Yet Another Garbage Object Tracker for Python

Release 0.5.0

Andreas Maier

Mar 07, 2020

Contents

CHAPTER 1

Introduction

Yagot (Yet Another Garbage Object Tracker) is a tool for Python developers to help find issues with garbage collection and memory leaks:

- It can determine the set of *collected objects* caused by a function or method.

  Collected objects are objects Python could not immediately release when they became unreachable and that were eventually released by the Python garbage collector. Frequently this is caused by the presence of circular references into which the object to be released is involved. The garbage collector is designed to handle circular references when releasing objects.

  Collected objects are not a problem per se, but they can contribute to large memory use and can often be eliminated.

- It can determine the set of *uncollectable objects* caused by a function or method.

  Uncollectable objects are objects Python was unable to release during garbage collection, even when running a full collection (i.e. on all generations of the Python generational garbage collector).

  Uncollectable objects remain allocated in the last generation of the garbage collector. On each run on its last generation, the garbage collector attempts to release these objects. It seems to be rare that these continued attempts eventually succeed, so these objects can basically be considered memory leaks.

See section *Background* for more detailed explanations about object release in Python.

Yagot is simple to use in either of the following ways:

- It provides a pytest plugin named *yagot* that detects collected and uncollectable objects caused by the test cases. This detection is enabled by specifying command line options or environment variables and does not require modifying the test cases.
- It provides a Python decorator named `garbage_checked()` that detects collected and uncollectable objects caused by the decorated function or method. This allows using Yagot independent of any test framework or with other test frameworks such as nose or unittest.

Yagot works with a normal (non-debug) build of Python.

1.1 Usage

Here is an example of how to use Yagot to detect collected objects caused by pytest test cases using the command line options provided by the *Yagot pytest plugin*:

```
$ cat examples/test_1.py
def test_selfref_dict():
    d1 = dict()
    d1['self'] = d1

$ pytest examples --yagot -k test_1.py
====================================== test session starts␣
↪=======================================
platform darwin -- Python 3.7.5, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /Users/maiera/PycharmProjects/yagot/python-yagot
plugins: cov-2.8.1, yagot-0.1.0.dev1
yagot: Checking for collected and uncollectable objects, ignoring types: (none)
collected 2 items / 1 deselected / 1 selected

examples/test_1.py .E                                                    ␣
↪   [100%]

============================================ ERRORS␣
↪============================================
_____________________________ ERROR at teardown of test_selfref_dict _____________________
↪___________

item = <Function test_selfref_dict>

    def pytest_runtest_teardown(item):
        """
        py.test hook that is called when tearing down a test item.

        We use this hook to stop tracking and check the track result.
        """
        config = item.config
        enabled = config.getvalue('yagot')
        if enabled:
            import yagot
            tracker = yagot.GarbageTracker.get_tracker()
            tracker.stop()
            location = "{file}::{func}". \
                format(file=item.location[0], func=item.name)
>           assert not tracker.garbage, tracker.assert_message(location)
E           AssertionError:
E             There were 1 collected or uncollectable object(s) caused by function␣
↪examples/test_1.py::test_selfref_dict:
E
E             1: <class 'dict'> object at 0x10df6ceb0:
E             {'self': <Recursive reference to dict object at 0x10df6ceb0>}
E
E           assert not [{'self': {'self': {'self': {'self': {'self': {...}}}}}}]
E            +  where [{'self': {'self': {'self': {'self': {'self': {...}}}}}}] =
↪<yagot._garbagetracker.GarbageTracker object at 0x10df15f10>.garbage

yagot_pytest/plugin.py:148: AssertionError
========================== 1 passed, 1 deselected, 1 error in 0.07s␣
↪===========================
```

(continues on next page)

(continued from previous page)

Here is an example of how to use Yagot to detect collected objects caused by a function using the *garbage_checked()* decorator on the function. The yagot pytest plugin is loaded in this example and it presence is reported by pytest, but it is not used:

```
$ cat examples/test_2.py
import yagot

@yagot.garbage_checked()
def test_selfref_dict():
    d1 = dict()
    d1['self'] = d1

$ pytest examples -k test_2.py
========================================= test session starts␣
↪=========================================
platform darwin -- Python 3.7.5, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /Users/maiera/PycharmProjects/yagot/python-yagot
plugins: cov-2.8.1, yagot-0.1.0.dev1
collected 2 items / 1 deselected / 1 selected

examples/test_2.py F                                                    ␣
↪    [100%]

============================================== FAILURES␣
↪==============================================
_____________________________________ test_selfref_dict ______________________________
↪___________

args = (), kwargs = {}, tracker = <yagot._garbagetracker.GarbageTracker object at␣
↪0x1078853d0>
ret = None, location = 'test_2::test_selfref_dict'
@py_assert1 = [{'self': {'self': {'self': {'self': {'self': {...}}}}}}], @py_assert3␣
↪= False
@py_format4 = "\n~There were 1 collected or uncollectable object(s) caused by␣
↪function test_2::test_selfref_dict:\n~\n~1: <class 'di...elf': {'self': {'self': {
↪'self': {...}}}}}}] = <yagot._garbagetracker.GarbageTracker object at 0x1078853d0>.
↪garbage\n}"

    @functools.wraps(func)
    def wrapper_garbage_checked(*args, **kwargs):
        "Wrapper function for the garbage_checked decorator"
        tracker = GarbageTracker.get_tracker()
        tracker.enable(leaks_only=leaks_only)
        tracker.start()
        tracker.ignore_types(type_list=ignore_types)
        ret = func(*args, **kwargs)  # The decorated function
        tracker.stop()
        location = "{module}::{function}".format(
            module=func.__module__, function=func.__name__)
>       assert not tracker.garbage, tracker.assert_message(location)
E       AssertionError:
E         There were 1 collected or uncollectable object(s) caused by function test_
↪2::test_selfref_dict:
E
E         1: <class 'dict'> object at 0x1078843c0:
```

(continues on next page)

(continued from previous page)

```
E         {'self': <Recursive reference to dict object at 0x1078843c0>}
E
E       assert not [{'self': {'self': {'self': {'self': {'self': {...}}}}}}]
E        +  where [{'self': {'self': {'self': {'self': {'self': {...}}}}}}] = <yagot._
↪garbagetracker.GarbageTracker object at 0x1078853d0>.garbage

yagot/_decorators.py:67: AssertionError
============================== 1 failed, 1 deselected in 0.07s␣
↪===============================
```

In both usages, Yagot reports that there was one collected or uncollectable object caused by the test function. The assertion message provides some details about that object. In this case, we can see that the object is a `dict` object, and that its 'self' item references back to the same `dict` object, so there was a circular reference that caused the object to become a collectable object.

That circular reference is simple enough for the Python garbage collector to break it up, so this object does not become uncollectable.

The failure location and source code shown by pytest is the wrapper function of the `garbage_checked` decorator and the `pytest_runtest_teardown` function since this is where it is detected. The decorated function or pytest test case that caused the objects to be created is reported in the assertion message using a "module::function" notation.

Knowing the test function `test_selfref_dict()` that caused the object to become a collectable object is a good start for identifying the problem code, and in our example case it is easy to do because the test function is simple enough. If the test function is too complex to identify the culprit, it can be split into multiple simpler test functions, or new test functions can be added to check out specific types of objects that were used.

As an exercise, test the standard `dict` class and the `collections.OrderedDict` class by creating empty dictionaries. You will find that on CPython 2.7, `collections.OrderedDict` causes collected objects (see issue9825).

The `garbage_checked` decorator can be combined with any other decorators in any order. Note that it always tracks the next inner function, so unless you want to track what garbage other decorators create, you want to have it directly on the test function, as the innermost decorator, like in the following example:

```
import pytest
import yagot

@pytest.mark.parametrize('parm2', [ ... ])
@pytest.mark.parametrize('parm1', [ ... ])
@yagot.garbage_checked()
def test_something(parm1, parm2):
    pass  # some test code
```

1.2 Installation

1.2.1 Supported environments

Yagot is supported in these environments:

- Operating Systems: Linux, Windows (native, and with UNIX-like environments), OS-X
- Python: 2.7, 3.4, and higher

1.2.2 Installing

- Prerequisites:
  - The Python environment into which you want to install must be the current Python environment, and must have at least the following Python packages installed:
    - setuptools
    - wheel
    - pip
- Install the yagot package and its prerequisite Python packages into the active Python environment:

```
$ pip install yagot
```

1.2.3 Installing a different version

The examples in the previous sections install the latest version of Yagot that is released on PyPI. This section describes how different versions of Yagot can be installed.

- To install an older released version of Yagot, Pip supports specifying a version requirement. The following example installs Yagot version 0.1.0 from PyPI:

```
$ pip install yagot==0.1.0
```

- If you need to get a certain new functionality or a new fix that is not yet part of a version released to PyPI, Pip supports installation from a Git repository. The following example installs yagot from the current code level in the master branch of the python-yagot repository:

```
$ pip install git+https://github.com/andy-maier/python-yagot.git@master#egg=yagot
```

1.2.4 Verifying the installation

You can verify that yagot is installed correctly by importing the package into Python (using the Python environment you installed it to):

```
$ python -c "import yagot; print('ok')"
ok
```

In addition, you can verify that pytest picks up the yagot plugin, by looking at the pytest help. If it shows the yagot options, the plugin has been properly picked up:

```
$ pytest --help | grep yagot
--yagot               Enables checking for collected and uncollectable
--yagot-leaks-only    Limits the checking to only uncollectable (=leak)
--yagot-ignore-types=TYPE[,TYPE[...]]
```

In case of trouble with the installation, see the *Troubleshooting* section.

CHAPTER 2

Background

2.1 Understanding object release in Python

This section explains how Python releases objects, the role of the Python garbage collector, the role of object references, and how memory leaks can exist in Python. It is a rather brief description just enough to understand what is relevant to your Python program. Unless otherwise stated, the description applies to CPython with its generational cyclic garbage collector (introduced in CPython 2.0).

Python has two mechanisms for releasing objects:

- Immediate release based on reference counting:

  If the reference count of an object drops to zero (e.g. because its referencing variable goes out of scope), the reference count of all objects it references are decreased by one, and the original object is immediately released.

  Those referenced objects whose reference count drops to zero as a result of being decreased will in turn be immediately released.

  If it triggers, this process always succeeds. If the original object is involved in circular references, the process does not trigger in the first place, because the reference count of the original object never drops to zero.

- Asynchronous release during garbage collection:

  Objects that can possibly be involved in circular references are tracked by the Python garbage collector. Python schedules runs of the garbage collector based on object allocation and deallocation heuristics. The garbage collector is able to release isolated sets of objects with circular references by breaking up these circular references.

  As an optimization, the garbage collector has 3 generations, and the heuristics are optimized such that younger generations are collected more often than older generations.

  There are circumstances whereby multiple collection runs are necessary to release objects, and where objects can end up as uncollectable, meaning they normally stay around until the Python process terminates.

When an object is created, Python decides whether or not the object is tracked by the garbage collector. Whether an object is tracked or not can change over the lifetime of the object. The `gc.is_tracked()` function returns whether a particular object is currently tracked or untracked.

Untracked objects are not listed in any generation of the garbage collector, so their release can only be done by the reference counting mechanism. Python guarantees that objects that are untracked are not involved in circular object references so their reference count does have a chance to drop to zero.

Tracked objects may or may not be involved in circular references. If they are not, their release happens via the reference counting mechanism. If they are, a garbage collection run (on the generation they are in) can release them. During each collection run on a generation, the garbage collector examines the objects in that generation to find isolated sets of objects with circular references that are unreachable from outside of themselves. The garbage collector tries to break up circular references to release such sets of objects.

Objects that become tracked (either at object creation or later) are always put into the first generation of the garbage collector. When a garbage collection run on a particular generation does not succeed in releasing an unreachable isolated set of objects, the objects are moved into the next (older) generation.

Objects in the last (oldest) generation that survive a collection run stay in the last generation. They will be attempted to be released again and again during future runs of the garbage collector on the last generation. In most cases, that will not succeed, but there are cases where it will. Unreachable isolated set of objects in the last generation of the garbage collector are called *uncollectable objects* and unless a future run succeeds in releasing them, they remain allocated until the Python process terminates.

Uncollectable objects may be considered memory leaks, but this distinction is not black and white because they may be successfully released in a future run of the garbage collector on the last generation.

Among the possible reasons for objects to become uncollectable are:

- Before Python 3.4, the presence of the __del__() method in a set of unreachable objects involved in circular references caused these objects to be uncollectable. The reason is that Python cannot safely decide for an order in which the `__del__()` methods should be invoked, because they might be using the reference to another object in the cycle. In Python 3.4, PEP 442 – Safe object finalization was implemented which ensures that the `__del__()` methods are invoked exactly once in all cases, but in a set of objects that has reference cycles, the order of invocation is undefined. Also, this change no longer causes objects with circular references to become uncollectable just because they have a `__del__()` method.
- Reference counting bugs in Python modules implemented in native languages such as C may cause objects to be uncollectable. For example, a Python module implemented in C could properly increase an object reference upon creation but forget to decrease it upon deletion. That will prevent the reference count from dropping to zero and thus will successfully prevent not only the immediate object release but also any future attempt of the garbage collector to release it.

If you want to understand this in more detail, here are a few good resources:

- Garbage collection in Python: things you need to know (Artem Golubin)
- Design of CPython's Garbage Collector (Pablo Galindo Salgado)
- Python garbage collection (Digi.com)
- Garbage Collection for Python (Neil Schemenauer)
- Safely using destructors in Python (Eli Bendersky)
- Python destructor and garbage collection notes (Ferry Boender)
- Finalizer (Wikipedia)
- Object resurrection (Wikipedia)

2.2 The issues with collected and uncollectable objects

For short-running Python programs (e.g. command line utilities), it is mostly not so important if there are some memory leaks and other resource leaks. On most operating systems, resource cleanup at process termination is very thorough and resources such as open files are cleaned up properly. This should not be understood as advocating to be careless there, but the negative effect is less severe on short-running programs compared to long-running programs.

If your Python package provides modules for use by other code, you usually cannot predict whether it will be used in short-running or long-running programs. Therefore, resource usage in a Python module should be designed with the worst case assumption in mind, i.e. that it is used by an infinitely running piece of code.

The remainder of this section explains the issues with collected and uncollectable objects caused during object release and how to address them:

- Issues with *uncollectable objects*:
  - They often stay around until the Python process terminates, and thus can be considered memory leaks.
  - The garbage collector attempts to release them again and again on every collection run of its last generation, causing repeated unnecessary processing.

  In Python 3.4 or higher, the reasons for uncollectable objects have diminished very much and their presence usually indicates a bug. You should use tools that can detect uncollectable objects and then analyze each case to find out what caused the object to be uncollectable.
- Issues with *collected objects*:
  - Increased processing overhead caused by the collector runs (compared to immediate release based on reference counting).
  - Suspension of all other activity in the Python process when the garbage collector runs.
  - The amount of memory bound to these objects until the garbage collector will run for the next time. Automatic runs of the garbage collector are triggered by heuristics that are based on the number of objects and not on the amount of memory bound to these objects, so it is possible to have a small number of collectable objects with large amounts of memory allocated, that are still not triggering a garbage collector run.

  Suitable measures to address these issues with collected objects:
  - Redesign to avoid circular references.
  - Replacement of normal references with weak references to get rid of circular references. Using weak references requires to be able to handle the case where the referenced object is unexpectedly gone, which can be properly detected.
  - Manually triggering additional garbage collector runs via `gc.collect()`. There are very few cases though where this actually improves anything. One reasonable case might be to trigger a collection after application startup to release the many objects that have been used temporarily during configuration and initial startup processing.
- Issues with the `__del__()` method on objects that are involved in circular references on Python before 3.4:
  - The `__del__()` methods are not invoked, so the resource cleanup designed to be done by them does not happen.
  - In addition, the objects become uncollectable.

  Suitable measures to address these issues with the `__del__()` method:
  - The use of context managers is a good alternative to the use of the `__del__()` method, particularly on Python before 3.4.

2.3 Circular reference examples and detection

This section shows some simple examples of circular references.

Let's first look at a simple way to surface circular references. The approach is to create an object, make it unreachable, and check whether a run of the garbage collector releases an object. If the object is involved in circular references, its reference count will not drop to zero when it becomes unreachable, but the garbage collector will be able to break up the circular references and release it. If the object is not involved in circular references, it will be released when it becomes it unreachable, and the garbage collector does not have anything to do with it (even when it was tracked).

This is basically the approach Yagot uses, although in a more automated fashion.

```
$ python
>>> import gc
>>> gc.collect()    # Run full garbage collection to have a reference
0                   # No objects collected initially (in this simple case)
>>> obj = dict()
>>> len(gc.get_referrers(obj))
1                  # The dict object has one referrer (the 'obj' variable)
>>> obj['self'] = obj
>>> len(gc.get_referrers(obj))
2                  # The dict object now in addition has its 'self' item as a referrer
>>> gc.collect()
0                   # Still no new objects collected
>>> del obj         # The dict object becomes unreachable ...
>>> gc.collect()
1                   # ... and was released by the next garbage collection run
```

The interesting part happens during the `del obj` statement. The `del obj` statement removes the name `obj` from its namespace. That causes the reference count of the `dict` object to drop by one. Because of the circular reference back from its 'self' item, the reference count is still 1, so it will not be released at that point. The call to `gc.collect()` triggers a full garbage collection run on all generations, which successfully breaks up the circular reference and releases the object, as reported by its return value of 1.

Here are some examples for circular references. You can inspect them using the approach described above:

- List with a self-referencing item. This is not really useful code, but just a simple way to demonstrate a circular reference:

```
obj = list()
obj.append(obj)
```

- Class with a self-referencing attribute. Another simple example for demonstration purposes:

```
class SelfRef(object):

    def __init__(self):
        self.ref = self

obj = SelfRef()
```

- A tree node class that knows its parent and children. This is a more practical example and is very similar to what is done in xml.dom.minidom:

```
class Node(object):

    def __init__(self):
        self.parentNode = None
```

(continues on next page)

(continued from previous page)

```
        self.childNodes = []

    def appendChild(self, node):
        node.parentNode = self
        self.childNodes.append(node)

obj = Node()
obj.appendChild(Node())
```

2.4 Tools

This section lists some tools that can be used to detect memory leaks, garbage objects, and memory usage in Python.

TODO: Write section

CHAPTER 3

Yagot pytest plugin

The `yagot` pytest plugin is automatically installed along with the yagot package. It adds the following group of command line options to pytest:

```
Garbage object tracking using Yagot:

--yagot               Enables checking for collected and uncollectable objects caused␣
↪by
                      pytest test cases. Default: Env.var YAGOT (set to non-empty),␣
↪or False.

--yagot-leaks-only    Limits the checking to only uncollectable (=leak) objects.␣
↪Default:
                      Env.var YAGOT_LEAKS_ONLY (set to non-empty), or False.

--yagot-ignore-types=TYPE[,TYPE[...]]
                      Type name or module.path.class name of collected and␣
↪uncollectable
                      objects for which test cases will be ignored. Multiple comma-
↪separated
                      type names can be specified on each option, and in addition the␣
↪option
                      can be specified multiple times. The types must be specified as
                      represented by the str(type) function (for example, "int" or
                      "mymodule.MyClass"). Default: Env.var YAGOT_IGNORE_TYPES, or␣
↪empty list.
```

CHAPTER 4

API reference

This section describes the API of Yagot.

There are two main elements of the API:

- *yagot.garbage_checked()*: A decorator that checks for *uncollectable objects* and optionally for *collected objects* caused by the decorated function or method and raises AssertionError if detected.
- *yagot.GarbageTracker*: A class that checks for *uncollectable objects* and optionally for *collected objects* caused during a tracking period. This is a plumbing class the *yagot.garbage_checked()* decorator and the pytest plugin of Yagot use, and that other packages building on Yagot can also use.

4.1 yagot.garbage_checked

yagot.**garbage_checked**(*leaks_only=False*, *ignore_types=None*)

Decorator that checks for *uncollectable objects* and optionally for *collected objects* caused by the decorated function or method, and raises AssertionError if such objects are detected.

The decorated function or method needs to make sure that any objects it creates are deleted again, either implicitly (e.g. by a local variable going out of scope upon return) or explicitly. Ideally, no garbage is created that way, but whether that is actually the case is exactly what the decorator tests for. Also, it is possible that your code is clean but other modules your code uses are not clean, and that will surface this way.

Note that this decorator has arguments, so it must be specified with parenthesis, even when relying on the default argument values:

```
@yagot.garbage_checked()
test_something():
    # do some tests
```

Parameters

- **leaks_only** (*bool*) – Boolean to limit the checks to only *uncollectable objects*. By default, *collected objects* and *uncollectable objects* are checked for.

- **ignore_types** (iterable) – *None* or iterable of Python types or type names that are set as additional garbage types to ignore, in addition to `frame` and `code` that are always ignored.

  If any detected object has one of the types to be ignored, the entire set of objects caused by the decorated function or method is ignored.

  Each type can be specified as a type object or as a string with the type name as represented by the `str(type)` function (for example, "int" or "mymodule.MyClass").

  None or an empty iterable means not to ignore any types.

4.2 yagot.GarbageTracker

class yagot.**GarbageTracker**

The GarbageTracker class provides a singleton garbage tracker that can track *uncollectable objects* and optionally *collected objects* that emerged during a tracking period.

Methods

assert_message	Return a formatted multi-line string for the assertion message for the *collected objects* or *uncollectable objects* detected during the tracking period.
disable	Disable the garbage tracker.
enable	Enable the garbage tracker and control what objects it checks for.
format_obj	Return a formatted string for a single object.
get_tracker	Returns the singleton garbage tracker object.
ignore	Ignore the current tracking period for this garbage tracker, if it is enabled.
ignore_types	Set additional Python types to be ignored as *collected objects* or *uncollectable objects*.
start	Start the tracking period for this garbage tracker.
stop	Stop the tracking period for this garbage tracker.

Attributes

enabled	Boolean indicating the enablement status of the garbage tracker.
garbage	List of new *collected objects* or *uncollectable objects* that emerged during the last tracking period.
ignored	Boolean indicating whether the current tracking period should be ignored.
ignored_type_names	Return the Python type names to be ignored as *collected objects* or *uncollectable objects*.
leaks_only	Boolean indicating whether the tracker limits the checks to *uncollectable objects* (= leaks) only.

Details

static get_tracker()
Returns the singleton garbage tracker object.

The object is created when accessed through this method for the first time.

enabled
Boolean indicating the enablement status of the garbage tracker.

Type bool

ignored
Boolean indicating whether the current tracking period should be ignored.

This flag is set via *ignore()*.

Type bool

leaks_only
Boolean indicating whether the tracker limits the checks to *uncollectable objects* (= leaks) only.

This flag can be set via *enable()*.

Type bool

garbage
List of new *collected objects* or *uncollectable objects* that emerged during the last tracking period.

Type list

ignored_type_names
Return the Python type names to be ignored as *collected objects* or *uncollectable objects*.

The types `frame` and `code` that are always ignored are included in the returned list.

Returns

List of Python type names to be ignored as represented by the `str(type)` function (for example "int" or "mymodule.MyClass").

Return type list

enable(*leaks_only=False*)
Enable the garbage tracker and control what objects it checks for.

Parameters **leaks_only** (*bool*) – Boolean limiting the checks to *uncollectable objects* (=leaks) only.

disable()
Disable the garbage tracker.

ignore()
Ignore the current tracking period for this garbage tracker, if it is enabled. This causes *ignored* to be set.

ignore_types(*type_list*)
Set additional Python types to be ignored as *collected objects* or *uncollectable objects*.

The specified types are in addition to the following list of types that are aways ignored because they often appear as collectable objects when catching exceptions (e.g. when using `pytest.raises()`):

- `frame`
- `code`

If the list of collected or uncollectable objects detected during the tracking period contains an object with a type that is to be ignored, the entire tracking period is ignored.

Parameters **type_list** (iterable) – Iterable of Python types, or *None*.

Each type can be specified as a type object or as a string with the type name as represented by the `str(type)` function (for example, "int" or "mymodule.MyClass").

None or an empty iterable means not to set additional types.

start()
Start the tracking period for this garbage tracker.

Must be called before the code to be tracked is run.

stop()
Stop the tracking period for this garbage tracker.

Must be called after the code to be tracked is run.

assert_message(*location=None*, *max=10*)
Return a formatted multi-line string for the assertion message for the *collected objects* or *uncollectable objects* detected during the tracking period.

Parameters

- **location** (*string*) – Location of the function that created the objects, e.g. in the notation "module::function".
- **max** (*int*) – Maximum number of objects to be included in the returned string.

Returns Formatted multi-line string.

Return type *unicode string*

static format_obj(*obj*)
Return a formatted string for a single object.

Parameters **obj** (*object*) – The object.

Returns Formatted string for the object.

Return type *unicode string*

4.3 yagot.__version__

The version of the yagot package can be accessed by programs using the `yagot.__version__` variable:

`yagot._version.`**`__version__ = '0.5.0'`**
The full version of this package including any development levels, as a *string*.

Possible formats for this version string are:

- "M.N.P.dev1": Development level 1 of a not yet released version M.N.P
- "M.N.P": A released version M.N.P

Note: For tooling reasons, the variable is shown as `yagot._version.__version__`, but it should be used as `yagot.__version__`.

CHAPTER 5

Development

This section only needs to be read by developers of the Yagot project, including people who want to make a fix or want to test the project.

5.1 Repository

The repository for the Yagot project is on GitHub:

https://github.com/andy-maier/python-yagot

5.2 Setting up the development environment

1. If you have write access to the Git repo of this project, clone it using its SSH link, and switch to its working directory:

   ```
   $ git clone git@github.com:andy-maier/python-yagot.git
   $ cd python-yagot
   ```

   If you do not have write access, create a fork on GitHub and clone the fork in the way shown above.

2. It is recommended that you set up a virtual Python environment. Have the virtual Python environment active for all remaining steps.

3. Install the project for development. This will install Python packages into the active Python environment:

   ```
   $ make develop
   ```

4. This project uses Make to do things in the currently active Python environment. The command:

   ```
   $ make
   ```

   displays a list of valid Make targets and a short description of what each target does.

5.3 Building the documentation

The ReadTheDocs (RTD) site is used to publish the documentation for the project package at https://yagot.readthedocs.io/

This page is automatically updated whenever the Git repo for this package changes the branch from which this documentation is built.

In order to build the documentation locally from the Git work directory, execute:

```
$ make builddoc
```

The top-level document to open with a web browser will be `build_doc/html/docs/index.html`.

5.4 Testing

All of the following *make* commands run the tests in the currently active Python environment. Depending on how the *yagot* package is installed in that Python environment, either the directories in the main repository directory are used, or the installed package. The test case files and any utility functions they use are always used from the *tests* directory in the main repository directory.

The *tests* directory has the following subdirectory structure:

```
tests
 +-- plugintest          Plugin tests
 +-- unittest            Unit tests
```

The unit tests and plugin tests are run by executing:

```
$ make test
```

Test execution can be modified by a number of environment variables, as documented in the make help (execute *make help*).

To run the unit and plugin tests in all supported Python environments, the Tox tool can be used. It creates the necessary virtual Python environments and executes *make test* (i.e. the unit and function tests) in each of them.

For running Tox, it does not matter which Python environment is currently active, as long as the Python *tox* package is installed in it:

```
$ tox                               # Run tests on all supported Python versions
$ tox -e py27                       # Run tests on Python 2.7
```

5.5 Contributing

Third party contributions to this project are welcome!

In order to contribute, create a Git pull request, considering this:

- Test is required.
- Each commit should only contain one "logical" change.
- A "logical" change should be put into one commit, and not split over multiple commits.
- Large new features should be split into stages.

- The commit message should not only summarize what you have done, but explain why the change is useful.

What comprises a "logical" change is subject to sound judgement. Sometimes, it makes sense to produce a set of commits for a feature (even if not large). For example, a first commit may introduce a (presumably) compatible API change without exploitation of that feature. With only this commit applied, it should be demonstrable that everything is still working as before. The next commit may be the exploitation of the feature in other components.

For further discussion of good and bad practices regarding commits, see:

- OpenStack Git Commit Good Practice
- How to Get Your Change Into the Linux Kernel

Further rules:

- The following long-lived branches exist and should be used as targets for pull requests:
  - `master` - for next functional version
  - `stable_$MN` - for fix stream of released version M.N.
- We use topic branches for everything!
  - Based upon the intended long-lived branch, if no dependencies
  - Based upon an earlier topic branch, in case of dependencies
  - It is valid to rebase topic branches and force-push them.
- We use pull requests to review the branches.
  - Use the correct long-lived branch (e.g. `master` or `stable_0.2`) as a merge target.
  - Review happens as comments on the pull requests.
  - At least one approval is required for merging.
- GitHub meanwhile offers different ways to merge pull requests. We merge pull requests by rebasing the commit from the pull request.

5.6 Releasing a version to PyPI

This section describes how to release a version of Yagot to PyPI.

It covers all variants of versions:

- Releasing the master branch as a new (major or minor) version
- Releasing a fix stream branch of an already released version as a new fix version

The description assumes that the project repo is cloned locally. Their upstream repos are assumed to have the remote name `origin`.

1. Switch to your work directory of the project repo (this is where the `Makefile` is), and perform the following steps in that directory.
2. Set shell variables for the version and branch to be released.

   When releasing the master branch:

```
$ MNP="0.2.0"           # Full version number M.N.P of version to be released
$ MN="0.2"              # Major and minor version number M.N of version to be␣
↪released
$ BRANCH="master"       # Branch to be released
```

When releasing a fix stream branch:

```
$ MNP="0.1.1"           # Full version number M.N.P of version to be released
$ MN="0.1"              # Major and minor version number M.N of version to be␣
↪released
$ BRANCH="stable_$MN"   # Branch to be released
```

3. Check out the branch to be released, make sure it is up to date with upstream, and create a topic branch for the version to be released:

```
$ git checkout $BRANCH
$ git pull
$ git checkout -b release_$MNP
```

4. Edit the version file:

```
$ vi yagot/_version.py
```

and set the version to be released:

```
__version__ = 'M.N.P'
```

where M.N.P is the version to be released, e.g. *0.2.0*.

You can verify that this version is picked up by setup.py as follows:

```
$ ./setup.py --version
0.2.0
```

5. Edit the change log:

```
$ vi docs/changes.rst
```

To make the following changes for the version to be released:

- Finalize the version to the version to be released.
- Remove the statement that the version is in development.
- Update the statement which fixes of the previous stable version are contained in this version. If there is no fix release of the previous stable version, the line can be removed.
- Change the release date to today´s date.
- Make sure that all changes are described. This can be done by comparing the changes listed with the commit log of the master branch.
- Make sure the items in the change log are relevant for and understandable by users of the project.
- In the "Known issues" list item, remove the link to the issue tracker and add text for any known issues you want users to know about.

  Note: Just linking to the issue tracker quickly becomes incorrect for a released version and is therefore only good during development of a version. In the "Starting a new version" section, the link will be added again for the new version.

6. Perform a complete build (in your favorite Python virtual environment):

```
$ make clobber
$ make all
```

If this fails, fix and iterate over this step until it succeeds.

7. Commit the changes and push to upstream:

```
$ git status    # to double check which files have been changed
$ git commit -asm "Release $MNP"
$ git push --set-upstream origin release_$MNP
```

8. On GitHub, create a Pull Request for branch `release_$MNP`. This will trigger the CI runs in Travis and Appveyor.

   Important: When creating Pull Requests, GitHub by default targets the `master` branch. If you are releasing a fix version, you need to change the target branch of the Pull Request to `stable_$MN`.

9. Perform a complete test using Tox:

```
$ tox
```

   This will create virtual Python environments for all supported Python versions that are installed on your system and will invoke `make test` in each of them.

10. If any of the tests mentioned above fails, fix the problem and iterate back to step 6. until they all succeed.

11. On GitHub, once the CI runs for the Pull Request succeed:

    - Merge the Pull Request (no review is needed)
    - Delete the branch of the Pull Request (`release_$MNP`)

12. Checkout the branch you are releasing, update it from upstream, and delete the local topic branch you created:

```
$ git checkout $BRANCH
$ git pull
$ git branch -d release_$MNP
```

13. Tag the version:

    This step tags the local repo and pushes it upstream:

```
$ git status    # double check that the branch to be released (`$BRANCH`) is␣
↪checked out
$ git tag $MNP
$ git push --tags
```

14. If you released the master branch it will be fixed separately, so it needs a new fix stream.

    - Create a branch for its fix stream and push it upstream:

```
$ git status    # double check that the branch to be released (`$BRANCH`) is␣
↪checked out
$ git checkout -b stable_$MN
$ git push --set-upstream origin stable_$MN
```

    - Log on to RTD, go to the project, and activate the new branch `stable_$MN` as a version to be built.

15. On GitHub, edit the new tag, and create a release description on it. This will cause it to appear in the Release tab.

16. On GitHub, close milestone M.N.P.

    Note: Issues with that milestone will be moved forward in the section "Starting a new version".

17. Upload the package to PyPI:

```
$ make upload
```

Attention!! This only works once. You cannot re-release the same version to PyPI.

Verify that it arrived on PyPI: https://pypi.python.org/pypi/yagot/

5.7 Starting a new version

This section shows the steps for starting development of a new version of the Yagot project in its Git repo.

It covers all variants of new versions:

- A new (major or minor) version for new development based upon the master branch.
- A new fix version based on a `stable_$MN` fix stream branch.

1. Switch to the work directory of your repo clone and perform the following steps in that directory.
2. Set shell variables for the version to be started and for the branch it is based upon.

   When starting a new major or minor version based on the master branch:

   ```
   $ MNP="0.2.0"           # Full version number M.N.P of version to be started
   $ MN="0.2"              # Major and minor version number M.N of version to be␣
   ↪started
   $ BRANCH="master"       # Branch the new version is based on
   ```

   When releasing a fix version based on a fix stream branch:

   ```
   $ MNP="0.1.1"           # Full version number M.N.P of version to be started
   $ MN="0.1"              # Major and minor version number M.N of version to be␣
   ↪started
   $ BRANCH="stable_$MN"   # Branch the new version is based on
   ```

3. Check out the branch the new version is based upon, make sure it is up to date with upstream, and create a topic branch for the new version:

   ```
   $ git checkout $BRANCH
   $ git pull
   $ git checkout -b start_$MNP
   ```

4. Edit the version file:

   ```
   $ vi yagot/_version.py
   ```

   and set the version to the new development version:

   ```
   __version__ = 'M.N.P.dev1'
   ```

   where M.N.P is the new version to be started, e.g. *0.2.0*.

5. Edit the change log:

   ```
   $ vi docs/changes.rst
   ```

   To insert the following section before the top-most section:

```
yagot 0.2.0.dev1
----------------------------------------

This version contains all fixes up to yagot 0.1.x.

Released: not yet

**Incompatible changes:**

**Deprecations:**

**Bug fixes:**

**Enhancements:**

**Cleanup:**

**Known issues:**

* See `list of open issues`_.

.. _`list of open issues`: https://github.com/andy-maier/python-yagot/issues
```

6. Commit the changes and push to upstream:

```
$ git status    # to double check which files have been changed
$ git commit -asm "Start $MNP"
$ git push --set-upstream origin start_$MNP
```

7. On Github, create a Pull Request for branch `start_$MNP`.

   Important: When creating Pull Requests, GitHub by default targets the master branch. If you are starting a fix version, you need to change the target branch of the Pull Request to `stable_$MN`.

8. On GitHub, once all of these tests succeed:

   - Merge the Pull Request (no review is needed)
   - Delete the branch of the Pull Request (`release_$MNP`)

9. Checkout the branch the new version is based upon, update it from upstream, and delete the local topic branch you created:

```
$ git checkout $BRANCH
$ git pull
$ git branch -d start_$MNP
```

10. On GitHub, create a new milestone M.N.P for the version that is started.

11. On GitHub, list all open issues that still have a milestone of less than M.N.P set, and update them as needed to target milestone M.N.P.

CHAPTER 6

Appendix

This section contains information that is referenced from other sections, and that does not really need to be read in sequence.

6.1 Compatibility and deprecation policy

The Yagot project uses the rules of Semantic Versioning 2.0.0 for compatibility between versions, and for deprecations. The public interface that is subject to the semantic versioning rules and specificically to its compatibility rules are the APIs and commands described in this documentation.

The semantic versioning rules require backwards compatibility for new minor versions (the 'N' in version 'M.N.P') and for new patch versions (the 'P' in version 'M.N.P').

Thus, a user of an API or command of the Yagot project can safely upgrade to a new minor or patch version of the yagot package without encountering compatibility issues for their code using the APIs or for their scripts using the commands.

In the rare case that exceptions from this rule are needed, they will be documented in the *Change log*.

Occasionally functionality needs to be retired, because it is flawed and a better but incompatible replacement has emerged. In the Yagot project, such changes are done by deprecating existing functionality, without removing it immediately.

The deprecated functionality is still supported at least throughout new minor or patch releases within the same major release. Eventually, a new major release may break compatibility by removing deprecated functionality.

Any changes at the APIs or commands that do introduce incompatibilities as defined above, are described in the *Change log*.

Deprecation of functionality at the APIs or commands is communicated to the users in multiple ways:

- It is described in the documentation of the API or command
- It is mentioned in the change log.

- It is raised at runtime by issuing Python warnings of type `DeprecationWarning` (see the Python `warnings` module).

Since Python 2.7, `DeprecationWarning` messages are suppressed by default. They can be shown for example in any of these ways:

- By specifying the Python command line option: `-W default`
- By invoking Python with the environment variable: `PYTHONWARNINGS=default`

It is recommended that users of the Yagot project run their test code with `DeprecationWarning` messages being shown, so they become aware of any use of deprecated functionality.

Here is a summary of the deprecation and compatibility policy used by the Yagot project, by version type:

- New patch version (M.N.P -> M.N.P+1): No new deprecations; no new functionality; backwards compatible.
- New minor release (M.N.P -> M.N+1.0): New deprecations may be added; functionality may be extended; backwards compatible.
- New major release (M.N.P -> M+1.0.0): Deprecated functionality may get removed; functionality may be extended or changed; backwards compatibility may be broken.

6.2 Troubleshooting

Here are some trouble shooting hints for ...

6.3 Glossary

collectable objects objects Python cannot immediately release when they become unreachable (e.g. when their variable goes out of scope) and that are therefore supposed to be released by the Python garbage collector. Most of the time, this is caused by the presence of circular references into which the object to be released is involved. The Python garbage collector is designed to handle circular references.

collected objects *collectable objects* that have successfully been released by the Python garbage collector.

uncollectable objects *collectable objects* that could not be released by the Python garbage collector, even when running a full collection. Uncollectable objects remain allocated in the last generation of the garbage collector and their memory remains allocated until the Python process terminates. They can be considered memory leaks.

string a *unicode string* or a *byte string*

unicode string a Unicode string type (`unicode` in Python 2, and `str` in Python 3)

byte string a byte string type (`str` in Python 2, and `bytes` in Python 3). Unless otherwise indicated, byte strings in this project are always UTF-8 encoded.

number one of the number types `int`, `long` (Python 2 only), or `float`.

integer one of the integer types `int` or `long` (Python 2 only).

callable a callable object; for example a function, a class (calling it returns a new object of the class), or an object with a `__call__()` method.

hashable a hashable object. Hashability requires an object not only to be able to produce a hash value with the `hash()` function, but in addition that objects that are equal (as per the == operator) produce equal hash values, and that the produced hash value remains unchanged across the lifetime of the object. See term "hashable" in the Python glossary, although the definition there is not very crisp. A more exhaustive discussion of these requirements is in "What happens when you mess with hashing in Python" by Aaron Meurer.

6.4 References

Python Glossary

- Python 2.7 Glossary
- Python 3.4 Glossary

CHAPTER 7

Change log

7.1 yagot 0.5.0

Released: 2020-03-07

Initial version releaed to Pypi.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

N

P

S

U