom has the given observer for a given topic.

has_observers()
Get whether the atom has observers for a given topic.

notifications_enabled()
Get whether notification is enabled for the atom.

notify()
Call the registered observers for a given topic with positional and keyword arguments.

observe()
Register an observer callback to observe changes on the given topic(s).

set_notifications_enabled()
Enable or disable notifications for the atom.

unobserve()
Unregister an observer callback for the given topic(s).

class atom.catom.**DefaultValue**
Bases: atom.intenum.IntEnum

CallObject = <enum: DefaultValue.CallObject [value=6]>

CallObject_Object = <enum: DefaultValue.CallObject_Object [value=7]>

CallObject_ObjectName = <enum: DefaultValue.CallObject_ObjectName [value=8]>

Delegate = <enum: DefaultValue.Delegate [value=5]>

Dict = <enum: DefaultValue.Dict [value=4]>

List = <enum: DefaultValue.List [value=2]>

MemberMethod_Object = <enum: DefaultValue.MemberMethod_Object [value=11]>

NoOp = <enum: DefaultValue.NoOp [value=0]>

ObjectMethod = <enum: DefaultValue.ObjectMethod [value=9]>

ObjectMethod_Name = <enum: DefaultValue.ObjectMethod_Name [value=10]>

Set = <enum: DefaultValue.Set [value=3]>

Static = <enum: DefaultValue.Static [value=1]>

class atom.catom.**DelAttr**
Bases: atom.intenum.IntEnum

Constant = <enum: DelAttr.Constant [value=2]>

Delegate = <enum: DelAttr.Delegate [value=6]>

Event = <enum: DelAttr.Event [value=4]>

NoOp = <enum: DelAttr.NoOp [value=0]>

Property = <enum: DelAttr.Property [value=7]>

ReadOnly = <enum: DelAttr.ReadOnly [value=3]>

Signal = <enum: DelAttr.Signal [value=5]>

Slot = <enum: DelAttr.Slot [value=1]>

class atom.catom.**GetAttr**
Bases: atom.intenum.IntEnum

CachedProperty = <enum: GetAttr.CachedProperty [value=6]>

CallObject_Object = <enum: GetAttr.CallObject_Object [value=7]>

CallObject_ObjectName = <enum: GetAttr.CallObject_ObjectName [value=8]>

Delegate = <enum: GetAttr.Delegate [value=4]>

Event = <enum: GetAttr.Event [value=2]>

MemberMethod_Object = <enum: GetAttr.MemberMethod_Object [value=11]>

NoOp = <enum: GetAttr.NoOp [value=0]>

ObjectMethod = <enum: GetAttr.ObjectMethod [value=9]>

ObjectMethod_Name = <enum: GetAttr.ObjectMethod_Name [value=10]>

Property = <enum: GetAttr.Property [value=5]>

Signal = <enum: GetAttr.Signal [value=3]>

Slot = <enum: GetAttr.Slot [value=1]>

class atom.catom.**Member**
Bases: object

add_static_observer()
Add the name of a method to call on all atoms when the member changes.

clone()
Create a clone of this member.

copy_static_observers()
Copy the static observers from one member into this member.

default_value_mode
Get the default value mode for the member.

del_slot()
Delete the atom's slot value directly.

delattr_mode
Get the delattr mode for the member.

do_default_value()
Run the default value handler for member.

do_delattr()
Run the delattr handler for the member.

do_full_validate()
Run the validation and post validation handlers for the member.

do_getattr()
Run the getattr handler for the member.

do_post_getattr()
Run the post getattr handler for the member.

do_post_setattr()
Run the post setattr handler for the member.

do_post_validate()
Run the post validation handler for the member.

do_setattr()
Run the setattr handler for the member.

do_validate()
Run the validation handler for the member.

get_slot()
Get the atom's slot value directly.

getattr_mode
Get the getattr mode for the member.

has_observer()
Get whether or not the member already has the given observer.

has_observers()
Get whether or not this member has observers.

index
Get the index to which the member is bound

metadata
Get and set the metadata for the member.

name
Get the name to which the member is bound.

notify()
Notify the static observers for the given member and atom.

post_getattr_mode
Get the post getattr mode for the member.

post_setattr_mode
Get the post setattr mode for the member.

post_validate_mode
Get the post validate mode for the member.

remove_static_observer()
Remove the name of a method to call on all atoms when the member changes.

set_default_value_mode()
Set the default value mode for the member.

set_delattr_mode()
Set the delattr mode for the member.

set_getattr_mode()
Set the getattr mode for the member.

set_index()
Set the index to which the member is bound. Use with extreme caution!

set_name()
Set the name to which the member is bound. Use with extreme caution!

set_post_getattr_mode()
Set the post getattr mode for the member.

set_post_setattr_mode()
Set the post setattr mode for the member.

set_post_validate_mode()
Set the post validate mode for the member.

set_setattr_mode()
Set the setattr mode for the member.

set_slot()
Set the atom's slot value directly.

set_validate_mode()
Set the validate mode for the member.

setattr_mode
Get the setattr mode for the member.

static_observers()
Get a tuple of the static observers defined for this member

tag()
Tag the member with metatdata.

validate_mode
Get the validate mode for the member.

class atom.catom.**PostGetAttr**
Bases: atom.intenum.IntEnum

Delegate = <enum: PostGetAttr.Delegate [value=1]>

MemberMethod_ObjectValue = <enum: PostGetAttr.MemberMethod_ObjectValue [value=4]>

NoOp = <enum: PostGetAttr.NoOp [value=0]>

ObjectMethod_NameValue = <enum: PostGetAttr.ObjectMethod_NameValue [value=3]>

ObjectMethod_Value = <enum: PostGetAttr.ObjectMethod_Value [value=2]>

class atom.catom.**PostSetAttr**
Bases: atom.intenum.IntEnum

Delegate = <enum: PostSetAttr.Delegate [value=1]>

MemberMethod_ObjectOldNew = <enum: PostSetAttr.MemberMethod_ObjectOldNew [value=4]>

NoOp = <enum: PostSetAttr.NoOp [value=0]>

ObjectMethod_NameOldNew = <enum: PostSetAttr.ObjectMethod_NameOldNew [value=3]>

ObjectMethod_OldNew = <enum: PostSetAttr.ObjectMethod_OldNew [value=2]>

class atom.catom.**PostValidate**

Bases: atom.intenum.IntEnum

Delegate = <enum: PostValidate.Delegate [value=1]>

MemberMethod_ObjectOldNew = <enum: PostValidate.MemberMethod_ObjectOldNew [value=4]>

NoOp = <enum: PostValidate.NoOp [value=0]>

ObjectMethod_NameOldNew = <enum: PostValidate.ObjectMethod_NameOldNew [value=3]>

ObjectMethod_OldNew = <enum: PostValidate.ObjectMethod_OldNew [value=2]>

class atom.catom.**SetAttr**

Bases: atom.intenum.IntEnum

CallObject_ObjectNameValue = <enum: SetAttr.CallObject_ObjectNameValue [value=9]>

CallObject_ObjectValue = <enum: SetAttr.CallObject_ObjectValue [value=8]>

Constant = <enum: SetAttr.Constant [value=2]>

Delegate = <enum: SetAttr.Delegate [value=6]>

Event = <enum: SetAttr.Event [value=4]>

MemberMethod_ObjectValue = <enum: SetAttr.MemberMethod_ObjectValue [value=12]>

NoOp = <enum: SetAttr.NoOp [value=0]>

ObjectMethod_NameValue = <enum: SetAttr.ObjectMethod_NameValue [value=11]>

ObjectMethod_Value = <enum: SetAttr.ObjectMethod_Value [value=10]>

Property = <enum: SetAttr.Property [value=7]>

ReadOnly = <enum: SetAttr.ReadOnly [value=3]>

Signal = <enum: SetAttr.Signal [value=5]>

Slot = <enum: SetAttr.Slot [value=1]>

class atom.catom.**Validate**

Bases: atom.intenum.IntEnum

Bool = <enum: Validate.Bool [value=1]>

Bytes = <enum: Validate.Bytes [value=6]>

BytesPromote = <enum: Validate.BytesPromote [value=7]>

Callable = <enum: Validate.Callable [value=19]>

Coerced = <enum: Validate.Coerced [value=22]>

ContainerList = <enum: Validate.ContainerList [value=12]>

Delegate = <enum: Validate.Delegate [value=23]>

Dict = <enum: Validate.Dict [value=14]>

Enum = <enum: Validate.Enum [value=18]>

Float = <enum: Validate.Float [value=4]>

FloatPromote = <enum: Validate.FloatPromote [value=5]>

FloatRange = <enum: Validate.FloatRange [value=20]>

Instance = <enum: Validate.Instance [value=15]>

Int = <enum: Validate.Int [value=2]>

IntPromote = <enum: Validate.IntPromote [value=3]>

List = <enum: Validate.List [value=11]>

MemberMethod_ObjectOldNew = <enum: Validate.MemberMethod_ObjectOldNew [value=26]>

NoOp = <enum: Validate.NoOp [value=0]>

ObjectMethod_NameOldNew = <enum: Validate.ObjectMethod_NameOldNew [value=25]>

ObjectMethod_OldNew = <enum: Validate.ObjectMethod_OldNew [value=24]>

Range = <enum: Validate.Range [value=21]>

Set = <enum: Validate.Set [value=13]>

Str = <enum: Validate.Str [value=8]>

StrPromote = <enum: Validate.StrPromote [value=9]>

Subclass = <enum: Validate.Subclass [value=17]>

Tuple = <enum: Validate.Tuple [value=10]>

Typed = <enum: Validate.Typed [value=16]>

class atom.catom.**atomclist**

Bases: *atom.catom.atomlist*

append()
L.append(object) – append object to end

extend()
L.extend(iterable) – extend list by appending elements from the iterable

insert()
L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

class atom.catom.**atomdict**

Bases: dict

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F

class atom.catom.**atomlist**

Bases: list

append()
L.append(object) – append object to end

extend()
L.extend(iterable) – extend list by appending elements from the iterable

insert()
L.insert(index, object) – insert object before index

class atom.catom.**atomref**
Bases: object

class atom.catom.**atomset**
Bases: set

add()
Add an element to a set.

difference_update()
Update a set with the difference of itself and another.

intersection_update()
Update a set with the intersection of itself and another.

symmetric_difference_update()
Update a set with the symmetric difference of itself and another.

update()
Update a set with the union of itself and another.

atom.catom.**reset_property**()
Reset a Property member. For internal use only!

5.2.4 atom.coerced module

class atom.coerced.**Coerced**(*kind*, *args=None*, *kwargs=None*, *factory=None*, *coercer=None*)
Bases: *atom.catom.Member*

A member which will coerce a value to a given instance type.

Unlike Typed or Instance, a Coerced value is not intended to be set to None.

5.2.5 atom.containerlist module

class atom.containerlist.**ContainerList**(*item=None*, *default=None*)
Bases: *atom.list.List*

A List member which supports container notifications.

5.2.6 atom.delegator module

class atom.delegator.**Delegator**(*delegate*)
Bases: *atom.catom.Member*

A member subclass which delegates all work to a wrapped member.

The only behaviors not delegated are GetAttr and SetAttr. Subclasses should override behavior as needed to suit their needs. In order to change a particular mode, the relevant change method must be called via super(Delegator, ...).

add_static_observer(*observer*)
Add a static observer to the member.

This method also adds the static observer to the delegate.

clone()
Create a clone of the declarative property.

This method also creates a clone of the internal delegate for mode handlers which use the original delegate as the context.

delegate

remove_static_observer(*observer*)
Remove a static observer from the member.

This method also removes the static observer from the delegate.

set_default_value_mode(*mode*, *context*)
Set the default value mode for the member.

This method proxies the change to the delegate member.

set_index(*index*)
Assign the index to this member.

This method keeps the index of the delegate member in sync.

set_name(*name*)
Assign the name to this member.

This method keeps the name of the delegate member in sync.

set_post_getattr_mode(*mode*, *context*)
Set the post getattr mode for the member.

This method proxies the change to the delegate member.

set_post_setattr_mode(*mode*, *context*)
Set the post getattr mode for the member.

This method proxies the change to the delegate member.

set_post_validate_mode(*mode*, *context*)
Set the default value mode for the member.

This method proxies the change to the delegate member.

set_validate_mode(*mode*, *context*)
Set the default value mode for the member.

This method proxies the change to the delegate member.

5.2.7 atom.dict module

class atom.dict.**Dict**(*key=None*, *value=None*, *default=None*)
Bases: *atom.catom.Member*

A value of type *dict*.

clone()
Create a clone of the member.

This will clone the internal dict key and value members if they exist.

set_index(*index*)
Assign the index to this member.

This method is called by the Atom metaclass when a class is created. This makes sure the index of the internal members are also updated.

set_name(*name*)
Assign the name to this member.

This method is called by the Atom metaclass when a class is created. This makes sure the name of the internal members are also updated.

5.2.8 atom.enum module

class atom.enum.**Enum**(**items*)
Bases: *atom.catom.Member*

A member where the value can be one in a sequence of items.

added(**items*)
Create a clone of the Enum with added items.

Parameters ***items** – Additional items to include in the Enum.

Returns **result** – A new enum object which contains all of the original items plus the new items.

Return type *Enum*

items
A readonly property which returns the items in the enum.

removed(**items*)
Create a clone of the Enum with some items removed.

Parameters ***items** – The items to remove remove from the new enum.

Returns **result** – A new enum object which contains all of the original items but with the given items removed.

Return type *Enum*

5.2.9 atom.event module

class atom.event.**Event**(*kind=None*)
Bases: *atom.catom.Member*

A member which acts like a stateless event.

set_index(*index*)
A reimplemented parent class method.

This method ensures that the delegate index is also set, if a delegate validator is being used.

set_name(*name*)
A reimplemented parent class method.

This method ensures that the delegate name is also set, if a delegate validator is being used.

5.2.10 atom.instance module

class atom.instance.**ForwardInstance**(*resolve*, *args=None*, *kwargs=None*, *factory=None*)
Bases: *atom.instance.Instance*

An Instance which delays resolving the type definition.

The first time the value is accessed or modified, the type will be resolved and the forward instance will behave identically to a normal instance.

args

clone()
Create a clone of the ForwardInstance object.

default(*owner*)
Called to retrieve the default value.

This is called the first time the default value is retrieved for the member. It resolves the type and updates the internal default handler to behave like a normal Instance member.

kwargs

resolve

validate(*owner*, *old*, *new*)
Called to validate the value.

This is called the first time a value is validated for the member. It resolves the type and updates the internal validate handler to behave like a normal Instance member.

class atom.instance.**Instance**(*kind*, *args=None*, *kwargs=None*, *factory=None*)
Bases: *atom.catom.Member*

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

5.2.11 atom.list module

class atom.list.**List**(*item=None*, *default=None*)
Bases: *atom.catom.Member*

A member which allows list values.

Assigning to a list creates a copy. The orginal list will remain unmodified. This is similar to the semantics of the assignment operator on the C++ STL container classes.

clone()
Create a clone of the list.

This will clone the internal list item if one is in use.

item

set_index(*index*)
Assign the index to this member.

This method ensures that the item member index is also updated.

set_name(*name*)
Set the name of the member.

This method ensures that the item member name is also updated.

5.2.12 atom.property module

class atom.property.**Property**(*fget=None*, *fset=None*, *fdel=None*, *cached=False*)
Bases: *atom.catom.Member*

A Member which behaves similar to a Python property.

cached
Test whether or not this is a cached property.

deleter(*func*)
Use the given function as the property deleter.

This method is intended to be used as a decorator. The original function will still be callable.

fdel
Get the deleter function for the property.

This will not find a specially named _del_* function.

fget
Get the getter function for the property.

This will not find a specially named _get_* function.

fset
Get the setter function for the property.

This will not find a specially named _set_* function.

getter(*func*)
Use the given function as the property getter.

This method is intended to be used as a decorator. The original function will still be callable.

reset(*owner*)
Reset the value of the property.

The old property value will be cleared and the notifiers will be run if the new value is different from the old value. If the property is not cached, notifiers will be unconditionally run using None as the old value.

setter(*func*)
Use the given function as the property setter.

This method is intended to be used as a decorator. The original function will still be callable.

atom.property.**cached_property**(*fget*)
A decorator which converts a function into a cached Property.

Parameters **fget** (*callable*) – The callable invoked to get the property value. It must accept a single argument which is the owner object.

5.2.13 atom.scalars module

class atom.scalars.**Bool**(*default=False*, *factory=None*)
Bases: *atom.scalars.Value*

A value of type *bool*.

class atom.scalars.**Bytes**(*default=b”*, *factory=None*, *strict=False*)
Bases: *atom.scalars.Value*

A value of type *bytes*.

By default, unicode strings will be promoted to byte strings. Pass strict=True to the constructor to enable strict byte sting checking.

class atom.scalars.**Callable**(*default=None*, *factory=None*)
Bases: *atom.scalars.Value*

A value which is callable.

class atom.scalars.**Constant**(*default=None*, *factory=None*)
Bases: *atom.scalars.Value*

A value which cannot be changed from its default.

class atom.scalars.**Float**(*default=0.0*, *factory=None*, *strict=False*)
Bases: *atom.scalars.Value*

A value of type *float*.

By default, ints and longs will be promoted to floats. Pass strict=True to the constructor to enable strict float checking.

class atom.scalars.**FloatRange**(*low=None*, *high=None*, *value=None*)
Bases: *atom.scalars.Value*

A float value clipped to a range.

class atom.scalars.**Int**(*default=0*, *factory=None*, *strict=True*)
Bases: *atom.scalars.Value*

A value of type *int*.

By default, ints are strictly typed. Pass strict=False to the constructor to enable int casting for longs and floats.

class atom.scalars.**Range**(*low=None*, *high=None*, *value=None*)
Bases: *atom.scalars.Value*

An integer value clipped to a range.

class atom.scalars.**ReadOnly**(*default=None*, *factory=None*)
Bases: *atom.scalars.Value*

A value which can be assigned once and is then read-only.

class atom.scalars.**Str**(*default=”*, *factory=None*, *strict=False*)
Bases: *atom.scalars.Value*

A value of type *str*.

By default, bytes strings will be promoted to unicode strings. Pass strict=True to the constructor to enable strict unicode checking.

class atom.scalars.**Value**(*default=None*, *factory=None*)
Bases: *atom.catom.Member*

A member class which supports value initialization.

A plain *Value* provides support for default values and factories, but does not perform any type checking or validation. It serves as a useful base class for scalar members and can be used for cases where type checking is not needed (like private attributes).

5.2.14 atom.signal module

class atom.signal.**Signal**
Bases: *atom.catom.Member*

A member which acts similar to a Qt signal.

5.2.15 atom.subclass module

class atom.subclass.**ForwardSubclass**(*resolve*)
Bases: *atom.subclass.Subclass*

A Subclass which delays resolving the type definition.

The first time the value is accessed or modified, the type will be resolved and the forward subclass will behave identically to a normal subclass.

clone()
Create a clone of the ForwardSubclass object.

default(*owner*)
Called to retrieve the default value.

This is called the first time the default value is retrieved for the member. It resolves the type and updates the internal default handler to behave like a normal Subclass member.

resolve

validate(*owner*, *old*, *new*)
Called to validate the value.

This is called the first time a value is validated for the member. It resolves the type and updates the internal validate handler to behave like a normal Subclass member.

class atom.subclass.**Subclass**(*kind*, *default=None*)
Bases: *atom.catom.Member*

A value which allows objects subtypes of a given type.

Values will be tested using the *PyObject_IsSubclass* C API call. This call is equivalent to *issubclass(value, kind)* and all the same rules apply.

A Subclass member cannot be set to None.

5.2.16 atom.tuple module

class atom.tuple.**Tuple**(*item=None*, *default=()*)
Bases: *atom.catom.Member*

A member which allows tuple values.

If item validation is used, then assignment will create a copy of the original tuple before validating the items, since validation may change the item values.

clone()
Create a clone of the tuple.

This will clone the internal tuple item if one is in use.

item

set_index(*index*)
Assign the index to this member.

This method ensures that the item member index is also updated.

set_name(*name*)
Set the name of the member.

This method ensures that the item member name is also updated.

5.2.17 atom.typed module

class atom.typed.**ForwardTyped**(*resolve*, *args=None*, *kwargs=None*, *factory=None*)
Bases: *atom.typed.Typed*

A Typed which delays resolving the type definition.

The first time the value is accessed or modified, the type will be resolved and the forward typed will behave identically to a normal typed.

args

clone()
Create a clone of the ForwardTyped instance.

default(*owner*)
Called to retrieve the default value.

This is called the first time the default value is retrieved for the member. It resolves the type and updates the internal default handler to behave like a normal Typed member.

kwargs

resolve

validate(*owner*, *old*, *new*)
Called to validate the value.

This is called the first time a value is validated for the member. It resolves the type and updates the internal validate handler to behave like a normal Typed member.

class atom.typed.**Typed**(*kind*, *args=None*, *kwargs=None*, *factory=None*)
Bases: *atom.catom.Member*

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_TypeCheck* C API call. This call is equivalent to *type(obj) in cls.mro()*. It is less flexible but faster than Instance. Use Instance when allowing heterogenous values and Typed when the value type is explicit.

The value of a Typed may be set to None

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V