thinknode™ Examples

These examples provide a starting point for issuing http connections and requests to the dicom 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

Python: Overview

The provided python scripts and libraries are meant to be a foundation and starting point for using the astroid apps on the thinknode™ framework. The provided scripts outline the basics of using ISS to store objects, as well as constructing and making calculation requests to the calculation provider. The below sections detail the basic usage for each script.

Download: The python astroid_script_library can be downloaded from the .decimal GitHub repository.

thinknode.cfg

There is a simple configuration file (thinknode.cfg) that is used to store user data for connecting to the astroid app on the thinknode™ framework. This file is required by all scripts in the python astroid_script_library to authenticate and use the app. A sample file with no user data is available in the repository and the details of the information to include in the file are provided below.

- *basic_user* being a base64 encoded username and password. Refer to the thinknode documentation for more information.
- *api_url* being the connection string to the thinknode™ framework.
- *apps*
  - *app_name* being the current app name (e.g. dosimetry or dicom).
    - *app_version* being the current version of the app existing on the thinknode™ framework being used. If left blank the thinknode_worker will select the first app's version returned by the Realm Versions GET request.
    - *branch_name* not currently implemented
- *realm_name* thinknode realm
- *account_name* thinknode account name

thinknode.cfg

```
{
    "basic_user": "<Base64 encoded thinknode username:password>",
    "api_url": "https://<thinknode_account>.thinknode.io/api/v1.0",
```

```
    "apps":
    {
        "dosimetry":
        {
            "app_version": "1.0.0-beta1",
            "branch_name": "master"
        },
        "dicom":
        {
            "app_version": "",
            "branch_name": "master"
        },
        "rt_types":
        {
            "app_version": "",
            "branch_name": "master"
        }
    },
    "realm_name": "<thinknode realm>",
    "account_name": "<thinknode account>"
}
```

Python: Calculation Request

Posting a Dicom Patient

The below example shows how to post a dicom file directory to thinknode iss and return a *rt_study*. Using the *rt_study*, sobp and pbs dose calculations can be performed. See the examples *pbs_dose_calc_from_dicom.py* and *sobp_dose_calc_from_dicom.py* from the .decimal GitHub repository for more in depth examples of using the dicom app to call dosimetry calculations.

```python
# Copyright (c) 2015 .decimal, Inc. All rights reserved.
# Date:     09/25/2015
# Desc:     Post folder to thinknode and get back a dicom_study

import os.path
from lib import thinknode_worker as thinknode
from lib import dicom_worker as dicom
from lib import decimal_logging as dl

# Create a study
study_id = dicom.make_rt_study_from_dir(iam, 'E:/dicom/MGH_Phantom_min/')

# Combine uploaded CT image slices into an Image_3d datatype
```

```
study_calc = \
  thinknode.function('decimal', 'dicom', "merge_ct_image_slices",
      [
          thinknode.reference(study_id)
      ])
study_res = thinknode.do_calculation(iam, 'dicom', study_calc, False)
dl.data("Patient rt_tudy ISS ID: " + study_res)
```

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 = []
```

```
    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)
```

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 json_data: calculation request in json 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)
```

```
#   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):
```

dicom_worker

The dicom_worker module provides simplified function and calculation requests for common dicom 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. Reading and posting dicom file sets or individual files
2. Make a dicom study or patient from local files or thinknode data
3. Pull out dicom data from thinknode data

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

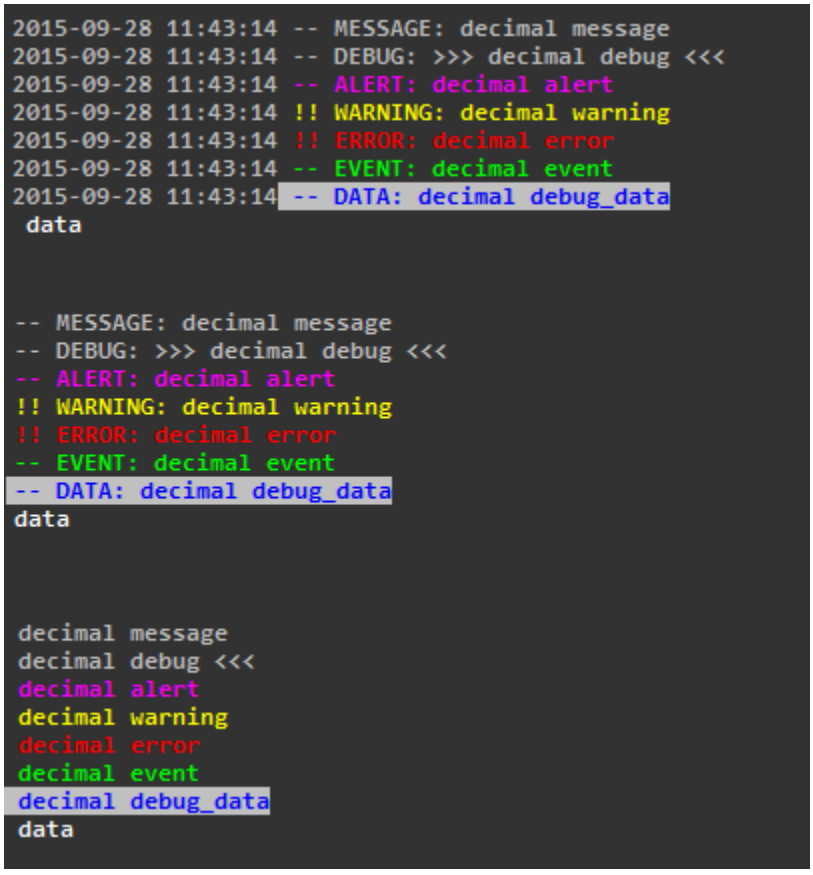
Debugging

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:

1. Timestamps = *True*; Types = *True*
2. Timestamps = *False*; Types = *True*
3. Timestamps = *False*; Types = *False*



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)

- log_data(data)

USR-001

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