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

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: Immutable Storage

Post Generic ISS Object

The *post_iss_object_generic.py* is a basic python script that provides an example to post any dosimetry type as an immutable object to the dosimetry app on the thinknode™ framework. This example can be used for any immutable storage post using any datatype by replacing the json iss file. The current example posts an rt_study DICOM App datatype object that is read in from the study.json data file.

Dependencies:

- thinknode.cfg
- .decimal Python Libraries
- study.json (or any other prebuilt json file of a dosimetry object as described in the Apps Manifest Guide)

post_iss_object_generic.py

```
# Copyright (c) 2015 .decimal, Inc. All rights reserved.
# Desc:     Post an immutable json object to the thinknode framework

from lib import thinknode_worker as thinknode
```

```
import requests
import json

iss_dir = "iss_files"
json_iss_file = "study.json"
obj_name = "rt_study"

# Get IAM ids
iam = thinknode.authenticate(thinknode.read_config('thinknode.cfg'))

# App object to post to iss
with open(iss_dir + '/' + json_iss_file) as data_file:
    json_data = json.load(data_file)

# Post immutable object to ISS
res = thinknode.post_immutable_named(iam, "dicom", json_data, obj_name)
```

Returns:

1. The ID (in json) of the object stored in Immutable Storage.

Python: Calculation Request

Generic Calc Request

The *post_calc_request_generic.py* is a basic example to post a calculation request to dosimetry. This example can be used for any calculation request using any datatype by replacing the calculation request json file. This request will post a calculation request, check the status using long polling with a specified timeout, and return the calculation result.

Dependencies:

- thinknode.cfg
- .decimal Python Libraries
- compute_aperture.json (or any other prebuilt json file of a dosimetry object as described in the Dosimetry Manifest Guide)

post_calc_request_generic.py

```
# Copyright (c) 2015 .decimal, Inc. All rights reserved.
# Desc:     Post a json calculation request to the thinknode framework

request_dir = "request_files"
json_calc_file = "compute_aperture.json"
```

```
# Get IAM ids
iam = thinknode.authenticate(thinknode.read_config('thinknode.cfg'))

# App calculation request
with open(request_dir + '/' + json_calc_file) as data_file:
    json_data = json.load(data_file)

# Send calc request and wait for answer
res = thinknode.do_calculation(iam, json_data)
dl.data("Calculation Result: ", str(res))
```

Returns:

1. The calculation result (in json) of the API function called.

SOBP Dose Calculation

The *post_calc_request_sobp_dose.py* and *post_calc_request_sobp_dose_with_shifter.py* are more complete examples that create input data and perform an sobp dose calculation function request to the dosimetry app on the thinknode™ framework.

The *post_calc_request_sobp_dose.py* example creates the entire calculation request inline using thinknode structure, array, and function requests. The entire dose calculation request is performed using one thinknode calculation provider call. While this structure of a request is a little more complicated to setup and perform, it removes the need to post to ISS or perform ancillary calculations separately.

The *post_calc_request_sobp_dose_with_shifter.py* adds in the complication of adding a degrader to the sobp calculation. This example performs three separate calculation requests. The first two requests are used to construct the proton degrader_geometry and the third performs the actual dose calculation request using the previously constructed degrader. The entire example could be condensed into a single more complicated thinknode calculation structure, eliminating the need to perform the separate requests, but in some instances it can be more straight-forward to perform some of the calculations separately as shown. As seen in the example, the first two calculation results for the proton degrader are what is placed into the sobp calculation request, instead of the actual function calls as was done in the case of the aperture in the previous example.

Dependencies:

- thinknode.cfg
- .decimal Python Libraries

Example

Below is an abbreviated version of the *post_calc_request_sobp_dose_with_shifter.py* file. The abbreviated

sections are denoted as "...". In the below sample, the *dose_calc* variable is a thinknode function request that is made of individually constructed arguments. Notice how some of the elements, like degrader, can be built upon seperate calculation requests.

- Modules used and explanation:
  - The *thinknode_worker* (thinknode) module is a library that provides worker functions for performing and building the authentication, iss, and calculation requests to the thinknode framework.
  - The *dosimetry_worker* (dosimetry) module is a library that provides simplified common dosimetry tasks.
  - The *decimal_logger* (dl) module is a library that provides nicely formatted log output. This includes optional file logging, timestamps, and message coloring (when run through command windows).

Refer to the .decimal Libraries section for more information on the provided decimal libraries.

```
import json
from lib import thinknode_worker as thinknode
from lib import dosimetry_worker as dosimetry
from lib import decimal_logging as dl

# Get IAM ids
iam = thinknode.authenticate(thinknode.read_config('thinknode.cfg'))

def make_dose_points(pointCount):
...

def make_layers(sad, range, mod):
    return \
        thinknode.function(iam["account_name"], "dosimetry",
"compute_double_scattering_layers",
            [
                thinknode.reference("55f70f5000c0a247563a909b6087ada0"), #
SOBP Machine from ISS
                thinknode.value(sad),
                thinknode.value(range),
                thinknode.value(mod)
            ])

def make_target():
    return \
        thinknode.function("dosimetry", "make_cube",
            [
                thinknode.value([-32, -20, -30]),
                thinknode.value([16, -10, 30])
            ])

def compute_aperture():
```

```
    return dosimetry.compute_aperture(iam, make_target(), beam_geometry, 20.0,
0.0, 250.5)

beam_geometry = \
...

# Get degrader geometry as calculation result
degrade_geom = \
    thinknode.function(iam["account_name"], "dosimetry", "make_shifter",
        [
            thinknode.value(18), # thickness
            thinknode.value("mm"), # units
            thinknode.value(200) # downstream edge
        ])
res_geom = thinknode.do_calculation(iam, degrade_geom, True)
degrader = \
    thinknode.function(iam["account_name"], "dosimetry", "make_degrader",
        [
            thinknode.value(res_geom),
            thinknode.reference("56030a9500c036a0c6393f984b25e303") # Material
spec from ISS
        ])
proton_degr = thinknode.do_calculation(iam, degrader)

# Call compute_sobp_pb_dose2
dose_calc = \
    thinknode.function("dosimetry", "compute_sobp_pb_dose2",
        [
            dosimetry.make_image_3d(iam, [-100, -100, -100], [200, 200, 200],
[2, 2, 2], 1), #stopping_power_image
            thinknode.value(make_dose_points(181)), # dose_points
            beam_geometry, #beam_geometry
            dosimetry.make_grid(iam, [-75, -75], [150, 150], [2, 2]), #
bixel_grid
            make_layers(2270.0, 152.0, 38.0),
            compute_aperture(), # aperture based on targets
            thinknode.value([proton_degr]) # degraders
        ])

# Perform calculation
res = thinknode.do_calculation(iam, dose_calc)
dl.data("Calculation Result: ", 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 as snippet from the rt_types module that shows the class for the *polyset* rt_type along with its default initializations and *.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 displayed by the *polygons* parameter of the *polyset* in this example.
- ***expand_data* function:** Each class's *.expand_data* function provides an ordered dictionary of each of the values in the class. This is explicitly an ordered dictionary since when calling a function in a calculation request, the order of the values provided matters if constructing the request by thinknode value type.
- ***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.

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, then using that data type to output the image as 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 http worker 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):
```

dosimetry_worker

The dosimetry_worker module provides simplified function 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 that can be visualized in Paraview. It's most useful for displaying image and 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 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

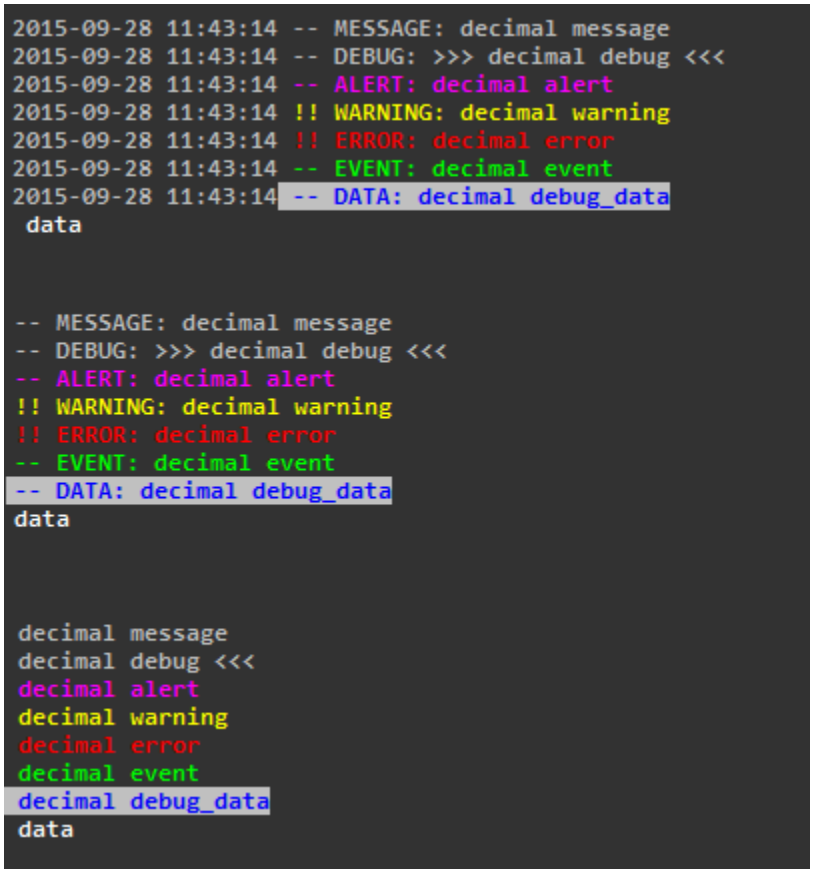
Debugging

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