Community Contribution Series

OSCAL-Pydantic: A python library for OSCAL

Credentive Security

Welcome to this Community Contribution Series presentation. Today we are discussing my python-based library for encoding and decoding OSCAL models.

Background

Before we dive into the details, let's start with some background.

Oscal-Pydantic

https://github.com/RS-Credentive/oscal-pydantic/

- An API for creating and manipulating OSCAL data models in Python
- Provides high-fidelity schemas of all OSCAL data elements
- Strives to be the reference implementation of OSCAL in Python

pip install oscal-pydantic

OSCAL-Pydantic, available from github, <NEXT>is a Python API for creating and manipulating OSCAL data models.

<NEXT>It provides high-fidelity schemas of all OSCAL data elements.

<NEXT>My goal is to develop the reference implementation of OSCAL in Python.

<NEXT>V1 can be obtained from pypi by running ‘pip install oscal-pydantic’

Building blocks

- Python (https://python.org)
  - Popular scripting/programming language
    - #2 behind Javascript on Github
  - Applications include
    - Web programming (server side)
    - Scientific Computing (Big Data/AI)
  - Strict, Dynamically typed language
- Pydantic (https://docs.pydantic.dev/latest/)
  - Most widely used data validation library for Python
  - Leverages type hints to provide strict, static typing
  - Supports serialization to/from JSON

<NEXT>OSCAL-Pydantic uses the popular Python programming language.

<NEXT>It is built on top of the Pydantic data validation library.

OSCAL Schema

https://pages.nist.gov/OSCAL-Reference/models/

- Built on top of NIST Metaschema
  - https://pages.nist.gov/metaschema/
  - a common, format-agnostic modeling framework supporting schema, code, and documentation generation

<NEXT>It provides validation of the OSCAL Schema, which is built on top of NIST's Metaschema modeling framework.

Oscal-Pydantic v1

V1 of the OSCAL-Pydantic library exists, and has been released. I'd like to spend a couple of minutes talking about how it was built and why I'm building a V2.

OSCAL-Pydantic v1

- Inspired by Compliance Trestle
  - https://github.com/IBM/compliance-trestle
- Dynamically generated from JSON Schema
  - OSCAL JSON Schema → Datamodel-code-generator → Pydantic Models
  - Hand tweaked to eliminate some issues with JSON Schema translation
    - JSON "format" vs Regex
    - Unicode Regex – e.g. "(\p{L}|_)(\p{L}|\p{N}|[.\-_])*"
- Outcome
  - Lightweight library to support generation and validation of OSCAL Data and import JSON objects

<NEXT>First, let me thank the compliance trestle project from IBM for inspiration. When I was first developing OSCAL tooling in Python, I imported the compliance trestle package just to use it's pydantic library. OSCAL-Pydantic v1 just split the validation library out as a standalone component.

<NEXT>Of course, compliance trestle is also built on top of other open source software, particularly a tool called "datamodel-code-generator". This tool takes a JSON Schema and converts it into pydantic models.

Since datamodel-code-generator is a little naïve and simplistic about how it does the conversion, some hand tweaking is required.

First, JSON schema supports two independent string format specifications for individual elements. It has something called a "format" – the documentation describes format as "basic semantic identification of certain kinds of string values that are commonly used". It also provides "pattern", used to restrict a string to a particular regular expression.

The interaction between "format" and "pattern" in JSON schema is undefined, as far

as I have been able to determine. If you define a “format” are you allowed to define a “pattern” as well? Apparently! Why would you? Who knows! What if the defined “pattern” conflicts with the “format”? It’s a mystery!

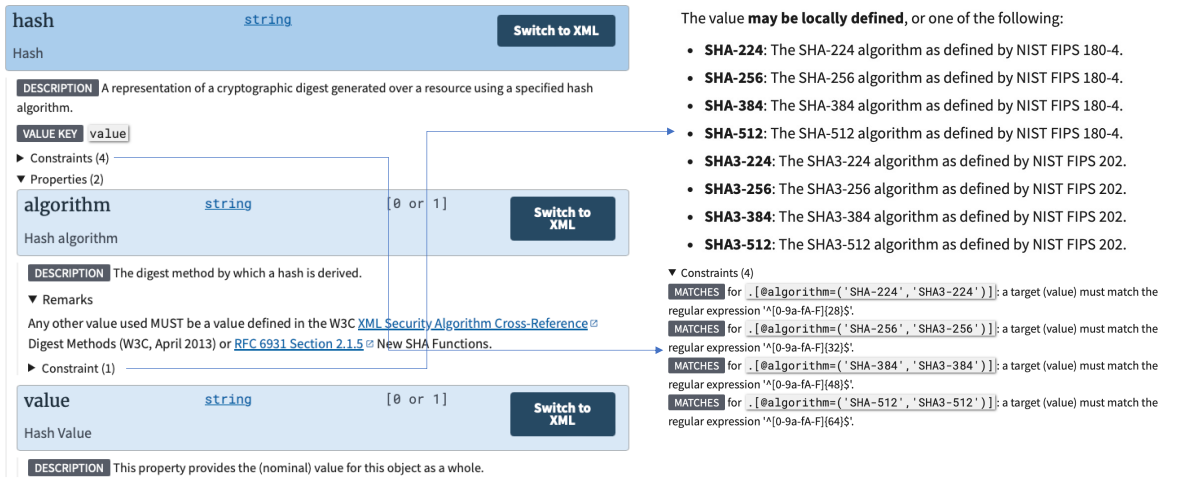
Pydantic has both concepts built in as well, but is very particular about how you mix and match them. Datamodel-code-generator will just try to map the JSON Schema format to the closest pydantic equivalent and then throw in the regex as well – that had to be cleaned up by hand.

Datamodel-code-generator also does not validate regex expressions during conversion, so you’ll get some regex that python can’t process, such as the Unicode regex on this slide. That also needs to be cleaned up by hand.

<NEXT>Nevertheless, it is possible to produce a basic. lightweight library for generation and validation of OSCAL Data in python with OSCAL-Pydantic v1.

There are some caveats however – more on that later.

Example OSCAL data element



This is an OSCAL data element which we can use as a simple example to show how OSCAL schema is translated into pydantic.

The hash element appears in “back matter”, under the “rlink” (Relative Link) element. It’s designed to ensure that the integrity of remote resources can be verified if included as part of an OSCAL model.

A hash consists of two values (both technically optional).

First is the algorithm used to define the hash, which must be expressed as a metaschema “string” and is further constrained <NEXT> to a list of supported algorithms (except “value may be locally defined” – which I think means that there is no constraint at all).

Second, there is a “value” which is the value of the hash. This is also a metaschema string, <NEXT> and is further constrained to valid hash values, which depends on the algorithm selected.

OSCAL-Pydantic v1 Example

```
class Hash(BaseModel):
     class Config:
          extra = Extra.forbid

   algorithm: Annotated[
      str,
      Field(
         description="Method by which a hash is derived",
         regex="^\\S(.*\\S)?$",
         title="Hash algorithm",
      ),
   ]
   value: str
```

Here is what the hash object looks like in OSCAL Pydantic v1.

Note:

- The Hash is a subclass of “BaseModel” – this is the basic model class in pydantic, and any additional requirements, such as the requirement that no other fields can be present (“Extra.forbid”), have to be added to each defined model.
- The fields are defined as “str” – which is just a string with no constraints. The constraints defined for the metaschema “string” are added as a regex to each field defined.
- Value does not have any constraints added at all. It’s just a string.

Issue: Machine code is not for humans

- Autogenerated schemas are tough to read and use
  - Lots of Root Models
  - A lot of repetition
- Difficult to extend/customize

We can see the limitations of automatically generating code from schema.

<NEXT>First, machine generated code is tough for humans to work with. The use of "root model" throughout, and the use of standard python/pydantic types means that there is a lot of repeated boilerplate code.

<NEXT>Secondly, it's very difficult to extend the models to create new elements, such as new properties.

Issue: Inherited JSON Schema limitations

- Constraints are limited to attribute values (regex) or “formats”
- No way to define relationships between attributes
  - IF “algorithm” == “SHA-224”
    - THEN ”value” must be a 28 character string
    - AND only comprised of the characters 0-9, a-f, or A-F

Of course, since oscal-pydantic v1 is built from the JSON schema, it inherits all the problems of that schema format.

<NEXT> We’ve already discussed the limitations of constraints, and how they can be either “formats” or regex, or both.

<NEXT>While this can be overcome for simple validation of string values, there’s no way to express more complex constraints or relationships, such as the ones identified on the slide.

There is no general way within JSON schema to restrict the value of an element based on the value of another element.

As a result, my application was producing OSCAL artifacts that conformed perfectly to the JSON Schema but were failing OSCAL validation.

Oscal-Pydantic v2

Back to the drawing board

So, for OSCAL-Pydantic v2, we went back to the drawing board.

Oscal-Pydantic v2 Approach

- Leverage Pydantic v2
  - 4x – 50x faster than Pydantic v1 (~17x in general)
- Hand produced
  - Less Repetition
  - Designed for humans to extend and customize
  - Closer alignment with underlying metaschema
  - (In Progress) Support for all validation rules

First, OSCAL pydantic v2 uses Pydantic v2 rather than pydantic v1. <NEXT>Pydantic v2 is between 4 and 50 times faster than v1, with an average improvement of 17 times.

<NEXT>Second, rather than automating the translation of the JSON Schema, we are hand coding the models. This has several benefits.

The code can be designed to improve clarity and minimize repetition, which makes it easier for humans to understand and extend.
Next, Pydantic models in oscal-pydantic v2 are much more closely aligned with the underlying metaschema.
Finally (and this is in-progress), oscal-pydantic v2 will support all of the OSCAL validation rules.

OSCAL-Pydantic v2 Example

```python
OscalString = Annotated[str, constr(pattern=r"\s.\S+")]

class Hash(base.OscalModel):
    algorithm: datatypes.OscalString | None = Field(
        description="""
        Method by which a hash is derived <…>

        """,
        default=None,
        pattern="^SHA3?-(224|256|384|512)$",
    )
    value: datatypes.OscalString | None = Field(
        description="""
        The value of the hash
        """,
        default=None,
    )
```

Before we dig into the details of the code, I'll start with an example that highlights some of the advantages.

1. <NEXT>Notice that we have defined a type called "OscalString" which is consisitantly validated according to the appropriate regex.
2. <NEXT>We use this type throughout the code as a way to ensure that the restrictions on model fields are consistently applied.
3. We also indicate when a field can be empty by allowing a value of "None"
4. <NEXT>Next, our Hash class is a subclass of "OscalModel" – this allows us to consolidate and centralize a lot of the configuration and other restrictions in a single place.
5. <NEXT>Note that we are able to add additional constraints when we define by using annotated types and fields together.

OSCAL-Pydantic v2 Example

```python
@model_validator(mode="after")
def validate_hash_for_algorithm(self):
    if self.algorithm is not None and self.value is None:
        raise ValueError("Hash Algorithm specified without Value")
    elif self.algorithm == "SHA-224" or self.algorithm == "SHA3-224":
        if len(self.value) == 28 and self.value_is_hex():
            return self
        else:
            raise ValueError("Hash value length or contents do not match algorithm")
    elif self.algorithm == "SHA-256" or self.algorithm == "SHA3-256":
        <…>
    elif self.algorithm == "SHA-384" or self.algorithm == "SHA3-384":
        <…>
    elif self.algorithm == "SHA-512" or self.algorithm == "SHA3-512":
        <…>
    else:
        return self
```

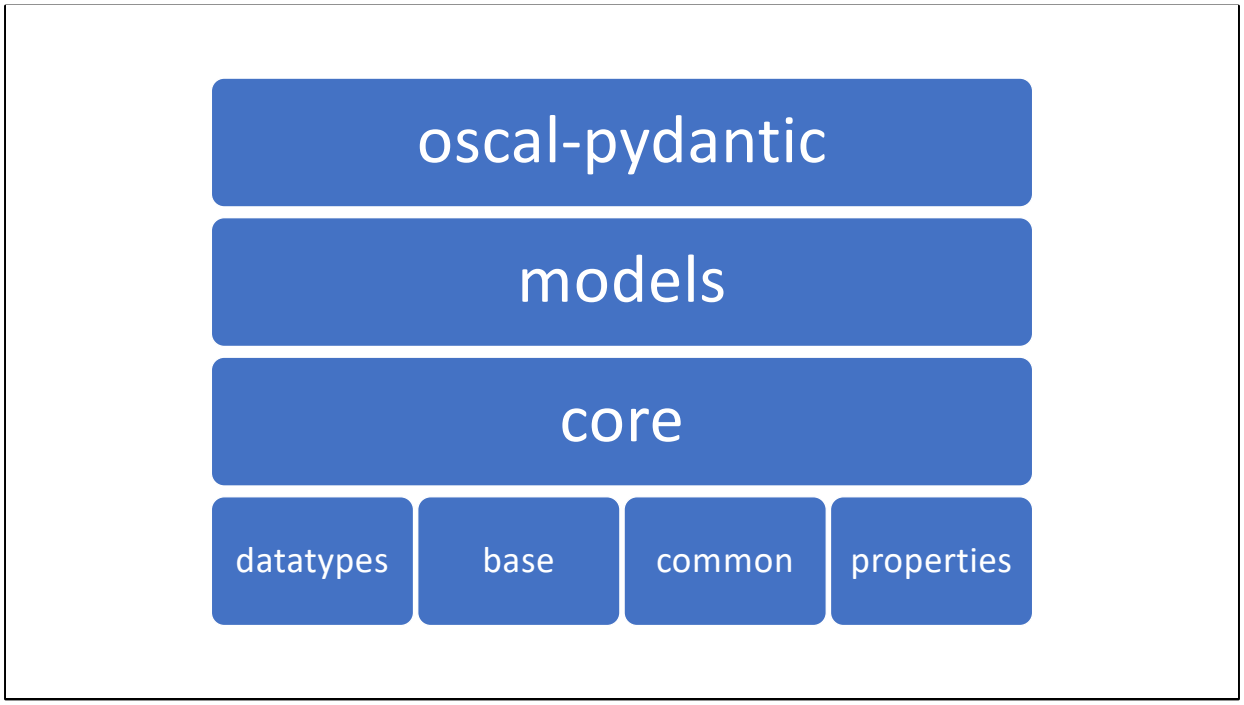
```python
def value_is_hex(self) -> bool:
    # Quick trick to check if a string is only HEX
    # try to convert it to an int.
    # If it doesn't work, there's a bad character in there.
    try:
        int(self.value, 16)
        return True
    except ValueError:
        return False
```

Finally, pydantic allows us to define validators to capture complex logic. We have the full power of the python language, so we can define arbitrarily complex rules.

A tour of OSCAL-Pydantic v2

Now that we understand the reason behind the new version, and have seen an example, I would like to take you on a brief tour of the package itself.

The package is composed of several individual modules.



The highest level of modules are the modules representing the models, such as catalog, profile, etc.

The “core” package underneath oscal-pydantic contains the fundamental building blocks of all the models. It can mostly be ignored unless you are extending the models.

Oscal-Pydantic core modules

- oscal_pydantic.core.datatypes
  - core Metaschema datatypes
- oscal_pydantic.core.base.OscalModel
  - Subclass of pydantic.BaseModel
  - Common Field Alias Generator
    - Json attributes (“-”) to snake_case (“_”)
    - Can’t use “class” as a field (reserved word)
  - Override “model_dump_json” method
    - Exclude null, always use alias, pretty print
  - Maybe common content validator code?

<NEXT>The most basic core module is “datatypes”. This module provides the basic validation rules for each of the metaschema datatypes used for any metaschema based schema, including the OSCAL schema.

<NEXT>The next core module is the ”OscalModel” – this module provides the basic constraints common to all of the Oscal Modules, and provides some important helper functions to support translation between OSCAL attributes and pydantic model fields.

<NEXT>Critically, it translates json attribute formats “like-this-example” to python camel case names “like_this_example”, and works around scenarios where oscal elements are named using reserved words like “class”

<NEXT>We also provide some nice features for exporting json from the models, such as excluding “null” attributes, and pretty printing.

<NEXT>One potentially useful feature is to provide a set of common validators that can be applied to any model if the proper inputs are applied – we’ll revisit this when we discuss the messy state of validation.

Oscal-Pydantic core modules

- oscal-pydantic.core.common
  - Common OSCAL elements that appear in many parts of models
- oscal-pydantic.core.properties
  - Implements documented OSCAL properties
  - Implements full set of constraints
  - User Extensible
  - UNDER CONSTRUCTION

<NEXT>The “common” submodule includes OSCAL elements that appear in many places. One example is the “metadata” element, which appears in every oscal model. "Hash" is in the common submodule.

<NEXT>The final module is “properties” - this is a shared module, like “common”, but due to the incredible complexity of the validation rules, properties have been split into their own module for now. We’ll discuss this again in a minute.

<NEXT>It should be noted that properties are ”UNDER CONSTRUCTION”, and is an area where assistance is needed.

Oscal-Pydantic modules

- oscal_pydantic.catalog <- Now
- oscal_pydantic.system_security_plan
- oscal_pydantic.profile
- oscal_pydantic.component_definition
- oscal_pydantic.assessment_plan
- oscal_pydantic.assessment_results
- oscal_pydantic.plan_of_action_and_milestones

Finally, we have the main modules of oscal-pydantic. We will eventually have one for each model, but right now we're still working out the catalog model. If we get the underlying core modules right, we will be able to quickly put the other top level models in place.

OSCAL Pydantic in Action

Help Wanted

This is an open-source project, so you are welcome to use it today! V1 has been published on Github and in PyPi, so you can use it today with “pip install oscal-pydantic”

Since it is an open-source project, you can also contribute to its development. All help and contributions are welcome, but I want to talk about some of the areas where the support is most needed.

Next Steps

- Fix Properties
- Finish Catalog
- Develop elements for Test Cases

The next steps for me are to finish implementing OSCAL Properties, finish the implementation of the catalog, and then start working on some basic data for test cases.

Help Wanted: Properties

- Properties are very special parts of the OSCAL specification
  - Complex validation rules
  - Numerous different types of properties
  - Designed to be extensible
- OSCAL Pydantic must also provide full validation for all the various properties, and must support extension
- Currently working to identify the best way to enforce validation rules in an easily extensible way.

Help Wanted: Test Cases

- Integration testing is critical to maintaining a high-quality implementation
- Testing requires a library of valid and invalid artifacts
- Needed for every project implementing OSCAL regardless of language or domain.
- Suggestion: Should test cases be a separate project maintained on behalf of the community?