MITRE

Sponsor: NIST
Dept. No.: T8B3
Contract No.: SB-1341-14-CQ-0010

McLean, VA

CASE User Guide

Author(s): Cory Hall
Vik Harichandran
Jared Stroud
Andrew Sovern

April 2018

MITRE Team:
Cory Hall, Vik Harichandran
Jared Stroud, Andrew Sovern

CASE USER GUIDE

NOTE: This is a work in progress. Changes may occur after feedback from the 2018 CASE workshops. Information that we expect to change or need clarification on is marked with [brackets].

TEMP NOTE: This guide is currently based off version 0.1.0 of the ontology (outdated).
Some of this information should be copied to the Github which will be a light weight, modular, non-narrative version of this, focused on "find what you need and how to do it quickly."

Contents

MITRE Team:
Cory Hall, Vik Harichandran
Jared Stroud, Andrew Sovern

CASE Introduction

What is it?

Cyber-investigation Analysis Standard Expression (CASE) is a community-developed specification language, which is intended to serve the needs of the broadest possible range of cyber-investigation domains, including digital forensic science, incident response, counter-terrorism, criminal justice, forensic intelligence and situational awareness.

The primary motivation for CASE is interoperability - to advance the exchange of cyber-investigation information between tools and organizations. CASE aligns with and extends the Unified Cyber Ontology (UCO).[1]

The end goal of UCO/CASE is to provide an evolved and structured approach that:

exchanges cyber investigation information in standardized form

achieves interoperability between systems and tools

maintains provenance at all phases of the cyber-investigation lifecycle

provides structured/linked data to support intelligent analysis

enhances tool testing and validation results

controls access to privileged, proprietary, and personal information

restricts use of data covered under license agreements[2]

CASE is referred to as a derivative of UCO because it inherits these main UCO objects:

Action	MarkingDefinition
Assertion	Relationship
Bundle	Role
ControlledVocabulary	Tool
Identity	Trace
Location	

While UCO focuses on being an umbrella ontology (capable of utilizing a variety of lower-level ontologies/specification languages) to span across all cyber domains, CASE is geared primarily towards digital investigations, [more listed here], and any metadata related to a case. [For information to be considered CASE the marked objects above must be present. <mark objects with asterisk or another symbol>] The full list of all CASE object types comprises the Natural Language Glossary (NLG).

For more on understanding the ontology fully visit: [link to Wiki where the whitepaper's ontology web figure is shown; link to whitepaper itself; link to the Github Wiki].

[1] https://sites.google.com/view/casework/home
[2] Casey E., Barnum, S., Griffith, R., CASE Presentation, DFRWS USA 2017

Why use it?

The benefits of CASE are aligned with that of any standard, but also includes some other perks:

Expressivity (ability to address all cases)
Integration reducing duplicated information
Flexibility
Extensibility
Automatability (data fusion from different organizations, cases, sources/devices, and domains)
Interoperability
Open-source (the ontology itself is community driven)
Ability to share sensitive information more easily (different licenses, classifications, and PII)
APIs and verifiers that make it easy to adopt
Custom flavors of the ontology can easily be made (by following the same conventions)
High-level analytics are possible due to capturing more case metadata (this may be done via pattern-recognition/machine learning, behavior analysis, visualization, pivoting, searching, and contextual analysis)

How should it be adopted?

Adoption of CASE is scoped into two primary, sequential tasks: 1) mapping 2) API integration and verification.

Mapping involves determining which CASE objects should be used to represent your information. Sometimes several objects may appear appropriate; [this object selection (visualization or web form?) tool can assist with this]. The second step is incorporation of the Python API (and optionally Glossary Verifier).

Next, the Python API should be integrated for automatic CASE export creation. [This API does ontology verification as well as type checking. During integration it is up to the adopter to either halt export if verification checks fail, or bypass this, allowing the user to receive output (where the object fails occurred are excluded).] Verification is done to ensure the output follows the CASE specification.

A glossary verifier is available for implementers using custom CASE-derived ontologies (or slightly altered CASE ontologies). It confirms that the names given to the ontology objects (for clarity, referred to as 'types') are being used – e.g. it would catch "Tools" being used as a CASE object when "Tool" is the correct type. The tool can also be used to obtain a diff of two ontology glossaries for ontology comparison purposes.

Refer to the subsequent sections based on which part of the process of adoption you are involved with. Following these sections is information for CASE developers to help contribute to its evolution and adoption (including how bugs and requests will be handled).

Mapping Objects

When is mapping in scope?

Not everyone need be concerned with mapping. However, it needs to be done before any official/stable integration of the API is performed. This is a critical step, as picking the wrong object type may introduce verification errors, cause misrepresentation of data, or simply inaccurately represent information as something it is not.

How should mapping be done?

[Visualization / web form filling tool?]

Python API

What does it do?

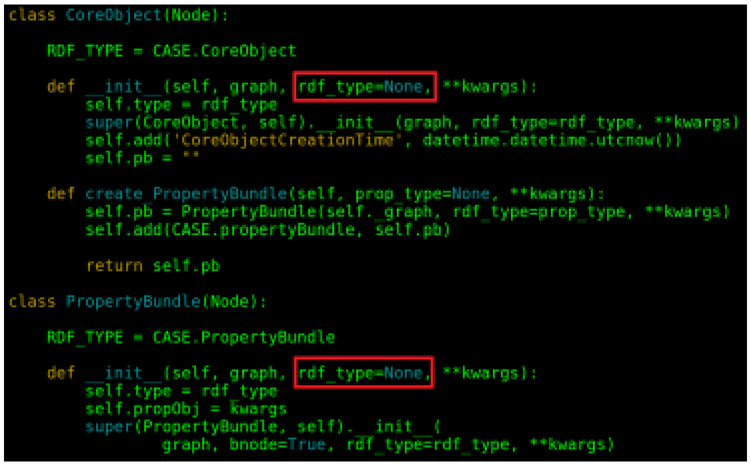
The Python API creates CASE objects in memory, inserting them into a Resource Description Framework (RDF) graph, which is used to properly encapsulate and ID the information, and then exportingit to a file. Note that different formats can be used for the serialized output, including JSON-LD (default), Turtle, n3, etc.

There are two files needed to use the API: *case.py* and *NLG.py*. The first can be used to make arbitrary RDF objects while the second is used to both create CASE-specific objects and verify that ontology specification is being followed (required properties are being included, format/type of the information matches expectation, and parent-child relationships are correct). It is ill-advised to use only the *case.py* file to create arbitrary objects as this means no verification of the CASE standard is performed, defeating most of the purpose of using a standard.

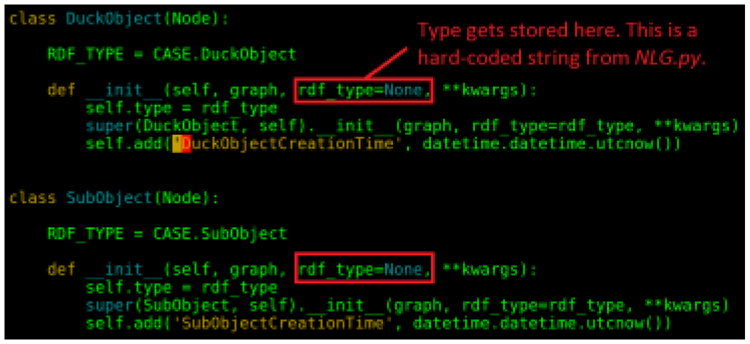
For absolute clarity (especially on Github), use these terms for aspects of the CASE API:

Classes (*case.py*)	= categories of the ontology object types (based on requirements/inheritance)
Types (*NLG.py*)	= CASE object names; group into Classes; not objects but return objects
Objects	= Python objects that are stored in an RDF graph and serialized to output

Examples are shown below. *CoreObject*, *PropertyBundle*, *DuckObject*, and *SubObject* are classes (categories of CASE object types). The *rdf_type* of objects is used to verify parent-child relationships.



```
class CoreObject(Node):

    RDF_TYPE = CASE.CoreObject

    def __init__(self, graph, rdf_type=None, **kwargs):
        self.type = rdf_type
        super(CoreObject, self).__init__(graph, rdf_type=rdf_type, **kwargs)
        self.add('CoreObjectCreationTime', datetime.datetime.utcnow())
        self.pb = ""

    def create_PropertyBundle(self, prop_type=None, **kwargs):
        self.pb = PropertyBundle(self._graph, rdf_type=prop_type, **kwargs)
        self.add(CASE.propertyBundle, self.pb)

        return self.pb

class PropertyBundle(Node):

    RDF_TYPE = CASE.PropertyBundle

    def __init__(self, graph, rdf_type=None, **kwargs):
        self.type = rdf_type
        self.propObj = kwargs
        super(PropertyBundle, self).__init__(
                graph, bnode=True, rdf_type=rdf_type, **kwargs)
```

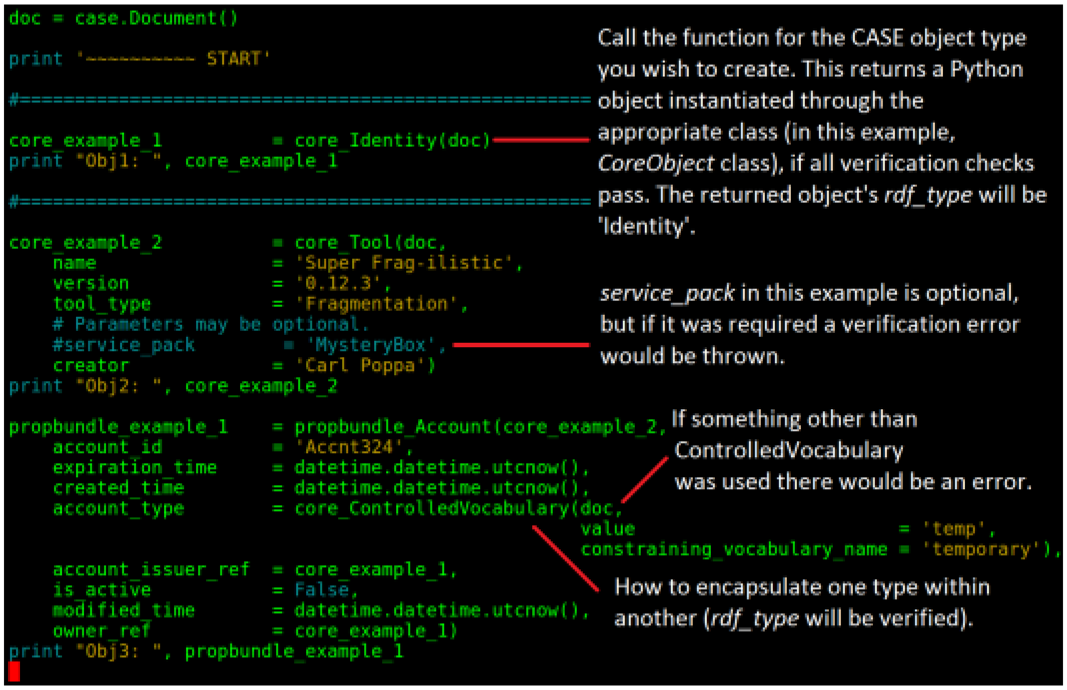


```
class DuckObject(Node):

    RDF_TYPE = CASE.DuckObject

    def __init__(self, graph, rdf_type=None, **kwargs):
        self.type = rdf_type
        super(DuckObject, self).__init__(graph, rdf_type=rdf_type, **kwargs)
        self.add('DuckObjectCreationTime', datetime.datetime.utcnow())


class SubObject(Node):

    RDF_TYPE = CASE.SubObject

    def __init__(self, graph, rdf_type=None, **kwargs):
        self.type = rdf_type
        super(SubObject, self).__init__(graph, rdf_type=rdf_type, **kwargs)
        self.add('SubObjectCreationTime', datetime.datetime.utcnow())
```

Type gets stored here. This is a hard-coded string from *NLG.py*.

How is it used?

To create a CASE object call the appropriate NLG entry for the desired object typeas in the figures below. **Note that the parameter names must match exactly for verification to occur.** To see the details about the *NLG.py* functions called, such as *core_Tool,* refer to the CASE Developer Conventions section.



```
doc = case.Document()

print '---------- START'

#=====================================================

core_example_1            = core_Identity(doc)
print "Obj1: ", core_example_1

#=====================================================

core_example_2            = core_Tool(doc,
    name                  = 'Super Frag-ilistic',
    version               = '0.12.3',
    tool_type             = 'Fragmentation',
    # Parameters may be optional.
    #service_pack          = 'MysteryBox',
    creator               = 'Carl Poppa')
print "Obj2: ", core_example_2

propbundle_example_1      = propbundle_Account(core_example_2,
    account_id            = 'Accnt324',
    expiration_time       = datetime.datetime.utcnow(),
    created_time          = datetime.datetime.utcnow(),
    account_type          = core_ControlledVocabulary(doc,
                                    value                       = 'temp',
                                    constraining_vocabulary_name = 'temporary'),

    account_issuer_ref    = core_example_1,
    is_active             = False,
    modified_time         = datetime.datetime.utcnow(),
    owner_ref             = core_example_1)
print "Obj3: ", propbundle_example_1
```

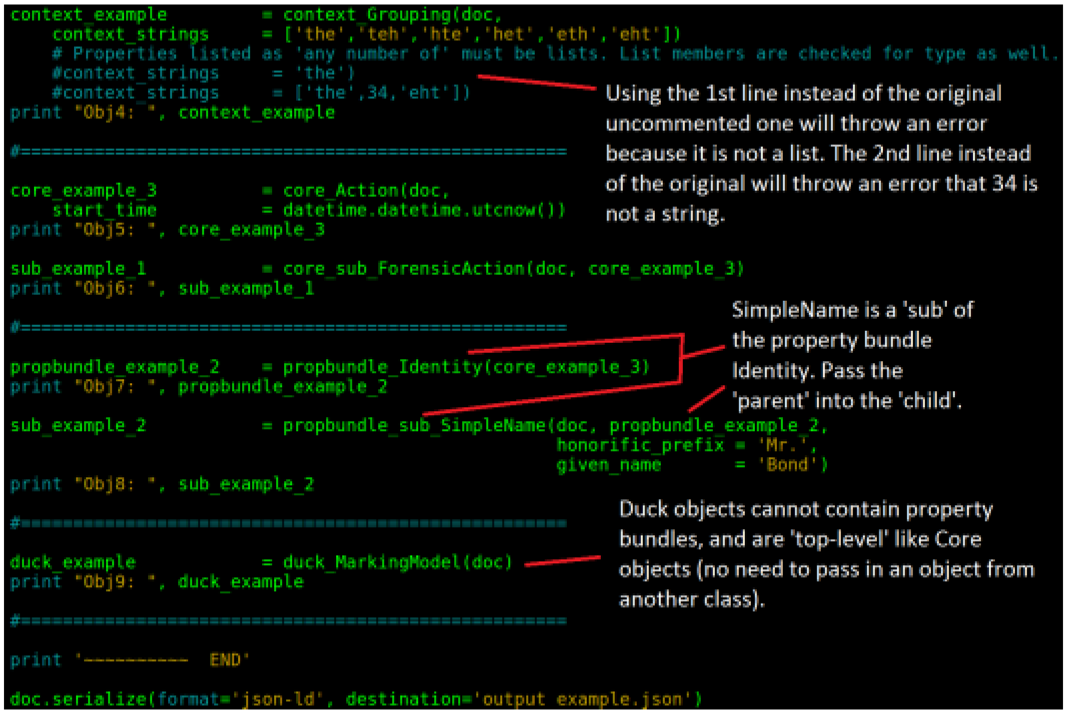
Call the function for the CASE object type you wish to create. This returns a Python object instantiated through the appropriate class (in this example, *CoreObject* class), if all verification checks pass. The returned object's *rdf_type* will be 'Identity'.

service_pack in this example is optional, but if it was required a verification error would be thrown.

If something other than ControlledVocabulary was used there would be an error.

How to encapsulate one type within another (*rdf_type* will be verified).

The simplest manner to illustrate how to adopt this in the real world is through an example with Volatility.



```
context_example          = context_Grouping(doc,
    context_strings      = ['the','teh','hte','het','eth','eht'])
    # Properties listed as 'any number of' must be lists. List members are checked for type as well.
    #context_strings      = 'the')
    #context_strings      = ['the',34,'eht'])
print "Obj4: ", context_example

#=====================================================

core_example_3           = core_Action(doc,
    start_time           = datetime.datetime.utcnow())
print "Obj5: ", core_example_3

sub_example_1            = core_sub_ForensicAction(doc, core_example_3)
print "Obj6: ", sub_example_1

#=====================================================

propbundle_example_2     = propbundle_Identity(core_example_3)
print "Obj7: ", propbundle_example_2

sub_example_2            = propbundle_sub_SimpleName(doc, propbundle_example_2,
                                                     honorific_prefix = 'Mr.',
                                                     given_name       = 'Bond')

print "Obj8: ", sub_example_2

#=====================================================

duck_example             = duck_MarkingModel(doc)
print "Obj9: ", duck_example

#=====================================================

print '----------  END'

doc.serialize(format='json-ld', destination='output_example.json')
```

Using the 1st line instead of the original uncommented one will throw an error because it is not a list. The 2nd line instead of the original will throw an error that 34 is not a string.

SimpleName is a 'sub' of the property bundle Identity. Pass the 'parent' into the 'child'.

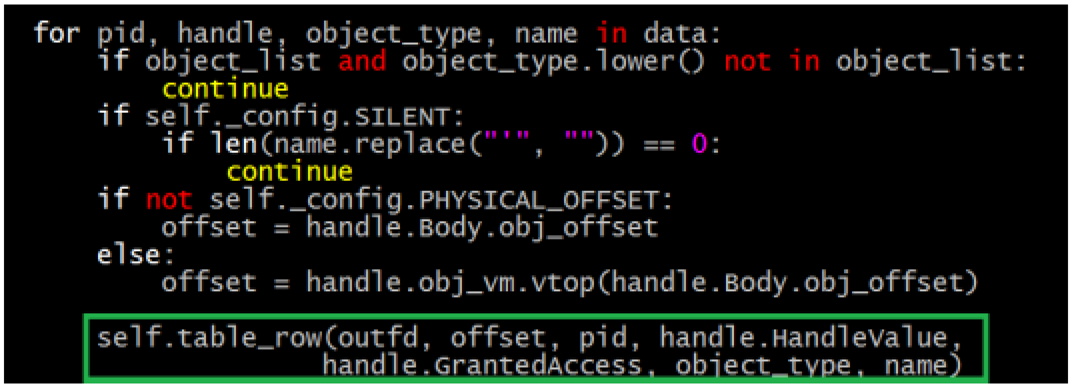
Duck objects cannot contain property bundles, and are 'top-level' like Core objects (no need to pass in an object from another class).

Volatility Proof-of-Concept

Volatility is a Python command-line memory forensics tool that analysis data structures from memory dumps to represent a what was occurring in a given snapshot of memory. Volatiltiy is made up of several independent modules that allow for analyzing memory dumps for specific data structures from a large variety of operating systems and associated releases (service packs, kernels, etc…). In this example, we will show how to partially adapt volatility output into CASE output.

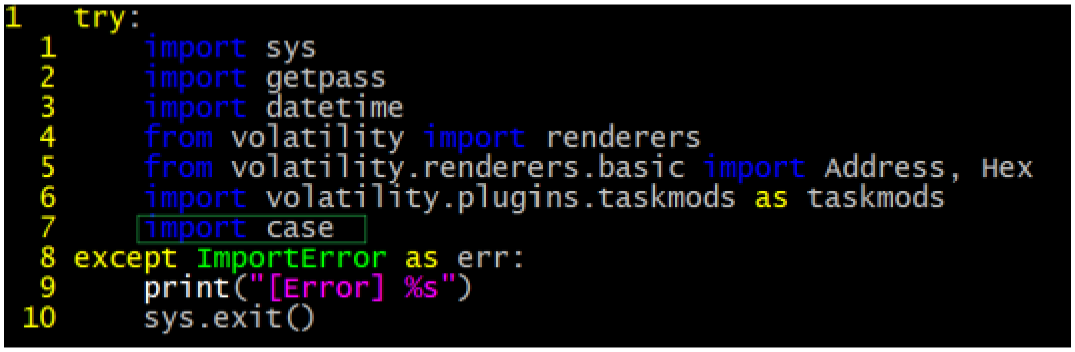
The `render text` function is the standard function within the Volatility framework which is responsible for sending data to standard out. For this example, we will be looking at `render_text` within "handles.py". This plugin prints out handle information (process, thread, etc…) pertaining to Windows memory dumps.

The green box in the image below shows which variables are responsible for representing handle information. This is an excellent starting point for what needs to be encapsulated into CASE.



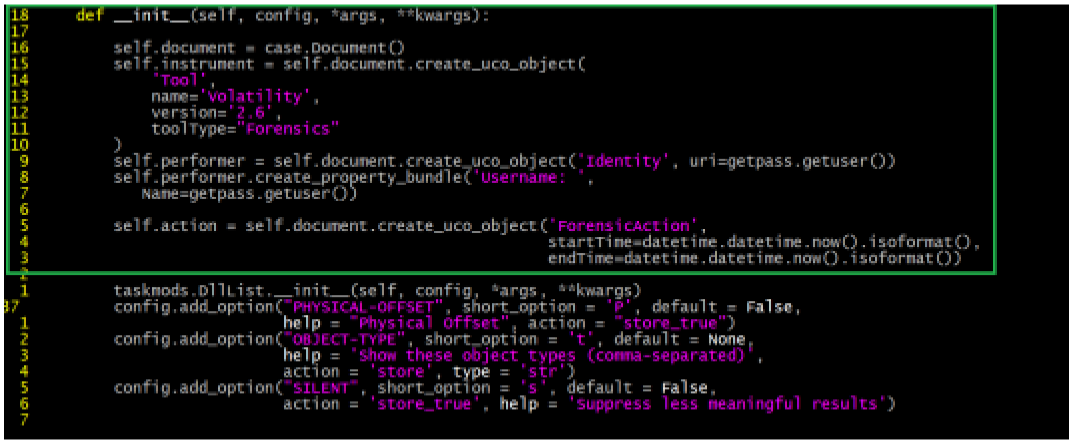
```
for pid, handle, object_type, name in data:
    if object_list and object_type.lower() not in object_list:
        continue
    if self._config.SILENT:
        if len(name.replace("'", "")) == 0:
            continue
    if not self._config.PHYSICAL_OFFSET:
        offset = handle.Body.obj_offset
    else:
        offset = handle.obj_vm.vtop(handle.Body.obj_offset)

    self.table_row(outfd, offset, pid, handle.HandleValue,
                   handle.GrantedAccess, object_type, name)
```

First, copy the handles.py file into casehandles.py within a development directory of your choice. Then using your favorite text editor open handles.py. Insert an import statement at the top of the file with "case".



```
try:
    import sys
    import getpass
    import datetime
    from volatility import renderers
    from volatility.renderers.basic import Address, Hex
    import volatility.plugins.taskmods as taskmods
    import case
except ImportError as err:
    print("[Error] %s")
    sys.exit()
```

Within the __init__ constructor in the CaseHandles class, the initialization of objects required for other class methods. Here, we will define some additional properties pertaining to the CASE API. The green square below shows the additions made for the CASE property bundle.

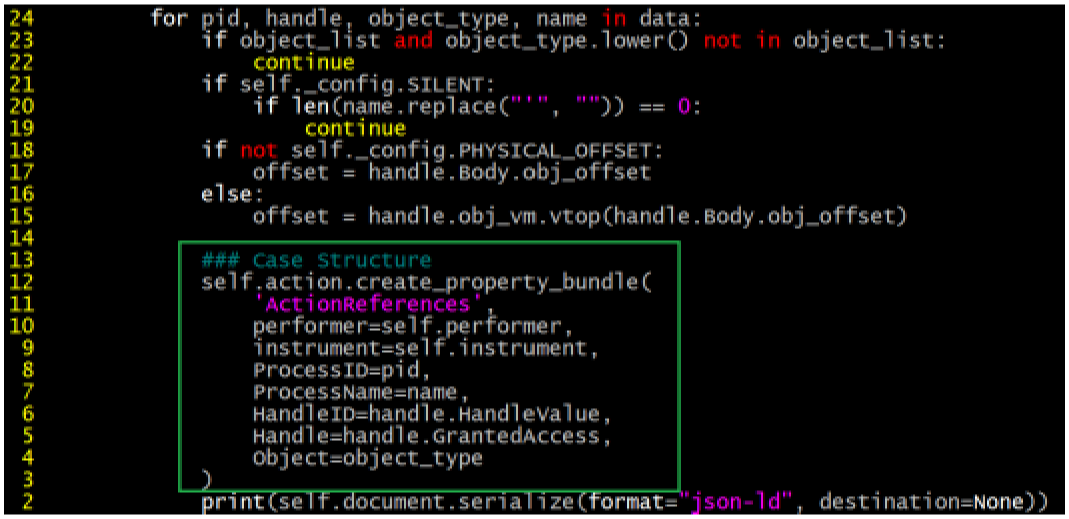


```
18      def __init__(self, config, *args, **kwargs):
17
16          self.document = case.Document()
15          self.instrument = self.document.create_uco_object(
14              'Tool',
13              name='Volatility',
12              version='2.6',
11              toolType="Forensics"
10          )
 9          self.performer = self.document.create_uco_object('Identity', uri=getpass.getuser())
 8          self.performer.create_property_bundle('Username: ',
 7             Name=getpass.getuser())
 6
 5          self.action = self.document.create_uco_object('ForensicAction',
 4                                                          startTime=datetime.datetime.now().isoformat(),
 3                                                          endTime=datetime.datetime.now().isoformat())
 2
 1          taskmods.DllList.__init__(self, config, *args, **kwargs)
37          config.add_option("PHYSICAL-OFFSET", short_option = 'P', default = False,
 1                            help = "Physical Offset", action = "store_true")
 2          config.add_option("OBJECT-TYPE", short_option = 't', default = None,
 3                            help = 'Show these object types (comma-separated)',
 4                            action = 'store', type = 'str')
 5          config.add_option("SILENT", short_option = 's', default = False,
 6                            action = 'store_true', help = 'Suppress less meaningful results')
 7
```

Line 16 declares the case.Document() method creates the initial CASE object. Line 15 utilizes this document object to create a "instrument" construct followed by lines 9 and 8 defining the user who's running this utility. This embeds the user name in the CASE output. Lines 5 through 3 shows when the "forensicAction" occurred. In this context this pertains to running the tool. You could adopt this to be updated when the tool finishes running assuming you have a large memory dump or a different time zone format.

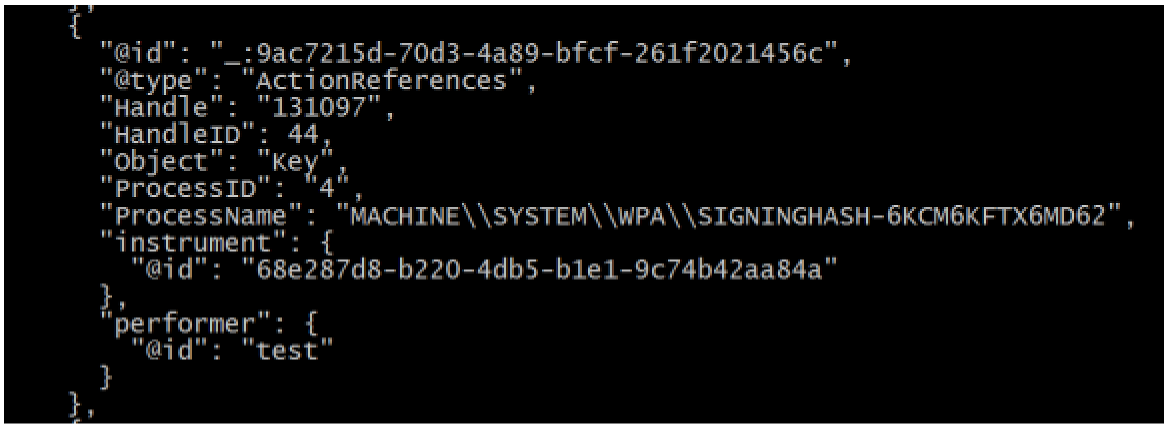
Finally, back within the previously seen for loop we create the CASE property bundle. All this does, is take the handle and process attributes and insert them into an RDF graph. The keys (performer, instrument, ProcessID, ProcessName, HandleID, Handle,Object) are given the values that are either defined in the class constructor or values obtained through Volatility after parsing data structures from a memory dump. Here is where you can reference the case ontology to see where mappings exist between your tools output and the CASE definition. In the event you find a missing attribute from the CASE ontology, please open an Github issue!

Line 2 (below the graph) serializes the RDF graph into JSON-LD, the standard CASE format and prints the information to the screen.



```
24          for pid, handle, object_type, name in data:
23              if object_list and object_type.lower() not in object_list:
22                  continue
21              if self._config.SILENT:
20                  if len(name.replace("'", "")) == 0:
19                      continue
18              if not self._config.PHYSICAL_OFFSET:
17                  offset = handle.Body.obj_offset
16              else:
15                  offset = handle.obj_vm.vtop(handle.Body.obj_offset)
14
13              ### Case Structure
12              self.action.create_property_bundle(
11                  'ActionReferences',
10                  performer=self.performer,
 9                  instrument=self.instrument,
 8                  ProcessID=pid,
 7                  ProcessName=name,
 6                  HandleID=handle.HandleValue,
 5                  Handle=handle.GrantedAccess,
 4                  Object=object_type
 3              )
 2              print(self.document.serialize(format="json-ld", destination=None))
```

At this point you should be able to successfully run the utility. Below is a snippet of the output shown below. Please note that the "`@id`" is a unique string generated by Python's UUID library.



```
  {
    "@id": "_:9ac7215d-70d3-4a89-bfcf-261f2021456c",
    "@type": "ActionReferences",
    "Handle": "131097",
    "HandleID": 44,
    "Object": "Key",
    "ProcessID": "4",
    "ProcessName": "MACHINE\\SYSTEM\\WPA\\SIGNINGHASH-6KCM6KFTX6MD62",
    "instrument": {
      "@id": "68e287d8-b220-4db5-b1e1-9c74b42aa84a"
    },
    "performer": {
      "@id": "test"
    }
  },
```

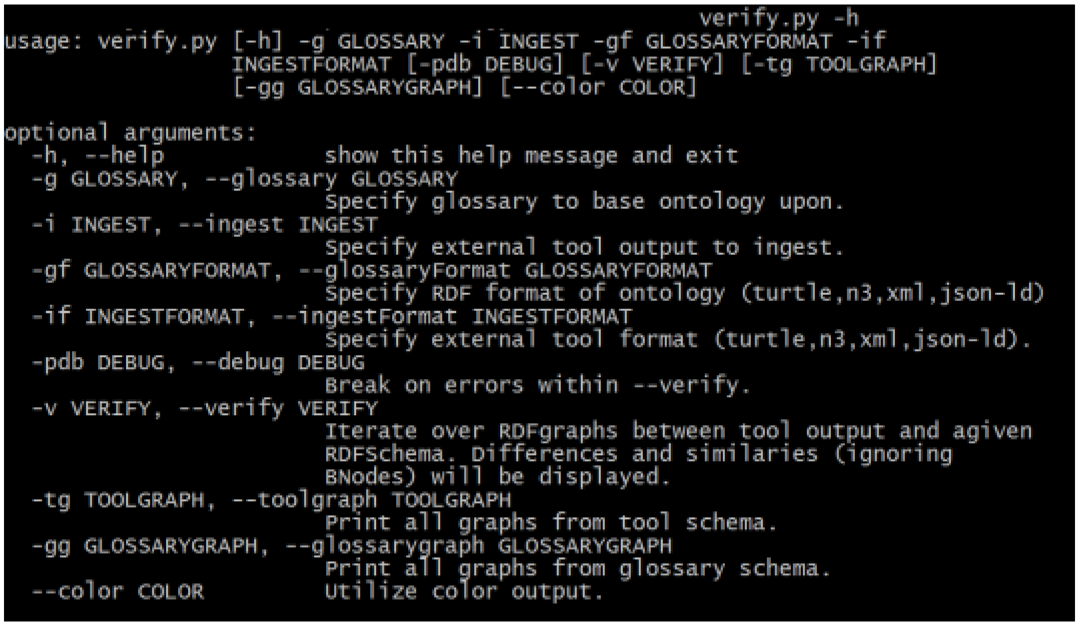
Glossary Verifier

What does it do?

The Glossary Verifier is a Python script that can assist adopters in ensuring that their mappings match core CASE objects. Verifier.py uses a 3rd party Python library, RDFLib[3] to interact with all RDF schema based files (n3, ttl, json-ld, etc..). Verifier.py will read in an Ontology, such as case.ttl and iterate through all the RDF schema objects and compare it against a tool's output. This effectively ends up as a large RDF graph diff showing what matches and what does not match. It's important to note blank nodes[4] are skipped for brevity.

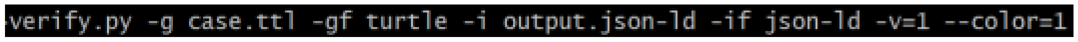
How is it used?

Verify.py has several options to display data from a RDF schema file. Entering the `-h` argument prints out all additional command line options that can be used to manipulate data.



```
                                            verify.py -h
usage: verify.py [-h] -g GLOSSARY -i INGEST -gf GLOSSARYFORMAT -if
                 INGESTFORMAT [-pdb DEBUG] [-v VERIFY] [-tg TOOLGRAPH]
                 [-gg GLOSSARYGRAPH] [--color COLOR]

optional arguments:
  -h, --help            show this help message and exit
  -g GLOSSARY, --glossary GLOSSARY
                        Specify glossary to base ontology upon.
  -i INGEST, --ingest INGEST
                        Specify external tool output to ingest.
  -gf GLOSSARYFORMAT, --glossaryFormat GLOSSARYFORMAT
                        Specify RDF format of ontology (turtle,n3,xml,json-ld)
  -if INGESTFORMAT, --ingestFormat INGESTFORMAT
                        Specify external tool format (turtle,n3,xml,json-ld).
  -pdb DEBUG, --debug DEBUG
                        Break on errors within --verify.
  -v VERIFY, --verify VERIFY
                        Iterate over RDFgraphs between tool output and agiven
                        RDFSchema. Differences and similaries (ignoring
                        BNodes) will be displayed.
  -tg TOOLGRAPH, --toolgraph TOOLGRAPH
                        Print all graphs from tool schema.
  -gg GLOSSARYGRAPH, --glossarygraph GLOSSARYGRAPH
                        Print all graphs from glossary schema.
  --color COLOR         Utilize color output.
```

The simplest option to execute would be to compare RDF graphs and show what's different between them. The following command will perform a "verification" between the case.ttl ontology and the tool output specified in output.json-ld.

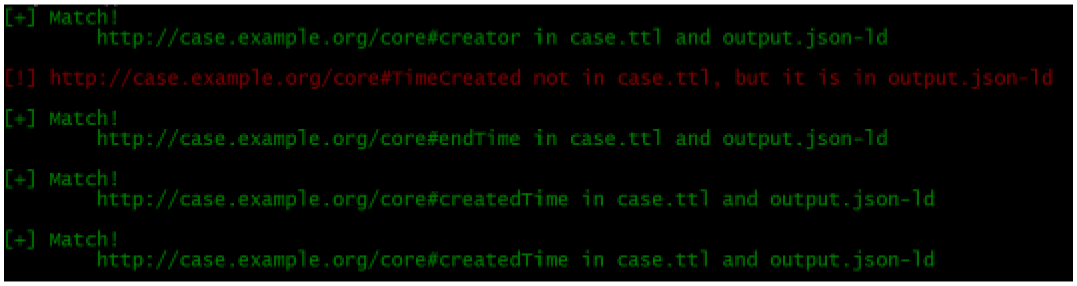


```
verify.py -g case.ttl -gf turtle -i output.json-ld -if json-ld -v=1 --color=1
```

[3] https://rdflib.readthedocs.io/en/stable/
[4] https://www.w3.org/TR/turtle/#BNodes

Figure 1 - verification

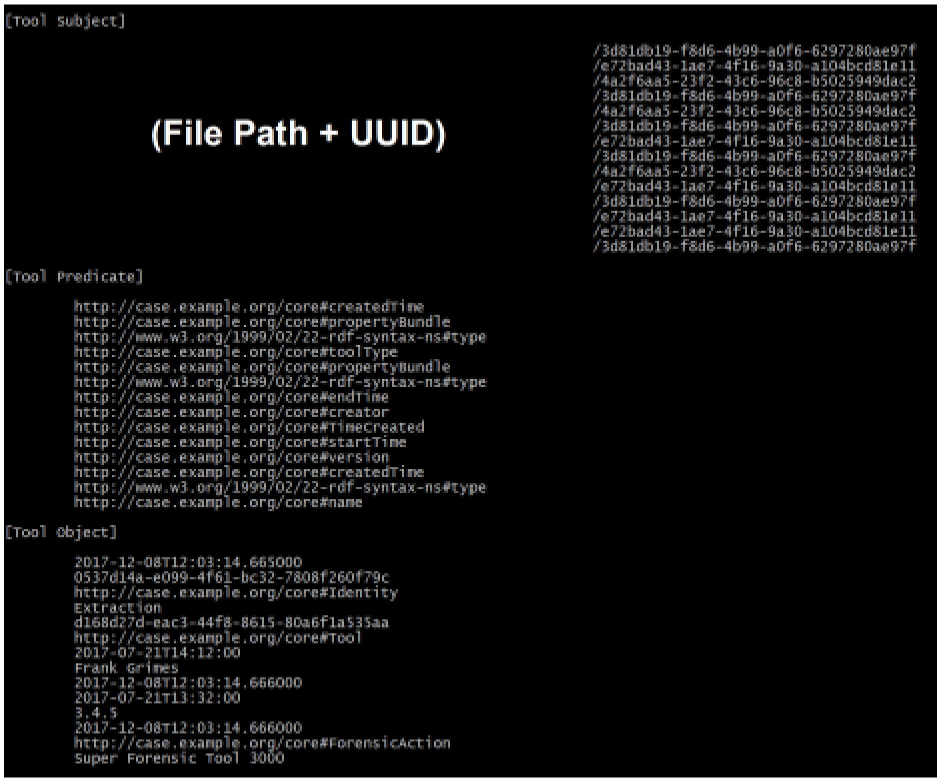
Shown in the image below in red is a attribute that is in the output.json-ld but not in case.ttl This immediate visual indication can aid the developer in understanding if something has been spelled incorrectly, implemented incorrectly or if their custom ontology matches their custom property bundle output.



```
[+] Match!
        http://case.example.org/core#creator in case.ttl and output.json-ld

[!] http://case.example.org/core#TimeCreated not in case.ttl, but it is in output.json-ld

[+] Match!
        http://case.example.org/core#endTime in case.ttl and output.json-ld

[+] Match!
        http://case.example.org/core#createdTime in case.ttl and output.json-ld

[+] Match!
        http://case.example.org/core#createdTime in case.ttl and output.json-ld
```

Showing Graphs in RDF

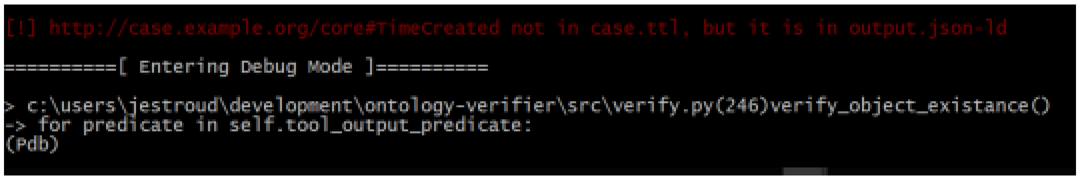
RDF stores data into tuples containing a subject, predicate and an object. To see how data from a turtle, n3 or json-ld file is stored into this tuple, the graph display functions of verify.py can be used. For example, if a user wanted to see what the tool output's RDF schema looked like, the command shown in figure one can be used with addition to the -tg=1 argument (tg representing "tool graph", gg representing glossary graph).



```
[Tool Subject]
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f
                                                        /e72bad43-1ae7-4f16-9a30-a104bcd81e11
                                                        /4a2f6aa5-23f2-43c6-96c8-b5025949dac2
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f
                                                        /4a2f6aa5-23f2-43c6-96c8-b5025949dac2
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f
                                                        /e72bad43-1ae7-4f16-9a30-a104bcd81e11
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f
                                                        /4a2f6aa5-23f2-43c6-96c8-b5025949dac2
                                                        /e72bad43-1ae7-4f16-9a30-a104bcd81e11
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f
                                                        /e72bad43-1ae7-4f16-9a30-a104bcd81e11
                                                        /e72bad43-1ae7-4f16-9a30-a104bcd81e11
                                                        /3d81db19-f8d6-4b99-a0f6-6297280ae97f

[Tool Predicate]

        http://case.example.org/core#createdTime
        http://case.example.org/core#propertyBundle
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
        http://case.example.org/core#toolType
        http://case.example.org/core#propertyBundle
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
        http://case.example.org/core#endTime
        http://case.example.org/core#creator
        http://case.example.org/core#TimeCreated
        http://case.example.org/core#startTime
        http://case.example.org/core#version
        http://case.example.org/core#createdTime
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
        http://case.example.org/core#name

[Tool Object]

        2017-12-08T12:03:14.665000
        0537d14a-e099-4f61-bc32-7808f260f79c
        http://case.example.org/core#Identity
        Extraction
        d168d27d-eac3-44f8-8615-80a6f1a535aa
        http://case.example.org/core#Tool
        2017-07-21T14:12:00
        Frank Grimes
        2017-12-08T12:03:14.666000
        2017-07-21T13:32:00
        3.4.5
        2017-12-08T12:03:14.666000
        http://case.example.org/core#ForensicAction
        Super Forensic Tool 3000
```

Debugging RDF

Going one step further for developers, including the `-pdb=1` line will enter the Python debugger when an inconsistency has been met. At this point the developer can manipulate the graphs and explore data at a deeper level. This is not expected to be used during normal implementation efforts, however it could provide useful for troubleshooting 3rd party RDF libraries. Upon re-running the command shown in figure-1 with `-pdb=1` you'll be greeted with the Python debugger when an inconsistency is matched.



```
[!] http://case.example.org/core#TimeCreated not in case.ttl, but it is in output.json-ld

==========[ Entering Debug Mode ]==========

> c:\users\jestroud\development\ontology-verifier\src\verify.py(246)verify_object_existance()
-> for predicate in self.tool_output_predicate:
(Pdb)
```

CASE Developer Conventions

What are the conventions to follow when contributing to the Python API development?

General structure of NLG entries is highlighted at the top of *NLG.py* (and a running list of all possibilities found in this file in *NLG_template.txt*), but here is a reiteration with further detail:

DECLARATION

- The naming convention for functions is *undercaseclass_CamelCaseNLGType*. If the function is for a 'sub' type use *undercaseclass_sub_CamelCaseNLGType*. Underscores were decided because they allow the convention to stay simple and easy to read, despite being lengthier.

PARAMETERS

- Parameter variable names use underscores and are set equal to a *Missing* object (always, regardless of whether the parameter is required or optional ontologically).
- If the type is of *PropertyBundle* a *core_* object must be passed in, additional to the document object included for all functions. If the type is a 'sub' of a CASE class then an object of the class must be passed into the function.

DOCSTRINGS

- Docstrings for functions define each parameter and the return object.
- **These docstrings should be the direct place that additions/edits to the ontology occur.** An HTML, human-legible version of the NLG is automatically generated from this.
- Again, a list of all variations of definition wording is included in the *NLG_template.txt* file. When the definition specifies the ability for "more than one" thing to belong to the property it necessitates a Python list.
- When writing these definitions use periods and precede a CamelCase object type name with 'type'.

BODY (VERIFICATION CHECKS)

- Each function's body has **2 or 3 blocks** separated by newlines. The first, if applicable, is a check for 'sub' functions to verify their 'parent' object was made first and passed in. The second is for required properties and the third is for optional. Follow the spacing convention for easier peer review.
- Each verification check (may) consist of:
  - These lines **only exist for 'sub' objects**:
    assert (isinstance(object_parent, case.ParentClass) and (object_parent.type=='CaseTypeName')),
    "[function_name] parameter_name must be of type CaseTypeName."

  - These lines **only exist for required parameters** (to check if they are missing):
    assert not isinstance(param_name, Missing),
    "[function_name] param_name is required."
  - These lines **exist for all parameters**, required or optional:
    if not isinstance(param_name, Missing):
    >>>> <assert statement for check goes here>

  - The type verification for the parameter itself goes where ***<assert statement for check goes here>*** was shown in the last bullet. This will check that it is of type *int* for instance, but may also include checking that it is of a specific NLG type like *ControlledVocabulary*. If the docstring indicates a parameter could include more than one thing, an **additional verification** check is performed to **verify the parameter is a list** (e.g. a list of type *int*). Referring to the *NLG_template.txt* file will make this much easier to understand (variables named *t_* should be replaced).

- **Developer notetaking can save a lot of time** (since the NLG is very large). If there is at least one proper parameter verification check but another property has the definition "any number of any type" then no verification can occur ; use "*# NO CHECK: <parameter_name* in the body of the code. This helps keep track of properties that need better defining. If there is not at least one valid verification check then the entire object is not properly defined – this should never exist in a stable version of the API. If the docstrings cannot motivate any checks use "*#TODO:NothingElseToCheck*" to specify incomplete/ambiguous ontology. Finally, if a standard has not been defined yet for how to check a type (say we want to make sure all timestamps are not just of type *datetime* but that they have a specific format) then use "*#TODO:<type_name>*" as a placeholder so that after a convention is declared it can easily be updated.

RETURN

- Each function returns an object (instantiation) of its class (*Core*, *Context*, *Duck*, or *Sub*). For this creation use CamelCase (and fully capitalize any abbreviations) because this is what the CASE output file will display. Although programmatically usually the first letter is under case, readability is better with pure CamelCase (capitalization of the first letter). E.g., httpConnection vs. HTTPConnection, x509certificate vs. X509Certificate).

How should a Github issue be created?

When creating an issue on Github (on repository website click on *Issues* tab and then *New issue* button) use one label from each column below:

CASE-Ontology	NewObject
Tool-Mapping	NewFeature
Python-API	Improvement (edits to object or feature)
Glossary-Verifier	Bug (edits to object or feature)
Continuous-Testing	Documentation