smqtk-core

Release 0.19.0

Kitware, Inc.

Sep 16, 2022

CONTENTS:

Source Code (GitHub)
ReadTheDocs Documentation

This pure-python package provides a bedrock of interfaces and utilities to supercharge abstract interfaces with the ability to find their own implementations, as well as to be able to factory construct arbitrary implementations of an abstract interface with an input JSON-compliant configuration dictionary. These features may be utilized separately but are designed to work synergistically.

```python
from smqtk_core import Pluggable, Configurable
from smqtk_core.configuration import from_config_dict


# An abstract interface.
class MyNewInterface (Pluggable, Configurable):
    @abc.abstractmethod
    def work(self) -> None:
        "Abstract method for implementations to define."


if __name__ == "__main__"
    # Discover currently available implementations of the defined interface.
    implementation_set: Set[Type[MyNewInterface]] = MyNewInterface.get_impls()

    # With a configuration from some source (see the smqtk_core.configuration module)
    # we can instantiate a concrete instance for use.
    from my_app import get_config_from_somewhere
    config_json: Dict[str, Any] = get_config_from_somewhere()
    instance: MyNewInterface = from_config_dict(config_json, implementation_set)

    # Use the new instance!
    instance.work()
```

CHAPTER

ONE

INSTALLATION

There are two ways to get ahold of SMQTK-Core. The simplest is to install via the **pip** command. Alternatively, the source tree can be acquired and be locally developed using Poetry.

For more information on the use of Poetry, follow these links fr installation and usage documentation.

1.1 From `pip`

```
$ pip install smqtk-core
```

This method will install all of the same functionality as when installing from source. If you have an existing installation and would like to upgrade your version, provide the `-U`/`--upgrade` option.

1.2 From Source

The following assumes Poetry is already installed.

1.2.1 Quick Start

```
$ cd /where/things/should/go/
$ git clone https://github.com/Kitware/smqtk-core.git ./
$ poetry install
$ # Since we're from source we can test the installation.
$ poetry run pytest
$ # We can also build the local documentation as it may be more up to date then ReadTheDocs.
$ cd docs
$ make html
```

1.2.2 Installing Python dependencies

This project uses Poetry for depedency management, environment consistency, version management, package building and publishing to PYPI. Dependencies are abstractly defined in the `pyproject.toml` file. Additionally, specifically pinned versions are specified in the `poetry.lock` file for *development* environment consistency. Both of these files can be found in the root of the source tree.

The following command installs both installation and development dependencies as specified in the `pyproject.toml` file, with versions specified (including for transitive dependencies) in the `poetry.lock` file:

```
$ poetry install
```

1.2.3 Building the Documentation

The documentation for SMQTK-Core is maintained as a collection of reStructuredText documents in the `docs/` folder of the project. This documentation can be processed by the **Sphinx** documentation tool into a variety of documentation formats, the most common of which is HTML.

Within the `docs/` directory is a Unix `Makefile` (for Windows systems, a `make.bat` file with similar capabilities exists). This `Makefile` takes care of the work required to run **Sphinx** to convert the raw documentation to an attractive output format. For example, as shown in the quickstart, calling `make html` will generate HTML format documentation rooted at `docs/_build/html/index.html`.

Calling the command `make help` here will show the other documentation formats that may be available (although be aware that some of them require additional dependencies such as **TeX** or **LaTeX**)

Live Preview

While writing documentation in a mark up format such as reStructuredText it is very helpful to be able to preview the formatted version of the text. While it is possible to simply run the `make html` command periodically, a more seamless workflow of this is available. Within the `docs/` directory is a small Python script called `sphinx_server.py` that can simply be called with:

```
$ python sphinx_server.py
```

This will run a small process that watches the `docs/` folder contents, as well as the source files in `smqtk_core/`, for changes. **`make html`** is re-run automatically when changes are detected. This will serve the resulting HTML files at http://localhost:5500. Having this URL open in a browser will provide you with a relatively up-to-date preview of the rendered documentation.

CHAPTER

TWO

PLUGINS AND CONFIGURATION

The general concept of abstract interfaces allows users to create functionality in terms of the interface, separating the concerns of usage and implementation. This tooling is intended to enhance that concept by providing a straightforward way to expose and discover implementations of an interface, as well as factory functionality to create instances of an implementation from a JSON-compliant configuration structure. We provide two mixin classes and a number of utility functions to achieve this. While the two mixin classes function independently and can be utilized on their own, they have been designed such that their combination is symbiotic.

2.1 The `Pluggable` Mixin

Motivation: We want to be able to define interfaces to generic concepts and structures that higher-level functionality can be defined around without strictly catering themselves to any particular implementation. We additionally want to allow freedom in implementation variety without much adding to the implementation burden.

In SMQTK-Core, this is addressed via the *`Pluggable`* abstract mixin class:

```python
import abc
from smqtk_core.plugin import Pluggable

class MyInterface(Pluggable):
    @abc.abstractmethod
    def my_behavior(self, x: str) -> int:
        """My fancy behavior."""

class NumLetters(MyInterface):
    def my_behavior(self, x: str) -> int:
        return len(x)

class IntCast(MyInterface):
    def my_behavior(self, x: str) -> int:
        return int(x)

if __name__ == "__main__":
    # Discover currently available implementations and print out their names
    impl_types = MyInterface.get_impls()
    print("MyInterface implementations:")
    for t in impl_types:
        print(f"- {t.__name__}")
```

Running the above in a `.py` file would then output:

```
MyInterface implementations:
- NumLetters
- IntCast
```

This is of course a naive example where implementations are defined right next to the interface. This is not a requirement. Implementations may be spread out across other sub-modules within a package, or even in other packages. In the below section, *Plugin Discovery Methods*, and in the example *Creating an Interface and Exposing Implementations*, we will show how to expose implementations of a plugin-enabled interface.

2.1.1 Interfaces vs. Implementations

Classes that inherit from the *Pluggable* mixin are considered either plugin implementations, or further pluggable interfaces, depending on whether they fully implement abstract methods or not, respectively.

2.1.2 Plugin Discovery Methods

SMQTK-Core's plugin discovery via the *get_impls()* method currently allows for finding a plugin implementations in 3 ways:

- sub-classes of an interface type defined in the current runtime.
- within python modules listed in the environment variable specified by `YourInterface.PLUGIN_ENV_VAR`. (default SMQTK-Core environment variable name is `SMQTK_PLUGIN_PATH`, which is defined in `Pluggable.PLUGIN_ENV_VAR`).
- within python modules specified under the entry point extensions namespace defined by `YourInterface.PLUGIN_NAMESPACE` (default SMQTK-Core extension namespace is `smqtk_plugins`, which is defined in `Pluggable.PLUGIN_NAMESPACE`).

When exposing interface implementations, it is generally recommended to use a package's entry point extensions (3rd bullet above).

2.2 The `Configurable` Mixin

Motivation: We want generic helpers to enable serializable configuration for classes while minimally impacting standard class development.

SMQTK-Core provides the *Configurable* mixin class as well as other helper utility functions in *smqtk_core.configuration* for generating class instances from configurations. These use python's `inspect` module to determine constructor parameterization and default configurations.

Currently this module uses the JSON-serializable format as the basis for input and output configuration dictionaries as a means of defining a relatively simple playing field for communication. Serialization and deserialization is detached from these configuration utilities so tools may make their own decisions there. Python dictionaries are used as a medium in between serialization and configuration input/output.

Classes that inherit from *Configurable* *do* need to at a minimum implement the *get_config()* instance method. This is due to currently lacking the ability to introspect the connection between constructor parameters and how those values are retained in the class. See this method's doc-string for more details.

2.3 The Convenient Combination: `Plugfigurable`

It will likely be desirable to utilize both the *Pluggable* and *Configurable* mixins when constructing your own new interfaces. To facilitate this, and to reduce excessive typing, we provide the *Plugfigurable* helper class. This class does not add or change any functionality. It is merely a convenience to indicate the multiply inherit from both mixin types. Also regarding multiple in inheritance, either `Pluggable' nor :class:.Configurable` define a *__init__* method, so

2.4 Examples

2.4.1 Creating an Interface and Exposing Implementations

In this example, we will show:

- A simple interface definition that inherits from both *Pluggable* and *Configurable* via the convenient *Plugfigurable* class.
- A sample implementation that is defined in a different module.
- The subsequent exposure of that implementation via a python package's entry point extensions.

Let's start with the example interface definition which, let's say, is defined in the `MyPackage.interface` module of our hypothetical `MyPackage` package:

```
# File: MyPackage/interface.py
import abc
from smqtk_core import Plugfigurable

class MyInterface(Plugfigurable):
    """
    A new interface that transitively inherits from Pluggable and Configurable.
    """

    @abc.abstractmethod
    def my_behavior(self, x: str) -> int:
        """My fancy behavior."""
```

Then, in another package module, `MyPackage.implementation`, let's say we define the following implementation. This implementation will need to define all parent class abstract methods in order for the class to satisfy the definition of an "implementation" (see *Interfaces vs. Implementations*).

```
# File: MyPackage/implementation.py
from MyPackage.interface import MyInterface
from typing import Any, Dict

class MyImplementation(MyInterface):

    def __init__(self, paramA: int = 1, paramB: int = 2):
        """Implementation constructor."""
        self.a = paramA
        self.b = paramB

    # Abstract method from the Configurable mixin.
    def get_config(self) -> Dict[str, Any]:
```

(continues on next page)

(continued from previous page)

```
        # As per Configurable documentation, this should return the same
        # non-self keys as the constructor.
        return {
            "paramA": self.a,
            "paramB": self.b,
        }

    # Abstract method from MyInterface
    def my_behavior(self, x: str) -> int:
        """My fancy implementation."""
        ...
```

Lastly, our implementation should be exposed via our package's entrypoint metadata, using the "smqtk_plugins" namespace. This namespace value is derived from the base *Pluggable* mixin's `PLUGIN_NAMESPACE` class property. Entry point metadata may be specified for a package either via the `setuptools.setup()` function, the *setup.cfg* file, or, when using poetry, a `[tool.poetry.plugins."..."]` section in the `pyproject.toml` file. This is illustrated in the following:

a) `setuptools.setup()` function

```
from setuptools import setup
setup(
    entry_points={
        "smqtk_plugins": [
            "unique_key = MyPackage.implementation",
            ...
        ]
    }
)
```

b) The `setup.cfg` file

```
[options.entry_points]
smqtk_plugins =
    unique_key = MyPackage.implementation
    ...
```

c) with Poetry in the `pyproject.toml` file

```
[tool.poetry.plugins."smqtk_plugins"]
"my_plugins" = "MyPackage.implementation"
...
```

Now, this implementation will show up as an available implementation of the interface class:

```
>>> from MyPackage.interface import MyInterface
>>> MyInterface.get_impls()
{<class 'MyPackage.implementation.MyImplementation'>}
```

The `MyImplementation` class above should also be all set for configuration because it defines the one required abstract method *get_config()* and because it's constructor is only anticipating JSON-compliant data-types. If more complicated types are desired by the constructor, that is completely OK! In such cases, additional methods would need to be overridden as defined in the *smqtk_core.configuration* module.

2.4.2 Supporting configuration with more complicated constructors

In the above example, only very simple, already-JSON-compliant data types were utilized in the constructor. This not intended to imply that there is a restriction in constructor specification. Instead, the *Configurable* mixin provides overridable methods to insert conversion to and from JSON-compliant dictionaries, to provide a class "default" configuration as well as for factory-generation of a class instance from an input configuration.

Instead of the above implementation, let us consider a slightly more complex implementation of `MyInterface` defined above. In this new implementation we show the implementation of two additional class methods from the *Configurable* mixin: *Configurable.get_default_config()* and *Configurable.from_config()*. These allow us to customize the how we maintain JSON-compliance.

```
from MyPackage.interface import MyInterface
from typing import Any, Dict, Type, TypeVar
from datetime import datetime


C = TypeVar("C", bound="DateContainer")


class DateContainer (MyInterface):
    """
    This example implementation takes in datetime instances as constructor
    parameters, one of which has a default.
    Both parameters in this example require a `datetime` instance value at
    construction time, but since `b_date` has a default value, it is not
    strictly required that an input configuration provide a value for the
    `b_date` parameter since there is a default to draw upon.
    """

    def __init__(
        self,
        a_date: datetime,
        b_date: datetime = datetime.utcfromtimestamp(0),
    ):
        self.a_date = a_date
        self.b_date = b_date

    # NEW FROM PREVIOUS EXAMPLE
    # Abstract method from the Configurable mixin.
    @classmethod
    def get_default_config(cls) -> Dict[str, Any]:
        # Utilize the mixin-class implementation to introspect our
        # constructor and make a parameter-to-value dictionary.
        cfg = super().get_default_config()
        # We are ourself, so we know that cfg['a_date'] has a default value
        # and it will be a datetime instance.
        cfg['b_date'] = datetime_to_str(cfg['b_date'])
        # We know that `a_date` is not given a default, so its "default"
        # value of None, which is JSON-compliant, is left alone.
        return cfg

    # NEW FROM PREVIOUS EXAMPLE
    # Abstract method from the Configurable mixin.
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def from_config(
        cls: Type[C],
        config_dict: Dict,
        merge_default: bool = True
    ) -> C:
        # Following the example found in the Configurable.from_config
        # doc-string.
        config_dict = dict(config_dict)
        # Convert required input data into the constructor-expected types.
        # This implementation will expectedly error if the expected input
        # is missing.
        config_dict['a_date'] = str_to_datetime(config_dict['a_date'])
        # b_date might not be there because there's a default that can fill
        # in.
        b_date = config_dict.get('b_date', None)
        if b_date is not None:
          config_dict['b_date'] = str_to_datetime(b_date)
        return super().from_config(config_dict, merge_default=merge_default)

    # Abstract method from the Configurable mixin.
    def get_config(self) -> Dict[str, Any]:
        # This now matches the same complex-to-JSON conversion as the
        # `get_default_config`. We show the use of a helper function to
        # reduce code duplication.
        return {
            "a_date": datetime_to_str(self.date),
        }

    # Abstract method from MyInterface
    def my_behavior(self, x: str) -> int:
        """My fancy implementation."""
        ...


def datetime_to_str(dt: datetime) -> str:
    """ Local helper function for config conversion from datetime. """
    # This conversion may be arbitrary to the level of detail that this
    # local implementation considers important. We choose to use strings
    # here as an example, but there's nothing special that requires that
    # other than JSON type compliance.
    return str(dt)


def str_to_datetime(s: str) -> datetime:
    """ Local helper function for config conversion into datetime """
    # Reverse of above converter.
    if '.' in s:  # has decimal seconds
        return datetime.strptime(s, "%Y-%m-%d %H:%M:%S.%f")
    return datetime.strptime(s, "%Y-%m-%d %H:%M:%S")
```

The above then allows us to create instances of this instance with the JSON config:

```
>>> inst = DateContainer.from_config({
...   "a_date": "2021-01-01 00:00:00.123123"
...   "b_date": "1970-01-01 00:00:01",
... })
>>> str(inst.a_date)
'2021-01-01 00:00:00.123123'
>>> str(inst.b_date)
'1970-01-01 00:00:01'
```

Or even just:

```
>>> inst = DateContainer.from_config({
...   "a_date": "2021-01-01 00:00:00.123123"
... })
>>> str(inst.a_date)
'2021-01-01 00:00:00.123123'
>>> str(inst.b_date)
'1970-01-01 00:00:00'
```

While such inline usage is less likely to be called so directly as opposed to just calling the constructor, this form is useful when constructing directly from a deserialized configuration:

```
>>> # Maybe received this from a web request!
>>> from_file = '{"a_date": "2021-01-01 00:00:00.123123", "b_date": "1970-01-01 00:00:01
↪"}'=
>>> import json
>>> inst = DateContainer.from_config(json.loads(from_file))
>>> str(inst.a_date)
'2021-01-01 00:00:00.123123'
>>> str(inst.b_date)
'1970-01-01 00:00:01'
```

Instances may also be "cloned" by creating a new instance from the current configuration output from another instance using the *Configurable.get_config()* instance method:

```
>>> # assume and `inst` from above
>>> inst2 = DataContainer.from_config(inst.get_config())
>>> assert inst.a_date == inst2.a_date
>>> assert inst.b_date == inst2.b_date
```

This is again a little silly in the demonstration context because we know what instance type and attributes are to construct a second one, however if the concrete instance is not known at runtime, this may be an alternate means of constructing a duplicate instance (at least "duplicate" in regards to construction).

2.4.3 Multiple Implementation Choices

One usage mode of *smqtk_core.configuration* configuration is when a configuration slot may be comprised of one of multiple *Configurable*-implementing choices. Also found in the *smqtk_core.configuration* module are additional helper functions for navigating this use-case:

- *make_default_config()*
- *to_config_dict()*
- *from_config_dict()*

These methods utilize JSON-compliant dictionaries to represent configurations that follow the schema:

```
{
    "type": "object",
    "properties": {
        "type": {"type": "string"},
    },
    "additionalProperties": {"type": "object"},
}
```

Getting Defaults from a group of types

When programmatically creating a multiple-choice configuration structure for output, the *make_default_config()* function may be convenient to use. This takes in some `Iterable` of *Configurable*-inheriting types and returns a JSON-compliant dictionary. This may be useful, for example, when a higher-order tool wants to programmatically generate a default configuration for itself for serialization or for some other interface.

This function may be called with an independently generated `Iterable` of inputs, however this also melds well with the:meth:.*Pluggable.get_impls* class method. For example, let us use the pluggable `MyInterface` class defined above:

```
>>> from smqtk_core.configuration import make_default_config
>>> cfg_dict = make_default_config(MyInterface.get_impls())
>>> cfg_dict
{
    "type": None,
    "__main__.MyImplementation": {
        "paramA": 1,
        "paramB": 2
    },
    "__main__.DateContainer": {
        "a_date": None,
        "b_date": "1970-01-01 00:00:00"
    }
}
```

See the method documentation for additional details.

Factory constructing from configuration

The opposite of above, the *from_config_dict()* will take a configuration multiple-choice dictionary structure and "hydrate" an instance of the configured type with the configured parameters (if it's available, of course). This function again takes an `Iterable` of *Configurable*-inheriting types which again melds well with the `Pluggable.get_impls` class method where applicable.

For example, if we take the "default" configuration output above, change the `"type"` value to refer to the "MyImplementation" type and pass it to this function, we get a fully constructed instance:

```
>>> from smqtk_core.configuration import from_config_dict
>>> # Let's modify the config away from default values.
>>> cfg_dict['__main__.MyImplementation'] = {'paramA': 77, 'paramB': 444}
>>> inst = from_config_dict(cfg_dict, MyInterface.get_impls())
>>> assert inst.a == 77
>>> assert inst.b == 444
```

See the method documentation for additional details.

Help with writing unit tests for Configurable-implementing types

When creating new implementations of things that includes *Configurable* functionality it is often good to include tests that make sure the expected configuration capabilities operate as they should. We include a helper function to lower the cost of adding such a test: *configuration_test_helper()*. This will exercise certain runtime-assumptions that are not strictly required to define and construct *Configurable*-inheriting types.

For an example, let's assume we are writing a unit test for the above-defined `MyImplementation` class:

```
>>> from smqtk_core.configuration import configuration_test_helper
>>>
>>> class TestMyImplementation:
...     def test_config(self):
...         inst = MyImplementation(paramA=77, paramB=444)
...         for i in configuration_test_helper(inst):
...             # Checking that yielded instance properties are as expected
...             assert i.a == 77
...             assert i.b == 444
```

See the method documentation for additional details.

2.5 Module References

2.5.1 smqtk_core

class smqtk_core.**Plugfigurable**

When you don't want to have to constantly inherit from two mixin classes all the time, we provide this as a convenience that descends from both mixin classes: Pluggable and Configurable.

2.5.2 smqtk_core.plugin

Helper functions and mixin interface for implementing class type discovery, filtering and a convenience mixin class.

This package provides a number of *discover_via_...* functions that return sets of type instances as found by the method described by that function.

These methods may be composed to create a pool of types that may be then filtered via the *filter_plugin_types* function to those types that are specifically "plugin types" for the given interface class. See the *is_valid_plugin* function documentation for what it means to be a "plugin" of an interface type.

While the above are defined in fairly general terms, the *Pluggable* class type defined last here is a mixin class that utilizes all of the above in a manner specific manner for the purposes of SMQTK. This mixin class defines the class-method `get_impls()` that will return currently discoverable plugins underneath the type it was called on. This discovery will follow the values of the `PLUGIN_ENV_VAR` and `PLUGIN_NAMESPACE` class variables defined in the interface class you are calling `get_impls()` from, using inherited values if not immediately specified.

Because these plugin semantics are pretty low level and commonly utilized, logging can be extremely verbose. Logging in this module, while still exists, is set to emit only at log level 1 or lower ("trace").

NOTE: The type annotations for *discover_via_subclasses* and *filter_plugin_types* are currently set to the broad *Type* annotation. Ideally these should use *Type[T]* instead, but there is currently a known issue with mypy where it aggressively assumes that an annotated type *must* be constructable, so it emits an error when the functions are called with an abstract *interface_type*. When this is resolved in mypy these annotations should be updated.

exception smqtk_core.plugin.**NotAModuleError**

Exception for when the *discover_via_entrypoint_extensions* function found an entrypoint that was *not* a module specification.

class smqtk_core.plugin.**Pluggable**

Interface for classes that have plugin implementations.

classmethod get_impls() → Set[Type[P]]

Discover and return a set of classes that implement the calling class.

See the various `discover_via_*()` functions in this module for more details on the logic of how implementing classes (aka "plugins") are discovered.

The class-level variables `PLUGIN_ENV_VAR` and `PLUGIN_NAMESPACE` may be overridden to change what environment and entry-point extension are looked for, respectively.

Returns

Set of discovered class types that are considered "valid" plugins of this type. See *is_valid_plugin()* for what we define a "valid" type to be relative to this class.

classmethod is_usable() → bool

Check whether this class is available for use.

Since certain plugin implementations may require additional dependencies that may not yet be available on the system, or other runtime conditions, this method may be overridden to check for those and return a boolean saying if the implementation is available for usable. When this method returns *True*, the class is declaring that it should be constructable and usable in the current environment.

By default, this method will return True unless a subclass overrides this class-method with their specific logic.

NOTES:

- This should be a class method

- **When an implementation is deemed not usable, this should emit a** (user) warning, or some other kind of logging, detailing why the implementation is not available for use.

Returns
Boolean determination of whether this implementation is usable in the current environment.

Return type
bool

smqtk_core.plugin.**discover_via_entrypoint_extensions**(*entrypoint_ns: str*) → Set[Type]

Discover and return types defined in modules exposed through the entry-point extensions defined for the given namespace by installed python packages.

Other installed python packages may define one or more extensions for a namespace, as specified by *ns*, in their "setup.py". This should be a single or list of extensions that specify modules within the installed package where plugins for export are implemented.

Currently, this method only accepts extensions that export a module as opposed to specifications of a specific attribute in a module. This is due to other methods of type discovery not necessarily honoring the selectivity that specific attribute specification provides (Looking at you *__subclasses__*...).

For example, as a single specification string:

```
...
entry_points={
    "smqtk_plugins": "my_package = my_package.plugins"
]
...
```

Or in list form of multiple specification strings:

```
...
entry_points = {
    "smqtk_plugins": [
        "my_package_mode_1 = my_package.mode_1.plugins",
        "my_package_mode_2 = my_package.mode_2.plugins",
    ]
}
...
```

Parameters
entrypoint_ns – The name of the entry-point mapping in to look for extensions under.

Returns
Set of discovered types from the modules and class types specified in the extensions under the specified entry-point.

smqtk_core.plugin.**discover_via_env_var**(*env_var: str*) → Set[Type]

Discover and return types specified in python-importable modules specified in the given environment variable.

We expect the given environment variable to define zero or more python module paths from which to yield all contained type definitions (i.e. things that descent from *type*). If there is an empty path element, it is skipped (e.g. "foo::bar:baz" will only attempt importing *foo*, *bar* and *baz* modules).

These python module paths should be separated with the same separator as would be used in the PYTHONPATH environment variable specification.

If a module defines no class types, then no types are included from that source for return.

An expected use-case for this discovery method is for modules that are not installed but otherwise accessible via the python search path. E.g. local modules, modules accessible through PYTHONPATH search path modification, modules accessible through *sys.path* modification.

Any errors raised from attempting to import a module are propagated upward.

Parameters
env_var – The name of the environment variable to read from.

Raises
ModuleNotFoundError – When one or more module paths specified in the given environment variable are not importable.

Returns
Set of discovered types from the modules specified in the environment variable's contents.

smqtk_core.plugin.**discover_via_subclasses**(*interface_type: Type*) → Set[Type]

Utilize the `__subclasses__` to discover nested subclasses for a given interface type.

This approach will be able to observe any implementations that have been defined, anywhere at all, at the point of invocation, which can circumvent efforts towards specificity that other discovery methods may provide. For example, *discover_via_entrypoint_extensions* may return a single type that was specifically exported from a module whereas this method will, called afterwards, yield all the other types defined in that entry-point-imported module.

The use of this discovery method may also result in different returns depending on the import state at the time of invocation. E.g. further imports may increase the quantity of returns from this function.

This function uses depth-first-search when traversing subclass tree.

Reference:
https://docs.python.org/3/library/stdtypes.html#class.__subclasses__

NOTE: subclasses are retained via weak-references, so if a normal condition
is exposing types from something that otherwise raised an exception or if a local definition is leaking, apparently an *import gc; gc.collect()* wipes out the return as long as it's not referenced, of course as long as its reference is not retained by something.

Parameters
interface_type – The interface type to recursively find subclasses under.

Returns
Set of recursive subclass types under *interface_type*.

smqtk_core.plugin.**filter_plugin_types**(*interface_type: Type*, *candidate_pool: Collection[Type]*) → Set[Type]

Filter the given set of types to those that are "plugins" of the given interface type.

See the documentation for *is_valid_plugin()* for what we define a "plugin type" to be relative to the given *interface_type*.

We consider that there may be duplicate type instances in the given candidate pool. Due to this we will consider an instance of a type only once and return a set type to contain the validated types.

Parameters

- **interface_type** – The parent type to filter on.
- **candidate_pool** – Some iterable of types from which to collect interface type plugins from.

Returns
Set of types that are considered "plugins" of the interface types following the above listed rules.

`smqtk_core.plugin.is_valid_plugin`(*cls: Type*, *interface_type: Type*) → bool

Determine if a class type is a valid candidate for plugin discovery.

In particular, the class type `cls` must satisfy several conditions:

1. It must not literally be the given interface type.
2. It must be a strict subtype of `interface_type`.
3. It must not be an abstract class. (i.e. no lingering abstract methods or properties if the *abc.ABCMeta* metaclass has been used).
4. If the cls is a subclass of Pluggable, it must report as usable via its is_usable() class method.

Logging for this function, when enabled can be very verbose, and is only active with a logging level of 1 or lower.

Parameters

- **cls** – The class type whose validity is being tested
- **interface_type** – The base class under consideration

Returns
`True` if the class is a valid candidate for discovery, and `False` otherwise.

Return type
bool

2.5.3 smqtk_core.configuration

Helper interface and functions for generalized object configuration, to and from JSON-compliant dictionaries.

While this interface and utility methods should be general enough to add JSON-compliant dictionary-based configuration to any object, this was created in mind with the SMQTK plugin module.

Standard configuration dictionaries should be JSON compliant take the following general format:

```
{
    "type": "one-of-the-keys-below",
    "ClassName1": {
        "param1": "val1",
        "param2": "val2"
    },
    "ClassName2": {
        "p1": 4.5,
        "p2": null
    }
}
```

The "type" key is considered a special key that should always be present and it specifies one of the other keys within the same dictionary. Each other key in the dictionary should be the name of a `Configurable` inheriting class type. Usually, the classes named within a block inherit from a common interface and the "type" value denotes a selection of a specific sub-class for use, though this is not required property of these constructs.

class `smqtk_core.configuration.Configurable`

Interface for objects that should be configurable via a configuration dictionary consisting of JSON types.

classmethod from_config(*config_dict: Dict*, *merge_default: bool = True*) → C

Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This base method is adequate without modification when a class's constructor argument types are JSON-compliant. If one or more are not, however, this method then needs to be overridden in order to convert from a JSON-compliant stand-in into the more complex object the constructor requires. It is recommended that when complex types *are* used they also inherit from the *Configurable* in order to hopefully make easier the conversion to and from JSON-compliant stand-ins.

When this method *does* need to be overridden, this usually looks like the following pattern:

```
D = TypeVar("D", bound="MyClass")

class MyClass (Configurable):

    @classmethod
    def from_config(
        cls: Type[D],
        config_dict: Dict,
        merge_default: bool = True
    ) -> D:
        # Perform a shallow copy of the input ``config_dict`` which
        # is important to maintain idempotency.
        config_dict = dict(config_dict)

        # Optionally guarantee default values are present in the
        # configuration dictionary.  This is useful when the
        # configuration dictionary input is partial and the logic
        # contained here wants to use config parameters that may
        # have defaults defined in the constructor.
        if merge_default:
            config_dict = merge_dict(cls.get_default_config(),
                                     config_dict)

        #
        # Perform any overriding of `config_dict` values here.
        #

        # Create and return an instance using the super method.
        return super().from_config(config_dict,
                                   merge_default=merge_default)
```

Note on type annotations: When defining a sub-class of configurable and override this class method, we will need to defined a new TypeVar that is bound at the new class type. This is because super requires a type to be given that descends from the implementing type. If *C* is used as defined in this interface module, which is upper-bounded on the base *Configurable* class, the type analysis will see that we are attempting to invoke super with a type that may not strictly descend from the implementing type (`MyClass` in the example above), and cause an error during type analysis.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns
Constructed instance from the provided config.

abstract get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns
JSON type compliant configuration dictionary.

Return type
dict

classmethod get_default_config() → Dict[str, Any]

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class's constructor takes as arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns
Default configuration dictionary for the class.

Return type
dict

```
>>> # noinspection PyUnresolvedReferences
>>> class SimpleConfig(Configurable):
...     def __init__(self, a=1, b='foo'):
...         self.a = a
...         self.b = b
...     def get_config(self):
...         return {'a': self.a, 'b': self.b}
>>> self = SimpleConfig()
>>> config = self.get_default_config()
>>> assert config == {'a': 1, 'b': 'foo'}
```

smqtk_core.configuration.**cls_conf_from_config_dict**(*config: Dict*, *type_iter: Iterable[Type[T]]*) → Tuple[Type[T], Dict]

Helper function for getting the appropriate type and configuration sub-dictionary based on the provided "standard" SMQTK configuration dictionary format (see above module documentation).

Parameters

- **config** – Configuration dictionary to draw from.
- **type_iter** – An iterable of class types to select from.

Raises

ValueError –

This may be raised if:

- type field not present in `config`.
- type field set to `None`
- type field did not match any available configuration in the given config.
- Type field did not specify any implementation key.

Returns

Appropriate class type from `type_iter` that matches the configured type as well as the sub-dictionary from the configuration. From this return, `type.from_config(config)` should be callable.

`smqtk_core.configuration.`**cls_conf_to_config_dict**(*cls: Type*, *conf: Dict*) → Dict

Helper function for creating the appropriate "standard" smqtk configuration dictionary given a *Configurable*-implementing class and a configuration for that class.

This very simple function simply arranges a semantic class key and an associated dictionary into a normal pattern used for configuration in SMQTK:

```
>>> class SomeClass (object):
```

... pass >>> cls_conf_to_config_dict(SomeClass, {0: 0, 'a': 'b'}) == { ... 'type': 'smqtk_core.configuration.SomeClass', ... 'smqtk_core.configuration.SomeClass': {0: 0, 'a': 'b'} ... } True

Parameters

- **cls** (*type[Configurable]*) – A class type implementing the *Configurable* interface.
- **conf** (*dict*) – SMQTK standard type-optioned configuration dictionary for the given class and dictionary pair.

Returns

"Standard" SMQTK JSON-compliant configuration dictionary

Return type

dict

`smqtk_core.configuration.`**configuration_test_helper**(*inst: C*, *config_ignored_params: Union[Set, FrozenSet] = frozenset({})*, *from_config_args: Sequence = ()*) → Tuple[C, C, C]

Helper function for testing the get_default_config/from_config/get_config methods for class types that in part implement the Configurable mixin class. This function also tests that `inst`'s parent class type's `get_default_config` returns a dictionary whose keys' match the constructor's inspected parameters (except "self" of course).

This constructs 3 additional instances based on the given instance following the pattern:

```
inst-1  ->  inst-2  ->  inst-3
        ->  inst-4
```

This refers to `inst-2` and `inst-4` being constructed from the config from `inst`, and `inst-3` being constructed from the config of `inst-2`. The equivalence of each instance's config is cross-checked with the other instances. This is intended to check that a configuration yields the same class configurations and that the config does not get mutated by nested instance construction.

This function uses assert calls to check for consistency.

We return all instances constructed in case the caller wants to make additional instance integrity checks.

Parameters

- **inst** (Configurable) – Configurable-mixin inheriting class to test.
- **config_ignored_params** (*set[str]*) – Set of parameter names in the instance type's constructor that are ignored by `get_default_config` and `from_config`. This is empty by default.
- **from_config_args** (*tuple*) – Optional additional positional arguments to the input `inst.from_config` method after the configuration dictionary.

Returns

Instance 2, 3, and 4 as described above.

Return type

(*Configurable*,*Configurable*,*Configurable*)

smqtk_core.configuration.**from_config_dict**(*config: Dict*, *type_iter: Iterable[Type[C]]*, **args: Any*) → C

Helper function for instantiating an instance of a class given the configuration dictionary `config` from available types provided by `type_iter` via the `Configurable` interface's `from_config` class-method.

`args` are additionally positional arguments to be passed to the type's `from_config` method on return.

Example: >>> class SimpleConfig(Configurable): … def __init__(self, a=1, b='foo'): … self.a = a … self.b = b … def get_config(self): … return {'a': self.a, 'b': self.b} >>> example_config = { … 'type': 'smqtk_core.configuration.SimpleConfig', … 'smqtk_core.configuration.SimpleConfig': { … "a": 3, … "b": "baz" … }, … } >>> inst = from_config_dict(example_config, {SimpleConfig}) >>> isinstance(inst, SimpleConfig) True >>> inst.a == 3 True >>> inst.b == "baz" True

Raises

- **ValueError** –

  This may be raised if:

  - type field not present in `config`.
  - type field set to `None`
  - type field did not match any available configuration in the given config.
  - Type field did not specify any implementation key.
- **AssertionError** – This may be raised if the class specified as the configuration *type*, is present in the given `type_iter` but is not a subclass of the `Configurable` interface.
- **TypeError** – Insufficient/incorrect initialization parameters were specified for the specified `type`'s constructor.

Parameters

- **config** – Configuration dictionary to draw from.
- **type_iter** – An iterable of class types to select from.
- **args** (*object*) – Other positional arguments to pass to the configured class' `from_config` class method.

Returns

Instance of the configured class type as specified in `config` and as available in `type_iter`.

smqtk_core.configuration.**make_default_config**(*configurable_iter: Iterable[Type[C]]*) → Dict[str, Union[None, str, Dict]]

Generated default configuration dictionary for the given iterable of Configurable-inheriting types.

For example, assuming the following simple class that descends from `Configurable`, we would expect the following behavior:

```
>>> # noinspection PyAbstractClass
>>> class ExampleConfigurableType (Configurable):
...     def __init__(self, a, b):
...         ''' Dummy constructor '''
>>> make_default_config([ExampleConfigurableType]) == {
...     'type': None,
...     'smqtk_core.configuration.ExampleConfigurableType': {
...         'a': None,
...         'b': None,
...     }
... }
True
```

Note that technically `ExampleConfigurableType` is still abstract as it does not implement `get_config`. The above call to `make_default_config` still functions because we only use the `get_default_config` class method and do not instantiate any types given to this function. While functionally acceptable, it is generally not recommended to draw configurations from abstract classes.

The `"type"` returned is `None` because we explicitly do not make any decisions about an appropriate default type. Additionally, this value stays `None` even when there is just one choice as we do not assume that is a valid choice as well as do not assume that the default configuration for that choice is valid for construction. This serves to cause the user to explicitly check that the multiple-choice configuration is set to point to a choice as well as that the choice is properly configured.

Parameters

configurable_iter – An iterable of class types class types that sub-class `Configurable`.

Returns

Base configuration dictionary with an empty `type` field, and containing the types and initialization parameter specification for all implementation types available from the provided getter method.

smqtk_core.configuration.**to_config_dict**(*c_inst:* Configurable) → Dict

Helper function that transforms the configuration dictionary retrieved from `configurable_inst` into the "standard" SMQTK configuration dictionary format (see above module documentation).

For example, with a simple Configurable derived class:

```
>>> class SimpleConfig(Configurable):
...     def __init__(self, a=1, b='foo'):
...         self.a = a
...         self.b = b
...     def get_config(self):
...         return {'a': self.a, 'b': self.b}
>>> e = SimpleConfig(a=2, b="bar")
>>> to_config_dict(e) == {
...     "type": "smqtk_core.configuration.SimpleConfig",
...     "smqtk_core.configuration.SimpleConfig": {
...         "a": 2,
```

(continues on next page)

(continued from previous page)

```
...         "b": "bar"
...     }
... }
True
```

Parameters
c_inst (Configurable) – Instance of a class type that subclasses the Configurable interface.

Returns
Standard format configuration dictionary.

Return type
dict

CHAPTER

THREE

SMQTK REVIEW PROCESS

The purpose of this document is to define the process for reviewing and integrating branches into SMQTK. This encompasses all SMQTK repositories.

See `CONTRIBUTING.md` for guidelines on contributing to SMQTK.

See release process for guidelines on the release process for SMQTK.

The review process consists of the following steps:

3.1 Pull Request

A PR is initiated by a user intending to integrate a branch from their forked repository. Before the branch is integrated into the SMQTK master branch, it must first go through a series of checks and a review to ensure that the branch is consistent with the rest of the repository and doesn't contain any issues.

3.1.1 Workflow Status

The submitter must set the status of their PR.

Draft

Indicates that the submitter does not think that the PR is in a mergeable state. Once they complete their work and think that the PR is mergeable, they may set the status to `Open`.

Open

Indicates that a PR is ready for review. This indicates that the submitter of the PR thinks that the branch is ready to be merged. If the submitter is still working on the PR and simply wants feedback, they must request it and leave their branch marked as a `Draft`.

Closed

Indicates that the PR is resolved or discarded.

3.2 Continuous Integration

3.2.1 kwrobot

Runs basic checks on the commits submitted in a PR. Should kwrobot find any issues with the build, the diagnostics are written out prompting the submitter to correct the reported issues. If there are no issues, kwrobot simply reports a successful build. The branch must pass this check before it can be merged.

Some reports such as whitespace issues will need to be corrected by rewriting the commit

3.2.2 LGTM Analysis

Runs a more advanced code analysis tool over the branch that can address issues that the submitter might not have noticed. Should LGTM find an issue, it will write a comment on the PR. The comment should be addressed by the submitter before continuing to submit for review.

3.2.3 lint

Runs flake8 to quality check the code style. You can run this check manually in your local repository with `poetry run flake8`.

3.2.4 MyPy

Performs static type analysis. You can run this check manually in your local repository with `poetry run mypy`.

3.2.5 Unittests

Runs the unittests created under `tests/`. You can run this check manually in your local repository with `poetry run pytest`.

3.3 Human Review

Once the automatic checks are either resolved or addressed, the submitted PR will need to go through a human review. Reviewers should add comments to provide feedback and raise potential issues. Should the PR pass their review, the reviewer should then indicate that it has their approval using the Github review interface to flag the PR as `Approved`.

A review can still be requested before the checks are resolved, but the PR must be marked as a `Draft`. Once the PR is in a mergeable state, it will need to undergo a final review to ensure that there are no outstanding issues.

If a PR is not a draft and has an approving review, it can be merged at any time.

3.4 Resolving a Branch

3.4.1 Merge

Once a PR receives an approving review and is no longer marked as a `Draft`, the repository maintainers can merge, closing the pull request. It is recommended that the submitter delete their branch after the PR is merged.

3.4.2 Close

If it is decided that the PR will not be integrated into SMQTK, then it can be closed through Github.

CHAPTER

FOUR

RELEASE PROCESS AND NOTES

4.1 Steps of the SMQTK Release Process

Three types of releases are expected to occur: - major - minor - patch

See the `CONTRIBUTING.md` file for information on how to contribute features and patches.

The following process should apply when any release that changes the version number occurs.

4.1.1 Create and merge version update branch

Major and Minor Releases

Major and minor releases may add one or more trivial or non-trivial features and functionalities.

1. Create a new branch off of the `master` named something like `update-to-v{NEW_VERSION}`, where `NEW_VERSION` is the new `X.Y` version.

   a. Use the `scripts/update_release_notes.sh` script to update the project version number, create `docs/release_notes/v{NEW_VERSION}.rst`, and add a new pending release notes stub file.

   ```
   $ ./scripts/update_release_notes.sh minor
   ```

   b. Add a descriptive paragraph under the title section of `docs/release_notes/v{NEW_VERSION}.rst` summarizing this release.

2. Push the created branch to the upstream repository, not your fork (this is an exception to the normal forky workflow).

3. Create a pull/merge request for this branch with `release` as the merge target. This is to ensure that everything passes CI testing before making the release. If there is an issue, then topic branches should be made and merged into this branch until the issue is resolved.

4. Get a positive review.

5. Merge version bump branch into the `release` branch.

6. Tag the resulting merge commit. See *Tag new version* below for how to do this.

7. As a repository administrator, merge the `release` branch into `master` locally and push the updated `master` to upstream. (Replace "upstream" in the example below with your applicable remote name.)

```
$ git fetch --all
$ git checkout upstream/master
$ git merge --log --no-ff upstream/release
$ git push upstream master
```

Patch Release

A patch release should only contain fixes for bugs or issues with an existing release. No new features or functionality should be introduced in a patch release. As such, patch releases should only ever be based on an existing release point (git tag).

This list assumes we are creating a new patch release off of the *latest* release version, i.e. off of the `release` branch. If a patch release for an older release version is being created, see the *Patching an Older Release* section.

1. Create a new branch off of the `release` branch named something like `update-to-v{NEW_VERSION}`, where `NEW_VERSION` is the target `X.Y.Z`, including the bump in the patch (`Z`) version component.

    a. Use the `scripts/update_release_notes.sh` script to update the project version number, create `docs/release_notes/v{NEW_VERSION}.rst`, and add a new pending release notes stub file. E.g.

    ```
    $ ./scripts/update_release_notes.sh patch
    ```

    b. Add a descriptive paragraph under the title section of `docs/release_notes/v{NEW_VERSION}.rst` summarizing this release.

2. Push the created branch to the upstream repository, not your fork (this is an exception to the normal forky workflow).
3. Create a pull/merge request for this branch with `release` as the merge target. This is to ensure that everything passes CI testing before making the release. If there is an issue, then topic branches should be made and merged into this branch until the issue is resolved.
4. Get a positive review.
5. Merge the pull/merge request into the `release` branch.
6. Tag the resulting merge commit. See *Tag new version* below for how to do this.
7. As a repository administrator, merge the `release` branch into `master` locally and push the updated `master` to upstream. (Replace "upstream" with your applicable remote name.)

```
$ git fetch --all
$ git checkout upstream/master
$ git merge --log --no-ff upstream/release
$ git push upstream master
```

Patching an Older Release

When patching a major/minor release that is not the latest pair, a branch needs to be created based on release version being patched to integrate the specific patches into. This branch should be prefixed with `release-` to denote that it is a release integration branch. Patch topic-branches should be based on this branch. When all fix branches have been integrated, follow the *Patch Release* section above, replacing `release` branch references (merge target) to be the `release-...` integration branch. Step 6 should be to merge this release integration branch into `release` first, and *then* `release` into `master`, if applicable (some patches may only make sense for specific versions).

4.1.2 Tag new version

Release branches are tagged in order to record where in the git tree a particular release refers to. All release tags should be in the history of the `release` and `master` branches (barring exceptional circumstances).

We will prefer to use local `git tag` commands to create the release version tag, pushing the tag to upstream. The version tag should be applied to the merge commit resulting from the above described `update-to-v{NEW_VERSION}` topic-branch ("the release").

See the example commands below, replacing `HASH` with the appropriate git commit hash, and `UPSTREAM` with the appropriate remote name. We show using Poetry's version command to consistently access the current package version.

```
$ git checkout HASH
# VERSION="v$(poetry version -s)"
$ git tag -a "$VERSION" -F docs/release_notes/"$VERSION".rst
$ git push UPSTREAM "$VERSION"
```

After creating and pushing a new version tag, a GitHub "release" should be made. Navigate to the releases page on GitHub and click the `Draft a new release` button in the upper right. The newly added tag should be selected in the "Choose a tag" drop-down. The "Release Title" should be the version tag (i.e. "v#.#.#"). Copy and paste this version's release notes into the `Describe this release` text box. Remember to check the `This is a pre-release` check-box if appropriate. Click the `Public release` button at the bottom of the page when complete.

4.2 Release Notes

4.2.1 Pending Release Notes

Updates / New Features

Fixes

4.2.2 v0.15.0

This is the initial release of `smqtk-core`, spinning off from v0.14.0 of the monolithic `smqtk` library.

4.2.3 v0.16.0

This minor release primarily introduces the *`Plugfigurable`* class type, which is a very simple type that combines the existing *`Pluggable`* and *`Configurable`* types that are commonly utilized together.

Updates / New Features

Interfaces

- Added a new *`Plugfigurable`* class that provides a convenient handle to inherit from both *`Pluggable`* and *`Configurable`* at the same time.

CI

- Updated gitlab CI rules to trigger the test jobs during merge requests, tag builds and branch pipelines when the branch in question is the default branch (i.e. master).
- Added python 3.9 to the gitlab matrix of versions tested. Added 3.9 to the `setup.py` metadata.

- Updated requirements versions to the latest for each package.

Fixes

Docs

- Minor fixes to the `sphinx_server.py` helper tool: Removed a watch to a directory that does not exist.

Misc.

- Fixed `MANIFEST.in` file to not include a no-longer-present file and to include all `requirements/*.txt` files.
- Removed unnecessary `.dockerignore` file.

4.2.4 v0.17.0

Updates / New Features

Pluggable

- Removed *__init__* method and added a *__new__* in its place. The behavior is the same, but is now less fragile from override and addresses some issues with type-checking during some multiple inheritance situations.

Misc.

- Now standardize to using Poetry for environment/build/publish management.
  - Collapsed pytest configuration into the `pyproject.toml` file.
  - Updated release process documentation to reflect the use of Poetry.
- Add explicit ReadTheDocs configuration file `.readthedocs.yaml`.

Fixes

CI

- Update CI configurations to use Poetry.

Docs

- Fix incorrect filepath relative to this repository in the release process documentation.
- Fix for use with poetry where appropriate.

4.2.5 v0.18.0

This minor update removes the runtime requirement on the `setuptools` package in favor of equivalent functionality that is in the standard library from python version 3.8 and onward. For python versions 3.7 and lower, we introduce the common `importlib-metadata` backport package.

Additional updates may be found below.

Updates / New Features

Dependencies

- Remove dependency on `setuptool`'s `pkg_resources` module. Taking the stance of bullet #5 in from Python's Packaging User-guide with regards to getting this package's version. The "needs to be installed" requirement from before is maintained.
- Added dependency on the `importlib-metadata` backport package for installations in environments that use python version less-than 3.8.
- Added `ipython` (and appropriately supporting version of `jedi`) as development dependencies. Minimum versioning is set to support python 3.6 (current versions follow NEP 29 and thus require python 3.7+).

Documentation

- Clarified plugin implementation entrypoint example to include setuptools `setuptools.setup()` function and `setup.cfg` file.
- Revisions from proof-reading.

Plugins

- Entry-point discovery functionality now uses `importlib_metadata` / `importlib.metadata` as appropriate for the version of python being used.

Testing

- Added terminal-output coverage report in the standard pytest config in the `pyproject.toml` file.

Fixes

4.2.6 v0.18.1

Updates / New Features

CI

- Added use of CodeCov. Fixed/added unittests to show 100% of test code, e.g. no dead-code in the tests. CodeCov checks different coverage bars for test and package scopes.

Documentation

- Added an FAQ to cover some basic questions about SMQTK as a whole.
- Added additional examples around using the `smqtk_core.configuration` module: non-trivial constructor type configuration, and multiple-choice configuration specification, generation and utilization.
- Added a review process document that details how SMQTK PR's should be reviewed.
- Update the "Libraries" section of the `README.md` file for more detail.
- Added `release.yml` to github workflow to automate part of the release and publishing process. Releases will now be triggered after a new tag is pushed to the repo. The maintainer will no longer have to manually create releases and manually publish to PYPI.

Fixes

Documentation

- Fix some white-space and indentation issues in the `README.md` file.

4.2.7 v0.19.0

This minor update brings in a number of repository updates, deprecates python 3.6 support, and deprecates the *__new__* override in the *Pluggable* mixin interface to make it easier to use under real-world conditions.

Most repository updates have revolved around stream lining the mechanics going into creating and publishing releases.

The override of *__new__* provided functionality that, from experience and observation, was never actually made use of or added any actual security to anything. This feature came at the cost of interrupting some downstream tools (e.g. in ipython when inspecting a class with the ?-suffix) and adding complexity in how Pluggable needed to be used in defining interfaces (multiple inheritance and mixin UX). With the feature not being used, these costs don't make any sense any more. Thus, this override was removed for simplicity. Now, "not usable" implementations may be manually constructed if the user really desires to, under the presumption that manual construction implies manual consent. The *is_usable()* class method of course still exists and super-logic may continue to make use of it in relation to use-prevention or use-warning logic.

Updates / New Features

CI

- Reverted previous release automation due to unintended side-effects. Created a revised publish action to more simply publish the package to pypi, guarding against activating on forks of the repository. This workflow has been made to be reusable by other repositories' workflows.
- Modified CI unittests workflow to run for PRs targeting branches that match the *release** glob.
- Added additional step in unittest workflow to install optional package requirements.
- Reduced CodeCov report submission by skipping this step on scheduled runs.
- Update code-cov action usage to use v3.

Contribution Guide

- Added instructions to update pending release when making a contribution.

Dependencies

- Updated minimum required python version to 3.7 to follow python end of life.
- Updated development abstract dep versions to "*" since we do not currently require any specific versions.

Documentation

- Updated release instructions to be clear on where to push created release branches. This now includes instructions related to a `release` branch.
- Expanded top-level contributing document with more details.

Plugin

- Added a suggestion to fix *NotAModuleError*.
- Removed __new__ override to prevent construction of "not usable" implementations. This feature has never been observed/utilized in the wild and it's removal simplifies tool interactions and use complexity.

Miscellaneous

- Removed CODE_OF_CONDUCT file. This is not something that we can enforce at this time so it will be removed.
- Added SMQTK-Descriptors to the `README.md` package list and graphic.
- Added script to help with updating versioning and updating changelog during the release process.
- Updated README to include reference to the SMQTK-IQR package.
- Periodic update of pinned dependency versions in lock file.
- Added missing assert failure message to configuration test helper.
- Added properties file for use with SonarQube and SonarCloud.

Fixes

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS

5.1 What is SMQTK?

SMQTK is an open-source light-weight framework for developing interfaces that have built-in implementation discovery and factory construction from configuration.

5.2 Why would I use SMQTK over KWIVER?

You should use SMQTK if you want to define your own python algorithm implementations and don't want to develop in C++. However, if the algorithm implementations that you want to use are already defined in KWIVER, then KWIVER would be the better option.For more info, see the `README.md`

5.3 I've used SMQTK before, but what are these broken out packages?

In 2021, `SMQTK v0.14.0` was broken out from one monolithic library into several distinct libraries labeled as `SMQTK-Core`, `SMQTK-Classifier`, `SMQTK-Image-IO`, and more. The decision was part of a new effort to reduce technical debt and isolate functionality to preserve the light-weight design.

5.4 Can I contribute to SMQTK?

Of course! To add in your own implementation see `CONTRIBUTING.md`. Additionally you can contribute by helping review any outstanding branches in any one of the SMQTK repos. For guidelines on reviewing please see review_process

5.5 What does SMQTK encompass?

SMQTK is currently composed of 7 different libraries, all of which are pip installable. `SMQTK-Core` provides the underlying tooling and is utilized by the other 6 packages which provide more complex functionality and pluggable implementations. The following are all the packages associated with SMQTK:

- SMQTK-Core provides the basic tools for developing interfaces.
- SMQTK-Dataprovider provides data structure abstractions.
- SMQTK-Image-IO provides interfaces and implementations around image input/output.

- SMQTK-Descriptors provides algorithms and data structures around computing descriptor vectors.
- SMQTK-Classifier provides interfaces and implementations around classification.
- SMQTK-Indexing provides interfaces and implementations around the k-nearest-neighbor algorithm.
- SMQTK-Relevancy provides interfaces and implementations around providing search relevancy estimation.

CHAPTER

SIX

MISCELLANEOUS DOCUMENTATION

6.1 Setting Up smqtk-core with SonarCloud

6.1.1 Create a SonarCloud Organization

This will house our repository ("project") dashboard and is parallel to the GitHub organization.

- Go to SonarCloud
- Click the "+" button in the upper right near user drop-down.
- Select "Create a new organization".
- Follow instructions.
  - Part of this will be "Installing" the SonarCloud app at the organization level. Select either "All repositories" or select repositories to enable SonarCloud access to. Currently we have selected access only to the **`smqtk-*`_** set of repositories.

6.1.2 Create a Project for smqtk-core

- Go to SonarCloud.
- Click the "+" button in the upper right near user drop-down.
- Select the "Kitware" organization created above.
- Check our repo in the repos listed.

By default this will enable automatic scanning for common situations including PR submissions and master-branch updates. This is currently acceptable however this does not report coverage information due to the SonarCloud automatic scanning not having access to the unit test coverage reports generated during the GitHub Actions-based CI workflow. In the future we may want to switch to performing the SonarQube scanning action inside our CI workflow, however currently there is the hurdle where PRs from forks do not have the appropriate access to submit scan reports without learning private security tokens. The following section is provided as purely experimental/historical information.

6.1.3 Set Analysis Method to GitHub Actions

NOTE: *THIS IS NOT THE CURRENT CONFIGURATION.* This documentation here is notionally provided as it was written during experimentation and maybe has future value if we attempted again.

This is less setting something up but more deactivating the default setting to use SonarCloud Automatic Analysis. We chose to do this as SonarCloud automatic analysis has no ability to consider unittest code coverage, as this is only a byproduct of the CI unittests.

- Go to project page (e.g. https://sonarcloud.io/project/overview?id=Kitware_SMQTK-Core)
- Administration –> Analysis Method
- Toggle off "SonarCloud Automatic Analysis"

We "enabled" the GitHub action method by explicitly configuring a job in our GitHub CI workflow to use the SonarCloud GitHub Action to run the scanner and submit the report.

```
jobs:

  # Add the following step to the end of the `unittests` job.
  unittests:
    steps:
      - name: Upload test coverage artifact
        uses: actions/upload-artifact@v2
        with:
          name: test-coverage-${{ matrix.python-version }}-${{ matrix.opt-extra }}
          path: coverage.xml

  # For analysis of codebase and submission to sonarcloud.io.
  sonarcloud:
    runs-on: ubuntu-latest

    # Requires coverage info generated from a previous unit-test run.
    needs: unittests

    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Pull coverage XML from test run
        uses: actions/download-artifact@v2
        with:
          # Just one version of the coverage. Sonar can't merge a matrix of
          # coverages yet (2021-06).
          name: test-coverage-3.7-
      - name: Munge coverage.xml source path for SonarCloud container use
        # Bridge the gap between what pytest-cov outputs and SonarCloud repo
        # source directory assumptions of "/github/workspace/".
        # We check that the coverage source line is as expected before sed call.
        run: |
          grep -E '^\s*<source>.*</source>$' coverage.xml
          sed -Ei 's|^(\s*)<source>.*</source>|\1<source>/github/workspace/smqtk_core</
↪source>|g' coverage.xml
      - name: SonarCloud Scan
        uses: sonarsource/sonarcloud-github-action@master
```

(continues on next page)

(continued from previous page)

```
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

This *requires* that a `SONAR_TOKEN` secret to be defined in the github *repository* settings to be accessed within the workflow. The application to the *repository* is important because PRs from forks will not have access the secret defined in the upstream repository, thus the job will fail for those fork-based PRs.

The value for this secret is from a SonarCloud personal security token (see below on how to make one of these). Currently, Paul Tunison (paul.tunison@kitware.com) holds the security token that is set to the `SONAR_TOKEN` secret in the upstream smqtk-core repository on GitHub. In the future this may be changed by a repo admin, as described below.

Create a Personal Security Token

- Go to SonarCloud.
- At the drop-down user option in the upper right –> select "My Account".
- Click "Security" tab.
- Enter the descriptive label for the token in the editable box –> click "Generate".
- Retain one-time-exposed value of token appropriately.

Set GitHub Repository `SONAR_TOKEN` Secret

- Go to the SMQTK-Core repository page.
- Click on "Settings" –> "Secrets"
- If no existing `SONAR_TOKEN` secret, click on the "New repository secret" in the upper right.
  - This will open a new page to enter the name of the secret, which should be "SONAR_TOKEN" and a space to paste the value of the secret, which should be the token hash as generated above in *Create a personal security token*.
- Otherwise, update the existing secret value by clicking on the "Update" button to the right of the secret entry.
  - This will open a new page to enter a new value for the existing `SONAR_TOKEN` secret (i.e. cannot change the name of the secret). There should be a space to paste the value of the secret, which should be the token hash as generated above in *Create a personal security token*.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX