2025/12/21 09:48 1/6 thinknode™ Examples

# thinknode™ Examples

These examples provide a starting point for issuing http connections and requests to the dosimetry app on the thinknode™ framework. They are provided as is, and are written in python. Any further dependencies are listed along with the provided scripts.

# **Python: decimal Libraries**

### rt\_types

The *rt\_types* module is a reconstruction of all astroid types in python class format. This includes interdependencies between types (e.g. the class "polyset" requires the class "polygon2").

Each data type detailed in the astroid Manifest Guide has a corresponding class in this python module.

Below you will see a snippet from the rt\_types module that shows the class for the *polyset* rt\_type along with its default initialization, *expand data* and *from json* functions.

```
class polygon2(object):
    #Initialize
    def init (self):
        blob = blob type()
        self.vertices = blob.toStr()
    def expand data(self):
        data = {}
        data['vertices'] =
parse bytes 2d(base64.b64decode(self.vertices['blob']))
        return data
    def from_json(self, jdict):
        for k, v in jdict.items():
            if hasattr(self,k):
                setattr(self, k, v)
class polyset(object):
    #Initialize
    def init (self):
        self.polygons = []
        self.holes = []
```

2025/12/21 09:48 2/6 thinknode™ Examples

```
def expand data(self):
    data = \{\}
    polygon = []
    for x in self.polygons:
        s = polygon2()
        s.from json(x)
        polygon.append(s.expand data())
    data['polygons'] = polygon
    hole = []
    for x in self.holes:
        s = polygon2()
        s.from json(x)
        hole.append(s.expand data())
    data['holes'] = hole
    return data
def from json(self, jdict):
    for k, v in jdict.items():
        if hasattr(self,k):
            setattr(self, k, v)
```

- **Interdependence:** When rt\_types are constructed of other or multiple named types, they will be constructed as such in each class as seen in the *polygons* parameter of the *polyset* in the above example.
- expand\_data function: Each class's expand\_data function returns a python dictionary containing
  each of the values in the class, with all data values expanded out to remove compression or other
  encodings (i.e. providing results in a format more useful for send to other applications or for
  human-readability).
- **from json function:** Each class's *from json* function provides a method to turn a raw json string (e.g. a result from a thinknode calculation or ISS object) into an rt\_type data type. Proper use is to first construct an empty class instance, then to call the *from json* method on that instance, passing in the desired json data string.

Below is an example usage of getting a thinknode dose image (image\_3d data type in the astroid manifest) and turning it into a rt\_types image\_3d data type, so that it can be expanded and then used to output the image into a VTK graphics file:

```
def dose_to_vtk(dose_id):
    img_data = json.loads(thinknode.get_immutable(iam, 'dicom', dose_id))

img = rt_types.image_3d()
    img.from_json(img_data)
    img2 = img.expand_data()

vtk.write_vtk_image3('E:/dicom/dose.vtk', img2)
```

2025/12/21 09:48 3/6 thinknode™ Examples

## thinknode worker

The thinknode\_worker module is the main work horse for communication with the astroid app and thinknode. The module will handle authentication, posting objects to ISS, creating most of the common calculation request structures, and posting the calculation request.

Refer to the .decimal GitHub repository for the complete module. Below are a few of the more common thinknode\_worker functions and their intended usages:

```
# Authenticate with thinknode and store necessary ids.
# Gets the context id for each app detailed in the thinknode config
# Gets the app version (if non defined) for each app in the realm
    param config: connection settings (url and unique basic user
authentication)
def authenticate(config):
# Send calculation request to thinknode and wait for the calculation to
perform. Caches locally calculation results so if the same
# calculation is performed again, the calculation
# does not have to be repeatedly pulled from thinknode. Saves one calculation
time and bandwidth.
   note: see post calculation if you just want the calculation ID and don't
need to wait for the calculation to finish or get results
   param config: connection settings (url, user token, and ids for context
and realm)
   param ison data: calculation request in ison format
   param return data: When True the data object will be returned, when false
the thinknode id for the object will be returned
   param return error: When False the script will exit when error is found,
when True the sciprt will return the error
def do calculation(config, json data, return data=True, return error=False):
# Post immutable named type object to ISS
   param config: connection settings (url, user token, and ids for context
and realm)
   param app name: name of the app to use to get the context id from the iam
config
   param json data: immutable object in json format
    param obj name: object name of app to post to
def post immutable named(config, app name, json data, obj name):
    scope = '/iss/named/' + config["account name"] + '/rt types' + '/' +
obj name
    return post_immutable(config, app_name, json_data, scope)
# Post immutable object to ISS
    param config: connection settings (url, user token, and ids for context
and realm)
```

2025/12/21 09:48 4/6 thinknode™ Examples

```
# param app_name: name of the app to use to get the context id from the iam
config
# param obj_id: thinknode iss reference id for object to get
def get_immutable(config, app_name, obj_id):
```

# dosimetry worker

The dosimetry\_worker module provides high-level functions for building data types and calculation requests for common dosimetry tasks. This library is constantly growing as more routine tasks are programmed in python.

Refer to the .decimal GitHub repository for the complete module. Some basic examples of provided functionality are:

- 1. Aperture creation (using structures/beams or basic geometric)
- 2. Dose comparison
- 3. Grid creation
- 4. Image creation
- 5. PBS Spot functions

## vtk worker

The VTK worker provides a means to write out common rt\_types to a vtk file format (The Visualization TooKit) that can be visualized in Paraview. It's most useful for displaying and post-processing image, mesh, and other primitive object data types.

Below is an example of turning a dose image 3d into a vtk file for visualization in Paraview:

```
def dose_to_vtk(dose_id):
    img_data = json.loads(thinknode.get_immutable(iam, 'dicom', dose_id))

img = rt_types.image_3d()
    img.from_json(img_data)
    img2 = img.expand_data()

vtk.write_vtk_image3('E:/dicom/dose.vtk', img2)
```

# decimal\_logging

The decimal logging module provides formatted and detailed output window messages and file logging.

The following settings are available in the decimal\_logging.py file: **display\_timestamps:** display timestamps in the output window/logfile **display\_types:** display message types (e.g. debug, data, alert) in the output window/logfile **log file:** sets the logfile name and location

2025/12/21 09:48 5/6 thinknode™ Examples

#### **Debugging**

When debugging, use the dl.debug() function and set the *isDebug* flag in the decimal\_logging library to True. This toggles on the output for each of the dl.debug calls. By default we keep debugging off, but it can be turned on as needed.

### **Other Flags**

The following image shows the logging settings for each message type as:

```
    Timestamps = True; Types = True
    Timestamps = False; Types = True
    Timestamps = False; Types = False
```

```
2015-09-28 11:43:14 -- MESSAGE: decimal message
2015-09-28 11:43:14 -- DEBUG: >>> decimal debug <<<
2015-09-28 11:43:14 -- ALERT: decimal aler
2015-09-28 11:43:14 !! WARNING: decimal warning
2015-09-28 11:43:14
2015-09-28 11:43:14 -- EVENT: decimal event
2015-09-28 11:43:14 -- DATA: decimal debug_data
-- MESSAGE: decimal message
-- DEBUG: >>> decimal debug <<<
-- ALERT: decimal alert
!! WARNING: decimal warning
 -- DATA: decimal debug data
data
decimal message
decimal debug <<<
decimal warning
decimal debug_data
data
```

### File Logging

The decimal\_logging library also provides simple file logging. The *log\_file* variable at the top of the library sets the log file. By using any of the following functions, you can easily log data to the specified file:

- log(message)
- log\_debug\_data(message,data)

2025/12/21 09:48 6/6 thinknode™ Examples

• log\_data(data)

### USR-001

.decimal LLC, 121 Central Park Place, Sanford, FL. 32771

From:

http://apps.dotdecimal.com/ - decimal App Documentation

Permanent link:

http://apps.dotdecimal.com/doku.php?id=dicom:userguide:thinknode&rev=1436537307



