importjson Documentation

Release 0.2

Tony Flury

May 27, 2019

Contents

1 Installation **3**

2 Getting Started **5**
- 2.1 Json Format . . . 5
- 2.2 Search Path . . . 6
- 2.3 Example Json . . . 6
- 2.4 Importing the JSON file . . . 6
- 2.5 Imported Module Content . . . 7

3 Constraints **11**
- 3.1 Constraints with Inheritance . . . 12
- 3.2 Extending constraints . . . 13

4 Details **15**
- 4.1 1. Module Configuration . . . 15
- 4.2 2. JSON file structure . . . 15
- 4.3 3. Top Level content . . . 15
- 4.4 4. Content of `__classes__` dictionary . . . 16
- 4.5 5. Content of a class defining dictionary . . . 16
- 4.6 6. Content of the `__class_attributes__` dictionary . . . 16
- 4.7 7. Content of the `__constraints__` dictionary . . . 16

5 repr and str format **19**
- 5.1 Customising repr . . . 19
- 5.2 Customising str . . . 20
- 5.3 Format String attrributes . . . 20

6 Introspection **23**
- 6.1 Introspection objects . . . 23
- 6.2 Module level Introspection . . . 24
- 6.3 Class Level Introspection . . . 24

7 Extra Information **25**
- 7.1 Notes and Comments . . . 25
- 7.2 Shortcomings . . . 25
- 7.3 Future . . . 26
- 7.4 Explicit Classes . . . 26

It is sometimes useful to be able to use json data to initialise classes and other data structures, giving your application a portable and human readable configuration capability. To do this you will probably write some level of functionality around the json standard library, and use the resulting data loaded from the json file, to populate classes and instances implemented in your application. This separates your data and functionality, which can often present challenges later down the line as you need to keep the data and functionality in step. It would be better in many cases to be able to combine the data and functionality in a single place, and with the importjson library you can do that.

The library allows you to import a json file direct into your python application, and automatically build a real python module, complete with classes, class attributes, and instance data attributes (implemented with set and get descriptors).

Your code can use these classes, attributes and methods just as if you have written the code yourself.

The importjson library also allows you to set constraints on your instance attributes, checking for the data type and simple range checks on the values your attempt to set when you create instances of the classes. You can also determine whether attributes are read only, or whether they will be allowed to be set to None (or not).s if you had written the code yourself.

Note: Every care is taken to try to ensure that this code comes to you bug free. If you do find an error - please report the problem on :

- GitHub Issues
- By email to : Tony Flury

CHAPTER 1

Installation

Installation is very simple :

```
$ pip install py-importjson
```

There is no further configuration or setup required - by default the library can now be used in your python application.

To start using the *importjson* library in your python code - simply import the library before you try to import your json file

```
>>> import importjson
```

You can now import json files - as described in *Getting Started*, and *Details*

Once imported (and before you import json files) there are some configurations you can change to change how the imports work - see *1. Module Configuration* for more details. These configurations are optional.

CHAPTER 2

Getting Started

2.1 Json Format

By default `importjson` creates the module according to some simple rules (see *Details* for exact details on the required format for the json) :

- The json file must define a dictionary - i.e. the first and last characters must be a `{` and `}` respectively
- name values in the top-level dictionary are converted to module data attributes
- sub-dictionaries in the top-level dictionary are converted to classes - the name of the dictionary becomes the name of the class
- name value pairs in the class dictionaries are converted to instance data attributes with a few expections :
  - a name of `__doc__` can be used to define the documentation string for the class
  - a name of `__parent__` can be used to define that the class is subclassed from another class within the json file.
  - a name of `__class_attributes__` can be used to define class data attributes (rather than instance attributes). name/value pairs in the `__class_attributes__` dictionary are converted to class data attributes with the appropriate name and starting value.
  - a name of `__constraints__` can be used to define type, range and other constraints for the instance data attributes (See *Constraints* for details on how to specify constraints)
  - a name of ‘’__repr__” can be used to define a customised repr format, and the name of ‘’__str__” can be used to define a default str format - see *repr and str format* for more details.
- within a class dictionary - any name/value pairs where the value is a dictionary is defined as instance data variable with a default value of a dictionary.

You can define multiple classes per json file.

2.2 Search Path

The `importjson` library will look for json files across all directories and files specified in `sys.path`, ie. the same search path as normal python modules. With the `importjson` library in use it will take any attempted import first and try to find a relevant JSON file to import, and only if it is unable to find a JSON file of the appropriate name to import in any `sys.path` entry will it then hand over to the default python import to search for .py or .pyc files. This means that for instance if you have `classes.json` and `classes.py` in the same directory or package and your code either implicitly or explicitly imports `classes`, then classes.json file will be found first and will be imported and the classes.py file will be effectively hidden, and cannot be imported. This will cause unexpected behaviour unless you are careful about your file naming.

2.3 Example Json

Using the following json file as an example to illustrate the key features of the importjson library:

Place the json file called `example.json` exists in your applications current directory

```
{
    "__version__":"0.1",
    "__author__":"Tony",
    "point": {
        "__doc__": "Example class built from json",
        "__class_attributes__": {
                          "_val1": 1,
                          "_val2": 2
                        },
        "x": 0,
        "y": 0,
        "colour": [0,0,0]
         }
}
```

2.4 Importing the JSON file

Given a valid json file (such as the above file), the importjson library can be used to import the json file and create a module including attributes, classes, and attributes on those classes.

2.4.1 Using import command

Importing this json file is easy :

```
>>> import importjson        # Importjson library - must be imported before any json␣
↪files
>>> import example           # Will import example.json
```

If a classes.json is found the above import will try to read the JSON and convert it following the rules described above. It it fails (due to permisssions, or malformed JSON for instance), and `ImportError` exception will be raised. Assuming though that the above import works, with the JSON example above, then a python module is created, and can be used as any normal module:

2.4.2 Using importlib library

A json 'module' can also be imported using the importlib mechanism :

```
>>> import importlib
>>> import importjson
>>> example = importlib.import_module('example')
>>> dir(example.Point)
['ClassAttributeInfo', 'ClassInfo', 'InstanceAttributeInfo', 'ModuleAttributeInfo',
'__author__', '__builtins__', '__doc__', '__file__', '__json__', '__loader__',
'__name__', '__package__', '__version__', 'get_attributes', 'get_classes',
'namedtuple', 'point', 'six']
```

2.5 Imported Module Content

2.5.1 Module Data Attributes

Using the json file above as an example, and importing it using either method, the imported module contains a number of attributes :

```
>>> # Module attributes
>>> example.__author__, example.__version__
u'Tony', u'0.1'
```

As per the json implementation in the python standard library, all strings are treated as unicode.

By default the module has a auto generated documentation string

```
>>> print example.__doc__
Module classes - Created by JSONLoader
   Original json data : /home/tony/Development/python/importjson/src/classes.json
   Generated Mon 12 Oct 2015 22:30:54 BST (UTC +0100)
```

```
>>> dir(example)
['ClassAttributeInfo', 'ClassInfo', 'InstanceAttributeInfo', 'ModuleAttributeInfo',
'__author__', '__builtins__', '__doc__', '__file__', '__json__', '__loader__',
'__name__', '__package__', '__version__', 'get_attributes', 'get_classes', 'namedtuple
↪', 'point', 'six']
```

As can be seen from the `dir` listing above there are a number of special module variables :

- `__builtins__` : as per all modules this is the standard python builtins modules
- `__doc__` : as demonstrated above this is the module documentation string (either the auto generated or defined in the json file).
- `__file__` : this is the full path to the json file - in a normal module this would be the path of the `.py` or `.pyc` file
- `__json__` : the original json file imported as a dictionary. It is included for interest only, it should not ever be necessary to use the data in this dictionary (as it has all been converted to the specific module data attributes, classes and other content).
- `__loader__` : This is the custom loader object (which the importjson library implements).
- `__name__` : As with all other modules - this is the fully qualified module name.

- `__package__` : This is False, as the json file cannot ever define a package
- `ClassAttributeInfo`, `ClassInfo`, `InstanceAttributeInfo`, `ModuleAttributeInfo` are all introspection objects - these may not exist in future versions.
- `get_attributes` and `get_classes` : both functions related to introspection
- `namedtuple`, `six` : The code produced by importjson currently depends on the amedtuple and six libraries: this may change in future versions.

In the above output the `__version__` and `__author__` variables are not special variables - as they are defined by the json file.

2.5.2 Classes

The `point` dictionary in the example json file will have been converted to the `example.point` class.

The classes which are created have all the properties you might expect - for instance as defined by the `__doc__` and the `__class__attributes__` dictionary in the json file we can define class data attributes (see *Details* for details)

```
>>> example.point._val1
1
>>> example.point._val2
2
>>> example.point.__doc__
'Example class built from json'
```

2.5.3 Creating Instances

There is nothing special about these classes, instances of these classes can be created in just the same way as other classes.

Instances which are created from these classes have the expected Instance data attributes with default values derived from the relevant entries in the json. Instance Data Attributes can be retrieved by name (as expected).

```
>>> inst = example.point()
>>> inst.x, inst.y, inst.colour
0, 0, [0, 0, 0]
```

2.5.4 Instance Initialiser

The class initialiser accepts both keyword and position arguments; if positional arguments are used the arguemnts appear in the order that they are defined within the JSON file.

```
>>> insta = classes.point(0, 1)
>>> insta.x, insta.y, insta.colour
0, 1, [0, 0, 0]
```

Arguments to the initializer can be keyword arguments too - using the same names in the json file.

```
>>> instb = classes.point(colour=[1,1,1])
>>> instb.x, instb.y, instb.colour
0, 0, [1, 1, 1]
```

Instance Data attributes are implemented as data descriptors, and so attributes are accessed using the normal dot syntax :

```
>>> insta.x = 23
>>> insta.x, insta.y, insta.colour
23, 0, [0,0,0]
```

See also:

- Detailed Specification of the JSON format : *Details*
- Disovering what python classes and attributes have been imorted *Introspection*
- Type and range checking of Instance Data Attributes : *Constraints*
- Customised repr and str formatting : *repr and str format*
- Known Issues and Gotchas : *Shortcomings*

CHAPTER 3

Constraints

It is possible to define constraint criteria for the Instance Data Attributes, by using a `__constraints__` sub dictionary within the class definition - as an example :

```
{
    "point": {
        "x": 0,
        "y": 0,
        "__constraints__": {
            "x":{
                "type":"int",
                "min":-100,
                "max":100
                }
            }
    }
}
```

This would implement a definition of the `x` attribute on instances of the `point` class could only ever be set to an integer (or boolean), and must between -100 and 100 inclusive. The allowed criteria are `type`, `min`, `max`, `read_only` and `not_none`. The‘‘type‘‘ can be any one of `list`, `str`, `int`, `float`, `dict` or `bool` or the name of a class which is also defined in the JSOn file.

- A `type` of `float` will allow both floats and integer values
- A `type` of `int` will allow both integers and booleans values
- A `type` of `bool` will only allow either True or False values
- If the constraint of `not_none` is True, a `ValueError` will be raised if any attempt is made to assign a `None` value to the attribute which is not None. For lists and dictionaries an empty list or dict is not the same as a `None` value.
- If the constraint of `read_only` is True, a `ValueError` will be raised if an attempt is made to assign the attribute (other than the assignment made during initialisation).

- If an attempt is made to set an attribute to a value outside the range defined by `min` and `max`, a `ValueError` exception will be raised.
- If an attempt is made to set an attribute to a value which does not match the type criteria, then a `TypeError` exception will be raised.
- All criteria are optional - but an empty or missing constraints section has no effect (and specifically `not_none`, and `read_only` default to False when omitted)

Warning: You must ensure that the constraints for each instance attribute are self consistent, and don't contradict the specified default value for that attribute. The constraints section is not validate at the time of import, but if the constraints are wrong, or non-consistent then there will be exceptions raised during instance initialisation or other attribute assignment.

3.1 Constraints with Inheritance

when one class inherits from another, and both define constraints, then the constraints are applied in order (with the superclass constraints applied first, and so on through the list of subclasses). This has the effect that the most restrictive constraint will be applied.

As an example :

```
{
    "ClassA":{
            "a1":1,
        "__constraints__":{
                "a1":{
                    "min":-5,
                    "max":5
                    }
                }
            },
    "ClassB":{
        "__parent__":"classa",
        "a1":2,
        "__constraints__":{
                "a1":{
                    "min":-2,
                    "max":2
                    }
                }
            }
}
```

The JSON definition above is for two classes - `ClassA` and `ClassB` (which is a sub class of `ClassA`). On instances of `ClassA` the attribute `a1` can be set to any value between -5 and 5, whereas on instances of `ClassB` the same attribute is restricted to values between -2 and 2.

A more interesting example can be generated by this JSON file :

```
{
    "Class1":{
            "x":1,
        "__constraints__":{
                "x":{
```

(continues on next page)

(continued from previous page)

```
                    "min":0
                    }
                }
            },
    "Class2":{
        "__parent__":"classa",
        "x":2,
        "__constraints__":{
                "x":{
                    "max":6
                    }
                }
            }
}
```

The JSON definition above is for two classes - `Class1` and `Class2` (which is a sub class of `Class1`). On instances of `Class1` of the attribute `x` can be set to any value greater or equal to zero, whereas on instances of `ClassB` the `x` is restricted to values between 0 and 6 inclusive (even though `Class2` does not define a minimum constraint, the constraints defined on `Class1` are also applied).

3.2 Extending constraints

The constraints system has been constructed to allow simple extensions. By subclassing the class, and creating a method on the subclass of `_constrain_<attr_name>(value)` you can add further constraints to the named attribute (e.g. to extend the constraints testing of the `classes.point.x` attribute, your code should sub class `classes.point` and implement a method `_constrain_x(value)`).

constrain<attr_name>(self, value)

Implements constraints for the attribute <attr_name>.

If you need to access the existing current value of the attribute you can simply use `self.<attr_name>`.

param value The attempted new value for this attribute - i.e. the value to be validated

return The value if valid (there is nothing to stop the method from changing the value although that isn't recommended)

raises ValueError raised if the value of the `value` argument is not valid for that attribute

raises TypeError raised if the type of the `value` argument is not valid for that attribute

As shown in the example any extension should ideally call the `<super class> _constrain` method first, as it is that method which applies all of the constrains defined in the JSON file - including any type checks. By allowing the superclass method to execute first, you can be sure that the value returned is the expected type (assuming that the JSON file constrains the type).

3.2.1 Extending constraints example

As an example :

Listing 1: json_classes.json (in a top level directory)

```
{
   "classa":{
      "x":0,
```

(continues on next page)

(continued from previous page)

```
        "__constraints__":{
            "x":{
                "type":"int",
                "min":0,
                "max":1024
            }
        }
    }
}
```

Listing 2: Extending the json_classes.classa to constrain x attribute to be even only

```
>>> import importjson
>>> import json_classes # As above
>>>
>>> class XMustBeEven(json_classes.classa):
...     def _constrain_x(self, value):
...             value = super(XMustBeEven,self)._constrain_x(value)
...
...             if value % 2 == 0:
...                 return value
...             else:
...                 raise ValueError("Value Error : x must be an even number")
>>>
>>> e = XMustBeEven()
>>> e.x = 2 # will be fine - no exceptions expected
>>> e.x = 3
Value Error : x must be an even number
```

CHAPTER 4

Details

4.1 1. Module Configuration

The importjson module supports one configuration options, set using *importjson.configure(<config_item>,<value>)*. The config_items supported are :

- `JSONSuffixes` : A list of valid JSON file name suffixes which are used when searching for potential JSON files to import. The default is [".json"]. Setting this value incorrectly will prevent the library from finding or importing any JSON files - so take care.

A previous configuration item `AllDictionariesAsClasses` has been rendered obsolete due to changes in *0.0.1a5* and a exception is raised if this item is attempted to be used.

4.2 2. JSON file structure

The json file must be in a specific format :

The Top level of the json file **must** be a dictionary - ie it must start with `{` and end with `}` - see *3. Top Level content* for details.

4.3 3. Top Level content

All name, value pairs in the top level are created as module level attributes (see example of `__version__` above) with the following notes and exceptions:

- An optional name of `__doc__` is found then the value is used as the module documentation string instead of an automatically generated string. While it is normal that the value is a string if a different object is provided the documentation string will be set to the string representation of that object.
- Within the top level dictionary, a name of `__classes__` is optional :

- If an json object with the name of `__classes__` does **not** exist: all dictionaries under the Top Level areas are used to define the classes in this module - see see *5. Content of a class defining dictionary*. Although this form of JSON is more 'natural', in this case it is not possible to define a Module Data Attribute with a dictionary value.
- If an json object with the name of `__classes__` does exist: the content of this dictionary are used as the definitions of the classes in this module - see *4. Content of __classes__ dictionary*. In this case any other dictionary under the Top Level JSON is treated as a Module Data Attributes whose initial value is a dictionary.

4.4 4. Content of `__classes__` dictionary

When the `__classes__` dictionary exists in the json file, each key,value within that dictionary is a separate class to be created. The key is the class name, and the value must be a dictionary (called the class defining dictionary) - see section 4. An example of this form of JSON file is used above.

4.5 5. Content of a class defining dictionary

Within the class defining dictionary, each key,value pair is used as instance attributes; the value in the json file is used as the default value for that attribute, and is set as such in the initializer method for the class. This is true for all key,value pairs with the following notes and exceptions:

- An optional key of `__doc__` will set the documentation string for the class - unlike at module level there is no automatically generated documentation string for the class. While it is normal that the value is a string if a different object is provided the documentation string will be set to the string representation of that object
- An optional key of `__class_attributes__` will have the value which is a dictionary : This dictionary defines the names and values of the class data attributes (as opposed to the instance data attributes) - see *6. Content of the __class_attributes__ dictionary*
- An optional key of `__parent__` will have a string value which is used as the name of a superclass for this class.
- An optional key `__constraints__` which will have a dictionary value - and define constraint to be applied to the value of individual Instance Data Attributes - see *7. Content of the __constraints__ dictionary*

4.6 6. Content of the `__class_attributes__` dictionary

Within the `__class_attributes__` dictionary each key, value pair defines the name and value of a class data attribute. There are no exceptions to this rule.

4.7 7. Content of the `__constraints__` dictionary

Within the `__constraints__` dictionary each key is the the name Instance Data attribute, as defined within the class defining dictionary. It is not neccessary for every Instance Data Attribute to be represented by a key in the `__constraints__` dictionary.

Each key has the value of a dictionary, and this dictionary has zero or more keys within it (every key being optional) :

- *type* : Can be used to constrain the type of value allowed for the attribute
  - *list* : constrains the type to be a list (the values of the items are not restricted)

  - *str* : constrains the type to be a string or basestring
  - *int* : constrains the type to be a integer or boolean
  - *float* : constrains the type to be a float or integer
  - *dict* : constrains the type to be a dictionary (keys and values are not restricted)
  - *bool* : constrains the type to be boolean (i.e. True or False Only)
  - Any other value must be the name of a class defined in the JSON file.
- *min* : Constrain the minimum value allowed for the attribute - applied to strings and numeric values only
- *max* : Constrain the maximum value allowed for the attribute - applied to strings and numeric values only
- *not_none* : determines if the value is allowed to be a None value
- *read_only* : determine if the value can be changed after the instance is created

If an attempt is made to set an attribute to a value outside the range defined by *min* and *max* the `ValueError` exception will be raised. This include setting the value within the Instance initializer.

If an attempt is made to set an attribute to a value which does not match the type criteria, then a `TypeError` exception will be raised. This includes setting the value within the Instance initializer.

If an attempt is made to set an attribute to None when *not_none* is set to True, a `ValueError` exception will be raised. This value defaults to false - i.e. None values are allowed.

- If an attempt is made to set an attribute when *read_only* is set to True, a `ValueError` exception will be raised. This does not include setting the attribute in the initialiser/constructor. This value defaults to False, i.e. attributes can be changed at any time.

All criteria are optional - an empty or missing constraints section for a given attribute has no effect.

Warning: Since the constraints are applied every time the value is set, including the initializer, you must ensure that the default value given for the data attribute is valid based on any constraints defined for that attribute. If the default value is invalid, then the JSON will import successfully, but class instances will not be able to be created with it's default values. The values in the constraints section are not cross checked currently at the time of import, and any errors (such as incorrect numeric ranges or invalid types) will only be detectable when instances are created. It is relatively simple though to change the json file and reload the module.

CHAPTER 5

repr and str format

within a class definition, you can define a customised repr and str format for this class.

the default formatting is such that the repr string of an instance is a string representation of the constructor call - as an example :

With a class defined with this json fragment :

```
{
    "point": {
        "x": 0,
        "y": 0,
        "colour": [0,0,0]
         }
}
```

then we can see the following result :

```
>>> p = Point(x=10,y=-10,colour=[1,0,0])
>>> repr(p)
"Point(x=10, y=-10, colour=[1,0,0])"
>>> str(p)
"Point(x=10, y=-10, colour=[1,0,0])"
```

As seen by default the str response is the same as the repr response.

5.1 Customising repr

The repr format can be customised by defining the '__repr__' key within the json file; the value for this key must be a string. This string can contain placeholders for the class and instance attributes, as well as the class name :

As an example the repr for the above class is equivalent to defiiing the '__repr__' format as follows :

```
{
    "point": {
        "x": 0,
        "y": 0,
        "colour": [0,0,0],
         "__repr__":"{class_name}(x={x}, y={y}, colour={colour})"
         }
}
```

note the `{class_name}` placeholder within the format string for the class name (in this case 'point').

As well as the placeholders, the format string can contain all of the formatting in a valid python format string

5.2 Customising str

By default the str of an instance is the same as the repr - this is default behaviour for all python classes and the importjson module does not change this.

A customised str format can be provided within the json definition by using a *__str__* key within the json (in a smiliar fashion to the *__repr__* format above.

```
{
    "person": {
        "first_name": "John",
        "last_name": "Smith",
        "birth_place": "London",
         "__str__":"{first_name} {last_name} born in {birth_place}"
         }
}
```

defines a class so that :

```
>>> p = Person(first_name='Michael', last_name='Palin', birth_place='Sheffield')
>>> repr(p)
"Person(first_name='Michael', last_name='Palin', birth_place='Sheffield')"
>>> str(p)
"Michael Palin born in Sheffield"
```

As you can see the repr result is the default described above, while the str result is now customised.

5.3 Format String attrributes

Both the repr and str formats support field names in the format strings for all of the class and instance attributes by name, as well as the `module_name` and `class_name` field names for the name of the module and class respectively.

The use of those field names includes accessing the items within attributes which are lists and dictionaries, and attributes can even be used as field fill and alignment values for other fields - as an example :

```
{
    "formatter": {
        "words": ["Monty", "Python"],
        "fill": "",
        "align": "",
        "width":"",
```

(continues on next page)

(continued from previous page)

```
        "__str__":"{words[0]:{fill}{align}{width}} {words[1]}"
        }
}
```

```
>>> p = formatter()
>>> str(p)
"Monty Python"
>>> p.width=10
>>> str(p)
"Monty      Python"
>>> p.align='^'
>>> p.fill='~"'
>>> str(p)
'~~Monty~~~ Python'
```

CHAPTER 6

Introspection

Sometimes there will be a need to discover what attributes and classes are defined in any imported json module.

Starting from vs 0.1.2 the *importjson* module does provide a comprehensive set of functions and methods for introspection of the contents of the python module generated by the imported json file.

6.1 Introspection objects

object	fields
ModuleAttributeInfo	name - The name of the attribute default - The default value of the attribute
ClassInfo	name - The name of the Attribute cls_ - The actual class object of this name parent - The name of the parent class (or 'object')
ClassAttributeInfo	name - The name of the attribute default - The default value of the attribute
InstanceAttributeInfo	name - The name of the attribute default - The default value of the attribute

6.2 Module level Introspection

The imported module has two functions:

<module>.get_attributes() A generator which will yield one or more `ModuleAttributeInfo` objects. The following code can be used to print the names of all module level fields :

```
import importjson
import jsonmodule

for attribute in jsonmodule.get_attributes():
    print( attribute.name )
```

<module>.get_classes() A generator which will yield one or more `ClassInfo` objects. The following code can be used to print the names of all module level fields :

```
import importjson
import jsonmodule

for class_info in jsonmodule.get_classes():
    print( class_info_.name )
```

6.3 Class Level Introspection

Each class is provided with two introspection class methods :

<class>.get_class_attributes() A generator which will yield one or more `ClassAttributeInfo` objects. The following code can be used to print the names of all class attributes of all classes in a module:

```
import importjson
import jsonmodule

for class_info in jsonmodule.get_classes():
    for attribute in class_info.cls_.get_class_attributes():
        print(attribute.name)
```

<class>.get_instance_attributes() A generator which will yield one or more `InstanceAttributeInfo` objects. The following code can be used to print the names of all instance attributes of all classes in a module:

```
import importjson
import jsonmodule

for class_info in jsonmodule.get_classes():
    for attribute in class_info.cls_.get_instance_attributes():
        print(attribute.name)
```

CHAPTER 7

Extra Information

This section contains some information that might be useful, or which might trip you up, and also some musings about the future.

7.1 Notes and Comments

1. Instance data attributes are actually created with the name prefixed by a ‘‘_‘‘, thus marking the attribute as private. A read/write descriptor is then created with the name as given in the json file.
2. If the json defines Instance data attribute with a default value which is a mutable type (list or dictionary), the initializer ensures that changes to the instance are not propagated to other instances. See Common Python Gotchas for a description of this issue. There are no plans to allow this protection to be turned off.
3. All strings are imported as Unicode - as can be seen from the ‘‘**__version__**‘‘ example above.
4. The module works by creating a python code block which is then compiled into the module and made available to the application. That code block is available for information : ‘‘**<module>.__loader__.get_source(<module_name)**‘‘ - while the json file is available through the ‘‘**__file__**‘‘ module attribute, and the imported dictionary can be seen by inspecting ‘‘**__json__**‘‘ module attribute. Under normal circumstance it should not be necessary to use either the json dictionary or the generated code.

7.2 Shortcomings

1. It is not possible to use json to define tuples, sets or other complex python data types. json only supports strings, lists, numbers and dictionaries. This is not a limitation of the importjson library, and cannot be fixed easily.
2. It is not possible to set a documentation string for any of the instance data attributes - see Futures
3. Keys in the ‘‘**__constraints__**‘‘ section of each class are lower case only.

7.3 Future

Possible future enhancements :

- Auto generation of factory methods, using a specific attribute as the key
- Documentation strings for the Instance Data Attributes
- Keys in **‘‘__constrains__‘‘** section should be case insensitive
- validity of **‘‘__constrains_‘‘** items should be performed at import time.

7.4 Explicit Classes

Note : From v0.0.1a5 onwards the example JSON used at the top of this README could be changed to be as follows :

```
{
    "__version__":"0.1",
    "__author__":"Tony",
    "__classes__":{
        "point": {
            "__doc__": "Example class built from json",
            "__class_attributes__": {
                                    "_val1": 1,
                                    "_val2": 2
                                   },
            "x": 0,
            "y": 0,
            "colour": [0,0,0]
        }
    }
}
```

Note the existence of the “__classes__” dictionary. This form is termed as the **explicit** form. The advantage of this form is that it is possible to define Module Data Attributes which are dictionaries, something which impossible in the other form of json.

Index

A

attributes
    module data, 7

C

classes, 8
    instances, 8

D

data
    attributes, module, 7

E

example
    json, 6

F

format
    json, 5

I

import
    json, 6
Instance Initialiser
    keyword argument, 8
    positional argument, 8
instances
    classes, 8

J

json
    example, 6
    format, 5
    import, 6

K

keyword argument
    Instance Initialiser, 8

M

module
    data attributes, 7

P

path
    search, 5
positional argument
    Instance Initialiser, 8

S

search
    path, 5