Aglyph Documentation

Release 3.0.0.post1

Matthew Zipay

Aug 29, 2018

Contents

Release 3.0.0.post1

Aglyph is a Dependency Injection framework for Python, supporting type 2 (setter) and type 3 (constructor) injection.

Aglyph runs on CPython 2.7 and 3.4+, and on recent versions of the PyPy, Jython, IronPython, and Stackless Python variants. See *Aglyph 3.0.0.post1 testing summary* for a complete list of the Python versions and variants on which Aglyph has been tested.

Aglyph can assemble *prototype* components (a new instance is created every time), *singleton* components (the same instance is returned every time), *borg* components (a new instance is created every time, but all instances of the same class share the same internal state), and *weakref* components (the same instance is returned as long as there is at least one "live" reference to that instance in the application).

Aglyph can be configured using a declarative XML syntax, or programmatically in pure Python.

CHAPTER 1

Table of Contents

1.1 What's new in release 3.0.0.post1?

This is just a small "housekeeping" (post-)release.

Note: There is no immediate reason to upgrade from **3.0.0**, as no core functionality has changed.

The updates to *Aglyph 3.0.0.post1 testing summary* are valid for release **3.0.0** as well as this post-release.

- Python 3.7 is now officially supported. (Cumulatively, Python 2.7, 3.4, 3.5, 3.6 and 3.7 are officially supported.)
- *Aglyph 3.0.0.post1 testing summary* has been updated with recent Python versions, variants and platforms to match the officially supported versions.
- A small change in the `aglyph._compat` module related to IronPython detection (the purpose of which is to keep detection logic the same between Aglyph and Autologging).
- Some documentation-related updates:
  - upgraded the Sphinx version (and therefore the documentation *Makefile*) used to generate the HTML docs
  - switch to using the (default) Alabaster Sphinx theme
  - added a warning in *The Aglyph Context fluent API* regarding the absence of `__qualname__` in Python versions < 3.3
- PyPI/Setuptools changes:
  - use Markdown Descriptions on PyPI
  - locked the Autologging version to `1.2.0` (see *requirements.txt*)
- One deprecation: the `aglyph.version_info` module attribute (which should not have been public in the first place)

1.1.1 Previous releases of Aglyph

What's new in release 3.0.0?

- The Python 3.2 lifespan and Python 3.3 lifespan have ended; those versions are no longer actively supported in Aglyph.
- Python 3.6 is now supported. (Cumulatively, Python 2.7, 3.4, 3.5, and 3.6 are currently supported.)
- All deprecated functions, classes, and behaviors have been replaced or removed, most notably:
  - The `aglyph.binder` module has been removed; programmatic configuration is now handled by *The Aglyph Context fluent API*.
  - The `aglyph.cache` module has been removed (it is an internal implementation detail and should not be part of the public API).
  - The `aglyph.compat` package has become the *non-public* `aglyph._compat` module (it is an internal implementation detail and should not be part of the public API).
  - The built-in `eval()` function is no longer supported (see Eval really is dangerous) in favor of the safer `ast.literal_eval()`.
- Aglyph is now fully logged and traced via Autologging. Tracing is **disabled** by default and can enabled by setting the *"AGLYPH_TRACED"* environment variable.

What's new in release 2.1.1?

- The *Python 3.2 lifespan* has ended; Aglyph is no longer actively tested against Python 3.2.
- Fix for issues/2 allowing builtin immutables to be used as components.
- Documentation updates to fix various hyperlinks.

What's new in release 2.1.0?

- The Python 2.6 lifespan has ended and it is no longer actively supported in Aglyph. *(Aglyph 2.1.0 will not run on Python 2.6 without patching)*
- Aglyph now supports *lifecycle methods*, which may be declared at the context, template, and/or component level for the "after injection" and "before clear" lifecycle states.
- Aglyph now supports a form of component "inheritance" through *`aglyph.component.Template`* (XML `<template>`). Refer to *Component inheritance using templates* for examples.
- The `Aglyph context DTD` has been updated to support both lifecycle methods and templates.
- The `aglpyh.integration` package has been added to support integrating Aglyph with other projects. Aglyph 2.1.0 introduces CherryPy integration using the classes defined in `aglyph.integration.cherrypy`. Refer to *Integrating Aglyph* for examples.
- The caches defined in `aglyph.cache` are now implemented as Context Manager Types, and the public `lock` members have been deprecated.
- A "safe" representation is now used to log assembled objects, ensuring that possibly sensitive data is not logged.
- Deprecated classes and functions now issue `aglyph.AglyphDeprecationWarning`.

What's new in release 2.0.0?

- The Python 2.5 lifespan has ended and it is no longer actively supported in Aglyph. *(Aglyph 2.0.0 will not run on Python 2.5 without patching)*
- Creating components using `staticmethod`, `classmethod`, and nested classes (any level) is now supported via *`aglyph.component.Component.factory_name`*.
- Referencing class objects, nested class objects (any level), functions, or attributes as components is now supported via *`aglyph.component.Component.member_name`*.
- The *Aglyph cookbook* has been expanded to include many new recipes, including examples of the aforementioned *`aglyph.component.Component.factory_name`* and *`aglyph.component.Component.member_name`* configuration options.
- The `<eval>` element in declarative XML configuration is **deprecated**. Use a `<component>` and a `<reference>` (or `@reference` attribute) to configure anything that was previously declared as an `<eval>`.
- The `aglyph.has_importable_dotted_name()` function is **deprecated**. The *`aglyph.format_dotted_name()`* function now verifies that the dotted name is actually importable.
- The `aglyph.identify_by_spec()` function is **deprecated**. This really belonged in `aglyph.binder.Binder` to begin with, which is where it now resides as a non-public method.
- Multiple calls to `aglyph.binder._Binding.init()` and `aglyph.binder._Binding.attributes()` now have a cumulative effect, rather than replacing any previously-specified arguments or attributes, respectively.
- `aglyph.compat` and *`aglyph.context.XMLContext`* have been updated to avoid deprecated `xml.etree.ElementTree` methods.
- Python implementation detection in `aglyph.compat` has been improved.
- The `aglyph.compat.ipyetree.XmlReaderTreeBuilder` class is **deprecated**. IronPython applications no longer need to explicitly pass a parser to *`aglyph.context.XMLContext`* (Aglyph now uses a sensible default). *(note: with this change, the Aglyph API is now 100% cross-compatible with all tested Python versions and variants)*
- The *Getting started with Aglyph* tutorial and accompanying sample code have been revamped to better demonstrate the various Aglyph configuration approaches, as well as to provide more substantive component examples.
- Aglyph documentation now uses the Read the Docs Sphinx Theme.

What's new in release 1.1.1?

- general compatibility updates for Python 3.3
- a revamped logging approach
- improved testing (more versions/variants, now works with `python setup.py test`)
- updated to use Sphinx and a new theme for documentation
- added example applications to demonstrate Aglyph usage

What's new in release 1.1.0?

- The `aglyph.binder` package provides a more concise approach to programmatic configuration of Aglyph
- All documentation and sample code is updated to include details of using `aglyph.binder` programmatic configuration

- Comment reformatting in accordance with **PEP 8**

1.2 Getting started with Aglyph

Release 3.0.0.post1

During this brief tutorial, you will download and install Aglyph, build a simple Python application based on the *MovieLister* component discussed in Inversion of Control Containers and the Dependency Injection pattern, then modify the application to take advantage of Aglyph dependency injection. This process will allow you understand the Dependency Injection pattern in general, and the Aglyph approach to Dependency Injection in particular.

This tutorial is a "whirlwind tour" of Aglyph that covers only the basics. Once you have completed the steps, read the *Aglyph cookbook* for additional guidelines and examples. Also review the *Aglyph API reference*, *The Aglyph Context fluent API* and the `Aglyph context DTD` to understand the details.

The tutorial assumes that you are familiar with Python development in general, and that Python 2.7 or 3.4+ is already installed on your system.

- Download Python
- Browse Dive Into Python 2 and/or The Python 2 Tutorial
- Browse Dive Into Python 3 and/or The Python 3 Tutorial

Note: It is recommended, but not required, that you read the Inversion of Control Containers and the Dependency Injection pattern and Python Dependency Injection [PDF] articles before beginning this tutorial.

1.2.1 1. Download and install Aglyph

There are several options for downloading and installing Aglyph. Choose the method that best suits your needs or preferences.

Download and install a source or built distribution from SourceForge

If you use Windows, a source ZIP distribution and EXE and MSI installers are available from the Aglyph SourceForge project.

Run the EXE or MSI installer after downloading, or unpack the ZIP distribution and run the following command from within the distribution directory:

```
python setup.py install
```

Download and install a source distribution from the Python Package Index (PyPI)

The Aglyph source distribution can be downloaded from the Aglyph PyPI page.

Unpack the archive and run the following command from with the distribution directory:

```
python setup.py install
```

Clone the Aglyph repository from GitHub

To install the latest release from a clone of the Aglyph GitHub repository, execute the following commands from a shell:

```
git clone https://github.com/mzipay/Aglyph.git
cd Aglyph
python setup.py install
```

Install into a virtual environment

You can also create a Virtualenv (details not covered here) and install Aglyph into it by running the following commands from a shell (assumes the virtual environment is active):

```
pip install Aglyph
```

Regardless of installation method, verify that the installation was successful by importing the `aglyph` module from a Python interpreter. For example:

```
$ python
Python 3.5.4 (default, Oct  9 2017, 12:07:29)
[GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import aglyph
>>> aglyph.__version__
'3.0.0.post1'
```

1.2.2 2. Download, extract, and run the *movielisterapp* application

The sample code for this tutorial can be downloaded `here (movielisterapp-basic.zip)`. If you don't feel like typing everything out by hand and would prefer to just "follow along," you can also download `movielisterapp-aglyph.zip`, which contains the completed tutorial source code (including the already-populated SQLite database).

Note: Both ZIP files are also available under the *examples/* directory if you cloned the Aglyph GitHub repository.

Warning: Jython users will not be able to run the tutorial code because the standard Python `sqlite3` module (which Jython does not support) is used by the example code.

To begin the tutorial, extract the ZIP archive to a temporary location and navigate into the application directory:

```
$ unzip movielisterapp-basic.zip
...
$ cd movielisterapp-basic
```

The *movies.txt* file is a simple colon-delimited text file that contains a number of *title:director* records, one per line:

```
The Colossus of Rhodes:Sergio Leone
Once Upon a Time in the West:Sergio Leone
THX 1138:George Lucas
```

(continues on next page)

(continued from previous page)

```
American Graffiti:George Lucas
Once Upon a Time in America:Sergio Leone
Sixteen Candles:John Hughes
The Breakfast Club:John Hughes
Weird Science:John Hughes
Ferris Bueller's Day Off:John Hughes
```

This data file is read by a particular implementation of the `MovieFinder` class (`ColonDelimitedMovieFinder`), both of which can be found in the *movies/finder.py* module:

```python
from movies.movie import Movie


class MovieFinder:

    def find_all(self):
        raise NotImplementedError()


class ColonDelimitedMovieFinder(MovieFinder):

    def __init__(self, filename):
        movies = []
        f = open(filename)
        for line in f:
            (title, director) = line.strip().split(':')
            movies.append(Movie(title, director))
        f.close()
        self._movies = movies

    def find_all(self):
        return self._movies
```

As you can see, each record is processed as a simple `Movie` data holder object. The *movies/movie.py* module holds the `Movie` class definition:

```python
class Movie:

    def __init__(self, title, director):
        self.title = title
        self.director = director
```

Finally, we have a `MovieLister` class (defined in the *movies/lister.py* module), which uses a `ColonDelimitedMovieFinder` to find the movies directed by a particular director:

```python
from movies.finder import ColonDelimitedMovieFinder


class MovieLister:

    def __init__(self):
        self._finder = ColonDelimitedMovieFinder("movies.txt")

    def movies_directed_by(self, director):
        for movie in self._finder.find_all():
            if (movie.director == director):
                yield movie
```

The application can be executed using the *app.py* script, which simply asks a `MovieLister` for all movies directed by Sergio Leone:

```
$ python app.py
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

1.2.3 3. A *(very)* brief introduction to Dependency Injection

Examine the `MovieLister` class (in the *movies/lister.py* module) again. There are three things to note:

1. The `MovieLister` class depends on a concrete implementation of `MovieFinder`.
2. The `ColonDelimitedMovieFinder` class depends on a filename.
3. The `MovieLister` is responsible for resolving *both* dependencies.

As a consequence of (3), neither the concrete `MovieFinder` implementation nor the name/location of the data file can be changed without modifying `MovieLister`.

In other words, it is `MovieLister` that controls dependency resolution. It is this aspect of control that is being inverted ("Inversion of Control") when we talk about **Dependency Injection**. Rather than having `MovieLister` be responsible for *resolving* its dependencies, we instead give control to some other object (an "assembler"), which has the responsibility of *injecting* dependencies into `MovieLister`.

The dependency injection approach provides several benefits:

- easier testing ("mock" or "stub" objects for testing are easier to manage)
- lower general maintenance cost (e.g. the manner in which application/domain objects get initialized and connected to one another is "homogenized" in the assembler's configuration, which makes application-wide changes easier to apply and test)
- the separation of object *configuration* from object *use* means generally smaller and simpler application code that is focused on object behavior

Aglyph can inject dependencies using initializers – `__init__` methods – or "factory" functions (type 2 "constructor" injection); or member variables, setter methods, and properties (type 3 "setter" injection).

In order to take advantage of type 2 "constructor" injection, the `__init__` method or "factory" function must *accept* dependencies, which means we need to make some simple changes to *movielisterapp*. . .

1.2.4 4. Make some general improvements to the *movielisterapp* application

As written, the basic application is somewhat change-resistant. For example, if we wish to support another implementation of `MovieFinder` (e.g. one that connects to a database to retrieve movie information), then we would also need to change the `MovieLister` implementation.

A simple solution to this problem is to change `MovieLister` so that it can *accept* a `MovieFinder` at initialization time:

```python
class MovieLister:

    def __init__(self, finder):
        self._finder = finder

    def movies_directed_by(self, director):
```

(continues on next page)

(continued from previous page)

```
        for movie in self._finder.find_all():
            if (movie.director == director):
                yield movie
```

Next, we'll add a `SQLMovieFinder` class definition to the *movies/finder.py* module. This new implementation will use the standard Python `sqlite3` module to connect to a SQLite database which stores the movies information:

```
import sqlite3
from movies.movie import Movie


class MovieFinder:

    def find_all(self):
        raise NotImplementedError()


class ColonDelimitedMovieFinder(MovieFinder):

    def __init__(self, filename):
        movies = []
        f = open(filename)
        for line in f:
            (title, director) = line.strip().split(':')
            movies.append(Movie(title, director))
        f.close()
        self._movies = movies

    def find_all(self):
        return self._movies


class SQLMovieFinder(MovieFinder):

    def __init__(self, dbname):
        self._db = sqlite3.connect(dbname)

    def find_all(self):
        cursor = self._db.cursor()
        movies = []
        try:
            for row in cursor.execute("select title, director from Movies"):
                (title, director) = row
                movies.append(Movie(title, director))
        finally:
            cursor.close()
        return movies

    def __del__(self):
        try:
            self._db.close()
        except:
            pass
```

The `SQLMovieFinder` expects a database name (a filename, or *":memory:"* for an in-memory database). We'll create a *movies.db* file so that it contains the same records as the original *movies.txt* file:

```
>>> import sqlite3
>>> conn = sqlite3.connect("movies.db")
>>> c = conn.cursor()
>>> c.execute("create table Movies (title text, director text)")
>>> for movie_fields in [("The Colossus of Rhodes", "Sergio Leone"),
...                      ("Once Upon a Time in the West", "Sergio Leone"),
...                      ("THX 1138", "George Lucas"),
...                      ("American Graffiti", "George Lucas"),
...                      ("Once Upon a Time in America", "Sergio Leone"),
...                      ("Sixteen Candles", "John Hughes"),
...                      ("The Breakfast Club", "John Hughes"),
...                      ("Weird Science", "John Hughes"),
...                      ("Ferris Bueller's Day Off", "John Hughes")]:
>>>     c.execute("insert into Movies values (?, ?)", movie_fields)
...
>>> c.close()
>>> conn.commit()
>>> conn.close()
```

Finally, we'll change *app.py* so that the new `SQLMovieFinder` is used to initialize a `MovieLister`:

```
import sys

from movies.finder import SQLMovieFinder
from movies.lister import MovieLister

lister = MovieLister(SQLMovieFinder("movies.db"))
for movie in lister.movies_directed_by("Sergio Leone"):
    sys.stdout.write("%s\n" % movie.title)
```

Running the application again should give us the same results:

```
$ python app.py
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

The basic application is now more flexible: we can change the `MovieFinder` implementation without having to modify the `MovieLister` class definition. However, we are still required to modify *app.py* if we decide to change the `MovieFinder` implementation!

Note: An important aspect of Aglyph is that it is **non-intrusive**, meaning that it requires only minimal changes to your existing application code in order to provide dependency injection capabilities.

Notice that the changes made in this section, while adding flexibility to the application, did not require the use of Aglyph. In fact, as we add Aglyph dependency injection support in the next two sections, **no further changes** to the *movies/lister.py*, *movies/finder.py*, and *movies/movie.py* modules need to be made.

1.2.5 5. Add Dependency Injection support to the *movielisterapp* application

Recall that Dependency Injection gives reponsibility for injecting dependencies to an an external object (called an "assembler"). In Aglyph, this "assembler" is an instance of the `aglyph.assembler.Assembler` class.

An `aglyph.assembler.Assembler` requires a "context," which is a collection of component definitions. A *component* (`aglyph.component.Component`) is simply a description of some object, including how it is created/acquired and its dependencies. Any component can itself be a dependency of any other component(s).

In Aglyph, a context is defined by the `aglyph.context.Context` class. Objects of this class can be created and populated either directly or by using *The Aglyph Context fluent API*. A specialized subclass, `aglyph.context.XMLContext`, is also provided to allow a context to be defined declaratively in an XML document. Such XML documents should conform to the `Aglyph context DTD`.

In this section, we will create a declarative XML context **and** use *The Aglyph Context fluent API* for *movielisterapp* in order to demonstrate each approach.

Warning: In practice, you should choose **either** `aglyph.context.XMLContext` or `aglyph.context.Context` (*The Aglyph Context fluent API*) when configuring Aglyph for your application.

First, we'll create the XML context document as *movies-context.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<context id="movies-context">
    <component id="delim-finder"
               dotted-name="movies.finder.ColonDelimitedMovieFinder">
        <init>
            <arg><str>movies.txt</str></arg>
        </init>
    </component>
    <component id="movies.finder.MovieFinder"
               dotted-name="movies.finder.SQLMovieFinder">
        <init>
            <arg><str>movies.db</str></arg>
        </init>
    </component>
    <component id="movies.lister.MovieLister">
        <init>
            <arg reference="movies.finder.MovieFinder" />
        </init>
    </component>
</context>
```

Some interesting things to note here:

- A `<context>` requires an `id` attribute, which should uniquely identify the context.
- A `<component>` requires an `id` attribute, and has an optional `dotted-name` attribute. If `dotted-name` is not provided, then the `id` attribute is assumed to be a dotted name; otherwise, the `id` can be a user-defined identifier and the `dotted-name` **must** be provided (this is useful when describing multiple components of the same class, for example). A dotted name is a string that represents an **importable** module, class, or function.
- Initialization arguments are provided as `<arg>` child elements of a parent `<init>` element. An `<arg>` is a postional argument, while an `<arg keyword="...">` is a keyword argument. (As in Python, the order in which positional arguments are declared is significant, while the order of keyword arguments is not.)

Note: A dotted name is a *"dotted_name.NAME"* or *"dotted_name"* string that represents a valid absolute import statement according to the following productions:

```
absolute_import_stmt  ::=   "from" dotted_name "import" NAME
                            | "import" dotted_name
```

```
dotted_name          ::=  NAME ('.' NAME)*
```

See also:

Full Grammar specification

Notice that the *movies.lister.MovieLister* component is being injected with a reference to the *movies.finder.MovieFinder* component, which describes an instance of `movies.finder.SQLMovieFinder`. We could easily change back to using `movies.finder.ColonDelimitedMovieFinder` by changing the reference.

Next, we'll create an equivalent context, but this time using *The Aglyph Context fluent API*. In *movies/__init__.py*:

```python
from movies.finder import MovieFinder, SQLMovieFinder
from movies.lister import MovieLister

from aglyph.component import Reference as ref
from aglyph.context import Context


context = Context("movies-context")

(context.component("delim-finder").
    create("movies.finder.ColonDelimitedMovieFinder").
    init("movies.txt").
    register())

# makes SQLMovieFinder the default impl bound to "movies.finder.MovieFinder"
(context.component(MovieFinder).
    create(SQLMovieFinder).
    init("movies.db").
    register())

# will initialize MovieLister with an object of SQLMovieFinder
context.component(MovieLister).init(ref(MovieFinder)).register()
```

Compare this context carefully with the XML declarative context above; they are identical. However, there are several interesting things to note about initializing the context using the fluent API:

- Here we simply use the `component(...)` method, which results in all components being of the default type (prototype). Defining components of different types (i.e. prototype, singleton, borg, weakref) is simply a matter of using the corresponding method name. We'll use some of these in the next part of the tutorial. These methods are the "entry points" into the fluent configuration API.
- Each component definition is terminated by a call to the `register()` method. This method **must** be the final call, as it (a) terminates the chained-call sequence and, more importantly, (b) finalizes the compoonent definition in the context. (If you get "component not found" errors when using the fluent API, the first thing to check is that you remembered to call `register()`!)
- The component methods (`prototype(...)` / `singleton(...)` / `borg(...)` / `weakref(...)`) and the `create(...)` method can accept dotted-name strings *as well as* objects. If the argument is **not** a string, Aglyph determines its dotted-name and uses that value. So in the above context, for example, `create(SQLMovieFinder)` is actually equivalent to `create("movies.finder.SQLMovieFinder")`.
- Unlike the component and create methods, the `init(...)` and `set(...)` (not shown here) methods do **not** automatically convert non-string arguments to dotted names. This is so that classes and other callables may be used directly as arguments. This is why we must use `init(ref(MovieFinder))` (note the use of `ref(...)`) when defining the MovieLister component.

Now that we have created Aglyph configurations for *movielisterapp*, it's time to modify the *app.py* script to use dependency injection. To demonstrate the use of both types of configution, we'll create two different versions of the application script.

Note: As noted earlier, in practice you would choose **one** of the configuration options and set up your application entry point appropriately.

The *app_xml.py* script will use the declarative XML context:

```
import sys
from aglyph.assembler import Assembler
from aglyph.context import XMLContext

context = XMLContext("movies-context.xml")
assembler = Assembler(context)

lister = assembler.assemble("movies.lister.MovieLister")
for movie in lister.movies_directed_by("Sergio Leone"):
    sys.stdout.write("%s\n" % movie.title)
```

This script creates an assembler with a context that is read from the *movies-context.xml* XML document. Notice that we no longer need to create the `SQLMovieFinder` class directly; we have effectively separated the configuration of `MovieLister` from its use in the application.

Running the application produces the same results as usual:

```
$ python app_xml.py
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

The *app_fluent.py* script will use the context that was created in *movies/__init__.py*:

```
import sys

from aglyph.assembler import Assembler
from movies import context

assembler = Assembler(context)

lister = assembler.assemble("movies.lister.MovieLister")
for movie in lister.movies_directed_by("Sergio Leone"):
    sys.stdout.write("%s\n" % movie.title)
```

Again, running the application produces the expected results:

```
$ python app_fluent.py
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

1.2.6 6. Make changes to the *movielisterapp* application

Now that the application is configured to use Aglyph for dependency injection, let's make some changes to demonstrate application maintenance under Aglyph.

Note: The key point of this final exercise is that we will be able to make "significant" changes to the application without having to modify any of the application source code. This is possible because we have *separated the configuration of objects from their use*; this is the goal of Depdendency Injection.

Introducing assembly strategies

In our existing configurations, all components are using Aglyph's default assembly strategy, **prototype**, which means that each time a component is assembled, a new object is created, initialized, wired, and returned.

This is not always desired (or appropriate), so Aglyph also supports **singleton**, **borg**, and **weakref** assembly strategies.

For details of what each assembly strategy implies, please refer to *`aglyph.component.Strategy`*.

See also:

The Borg design pattern Alex Martelli's original Borg recipe (from ActiveState Python Recipes)

Module `weakref` Documentation of the `weakref` standard module.

Modify *movielisterapp* to use a singleton `ColonDelimitedMovieFinder`

We note that `ColonDelimitedMovieFinder` class parses its data file on every initialization. We don't expect the data file to change very often, at least not during application runtime, so we'd prefer to only create an instance of `ColonDelimitedMovieFinder` *once*, regardless of how many times during the application runtime that it is requested (i.e. assembled). For the sake of demonstration, preted for a moment that *movielisterapp* is a useful application in which `MovieFinder` objects are used by more than just a `MovieLister` ;)

To accomplish this goal, we'll modify our configurations so that the *"delim-finder"* component uses the **singleton** assembly strategy.

Also, we'll change the *movies.lister.MovieLister* component so that it uses the original `ColonDelimitedMovieFinder` instead of `SQLMovieFinder`.

The modified XML context looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<context id="movies-context">
    <component id="delim-finder"
               dotted-name="movies.finder.ColonDelimitedMovieFinder"
               strategy="singleton">
        <init>
            <arg><str>movies.txt</str></arg>
        </init>
    </component>
    <component id="movies.finder.MovieFinder"
               dotted-name="movies.finder.SQLMovieFinder">
        <init>
            <arg><str>movies.db</str></arg>
        </init>
    </component>
    <component id="movies.lister.MovieLister">
        <init>
            <arg reference="delim-finder" />
        </init>
    </component>
</context>
```

We added `strategy="singleton"` to the *"delim-finder"* component, and changed the `MovieLister` argument to specify `reference="delim-finder"`.

The modifed *movies/__init__.py* module looks like this:

```python
from movies.finder import MovieFinder, SQLMovieFinder
from movies.lister import MovieLister

from aglyph.component import Reference as ref
from aglyph.context import Context


context = Context("movies-context")

(context.singleton("delim-finder").
    create("movies.finder.ColonDelimitedMovieFinder").
    init("movies.txt").
    register())

# makes SQLMovieFinder the default impl bound to "movies.finder.MovieFinder"
(context.borg(MovieFinder).
    create(SQLMovieFinder).
    init("movies.db").
    register())

# will initialize MovieLister with an object of ColonDelimitedMovieFinder
context.component(MovieLister).init(ref("delim-finder")).register()
```

We used the `singleton(...)` method to define the *"delim-finder"* component. Also, because the component ID *"delim-finder"* is not a dotted name, we need to manually specify that the `MovieLister` argument is an *`aglyph.component.Reference`* to *"delim-finder"*.

Running either version of the application still produces the expected results:

```
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

Modify *movielisterapp* again to use a borg `SQLMovieFinder`

We also note that `SQLMovieFinder` doesn't really need to create a new database connection every time it is assembled. We *could* use the singleton assembly strategy, but instead we'll use a similar pattern called **borg**. Of course, we'll also change the application to again use the `SQLMovieFinder`.

The final modified XML context looks like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<context id="movies-context">
    <component id="delim-finder"
               dotted-name="movies.finder.ColonDelimitedMovieFinder"
               strategy="singleton">
        <init>
            <arg><str>movies.txt</str></arg>
        </init>
    </component>
    <component id="movies.finder.MovieFinder"
               dotted-name="movies.finder.SQLMovieFinder"
```

(continues on next page)

(continued from previous page)

```
                strategy="borg">
        <init>
            <arg><str>movies.db</str></arg>
        </init>
    </component>
    <component id="movies.lister.MovieLister">
        <init>
            <arg reference="movies.finder.MovieFinder" />
        </init>
    </component>
</context>
```

The final modifed *movies/__init__.py* looks like this:

```
from movies.finder import MovieFinder, SQLMovieFinder
from movies.lister import MovieLister

from aglyph.component import Reference as ref
from aglyph.context import Context


context = Context("movies-context")

(context.singleton("delim-finder").
    create("movies.finder.ColonDelimitedMovieFinder").
    init("movies.txt").
    register())

# makes SQLMovieFinder the default impl bound to "movies.finder.MovieFinder"
(context.borg(MovieFinder).
    create(SQLMovieFinder).
    init("movies.db").
    register())

# will initialize MovieLister with an object of SQLMovieFinder
context.prototype(MovieLister).init(ref(MovieFinder)).register()
```

Running either the *app_xml.py* or *app_fluent.py* version of the application with the final configuration changes still produces the expected results:

```
The Colossus of Rhodes
Once Upon a Time in the West
Once Upon a Time in America
```

1.2.7 Suggested next steps

There are many more context/configuration options available in Aglyph beyond those that have been presented in this tutorial, including support for type 2 “setter” injection using member variables, setter methods, and properties (which can also be combined with the type 3 “constructor” injection used in the *movielisterapp* sample application).

Suggested next steps:

1. Read the *Aglyph cookbook*.
2. Read the *Aglyph API reference* and *The Aglyph Context fluent API*.

3. Read the `Aglyph context DTD`. The DTD is fully commented, and explains some of the finer points of using XML configuration.
4. Examine the Aglyph test cases (part of the distribution; located in the *tests/* directory).
5. Start with either the `movielisterapp-basic` or `movielisterapp-aglyph` applications and make your own modifications to explore the features of Aglyph.

1.3 Aglyph cookbook

Release 3.0.0.post1

1.3.1 Understand the general features of Aglyph

Release 3.0.0.post1

Aglyph supports Python 2 and 3 (CPython), PyPy, Stackless, IronPython, and Jython

The single Aglyph distribution and API is 100% compatible across Python language versions 2.7 and 3.4+, and the implementations listed below:

- CPython
- PyPy
- Jython
- IronPython
- Stackless Python

The *Aglyph 3.0.0.post1 testing summary* provides detailed information for the Python implementation releases and platforms under which the Aglyph unit test suite is executed.

Aglyph does not require framework-specific changes to your application code

Adding Aglyph Dependency Injection support to a new *or existing* application is easy, because Aglyph doesn't require any framework-specific code to reside in your application's business logic code.

Conceptually, Aglyph acts like a layer or proxy that sits *between* your business logic code and the scaffolding/controlling code that uses it. Put another way: Aglyph knows about your application components, but your application components don't know about Aglyph.

Any object (application-specific, Python standard library, or 3rd-party) can be an Aglyph component

Because Aglyph doesn't require that component source code be modified, either to be injected with dependencies or to serve *as* a dependency (or both!), **any** object can serve as a component in your application.

Aglyph will happily assemble any object that is reachable from an absolute importable dotted name and, optionally, attribute access on that object (via `aglyph.component.Component.factory_name` or `aglyph.component.Component.member_name`).

The only requirements for a component are that:

1. `aglyph.component.Component.component_id` must be unique within an *aglyph.context.Context*.
2. *aglyph.component.Component.dotted_name* must be a "dotted_name.NAME" or "dotted_name" string that represents a valid absolute import statement according to the productions listed below.

```
absolute_import_stmt  ::=   "from" dotted_name "import" NAME
                            | "import" dotted_name
dotted_name           ::=   NAME ('.' NAME)*
```

Whether you want to describe a component as an object from your own application, from the Python standard library, from a 3rd-party library, or even from an implementation-specific library (e.g. a .NET class in IronPython, a Java class in Jython), Aglyph is up to the task. No adapter code, no monkey-patching, no fuss.

1.3.2 Choose a configuration approach for Aglyph

Release 3.0.0.post1

Aglyph explicitly supports two methods of configuration:

1. Declarative XML configuration conforming to the `Aglyph context DTD`
2. Programmatic configuration via *The Aglyph Context fluent API*

Opinions vary widely on the merits of XML (particularly for configuration), but in fairness there's also plenty of debate over the merits of "code as configuration" (or "configuration as code" if you prefer) as well. Aglyph strives to **not** have an opinion one way or the other by supporting *either* approach.

However, both approaches to Aglyph configuration have strengths and weaknesses that you should understand before choosing one over the other.

Declarative XML configuration

An Aglyph context can be defined in an XML document that conforms to the `Aglyph context DTD`.

The XML context document is parsed by *aglyph.context.XMLContext*, which is just a subclass of *aglyph.context.Context* that knows how to populate itself from the parsed XML document. Once populated, this context can then be used to create an *aglyph.assembler.Assembler*:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> my_app_context = XMLContext("my-application-context.xml")
>>> assembler = Assembler(my_app_context)
```

Note: The `Aglyph context DTD` is provided primarily as a reference. The *aglyph.context.XMLContext* class uses a non-validating parser by default.

Developers are encouraged to explicitly validate an application's context XML document during testing.

See also:

Use a custom XML parser for XMLContext This recipe could also be used to force Aglyph to use a **validating** XML parser.

XMLContext configures mutable builtin objects safely

Consider the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<context id="cookbook">
    <component id="cookbook.Example">
        <attributes>
            <attribute name="mutable">
               <list>
                  <int>1</int>
                  <int>2</int>
                  <int>3</int>
               </list>
            </attribute>
        </attributes>
    </component>
</context>
```

Because builtin `list` objects are mutable, Aglyph will automatically turn the *"mutable"* attribute above into an *`aglyph.component.Evaluator`* (which is very similar to a `functools.partial`). Whenever the *"cookbook.Example"* component is assembled, the `Evaluator` for the *"mutable"* attribute is called, which will produce a *new* `list` object.

Why is this important? Consider a corresponding programmatic configuration for the same component:

```
context.prototype("cookbook.Example").set(mutable=[1, 2, 3]).register()
```

This configuration leads to a (likely) logic error: **all** objects of the *"cookbook.Example"* component will share a reference to a single list object. An example illustrates the problem:

```
>>> example1 = assembler.assemble("cookbook.Example")
>>> example1.mutable
[1, 2, 3]
>>> example1.mutable.append(4)
>>> example2 = assembler.assemble("cookbook.Example")
>>> example2.mutable
[1, 2, 3, 4]
```

Uh-oh! That's almost certainly *not* what we intended. To guard against this behavior, we would need to modify the binding:

```
from functools import partial
context.prototype("cookbook.Example").set(mutable=partial(list, [1, 2, 3])).register()
```

Now we will get a "fresh" list every time the component is assembled, so modifying the list on one instance will not affect the lists of any other instances.

(And what if we were actually specifying a list-of-list, or a tuple-of-list, or a list-of-dict? Now we would need to account for mutability of *each* member!)

This is an easy thing to forget, and can lead to a great deal of (programmatic) configuration code, which is why *`aglyph.context.XMLContext`* handles it automatically for any `<list>`, `<tuple>`, and `<dict>` declared in the XML context document.

See also:

Defer the resolution of injection values until assembly time

XMLContext is Unicode-aware and supports automatic character set conversion

Aglyph properly handles Unicode text and encoded byte data in XML context documents, regardless of Python version.

Aglyph can also provide your application components with byte data encoded to a user-specified character set.

Consider the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<context id="cookbook">
    <component id="cookbook.TextAndData">
        <attributes>
            <attribute name="text">
                <unicode>ΑΦΔ</unicode>
            </attribute>
            <attribute name="data1">
                <bytes>ΑΦΔ</bytes>
            </attribute>
            <attribute name="data2">
                <bytes encoding="iso-8859-7">ΑΦΔ</bytes>
            </attribute>
        </attributes>
    </component>
</context>
```

The first thing to notice is that `<bytes>ΑΦΔ</bytes>` is missing a character encoding. This can be problematic on Python 2, because the default string encoding used by the Unicode implementation is typically ASCII:

```
$ python2.7
Python 2.7.9 (default, Dec 13 2014, 15:13:49)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.getdefaultencoding()
'ascii'
>>> from aglyph.context import XMLContext
>>> context = XMLContext("cookbook-context.xml")
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal␣
↪not in range(128)
```

One solution would be to add the `encoding=` attribute. Alternatively, you can instruct `XMLContext` to use a different default encoding (it uses the value of `sys.getdefaultencoding()` by default):

```
$ python2.7
Python 2.7.9 (default, Dec 13 2014, 15:13:49)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> context = XMLContext("cookbook-context.xml", default_encoding="UTF-8")
>>> assembler = Assembler(context)
>>> text_and_data = assembler.assemble("cookbook.TextAndData")
>>> text_and_data.text
u'\u0391\u03a6\u0394'
>>> text_and_data.data1
'\xce\x91\xce\xa6\xce\x94'
```

(continues on next page)

(continued from previous page)

```
>>> text_and_data.data2
'\xc1\xd6\xc4'
```

If we run the same example under Python 3 (which uses “UTF-8” as the default encoding), we still get correct results, but without the need to explicitly set the default encoding on the `XMLContext`:

```
$ python3.4
Python 3.4.2 (default, Nov 12 2014, 18:23:59)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> context = XMLContext("cookbook-context.xml")
>>> assembler = Assembler(context)
>>> text_and_data = assembler.assemble("cookbook.TextAndData")
>>> text_and_data.text
'ΑΦΔ'
>>> text_and_data.data1
b'\xce\x91\xce\xa6\xce\x94'
>>> text_and_data.data2
b'\xc1\xd6\xc4'
```

One important thing to notice is the difference in the *types* of the Unicode and byte strings, dependent upon which version of Python is used.

Unicode and character encoding differences between Python 2 and Python 3

The builtin `str` type has changed significantly between Python 2 and Python 3 (see Text Vs. Data Instead Of Unicode Vs. 8-bit).

In short: `str` represented encoded byte data up to and including Python 2, but representes *Unicode text* as of Python 3.0:

Version	Unicode text	Encoded byte data
Python 2	`unicode`	`str`
Python 3	`str`	`bytes`

The `Aglyph context DTD` defines `<bytes>`, `<str>`, and `<unicode>` elements that correspond to the types in the table above, but treats the element content differently depending on the version of Python under which Aglyph is running:

Version	`<unicode>` content	`<str>` content	`<bytes>` content
Python 2	`unicode`	`str`	`str`
Python 3	`str`	`str`	`bytes`

To summarize the above:

- `<unicode>` is interpreted as a `unicode` type in Python 2 and a `str` type in Python 3
- `<str>` is always interpreted as a `str` type
- `<bytes>` is interpreted as a `str` type in Python 2 and a `bytes` type in Python 3

Note: For clarity in XML context documents, it is always safe to use `<bytes>` for encoded byte data and `<unicode>` for Unicode text (regardless of Python version), avoiding entirely the ambiguity of `<str>`.

Warning: Althoug the DTD permits an *encoding=”. . . ”* attribute on `<str>` elements, the attribute is **ignored** in Python 3 (a *WARNING*-level log message is emitted to the *aglyph.context.XMLContext* channel if it is present).

Programmatic configuration using the Context fluent API

New in version 3.0.0.

Objects of *`aglyph.context.Context`* now support a chained call “fluent” API. Please refer to *The Aglyph Context fluent API* for details.

Custom configuration using Context

Do **neither** declarative XML nor fluent configuration suit your fancy?

An *`aglyph.context.Context`* is just a `dict` that maps component ID strings (i.e. `aglyph.component.Component.component_id`) to *`aglyph.component.Component`* instances, so embrace the open source philosophy and “roll your own” configuration mechanism!

Note: Look at *`aglyph.context.XMLContext`* for a starting point - it’s just a *`aglyph.context.Context`* subclass that populates itself from a parsed XML document.

1.3.3 Common usage scenarios

Release 3.0.0.post1

All examples are shown with XML configuration *and* programmatic configuration, where appropriate.

- *Describe a simple component (a class or an unbound factory function)*
- *Describe any Python builtin type as a component*
- *Use a reference to another component as a dependency*
- *Defer the resolution of injection values until assembly time*
- *Declare a method to be called on an object after its dependencies have been injected*
- *Declare a method to be called on a singleton, borg, or weakref object before it is cleared from cache*
- *Avoid circular dependencies*

Describe a simple component (a class or an unbound factory function)

The simplest and most common kind of component is one that describes a module-level class or unbound factory function.

To demonstrate, we will describe a component for the Python standard library `http.client.HTTPConnection` class.

Note: Although we are using the Python standard library `http.client.HTTPConnection` class here, this example applies to **any** class or unbound function, whether defined as part of your application, the Python standard library, or a 3rd-party library.

Using declarative XML configuration

In the XML context document *"cookbook-context.xml"*:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="http.client.HTTPConnection">
        <init>
            <arg><str>ninthtest.info</str></arg>
            <arg><int>80</int>
            <arg keyword="timeout"><int>5</int></arg>
        </init>
    </component>
</context>
```

To assemble and use this component:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> conx = assembler.assemble("http.client.HTTPConnection")
>>> conx.request("GET", '/')
>>> response = conx.getresponse()
>>> print(response.status, response.reason)
200 OK
```

If we want to describe more than one component of the same class or function, then we need to use something other than the dotted name as the component IDs so that they are unique:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="ninthtest-info-conx" dotted-name="http.client.HTTPConnection">
        <init>
            <arg><str>ninthtest.info</str></arg>
            <arg><int>80</int>
            <arg keyword="timeout"><int>5</int></arg>
        </init>
    </component>
    <component id="python-org-conx" dotted-name="http.client.HTTPConnection">
        <init>
            <arg><str>www.python.org</str></arg>
            <arg><int>80</int>
            <arg keyword="timeout"><int>5</int></arg>
        </init>
    </component>
</context>
```

Accordingly, we use the component IDs to assemble these components:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> ninthtest_info = assembler.assemble("ninthtest-info-conx")
>>> python_org = assembler.assemble("python-org-conx")
```

Using fluent API configuration

Using *The Aglyph Context fluent API* to describe a simple component in a *bindings.py* module:

```
from http.client import HTTPConnection
from aglyph.context import Context

context = Context("cookbook-context")
context.prototype(HTTPConnection).init("ninthtest.info", 80, timeout=5).register()
```

To assemble and use the component:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> conx = assembler.assemble("http.client.HTTPConnection")
>>> conx.request("GET", '/')
>>> response = conx.getresponse()
>>> print(response.status, response.reason)
200 OK
```

And like XML contexts, when we wish to use multiple components of the same dotted name, we must give them unique component IDs:

```
from http.client import HTTPConnection
from aglyph.context import Context

context = Context("cookbook-context")
(context.prototype("ninthtest-info-conx").
    create(HTTPConnection).
    init("ninthtest.info", 80, timeout=5).
    register())
(context.prototype("python-org-conx").
    create(HTTPConnection).
    init("www.python.org", 80, timeout=5).
    register())
```

Assembling these components now requires the custom component IDs:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> ninthtest_info = assembler.assemble("ninthtest-info-conx")
>>> python_org = assembler.assemble("python-org-conx")
```

Describe any Python builtin type as a component

Python builtin types (e.g. `int`, `list`) can be identified by an importable dotted name, and so may be defined as components in Aglyph.

Using declarative XML configuration

Warning: The name of the module in which builtin types are defined differs between Python 2 and 3, so any Aglyph XML configuration that uses this approach will, by definition, **not** be compatible across Python versions.

The example given below uses the Python 3 `builtins` module. To make this example work on Python 2, the `__builtin__` module would be used instead.

In the XML context document *"cookbook-context.xml"*:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="foods" dotted-name="builtins.frozenset">
        <init>
            <arg>
                <list>
                    <str>spam</str>
                    <str>eggs</str>
                </list>
            </arg>
        </init>
    </component>
    <component id="opened-file" dotted-name="builtins.open">
        <init>
            <arg><str>/path/to/file.txt</str></arg>
            <arg keyword="encoding"><str>ISO-8859-1</str></arg>
        </init>
    </component>
</context>
```

Using fluent API configuration

Because the builtin types are accessible without having to do an explicit import, the fluent configuration is very simple.

In a *bindings.py* module:

```
from aglyph.context import Context

context = Context("cookbook-context")
context.prototype("foods").create(frozenset).init(["spam", "eggs"]).register()
(context.prototype("opened-file").
    create(open).
    init("/path/to/file.txt", encoding="ISO-8859-1").
    register())
```

Use a reference to another component as a dependency

An *`aglyph.component.Reference`* is a powerful mechanism for creating cross-references between components.

A `Reference` value is just a component ID, but a `Reference` triggers special behavior within an *`aglyph.assembler.Assembler`* or *`aglyph.component.Evaluator`* when it is encountered during assembly or evaluation (respectively): wherever the `Reference` appears, it will be automatically replaced with the fully-assembled component it identifies.

A `Reference` may be used in *any* of the following places, allowing for extremely flexible configurations:

- an initialization argument value (positional or keyword) for an *aglyph.component.Component* or an *aglyph.component.Evaluator*
- an attribute value for an *aglyph.component.Component*
- a key and/or value of a `dict`
- an item of any sequence type (e.g. `list`, `tuple`)

In a nutshell: an *aglyph.component.Reference* may be used in *any* case where a value is being defined, and will be replaced at assembly-time by the fully-assembled component identified by that reference.

To demonstrate, we will describe components for the Python standard library `urllib.request.Request` class and `urllib.request.urlopen()` function. (The former will be referenced as a dependency for the latter.)

Using declarative XML configuration

In the *"cookbook-context.xml"* document:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="ninthtest-home-page" dotted-name="urllib.request.Request">
        <init>
            <arg><str>http://ninthtest.info/</str></arg>
        </init>
    </component>
    <component id="ninthtest-url" dotted-name="urllib.request.urlopen">
        <init>
            <arg reference="ninthtest-home-page" />
            <arg keyword="timeout"><int>5</int></arg>
        </init>
    </component>
</context>
```

When the *"ninthtest-url"* component is assembled, the assembler will automatically assemble and inject the *"ninthtest-home-page"* component:

```python
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> ninthtest_url = assembler.assemble("ninthtest-url")
>>> print(ninthtest_url.status, ninthtest_url.reason)
200 OK
```

Using fluent API configuration

In a *bindings.py* module:

```python
from urllib.request import Request, urlopen
from aglyph.context import Context
from aglyph.component import Reference as ref

```

(continues on next page)

(continued from previous page)

```
context = Context("cookbook-context")
(context.component("ninthtest-home-page").
    create(Request).
    init("http://ninthtest.info/").
    register())
(context.component("ninthtest-url").
    create(urlopen).
    init(ref("ninthtest-home-page"), timeout=5).
    register())
```

When the *"ninthtest-url"* component is assembled, the assembler will automatically assemble and inject the *"ninthtest-home-page"* component:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> ninthtest_url = assembler.assemble("ninthtest-url")
>>> print(ninthtest_url.status, ninthtest_url.reason)
200 OK
```

Defer the resolution of injection values until assembly time

When specifying the values that should be injected into an object of a component as it is assembled, it is sometimes desired (or necessary) that those values be resolved **at the time the component is being assembled.**

The textbook example of such a case is a component that accepts some mutable sequence type (e.g. a `list`) as an injection value. If the value (the list) were resolved at the time the component is being defined, then all objects of that component would share a reference to the same list This means that changes to the list belonging to *any* instance will actually apply to *all* instances.

In almost all cases, this is not desired behavior. What we actually desire is for each instance of the component to have its *own* copy of the list.

The solution to this problem is to specify a dependency such that its actual value is determined on-the-fly when the component is being assembled. Aglyph supports several ways of accomplishing this.

Use a Reference to defer the assembly of a component

Whenever an *`aglyph.component.Reference`* is used to identify a component as a dependency, that component is not assembled until the *parent* component is assembled.

Using declarative XML configuration

Aglyph automatically creates an *`aglyph.component.Reference`* for any `<reference>` element encountered, or for any `<arg>`, `<attribute>`, `<key>`, or `<value>` element that specifies a `reference` attribute:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="cookbook-formatter" dotted-name="logging.Formatter">
        <init>
            <arg><str>%(asctime)s %(levelname)s %(message)s</str></arg>
        </init>
```

(continues on next page)

(continued from previous page)

```
    </component>
    <component id="cookbook-handler" dotted-name="logging.handlers.RotatingFileHandler
↪">
        <init>
            <arg><str>/var/log/cookbook.log</str></arg>
            <arg keyword="maxBytes"><int>1048576</int></arg>
            <arg keyword="backupCount"><int>3</int></arg>
        </init>
        <attributes>
            <attribute name="setFormatter">
                <reference id="cookbook-formatter" />
            </attribute>
        </attributes>
    </component>
    <component id="cookbook-logger" dotted-name="logging.getLogger" strategy=
↪"singleton">
        <init>
            <arg><str>cookbook</str></arg>
        </init>
        <attributes>
            <attribute name="addHandler" ref="cookbook-handler" />
        </attributes>
    </component>
</context>
```

Using fluent API configuration

In a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("cookbook-context")
(context.component("cookbook-formatter").create("logging.Formatter").
    init("%(asctime)s %(levelname)s %(message)s").register())
(context.component("cookbook-handler").create("logging.handlers.RotatingFileHandler").
    init("/var/log/cookbook.log", maxBytes=1048576, backupCount=3).
    set(setFormatter=ref("cookbook-formatter")).
    register())
(context.singleton("cookbook-logger").create("logging.getLogger").
    init("cookbook").
    attributes(addHandler=ref("cookbook-handler")).
    register())
```

Use a partial function or an Evaluator to defer the evaluation of a runtime value

Though almost all scenarios can be addressed by using components and References, in some cases you may prefer that a dependency is *not* defined as another component. In Aglyph, you can defer the evaluation of such a value until component assembly time by using either a `functools.partial` object or an *`aglyph.component.Evaluator`*.

A partial function and an Evaluator serve the same purpose, and share the same signature - *(func, *args, **keywords)* - but an Evaluator is capable of recognizing and assembling any *`aglyph.component.Reference`* that appears in *args* or *keywords*, while a partial function is not.

Note: When the arguments and/or keywords to a callable must specify an `aglyph.component.Reference`, use `aglyph.component.Evaluator`. Otherwise, use either `functools.partial` *or* an Evaluator.

Using declarative XML configuration

When using XML configuration, an `aglyph.component.Evaluator` is *automatically* created for any `<list>`, `<tuple>`, or `<dict>`.

There is no support for explicitly specifying either a `functools.partial` or an `aglyph.component.Evaluator`. There is nothing *preventing* you from declaring a component of `functools.partial` or `aglyph.component.Evaluator`, though the usefulness of the latter is questionable (and would very strongly suggest that a simpler configuration is possible).

In the following *"cookbook-context.xml"* document, the "states" keyword argument is automatically turned into an `aglyph.component.Evaluator`:

```xml
<?xml version="1.0" ?>
<context id="cookbook-context">
    <component id="cookbook.WorkflowManager">
        <init>
            <arg keyword="states">
                <dict>
                    <item>
                        <key><str>UNA</str></key>
                        <value><str>Unassigned</str></value>
                    </item>
                    <item>
                        <key><str>OPE</str></key>
                        <value><str>Open (Assigned)</str></value>
                    </item>
                    <item>
                        <key><str>CLO</str></key>
                        <value><str>Closed</str></value>
                    </item>
                </dict>
            </arg>
        </init>
    </component>
</context>
```

Using fluent API configuration

Warning: Unlike XML configuration, there is no provision for automatically creating an `aglyph.component.Evaluator` when using *The Aglyph Context fluent API*. Any value that should be the result of a `functools.partial` or an `aglyph.component.Evaluator` must be **explicitly** specified as such.

In a *bindings.py* module:

```python
import functools
from aglyph.context import Context
```

(continues on next page)

(continued from previous page)

```
context = Context("cookbook-context")
(context.component("cookbook.WorkflowManager").
    init(states=functools.partial(
        dict,
        UNA="Unassigned",
        OPE="Open (Assigned)",
        CLO="Closed")).
    register())
```

Declare a method to be called on an object after its dependencies have been injected

At times it may be desirable (or necessary) to call an "initialization" method on an assembled object before it is returned to the caller for use. For this purpose, Aglyph allows you to declare such a method name at the context, template, and/or component level.

Note: Please refer to *The lifecycle method lookup process* to understand how Aglyph determines *which* lifecycle method to call on an object when multiple options are declared at the context, template, and/or component level for a given object.

In the following examples, we assume that all of the objects implement a `prepare()` method. In such cases, a context-level `after_inject` lifecycle method may be appropriate. Alternatively, it could be declared in a template (see *Component inheritance using templates*).

Regardless of the configuration approach, the behavior is that the assembled object's `prepare()` method is called **after** the object is injected but **before** the object is returned to the caller.

Using declarative XML configuration

Note the use of the *after-inject* attribute on the `<context>` element:

```
<?xml version="1.0" ?>
<context id="cookbook-context" after-inject="prepare">
   <component id="object-a" dotted-name="cookbook.PreparableObjectA" />
   <component id="object-b" dotted-name="cookbook.PreparableObjectB" />
   <component id="object-c" dotted-name="cookbook.PreparableObjectC" />
</context>
```

Using fluent API configuration

Note the use of the *after_inject* keyword argument to the `call(...)` method:

```
from aglyph.context import Context

context = Context("cookbook-context", after_inject="prepare")
context.component("object-a").create("cookbook.PreparableObjectA").register()
context.component("object-b").create("cookbook.PreparableObjectB").register()
context.component("object-c").create("cookbook.PreparableObjectC").register()
```

Additionally, the fluent API supports a `call(...)` method that may be used to specify a lifecycle method:

```
context.component("cookbook.AnotherObject").call(after_inject="ready").register()
```

Declare a method to be called on a *singleton*, *borg*, or *weakref* object before it is cleared from cache

At times it may be desirable (or necessary) to call a "finalization" method on a cached object before it is cleared from Aglyph's internal cache. For this purpose, Aglyph allows you to declare such a method name at the context, template, and/or component level.

Note: Please refer to *The lifecycle method lookup process* to understand how Aglyph determines *which* lifecycle method to call on an object when multiple options are declared at the context, template, and/or component level for a given object.

Following is an example that declares a *singleton* GNU dbm key/date store using Python's `dbm.gnu` module. The store is configured to be opened in "fast" mode, meaning that writes are not synchronized. When using GDBM, it is important that every file opened is also closed (which causes any pending writes to be synchronized - i.e. written to disk). The example shows how to declare that the `close()` method is to be called before the cached GDBM object is evicted from the Aglyph singleton cache.

Using declarative XML configuration

Here we declare the `close()` method for the `before_clear` lifecycle state of the component by using the *before-clear* attribute of the `<component>` element:

```
<?xml version="1.0" ?>
<context id="cookbook-context">
   <component id="store" dotted-name="dbm.gnu" factory-name="open"
         strategy="singleton" before-clear="close">
      <init>
         <arg><str>/var/cookbook-store.db</str></arg>
         <arg><str>cf</str></arg>
      </init>
   </component>
</context>
```

Using fluent API configuration

Here we use the *before_clear* keyword argument when binding the GDBM store object:

```
from aglyph.context import Context

context = Context("cookbook-context")
(context.singleton("store").
    create("dbm.gnu", factory="open").
    init("/var/cookbook-store.db", "cf").
    call(before_clear="close").
    register())
```

Warning: Be careful when declaring `before_clear` lifecycle methods for *weakref* component objects, as the nature of weak references means that Aglyph **cannot** guarantee that the object still exists when the `aglyph.assembler.Assembler.clear_weakrefs()` method is called! Please refer to `weakref` for details.

Avoid circular dependencies

Consider two components, **A** and **B**. If **B** is a dependency of **A**, and **A** is also a dependency of **B**, then a circular dependency exists:

```
<component id="cookbook.A">
    <init>
        <arg reference="cookbook.B"/>
    </init>
</comonent>
<component id="cookbook.B">
    <init>
        <arg reference="cookbook.A"/>
    </init>
</comonent>
```

Aglyph will raise `aglyph.AglyphError` when it detects a circular reference during assembly.

Note: In software design in general, circular dependencies are frowned upon because they can lead to problems ranging from increased maintenance costs to infinite recursion and memory leaks. The existence of a circular dependency usually implies that the design can be improved to avoid such a relationship.

1.3.4 Component inheritance using templates

Release 3.0.0.post1

Aglyph supports a form of inheritance by allowing developers to declare that a component or template has a "parent."

The initialization arguments (positional and keyword) and attributes of the parent behave similarly to those of `functools.partial()`. Any positional arguments declared for a child are *appended* to the parent's declared positional arguments, and any keyword arguments and attributes declared for a child take precedence over (possibly overriding) the same-named keyword arguments or attributes declared by the parent.

Besides initialization arguments and attributes, child components or templates may also inherit lifecycle method declarations from their parents. However, it is important to understand that Aglyph will only call **one** lifecycle method on an object for a given lifecycle state. Please refer to *The lifecycle method lookup process* to understand how Aglyph determines *which* lifecycle method to call.

Any component or template may only declare **one** parent. However, there is no theoretical limit to the depth of parent/child relationships, and either a component *or* a template may serve as a parent.

Note: The difference between a component and a template is that a component may be assembled, while a template **cannot** be assembled and may be used *only* as the parent of another component or template.

Following are examples of how to use components and templates together in an Aglyph configuration context.

- *Use a template to declare common dependencies, and child components to declare specific dependencies*
- *"Extend" a component by using another component as the parent*

- *Use templates to declare the lifecycle methods used by similar components*

Use a template to declare common dependencies, and child components to declare specific dependencies

In this example, we will declare components for two versions of an HTTP server - a "simple" version and a "CGI" version. These components have one common dependency (the server address), which will be declared in the template, and one component-specific dependency (the request handler class), which will be declared in each component.

Using declarative XML configuration

In the XML context document *"cookbook-context.xml"*:

```
<?xml version="1.0" encoding="utf-8" ?>
<context id="cookbook-context">
   <template id="base-server">
      <init>
        <!-- server_address -->
         <arg>
            <tutple>
               <str>localhost</str>
               <int>8000</int>
            </tuple>
         </arg>
      </init>
   </template>
   <component id="http.server.SimpleHTTPRequestHandler"
         dotted-name="http.server" member-name="SimpleHTTPRequestHandler" />
   <component id="simple-server" dotted-named="http.server.HTTPServer"
         parent-id="base-server">
      <init>
         <!-- RequestHandlerClass -->
         <arg reference="http.server.SimpleHTTPRequestHandler" />
      </init>
   </component>
   <component id="http.server.CGIHTTPRequestHandler"
         dotted-name="http.server" member-name="CGIHTTPRequestHandler" />
   <component id="cgi-server" dotted-named="http.server.HTTPServer"
         parent-id="base-server">
      <init>
         <!-- RequestHandlerClass -->
         <arg reference="http.server.CGIHTTPRequestHandler" />
      </init>
   </component>
</context>
```

Assembling "simple-server" and "cgi-server", we can see that the server address is common, but that the request handler class differs:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> simple_server = assembler.assemble("simple-server")
>>> simple_server.server_address
('localhost', 8000)
```

(continues on next page)

(continued from previous page)

```
>>> simple_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> cgi_server = assembler.assemble("cgi-server")
>>> cgi_server.server_address
('localhost', 8000)
>>> cgi_server.RequestHandlerClass
<class 'http.server.CGIHTTPRequestHandler'>
```

Using fluent API configuration

In a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("cookbook-context")
context.template("base-server").init(("localhost", 8000)).register()
(context.component("simple-handler").
    create("http.server", member="SimpleHTTPRequestHandler").
    register())
(context.component("simple-server", parent="base-server").
    create("http.server.HTTPServer").
    init(ref("simple-handler")).
    register())
(context.component("cgi-handler").
    create("http.server", member="CGIHTTPRequestHandler").
    register())
(context.component("cgi-server", parent="base-server").
    create("http.server.HTTPServer").
    init(ref("cgi-handler")).
    register())
```

As in the XML example, assembling the "simple-server" and "cgi-server" components shows that the server address is common, but that the request handler class differs:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> simple_server = assembler.assemble("simple-server")
>>> simple_server.server_address
('localhost', 8000)
>>> simple_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> cgi_server = assembler.assemble("cgi-server")
>>> cgi_server.server_address
('localhost', 8000)
>>> cgi_server.RequestHandlerClass
<class 'http.server.CGIHTTPRequestHandler'>
```

"Extend" a component by using another component as the parent

In this example, we have a "default" HTTP server with stock settings and a "custom" HTTP server that extends the default to redefine several settings. Either server is fully functional as a standalone component, and so we use the default server as the parent of the custom server.

This example does not require the use of templates; any component can serve as the parent of another component.

Using declarative XML configuration

In the XML context document *"cookbook-context.xml"*:

```
<?xml version="1.0" encoding="utf-8" ?>
<context id="cookbook-context">
   <component id="request-handler" dotted-name="http.server"
         member-name="CGIHTTPRequestHandler" />
   <component id="default-server" dotted-named="http.server.HTTPServer">
      <init>
        <!-- server_address -->
         <arg>
            <tutple>
               <str>localhost</str>
               <int>8000</int>
            </tuple>
         </arg>
         <!-- RequestHandlerClass -->
         <arg reference="request-handler" />
      </init>
   </component>
   <component id="custom-server" dotted-named="http.server.HTTPServer"
         parent-id="default-server">
      <attributes>
         <attribute name="request_queue_size"><int>15</int></attribute>
         <attribute name="timeout"><float>3</float></attribute>
      </attributes>
   </component>
</context>
```

Assembling "default-server" and "custom-server", we can see that the server address and request handler class are the same, but that the custom server has non-default values for the request queue size and socket timeout:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> default_server = assembler.assemble("default-server")
>>> default_server.server_address
('localhost', 8000)
>>> default_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> default_server.request_queue_size
5
>>> default_server.timeout is None
True
>>> custom_server = assembler.assemble("custom-server")
>>> custom_server.server_address
('localhost', 8000)
>>> custom_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> custom_server.request_queue_size
15
>>> custom_server.timeout
3.0
```

Using fluent API configuration

In a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("cookbook-context")
(context.component("request-handler").
    create("http.server", member="SimpleHTTPRequestHandler").
    register())
(context.component("default-server").
   create("http.server.HTTPServer").
   init(("localhost", 8000), ref("request-handler")).
   register())
(context.component("custom-server", parent="default-server").
   create("http.server.HTTPServer").
   set(request_queue_size=15, timeout=3.0).
   register())
```

As in the XML example, assembling the “default-server” and “custom-server” components shows that the server address and request handler class are common, but that the request queue size and timeout differ:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> default_server = assembler.assemble("default-server")
>>> default_server.server_address
('localhost', 8000)
>>> default_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> default_server.request_queue_size
5
>>> default_server.timeout is None
True
>>> custom_server = assembler.assemble("custom-server")
>>> custom_server.server_address
('localhost', 8000)
>>> custom_server.RequestHandlerClass
<class 'http.server.SimpleHTTPRequestHandler'>
>>> custom_server.request_queue_size
15
>>> custom_server.timeout
3.0
```

Use templates to declare the lifecycle methods used by similar components

In this example, assume that a *cookbook.py* module contains the following class and method definitions:

```
class Hydrospanner:
   def calibrate(self):
      ...
   def disengage(self):
      ...

class Nervesplicer:
```

(continues on next page)

(continued from previous page)

```
    def prepare(self):
        self.sterilize()
        self.calibrate()
    def sterilize(self):
        ...
    def calibrate(self):
        ...
    def disengage(self):
        ...

class Macrofuser:
    def ignite(self):
        ...
    def extinguish(self):
        ...

class Vibrotorch:
    def ignite(self):
        ...
    def extinguish(self):
        ...
```

In the example configurations below, the *"mechanical-tool"* template (used as a parent by the `Hydrospanner` and `Nervesplicer` components) declares the `calibrate` and `disengage` lifecycle methods, and the *"incendiary-tool"* template (used as a parent by the `Macrofuser` and `Vibrotorch` components) declares the `ignite` and `extinguish` lifecycle methods.

Note: The `Nervesplicer` component represents a special case. While it declares *"mechanical-tool"* as its parent, and implements the `calibrate` initialization method, there is an additional initilization method (`sterilize`) which should be called. To accomplish this, the `Nervesplicer.prepare()` initialization method is implemented to call `sterilize()` *and* `calibrate()`, and is declared as the "after injection" lifecycle method for `Nervesplicer`, specifically.

The configurations shown below result in the following behaviors during the application's lifetime:

- When the *"cookbook.Hydrospanner"* component is assembled and has not yet been cached, its `calibrate` method is called before the object is cached and returned to the caller.
- When the *"cookbook.Nervesplicer"* component is assembled and has not yet been cached, its `prepare` method is called before the object is cached and returned to the caller.
- When **either** the *"cookbook.Hydrospanner"* or *"cookbook.Nervesplicer"* component is cleared from cache (via `aglyph.assembler.Assembler.clear_singletons()`), its `disengage` method is called.
- When **either** the *"cookbook.Macrofuser"* or *"cookbook.Vibrotorch"* component is assembled and has not yet been cached, its `ignite` method is called before the object is cached and returned to the caller.
- When **either** the *"cookbook.Macrofuser"* or *"cookbook.Vibrotorch"* component is cleared from cache (via `aglyph.assembler.Assembler.clear_singletons()`), its `extinguish` method is called.

Using declarative XML configuration

In a *coookbook-context.xml* document:

```
<?xml version="1.0" encoding="utf-8" ?>
<context id="cookbook-context">
   <template id="mechanical-tool"
         after-inject="calibrate" before-clear="disengage" />
   <component id="cookbook.Hydrospanner" strategy="singleton"
         parent-id="mechanical-tool" />
   <component id="cookbook.Nervesplicer" strategy="singleton"
         parent-id="mechanical-tool" after-inject="prepare" />
   <template id="incendiary-tool"
         after-inject="ignite" before-clear="extinguish" />
   <component id="cookbook.Macrofuser" strategy="singleton"
         parent-id="incendiary-tool" />
   <component id="cookbook.Vibrotorch" strategy="singleton"
         parent-id="incendiary-tool" />
</context>
```

Using fluent API configuration

In a *bindings.py* module:

```python
from aglyph.context import Context

context = Context("cookbook-context")
(context.template("mechanical-tool").
    call(after_inject="calibrate", before_clear="disengage").
    register())
context.singleton("cookbook.Hydrospanner", parent="mechanical-tool").register()
(context.singleton("cookbook.Nervesplicer", parent="mechanical-tool").
    call(after_inject="prepare").
    register())
(context.template("incendiary-tool").
    call(after_inject="ignite", before_clear="extinguish").
    register())
context.singleton("cookbook.Macrofuser", parent="incendiary-tool").register()
context.singleton("cookbook.Vibrotorch", parent="incendiary-tool").register()
```

1.3.5 Other notable usage scenarios

Release 3.0.0.post1

- *Describe components for static methods, class methods, or nested classes*
- *Describe components for module or class members*
- *Describe components for Python implementation-specific objects (Stackless, PyPy, IronPython, Jython)*
- *Use a custom XML parser for XMLContext*
- *Clear the Aglyph singleton, weakref, and borg memory caches*

Describe components for static methods, class methods, or nested classes

You may encounter cases where objects for which you want to describe components are created through static methods (`@staticmethod`), class methods (`@classmethod`), or nested classes.

Recall that an Aglyph component must have an *importable* dotted name. Unfortunately, the dotted names for static methods, class methods, and nested classes are **not** importable.

Aglyph components support an optional *aglyph.component.Component.factory_name* property to address these cases. The `factory_name` is itself a dotted name, but it is *relative to* *aglyph.component.Component.dotted_name*.

The following sample *delorean.py* module helps to illustrate the concepts, and will be used in the code examples below:

```python
class EquipmentFoundry:

    class FluxCapacitor:

        @classmethod
        def with_capacitor_drive(capacitor_drive):
            """Return a fully-operational FluxCapacitor."""

    @staticmethod
    def get_default_capacitor_drive():
        """Return an experimental CapacitorDrive."""
```

The objects which we wish to describe as components (`FluxCapacitor` and `CapacitorDrive`), are created by callables having dotted names that are **not importable**:

1. The class named `delorean.EquipmentFoundry.FluxCapacitor`.
2. The class method named `delorean.EquipmentFoundry.FluxCapacitor.with_capacitor_drive`.
3. The static method named `delorean.EquipmentFoundry.get_default_capacitor_drive`.

However, we can use either of the dotted names that *are* importable (the module named `delorean` and the class named `delorean.EquipmentFoundry`) to define our components, and specify *relative* dotted names as *aglyph.component.Component.factory_name* values to enable Aglyph to assemble `FluxCapacitor` and `CapacitorDrive` as components.

Using declarative XML configuration

In a *delorean-context.xml* document:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<context id="delorean-context">
    <component id="default-drive"
            dotted-name="delorean"
            factory-name="EquipmentFoundry.get_default_capacitor_drive" />
    <component id="capacitor-nodrive"
            dotted-name="delorean.EquipmentFoundry"
            factory-name="FluxCapacitor" />
    <component id="capacitor-withdrive"
            dotted-name="delorean.EquipmentFoundry"
            factory-name="FluxCapacitor.with_capacitor_drive">
        <init>
            <arg ref="default-drive" />
        </init>
    </component>
</context>
```

Key things to note in this configuration:

- For **any** of the components, we have the option of using either `delorean` or `delorean.EquipmentFoundry` as a component's dotted name because both of these names are importable. Which we choose influences how the factory name must be specified - it must be *relative to* the dotted name.
- Any factory name is just a dotted name - but split into its individual names, it must represent a callable object that can be obtained via attribute access on the module or class identified by the dotted name.

We can now assemble the *"capacitor-nodrive"* and *"capacitor-withdrive"* components as we would any other Aglyph components:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("delorean-context.xml"))
>>> flux_capacitor_without_drive = assembler.assemble("capacitor-nodrive")
>>> flux_capacitor_with_drive = assembler.assemble("capacitor-withdrive")
```

Note: If you remember nothing else, remember this:

1. *aglyph.component.Component.dotted_name* must be **importable**.
2. *aglyph.component.Component.factory_name* must be *relative to* the importable `dotted_name`.

Using fluent API configuration

Here is an equivalent programmatic configuration in a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("delorean-context")
(context.component("default-drive").
    create("delorean", factory="EquipmentFoundry.get_default_capacitor_drive").
    register())
(context.component("capacitor-nodrive").
    create("delorean.EquipmentFoundry", factory="FluxCapacitor").
    register())
(context.component("capacitor-withdrive").
    create("delorean.EquipmentFoundry", factory="FluxCapacitor.with_capacitor_drive").
    init(ref("default-drive")).
    register())
```

Describe components for module or class members

Similar in nature to the `factory_name` property explained in the previous section, the *aglyph.component.Component.member_name* property provides a way to access objects that are not directly importable.

But there are two key differences between `member_name` and `factory_name`:

1. The object identified by a `member_name` is not required to be callable. Instead, the object itself is considered to **be** the component object.
2. Even if the object identified by a `member_name` *is* callable, Aglyph will **not** call it.

Note: As a consequence of #1, any initialization arguments or keywords that are specified for a component that also specifies a `member_name` are **ignored** (i.e. Aglyph does **not** initialize the `member_name` object). However, any "setter" dependencies (setter methods, fields, properties) defined for such a component **are** processed.

As a consequence of #2, you can define components whose objects are of *any* type, including class types, function types, and (sub)module types.

In the examples below, we will use the Python standard library `http.server.HTTPServer` class (whose initializer accepts a `class` object for the request handler class) to demonstrate one possible use of the *`aglyph.component.Component.member_name`* property.

Using declarative XML configuration

In a *cookbook-context.xml* document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="request-handler-class"
            dotted-name="http.server"
            member-name="BaseHTTPRequestHandler" />
    <component id="http-server" dotted-name="http.server.HTTPServer">
        <init>
            <arg>
                <tuple>
                    <str>localhost</str>
                    <int>8080</int>
                </tuple>
            </arg>
            <arg reference="request-handler-class" />
        </init>
    </component>
</context>
```

When the *"http-server"* component is assembled, its second initialization argument is actually the *class* `http.server.BaseHTTPRequestHandler` (as opposed to an *instance* thereof):

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> httpd = assembler.assemble("http-server")
>>> httpd.RequestHandlerClass
<class 'http.server.BaseHTTPRequestHandler'>
>>> assembler.assemble("request-handler-class") is httpd.RequestHandlerClass
True
```

Using fluent API configuration

Here is an equivalent programmatic configuration in a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("cookbook-context")
```

(continues on next page)

(continued from previous page)

```
(context.component("request-handler-class").
    create("http.server", member="BaseHTTPRequestHandler").
    register())
(context.component("http-server").
    create("http.server.HTTPServer").
    init(("localhost", 8080), ref("request-handler-class")).
    register())
```

Describe components for Python implementation-specific objects (Stackless, PyPy, IronPython, Jython)

Strictly speaking, there is nothing "special" (from an Aglyph perspective) about the examples presented in the following subsections. They just build upon the previous cookbook recipes *Describe a simple component (a class or an unbound factory function)*, *Use a reference to another component as a dependency*, and *Describe components for static methods, class methods, or nested classes* to once again demonstrate that Aglyph can assemble *any* component that can be described using dotted name notation, even when the class or function is only available to a specific implementation of Python.

Example 1: Describe a component for a Stackless Python or PyPy tasklet

The Stackless Python and PyPy Python implementations support the stackless.tasklet wrapper, which allows any callable to run as a microthread.

The examples below demonstrate the Aglyph configuration for a variation of the sample code given in the Stackless Python "Tasklets" Wiki article.

Using declarative XML configuration

In a *cookbook-context.xml* document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="cookbook-context">
    <component id="aCallable-func" dotted-name="cookbook"
            member-name="aCallable" />
    <component id="aCallable-task" dotted-name="stackless.tasklet">
        <init>
            <arg reference="aCallable-func" />
        </init>
    </component>
</context>
```

Assembling and running this tasklet looks like this:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("cookbook-context.xml"))
>>> task = assembler.assemble("aCallable-task")
>>> task.setup("assembled by Aglyph")
>>> task.run()
'aCallable: assembled by Aglyph'
```

Using fluent API configuration

Below is an example of programmatic configuration, but with a twist - we allow Aglyph to inject the function argument into the task so that we only need to assemble and run it. This works because the `stackless.tasklet.setup` method has setter method semantics.

In a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("cookbook-context")
context.component("aCallable-func").create("cookbook", member="aCallable").register()
(context.component("aCallable-task").
    create("stackless.tasklet").
    init(ref("aCallable-func")).
    set(setup="injected by Aglyph").
    register())
```

Assembling and running this tasklet looks like this:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> task = assembler.assemble("aCallable-task")
>>> task.run()
'aCallable: injected by Aglyph'
```

Example 2: Describe a component for a .NET XmlReader

IronPython developers have access to the .NET Framework Standard Library and any custom assemblies via the `clr` module, allowing any .NET namespace to be loaded into the IronPython runtime and used.

In the examples below, we use System.Xml.DtdProcessing, System.Xml.ValidationType, System.Xml.XmlReaderSettings, and System.Xml.XmlReader to configure an XML reader that parses a fictitious "AppConfig.xml" document.

Warning: When using IronPython, the .NET namespace for any class referenced in an Aglyph component **must** be loaded prior to asking Aglyph to assemble the component. (Otherwise, those classes would not be importable in IronPython.)

In the examples given below, this means that the following statements must be executed *before* `aglyph.assembler.Assembler.assemble()` or `aglyph.binder.Binder.lookup()` is called (because the "System.Xml" namespace is not present by default):

```
>>> import clr
>>> clr.AddReference("System.Xml")
```

Using declarative XML configuration

In a *dotnet-context.xml* document:

```
<?xml version="1.0" ?>
<context id="dotnet-context">
    <component id="dtd-parse" dotted-name="System.Xml"
            member-name="DtdProcessing.Parse" />
    <component id="dtd-validate" dotted-name="System.Xml"
            member-name="ValidationType.DTD" />
    <component id="xmlreader-settings" dotted-name="System.Xml.XmlReaderSettings">
        <attributes>
            <attribute name="IgnoreComments"><true /></attribute>
            <attribute name="IgnoreProcessingInstructions"><true /></attribute>
            <attribute name="IgnoreWhitespace"><true /></attribute>
            <attribute name="DtdProcessing" reference="dtd-parse" />
            <attribute name="ValidationType" reference="dtd-validate" />
        </attributes>
    </component>
    <component id="app-config-reader" dotted-name="System.Xml.XmlReader"
            factory-name="Create">
        <init>
            <arg><str>file:///C:/Example/Settings/AppConfig.xml</str></arg>
            <arg reference="xmlreader-settings" />
        </init>
    </component>
</context>
```

With the Aglyph context in place, we can now assemble an XML reader for our fictitious application configuration reader:

```
>>> import clr
>>> clr.AddReference("System.Xml")
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("dotnet-context.xml"))
>>> assembler.assemble("app-config-reader")
<System.Xml.XmlValidatingReaderImpl object at 0x000000000000002B [System.Xml.
↪XmlValidatingReaderImpl]>
```

Using fluent API configuration

Here is an equivalent programmatic configuration in a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference

context = Context("dotnet-context")
(context.component("dtd-parse").
    create("System.Xml", member="DtdProcessing.Parse").
    register())
(context.component("dtd-validate").
    create("System.Xml", member="ValidationType.DTD").
    register())
(context.component("xmlreader-settings").
    create("System.Xml.XmlReaderSettings").
    set(
        IgnoreComments=True,
        IgnoreProcessingInstructions=True,
```

(continues on next page)

(continued from previous page)

```
        IgnoreWhitespace=True,
        DtdProcessing=Reference("dtd-parse"),
        ValidationType=Reference("dtd-validate")).
    register())
(context.component("app-config-reader").
    create("System.Xml.XmlReader", factory_name="Create").
    init(
        "file:///C:/Example/Settings/AppConfig.xml",
        Reference("xmlreader-settings")).
    register())
```

The code to assemble the fictitious application configuration reader looks like this:

```
>>> import clr
>>> clr.AddReference("System.Xml")
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> assembler.assemble("app-config-reader")
<System.Xml.XmlValidatingReaderImpl object at 0x000000000000002B [System.Xml.
↪XmlValidatingReaderImpl]>
```

Example 3: Describe a component for a Java™ LinkedHashMap

Jython developers have direct access to the Java™ Platform API and any custom JARs in the runtime *CLASSPATH*.

In the examples below, we use java.util.Collections#synchronizedMap(java.util.Map) and java.util.LinkedHashMap to configure a thread-safe, insertion-order hash map.

Using declarative XML configuration

In a *java-context.xml* document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<context id="java-context">
    <component id="java.util.LinkedHashMap" />
    <component id="threadsafe-ordered-map" dotted-name="java.util.Collections"
            factory-name="synchronizedMap">
        <init>
            <arg reference="java.util.LinkedHashMap" />
        </init>
    </component>
</context>
```

To assemble our map:

```
>>> from aglyph.assembler import Assembler
>>> from aglyph.context import XMLContext
>>> assembler = Assembler(XMLContext("java-context.xml"))
>>> mapping = assembler.assemble("threadsafe-ordered-map")
>>> mapping.__class__
<type 'java.util.Collections$SynchronizedMap'>
```

Using fluent API configuration

Here is an equivalent programmatic configuration in a *bindings.py* module:

```
from aglyph.context import Context
from aglyph.component import Reference

context = Context("java-context")
context.component("java.util.LinkedHashMap")
(context.component("threadsafe-ordered-map").
    create("java.util.Collections", factory="synchronizedMap").
    init(Reference("java.util.LinkedHashMap")).
    register())
```

Assembling the map looks like this:

```
>>> from aglyph.assembler import Assembler
>>> from bindings import context
>>> assembler = Assembler(context)
>>> mapping = assembler.assemble("threadsafe-ordered-map")
>>> mapping.__class__
<type 'java.util.Collections$SynchronizedMap'>
```

Use a custom XML parser for XMLContext

Aglyph uses the `xml.etree.ElementTree` API for processing context documents. By default, ElementTree uses the Expat XML parser (via `xml.etree.ElementTree.XMLParser`) to build element structures.

However, developers may subclass `xml.etree.ElementTree.XMLParser` to use *any* XML parser; simply pass an instance of the subclass to *`aglyph.context.XMLContext`* as the `parser` keyword argument.

Note: For a real, working example, please refer to `aglyph._compat.CLRXMLParser`, which is an `xml.etree.ElementTree.XMLParser` subclass that uses the .NET System.Xml.XmlReader parser.

Clear the Aglyph singleton, weakref, and borg memory caches

`aglyph.assembler.Assembler` automatically caches objects of **singleton** and **weakref** components, as well as the shared-state dictionaries of **borg** components, in memory. There is no automatic eviction strategy.

These caches may be cleared explicitly by calling *`aglyph.assembler.Assembler.clear_singletons()`*, *`aglyph.assembler.Assembler.clear_weakrefs()`*, or *`aglyph.assembler.Assembler.clear_borgs()`*, respectively. Each method returns a list of component IDs that were evicted.

Warning: There are some limitations on weakref caching, particularly with respect to *lifecycle methods*. Please see *`aglyph.assembler.Assembler.clear_weakrefs()`* for details.

1.3.6 Integrating Aglyph

Release 3.0.0.post1

- *Use Aglyph to configure your CherryPy application*

- *Provide dependency injection support to your application using AglyphDIPlugin*
- *Manage the lifecycles of your application components*

Use Aglyph to configure your CherryPy application

CherryPy's custom plugins and tools already provide a DI-like way to manage your web application's runtime dependencies, but *configuring* those custom plugins and tools can still result in bootstrap code that tightly couples their configuration and use. In this example, we'll use Aglyph to configure a CherryPy application to use Jinja templating.

Using declarative XML configuration

In the *myapp/config/myapp-context.xml* file:

```
<?xml version="1.0" encoding="utf-8" ?>
<context id="myapp-context">
   <component id="jinja2-loader" dotted-name="jinja2.FileSystemLoader">
      <init>
         <arg><str>templates</str></arg>
      </init>
   </component>
   <component id="jinja2.Environment">
      <init>
         <arg reference="jinja2-loader" />
      </init>
   </component>
   <component id="template-tool"
         dotted-name="myapp.tools.jinja2tool.Jinja2Tool">
      <init>
         <arg reference="jinja2.Environment" />
      </init>
   </component>
   <component id="cherrypy-tools" dotted-name="cherrypy" member-name="tools">
      <attributes>
         <attribute name="template" reference="template-tool" />
      </attributes>
   </component>
</context>
```

Now in our application's main module we simply use Aglyph to assemble the *"cherrypy-tools"* component, which has the effect of setting `cherrypy.tools.template` to an instance of `myapp.tools.jinja2tool.Jinja2Tool`:

```python
from algpyh.assembler import Assembler
from aglyph.context import XMLContext

assembler = Assembler(XMLContext("config/myapp-context.xml"))
# note that we do not assign the assembled component - it's unnecessary
assembler.assemble("cherrypy-tools")
```

Alternatively, we could assemble just the *"template-tool"* component and assign it explicitly:

```python
import cherrypy
from algpyh.assembler import Assembler
from aglyph.context import XMLContext
```

(continues on next page)

(continued from previous page)

```
assembler = Assembler(XMLContext("config/myapp-context.xml"))

cherrypy.tools.template = assembler.assemble("template-tool")
```

Using fluent API configuration

In a *bindings.py* module for the *myapp* application:

```
from aglyph.context import Context
from aglyph.component import Reference as ref

context = Context("myapp-context")
(context.component("jinja2-loader").
    create("jinja2.FileSystemLoader").
    init("templates").
    register())
context.component("jinja2.Environment").init(loader=ref("jinja2-loader")).register()
(context.component("template-tool").
    create("myapp.tools.jinja2tool.Jinja2Tool").
    init(ref("jinja2.Environment")).
    register())
(context.component("cherrypy-tools").
    create("cherrypy", member="tools").
    set(template=ref("template-tool")).
    register())
```

Now in our application's main module we simply use Aglyph to assemble the *"cherrypy-tools"* component, which has the effect of setting `cherrypy.tools.template` to an instance of `myapp.tools.jinja2tool.Jinja2Tool`:

```
from aglyph.assembler import Assembler
from bindings import context

# note that we do not assign the assembled component - it's unnecessary
Assembler(context).assemble("cherrypy-tools")
```

Alternatively, we could assemble just the *"template-tool"* component and assign it explicitly:

```
import cherrypy
from aglyph.assembler import Assembler
from bindings import context

cherrypy.tools.template = Assembler(context).assemble("template-tool")
```

Provide dependency injection support to your application using `AglyphDIPlugin`

This example shows how to use `aglyph.integration.cherrypy.AglyphDIPlugin` (a `cherrypy.process.plugins.SimplePlugin`), allowing your application's other plugins, tools, and dispatchers to assemble components via CherryPy's Web Site Process Bus.

Using declarative XML configuration

Using an Aglyph XML context document *myapp/config/myapp-context.xml*, configure the Aglyph DI plugin in your application's main module like so:

```
import cherrypy
from aglyph.assembler import Assembler
from aglyph.context import XMLContext

assembler = Assembler(XMLContext("config/myapp-context.xml"))
cherrypy.engine.aglyph = AglyphDIPlugin(cherrypy.engine, assembler)
```

Components may now be assembled by publishing **"aglyph-assemble"** messages to the bus. For example:

```
my_obj = cherrypy.engine.publish("aglyph-assemble", "my-component-id").pop()
```

Using fluent API configuration

Using an application-specific *bindings.py* module, configure the Aglyph DI plugin in your application's main module like so:

```
import cherrypy
from aglyph.assembler import Assembler
from bindings import context

cherrypy.engine.aglyph = AglyphDIPlugin(cherrypy.engine, Assembler(context))
```

Components may now be assembled by publishing **"aglyph-assemble"** messages to the bus. For example:

```
my_obj = cherrypy.engine.publish("aglyph-assemble", "my-component-id").pop()
```

Manage the lifecycles of your application components

The `aglyph.integration.cherrypy.AglyphDIPlugin` subscribes to channels for controlling the lifecycles of Aglyph **singleton**, **borg**, and **weakref** components:

- "aglyph-init-singletons"
- "aglyph-clear-singletons"
- "aglyph-init-borgs"
- "aglyph-clear-borgs"
- "aglyph-clear-weakrefs"

Refer to the plugin class documentation for details.

1.4 Aglyph API reference

Release 3.0.0.post1

The Aglyph API consists of a single top-level package (`aglyph`) along with several modules and subpackages, as listed below.

1.4.1 `aglyph` — Dependency Injection for Python

Release 3.0.0.post1

This module defines a custom error type and several utility functions used by Aglyph.

Note: Aglyph uses the standard `logging` module, but by default registers a `logging.NullHandler` to suppress messages.

To enable Aglyph logging, configure a logger and handler for the *"aglyph"* log channel (see `logging.config`).

Note: New in version 3.0.0.

Aglyph framework functions and methods are fully traced using Autologging. However, all tracing is **deactivated** by default.

To activate tracing:

1. Configure a logger and handler for the *"aglyph"* log channel and set the logging level to `autologging.TRACE`.
2. Run Aglyph with the *AGLYPH_TRACED* environment variable set to a **non-empty** value.

exception `aglyph.`**`AglyphError`**(*message*, *cause=None*)

Bases: `exceptions.Exception`

Raised when Aglyph operations fail with a condition that is not sufficiently described by a built-in exception.

`aglyph.`**`format_dotted_name`**(*obj*)

Return the importable dotted-name string for *obj*.

Parameters **`obj`** – an **importable** class, function, or module

Returns a dotted name representing *obj*

Return type `str`

Raises ***`AglyphError`*** – if *obj* does not have a resolvable (importable) dotted name

The dotted name returned by this function is a *"dotted_name.NAME"* or *"dotted_name"* string for *obj* that represents a valid absolute import statement according to the following productions:

```
absolute_import_stmt  ::=  "from" dotted_name "import" NAME
                           | "import" dotted_name
dotted_name           ::=  NAME ('.' NAME)*
```

Note: This function is the inverse of *`resolve_dotted_name()`*.

Warning: This function will attempt to use the `__qualname__` attribute, which is only available in Python 3.3+. When `__qualname__` is **not** available, `__name__` is used instead.

See also:

PEP 3155, `aglyph._compat.name_of()`

aglyph.**resolve_dotted_name**(*dotted_name*)

Return the class, function, or module identified by *dotted_name*.

Parameters **dotted_name** (str) – a string representing an **importable** class, function, or module

Returns a class, function, or module

dotted_name must be a "dotted_name.NAME" or "dotted_name" string that represents a valid absolute import statement according to the following productions:

```
absolute_import_stmt  ::=   "from" dotted_name "import" NAME
                            | "import" dotted_name
dotted_name           ::=   NAME ('.' NAME)*
```

Note: This function is the inverse of *format_dotted_name()*.

1.4.2 aglyph.assembler — The Aglyph component assembler

Release 3.0.0.post1

The Aglyph assembler creates application objects from component definitions, injecting dependencies into those objects through initialization arguments and keywords, attributes, setter methods, and/or properties.

Application components and their dependencies are defined in an *aglyph.context.Context*, which is used to initialize an assembler.

An assembler provides thread-safe caching of **singleton** component instances, **borg** component shared-states (i.e. instance `__dict__` references), and **weakref** component instance weak references.

class aglyph.assembler.**Assembler**(*context*)

Bases: object

Create application objects using type 2 (setter) and type 3 (constructor) dependency injection.

Parameters **context** (aglyph.context.Context) – a context object mapping unique IDs to component and template definitions

assemble(*component_spec*)

Create an object identified by *component_spec* and inject its dependencies.

Parameters **component_spec** – a unique component ID, or an object whose dotted name is a unique component ID

Returns a complete object with all of its resolved dependencies

Raises

- **KeyError** – if *component_spec* does not identify a component in this assembler's context
- ***aglyph.AglyphError*** – if *component_spec* causes a circular dependency

If *component_spec* is a string, it is assumed to be a unique component ID and is used as-is. Otherwise, *aglyph.format_dotted_name()* is called to convert *component_spec* into a dotted name string, which is assumed to be the component's unique ID.

How a component object is assembled (created, initialized, wired, and returned) is determined by the component's *aglyph.component.Component.strategy*:

"prototype" A new object is always created, initialized, wired, and returned.

This is the default assembly strategy for Aglyph components.

"singleton" If the component has been assembled already during the current application lifetime **and** there has been no intervening call to *`clear_singletons()`*, then a cached reference to the object is returned.

Otherwise, a new object is created, initialized, wired, cached, and returned.

Singleton component objects are cached by their `aglyph.component.Component.unique_id`.

Note: Assembly of singleton components is a thread-safe operation.

"borg" If the component has been assembled already during the current application lifetime **and** there has been no intervening call to *`clear_borgs()`*, then a new instance is created and a cached reference to the shared-state is directly assigned to the instance `__dict__`.

Otherwise, a new instance is created, initialized, and wired; its instance `__dict__` is cached; and the instance is returned.

Borg component instance shared-states are cached by their `aglyph.component.Component.unique_id`.

Note: Assembly of borg components is a thread-safe operation.

Warning: The borg assembly strategy is **only** supported for components whose objects have an instance `__dict__`.

This means that components using builtin classes, or components using classes that define or inherit a `__slots__` member, **cannot** be declated as borg components.

New in version 2.1.0: support for the "weakref" assembly strategy

"weakref" In the simplest terms, this is a "prototype" that can exhibit "singleton" behavior: as long as there is at least one "live" reference to the assembled object in the application runtime, then requests to assemble this component will return the same (cached) object.

When the only reference to the assembled object that remains is the cached (weak) reference, the Python garbage collector is free to destroy the object, at which point it is automatically removed from the Aglyph cache. Subsequent requests to assemble the same component will cause a new object to be created, initialized, wired, cached (as a weak reference), and returned.

Note: Please refer to the `weakref` module for a detailed explanation of weak reference behavior.

New in version 2.0.0: **Either** *`aglyph.component.Component.factory_name`* **or** *`aglyph.component.Component.member_name`* may be defined to exercise more control over how a component object is created and initialized. Refer to the linked documentation for details.

Note: This method is called recursively to assemble any dependency of *component_spec* that is defined as a *`aglyph.component.Reference`*.

init_singletons()

Assemble and cache all singleton component objects.

Returns the initialized singleton component IDs

Return type list

This method may be called at any time to "prime" the internal singleton cache. For example, to eagerly initialize all singleton components for your application:

```
assembler = Assembler(my_context)
assembler.init_singletons()
```

Note: Only singleton components that do not *already* have cached objects will be initialized by this method.

Initialization of singleton component objects is a thread-safe operation.

clear_singletons()

Evict all cached singleton component objects.

Returns the evicted singleton component IDs

Return type list

Aglyph makes the following guarantees:

1. All cached singleton objects' "before_clear" lifecycle methods are called (if specified) when they are evicted from cache.
2. The singleton cache will be empty when this method terminates.

Note: Any exception raised by a "before_clear" lifecycle method is caught, logged, and issued as a RuntimeWarning.

Eviction of cached singleton component objects is a thread-safe operation.

init_borgs()

Assemble and cache the shared-states for all borg component objects.

Returns the initialized borg component IDs

Return type list

This method may be called at any time to "prime" the internal borg cache. For example, to eagerly initialize all borg component shared-states for your application:

```
assembler = Assembler(my_context)
assembler.init_borgs()
```

Note: Only borg components that do not *already* have cached shared-states will be initialized by this method.

Initialization of borg component shared-states is a thread-safe operation.

clear_borgs()

Evict all cached borg component shared-states.

Returns the evicted borg component IDs

Return type `list`

Aglyph makes the following guarantees:

1. All cached borg shared-states' "before_clear" lifecycle methods are called (if specified) when they are evicted from cache.
2. The borg cache will be empty when this method terminates.

Note: Any exception raised by a "before_clear" lifecycle method is caught, logged, and issued as a `RuntimeWarning`.

Eviction of cached borg component shared-states is a thread-safe operation.

`clear_weakrefs``()`

Evict all cached weakref component objects.

Returns the evicted weakref component IDs

Return type `list`

Aglyph makes the following guarantees:

1. **IF** a cached weakref object is still available **AND** the component definition specifies a "before_clear" lifecycle method, Aglyph will call that method when the object is evicted.
2. The weakref cache will be empty when this method terminates.

Note: Any exception raised by a "before_clear" lifecycle method is caught, logged, and issued as a `RuntimeWarning`.

Eviction of cached weakref component objects is a thread-safe operation.

Warning: While eviction of weakref components is a thread-safe operation with respect to *explicit* modification of the weakref cache (i.e. any other thread attempting to `assemble()` a weakref component or to `clear_weakrefs()` will be blocked until this method returns), the nature of weak references means that entries may still "disappear" from the cache *even while the cache lock is held.*

With respect to cache-clearing, this means that referent component objects may no longer be available even *after* the cache lock has been acquired and the weakref component IDs (keys) are retrieved from the cache. Practically speaking, this means that callers must be aware of two things:

1. Aglyph **cannot** guarantee that "before_clear" lifecycle methods are called on weakref component objects, because there is no guarantee that a cached weak references is "live." (This is the nature of weak references.)
2. Aglyph will only return the component IDs of weakref component objects that were "live" at the moment they were cleared.

Please refer to the `weakref` module for a detailed explanation of weak reference behavior.

`__contains__``(`*component_spec*`)`

Tell whether or not the component identified by *component_spec* is defined in this assembler's context.

Parameters **`component_spec`** – used to determine the dotted name or component unique ID

Returns `True` if *component_spec* identifies a component that is defined in this assembler's context, else `False`

Note: Any *component_spec* for which this method returns `True` can be assembled by this assembler.

Accordingly, this method will return `False` if *component_spec* actually identifies a *`aglyph.component.Template`* defined in this assembler's context.

1.4.3 `aglyph.component` — Defining components and their dependencies

Release 3.0.0.post1

The classes in this module are used to define components and their dependencies within an Aglyph context (*`aglyph.context.Context`*).

Components and Templates

`aglyph.component.Component` tells an *`aglyph.assembler.Assembler`* how to create objects of a particular type. The component defines the initialization and/or attribute dependencies for objects of that type, as well as the assembly strategy and any lifecycle methods that should be called.

`aglyph.component.Template` is used to describe dependencies (initialization and/or attribute) and lifecylce methods that are shared by multiple components. Templates are similar to abstract classes; they cannot be assembled, but are instead used as "parents" of other components (or templates) to achieve a sort of "configuration inheritance."

Note: Both `Component` and `Template` may serve as the parent of any other component or template; but only components may be assembled.

References and Evaluators

A *`aglyph.component.Reference`* may be used as the value of any initialization argument (positional or keyword) or attribute in a component or template. Its value must be the unique ID of a **component** in the same context. At assembly time, the assembler will resolve the reference into an object of the component to which it refers.

An *`aglyph.component.Evaluator`* is similar to a `functools.partial()` object. It stores a callable factory (function or class) and related initialization arguments, and can be called repeatedly to produce new objects. (Unlike a `functools.partial()` object, though, an `Evaluator` will resolve any initialization argument that is a *`aglyph.component.Reference`*, `functools.partial()`, or `Evaluator` **before** calling the factory.)

Strategies and Lifecycle methods

`aglyph.component.Strategy` defines the assembly strategies supported by Aglyph (*"prototype"*, *"singleton"*, *"borg"*, *"weakref"* and *"_imported"*).

`LifecycleState` defines assmebly states for components at which Aglyph supports calling named methods on the objects of those components. (Such methods may be used to perform specialized initialization or disposal, for example.)

`aglyph.component.`**`Strategy = Strategy(PROTOTYPE='prototype', SINGLETON='singleton', BORG='bo`**

Define the component assembly strategies implemented by Aglyph.

“prototype”

A new object is always created, initialized, wired, and returned.

Note: “prototype” is the default assembly strategy for Aglyph components that do not specify a member name.

“singleton”

The cached object is returned if it exists. Otherwise, the object is created, initialized, wired, cached, and returned.

Singleton component objects are cached by `Component.unique_id`.

“borg”

A new instance is always created. The shared-state is assigned to the new instance’s `__dict__` if it exists. Otherwise, the new instance is initialized and wired, its instance `__dict__` is cached, and then the instance is returned.

Borg component instance shared-states are cached by `Component.unique_id`.

Warning:

- The borg assembly strategy is **only** supported for components that are non-builtin classes.
- The borg assembly strategy is **not** supported for classes that define or inherit a `__slots__` member.

“weakref”

In the simplest terms, this is a “prototype” that can exhibit “singleton” behavior: as long as there is at least one “live” reference to the assembled object in the application runtime, then requests to assemble this component will return the same (cached) object.

When the only reference to the assembled object that remains is the cached weak reference, the Python garbage collector is free to destroy the object, at which point it is automatically removed from the Aglyph cache.

Subsequent requests to assemble the same component will cause a new object to be created, initialized, wired, cached (as a weak reference), and returned.

Note: Please refer to the `weakref` module for a detailed explanation of weak reference behavior.

“_imported”

New in version 3.0.0.

Note: The “_imported” strategy is only valid (and is the only allowed value) when *member_name* is specified for a component.

Since this strategy is implicitly assigned and is intended for internal use by Aglyph itself, it is not exposed on the `Strategy` named tuple.

An already-created (loaded) object is obtained from an imported module or class (as opposed to creating the object directly). Such components will always resolve (i.e. be assembled) to the same objects; but those objects are not cached by Aglyph as they will exhibit "natural" singleton behavior so long as the containing module is referenced in `sys.modules`.

It is not necessary to explicitly set the strategy to "_imported" when using *member_name* - Aglyph will default to "_imported" when it sees a non-empty *member_name* defined.

Warning: Explicitly setting strategy="_imported" **without** specifying *member_name* will raise `AglyphError`.

Specifying *member_name* with any explicit strategy other than "_imported" will ignore the explicit strategy, change it to "_imported" internally, and issue a `UserWarning` to that effect.

`aglyph.component.`**`LifecycleState = LifecycleState(AFTER_INJECT='after_inject', BEFORE_CLEAR=`**

Define the lifecycle states for which Aglyph will call object methods on your behalf.

Lifecycle methods

Lifecycle methods are called with **no arguments** (positional or keyword).

If a called lifecycle method raises an exception, the exception is caught, logged at `logging.ERROR` level (including a traceback) to the "aglyph.assembler.Assembler" channel, and a `RuntimeWarning` is issued.

A method may be registered for a lifecycle state by specifying the method name at the context (least specific), template, and/or component (most specific) level.

Note: Aglyph only calls **one** method on an object for any lifecycle state. Refer to **The lifecycle method lookup process** (below) for details.

Aglyph recognizes the following lifecycle states:

"after_inject"

A component object is in this state after **all** dependencies (both initialization arguments and attributes) have been injected into a newly-created instance, but before the object is cached and/or returned to the caller.

Aglyph will only call **one** "after_inject" method on any object, and will determine which method to call by using the lookup process described below.

"before_clear"

A component object is in this state after is has been removed from an internal cache (singleton, borg, or weakref), but before the object itself is actually discarded.

Aglyph will only call **one** "before_clear" method on any object, and will determine which method to call by using the lookup process described below.

The lifecycle method lookup process

Lifecyle methods may be specified at the context (least specific), template, and component (most specific) levels.

In order to determine which named method is called for a particular object, Aglyph looks up the appropriate lifecycle method name in the following order, using the **first** one found that is not `None` *and* is actually defined on the object:

1. The method named by the object's `Component.<lifecycle-state>` property.
2. If the object's `Component.parent_id` is not `None`, the method named by the corresponding parent `Template.<lifecycle-state>` or `Component.<lifecycle-state>` property. (If necessary, lookup continues by examining the parent-of-the-parent and so on.)
3. The method named by the `Context.<lifecycle-state>` property.

When Aglyph finds a named lifecycle method that applies to an object, but the object itself does not define that method, a `logging.WARNING` message is emitted.

Note: Either a `Component` or `Template` may serve as the parent identified by a `parent_id`.

However, only a `Component` may actually be assembled into a usable object. (A `Template` is like an abstract class - it defines common dependencies and/or lifecycle methods, but it cannot be assembled.)

class `aglyph.component.`**`Reference`**

Bases: `unicode`

A placeholder used to refer to another `Component`.

A `Reference` is used as an alias to identify a component that is a dependency of another component. The value of a `Reference` can be either a dotted-name or a user-provided unique ID.

A `Reference` can be used as an argument for an `Evaluator`, and can be assembled directly by an `aglyph.assembler.Assembler`.

Warning: A `Reference` value MUST correspond to a component ID in the same context.

Note: In Python versions < 3.0, a `Reference` representing a dotted-name *must* consist only of characters in the ASCII subset of the source encoding (see **PEP 0263**).

But in Python versions >= 3.0, a `Reference` representing a dotted-name *may* contain non-ASCII characters (see **PEP 3131**).

However, a `Reference` may also represent a user-defined identifier. To accommodate all cases, the super class of `Reference` is "dynamic" with respect to the version of Python under which Aglyph is running (`unicode` under Python 2, `str` under Python 3). This documentation shows the base class as `str` because the Sphinx documentation generator for Aglyph runs under CPython 3.

Create a new reference to *referent*.

Parameters **`referent`** – the object that the reference will represent

Raises ***`aglyph.AglyphError`*** – if *referent* is a class, function, or module but cannot be imported

If *referent* is a string, it is assumed to be a valid `Component.unique_id` and its value is returned as a `Reference`.

If *referent* is a class, function, or module, its **importable** dotted name is returned as a `Reference`.

> **Warning:** If *referent* is a class, function, or module, it **must** be importable.

class `aglyph.component.`**`Evaluator`**(*factory*, **args*, ***keywords*)
Bases: `aglyph.component._InitializationSupport`

Perform lazy creation of objects.

Parameters

- **`factory`** – any callable that returns an object
- **`args`** (*`tuple`*) – the positional arguments to *func*
- **`keywords`** (*`dict`*) – the keyword arguments to *func*

An `Evaluator` is similar to a `functools.partial()` in that they both collect a function and related arguments into a `callable` object with a simplified signature that can be called repeatedly to produce a new object.

Unlike a partial function, an `Evaluator` may have arguments that are not truly "frozen," in the sense that any argument may be defined as a *`Reference`*, a `functools.partial()`, or even another `Evaluator`, which needs to be resolved (i.e. assembled/called) before calling *factory*.

When an `Evaluator` is called, its arguments (positional and keyword) are each resolved in one of the following ways:

- If the argument value is a *`Reference`*, it is assembled (by an *`aglyph.assembler.Assembler`* reference passed to *`__call__()`*)
- If the argument value is an `Evaluator` or a `functools.partial()`, it is called to produce its value.
- If the argument is a dictionary or a sequence other than a string type, each item is resolved according to these rules.
- If none of the above cases apply, the argument value is used as-is.

`factory`
The `callable` that creates new objects *(read-only)*.

`__call__`(*assembler*)
Call `factory(*args, **keywords)` and return the new object.

Parameters **`assembler`** (*`aglyph.assembly.Assembler`*) – the assembler that will be used to assemble any *`Reference`* encountered in this evaluator's positional and keyword arguments

class `aglyph.component.`**`Template`**(*unique_id*, *parent_id=None*, *after_inject=None*, *before_clear=None*)
Bases: `aglyph.component._DependencySupport`

Support for configuring type 1 (setter) and type 2 (constructor) injection, and lifecycle methods.

Parameters

- **`unique_id`** (*`str`*) – context-unique identifier for this template
- **`parent_id`** (*`str`*) – specifies the ID of a template or component that describes the default dependencies and/or lifecyle methods for this template

- **`after_inject`** (*`str`*) – specifies the name of the method that will be called on objects of components that reference this template after all component dependencies have been injected
- **`before_clear`** (*`str`*) – specifies the name of the method that will be called on objects of components that reference this template immediately before they are cleared from cache

Raises **`ValueError`** – if *unique_id* is `None` or empty

Note: A `Template` cannot be assembled (it is equivalent to an abstract class).

However, a *`Component`* can also serve as a template, so if you need the ability to assemble an object *and* use its definition as the basis for other components, then define the default dependencies and/or lifecycle methods in a *`Component`* and use that component's ID as the `Component.parent_id` in other components.

unique_id must be a user-provided identifier that is unique within the context to which this template is added. A component may then be instructed to use a template by specifying the same value for `Component.parent_id`.

parent_id is **another** `Component.unique_id` or *`Template.unique_id`* in the same context that descibes **this** template's default dependencies and/or lifecycle methods.

after_inject is the name of a method *of objects of this component* that will be called after **all** dependencies have been injected, but before the object is returned to the caller. This method will be called with **no** arguments (positional or keyword). Exceptions raised by this method are not caught.

Note: `Template.after_inject` takes precedence over any *after_inject* method name specified for the template's parent or context.

before_clear is the name of a method *of objects of this component* that will be called immediately before the object is cleared from cache via *`aglyph.assembler.Assembler.clear_singletons()`*, *`aglyph.assembler.Assembler.clear_borgs()`*, or *`aglyph.assembler.Assembler.clear_weakrefs()`*.

Note: `Template.before_clear` takes precedence over any *before_clear* method name specified for the template's parent or context.

Warning: The *before_clear* keyword argument has no meaning for and is ignored by "prototype" components. If *before_clear* is specified for a prototype, a `RuntimeWarning` will be issued.

For "weakref" components, there is a possibility that the object no longer exists at the moment when the *before_clear* method would be called. In such cases, the *before_clear* method is **not** called. No warning is issued, but a `logging.WARNING` message is emitted.

`unique_id`
Uniquely identifies this template in a context *(read-only)*.

`parent_id`
Identifies this template's parent template or component *(read-only)*.

`after_inject`
The name of the component object method that will be called after **all** dependencies have been injected *(read-only)*.

`before_clear`

The name of the component object method that will be called immediately before the object is cleared from cache *(read-only)*.

> **Warning:** This property is not applicable to "prototype" component objects, and is **not guaranteed** to be called for "weakref" component objects.

class `aglyph.component.`**`Component`**(*component_id*, *dotted_name=None*, *factory_name=None*, *member_name=None*, *strategy=None*, *parent_id=None*, *after_inject=None*, *before_clear=None*)

Bases: `aglyph.component.Template`

Define a component and the dependencies needed to create a new object of that component at runtime.

Parameters

- **`component_id`** (`str`) – the context-unique identifier for this component
- **`dotted_name`** (`str`) – an **importable** dotted name
- **`factory_name`** (`str`) – names a `callable` member of objects of this component
- **`member_name`** (`str`) – names **any** member of objects of this component
- **`strategy`** (`str`) – specifies the component assembly strategy
- **`parent_id`** (`str`) – specifies the ID of a template or component that describes the default dependencies and/or lifecyle methods for this component
- **`after_inject`** (`str`) – specifies the name of the method that will be called on objects of this component after all of its dependencies have been injected
- **`before_clear`** (`str`) – specifies the name of the method that will be called on objects of this component immediately before they are cleared from cache

Raises

- **`aglyph.AglyphError`** – if both *factory_name* and *member_name* are specified
- **`ValueError`** – if *strategy* is not a recognized assembly strategy

component_id must be a user-provided identifier that is unique within the context to which this component is added. An **importable** dotted name may be used (see `aglyph.resolve_dotted_name()`).

dotted_name, if provided, must be an **importable** dotted name (see `aglyph.resolve_dotted_name()`).

Note: If *dotted_name* is not specified, then *component_id* is used as the component's dotted name and **must** be an importable dotted name.

factory_name is the name of a `callable` member of *dotted-name* (i.e. a function, class, staticmethod, or classmethod). When provided, the assembler will call this member to create an object of this component.

factory_name enables Aglyph to inject dependencies into objects that can only be initialized via nested classes, `staticmethod`, or `classmethod`. See *`factory_name`* for details.

member_name is the name of a member of *dotted-name*, which **may or may not** be callable.

member_name differs from *factory_name* in two ways:

1. *member_name* is not restricted to callable members; it may identify **any** member (attribute, property, nested class).

2. When an assembler assembles a component with a *member_name*, initialization of the object is *bypassed* (i.e. the assembler will not call the member, and any initialization arguments defined for the component will be **ignored**).

member_name enables Aglyph to reference class, function, `staticmethod`, and `classmethod` obejcts, as well as simple attributes or properties, as components and dependencies. See *member_name* for details.

Note: Both *factory_name* and *member_name* can be dot-separated names to reference nested members.

Warning: The *factory_name* and *member_name* arguments are mutually exclusive. An exception is raised if both are provided.

strategy must be a recognized component assembly strategy, and defaults to `Strategy.PROTOTYPE` (*"prototype"*) if not specified.

New in version 3.0.0: When *member_name* is specified, the strategy **must** be *"_imported"*. Aglyph will use the "_imported" strategy automatically for components that specify *member_name*; setting strategy to anything other than "_imported" when specifying *member_name* will issue `UserWarning`.

Please see *Strategy* for a description of the component assembly strategies supported by Aglyph.

Warning: The `Strategy.BORG` (*"borg"*) component assembly strategy is only supported for classes that **do not** define or inherit `__slots__`!

parent_id is the context-unique ID of a *Template* (or another `Component`) that defines default dependencies and/or lifecycle methods for this component.

after_inject is the name of a method *of objects of this component* that will be called after **all** dependencies have been injected, but before the object is returned to the caller. This method will be called with **no** arguments (positional or keyword). Exceptions raised by this method are not caught.

Note: `Component.after_inject` takes precedence over any *after_inject* method names specified for the component's parent or context.

before_clear is the name of a method *of objects of this component* that will be called immediately before the object is cleared from cache via *aglyph.assembler.Assembler.clear_singletons()*, *aglyph.assembler.Assembler.clear_borgs()*, or *aglyph.assembler.Assembler.clear_weakrefs()*.

Note: `Component.before_clear` takes precedence over any *before_clear* method names specified for the component's parent or context.

Warning: The *before_clear* keyword argument has no meaning for, and is ignored by, "prototype" components. If *before_clear* is specified for a prototype component, a `UserWarning` is issued **when the component is defined**, and the component's `before_clear` attribute is set to `None`.

> **Warning:** For "weakref" components, there is a possibility that the object no longer exists at the moment when the *before_clear* method would be called. In such cases, the `aglyph.assembler.clear_weakrefs()` method will issue a `RuntimeWarning` (see that method's documentation for more details).

Once a `Component` instance is initialized, the `args` (`list`), `keywords` (`dict`), and `attributes` (`collections.OrderedDict`) members can be modified in-place to define the dependencies that must be injected into objects of this component at assembly time. For example:

```
component = Component("http.client.HTTPConnection")
component.args.append("ninthtest.info")
component.args.append(80)
component.keywords["strict"] = True
component.attributes["set_debuglevel"] = 1
```

In Aglyph, a component may:

- be assembled directly by an *`aglyph.assembler.Assembler`*
- identify other components as dependencies (using a *`Reference`*)
- be used by other components as a dependency
- use common dependencies and behaviors (*after_inject*, *before_clear*) defined in a *`aglyph.component.Template`*
- use any combination of the above behaviors

`dotted_name`
The importable dotted name for objects of this component *(read-only)*.

`factory_name`
The name of a `callable` member of *`dotted_name`* *(read-only)*.

`factory_name` can be used to initialize objects of the component when a class is not directly importable (e.g. the component class is a nested class), or when component objects need to be initialized via `staticmethod` or `classmethod`.

Consider the following:

```
# module.py
class Example:
    class Nested:
        pass
```

The dotted name "module.Example.Nested" is not importable, and so cannot be used as a component's `unique_id` or `dotted_name`. To assemble objects of this type, use `factory_name` to identify the callable factory (the Nested class, in this example) that is accessible through the importable "module.Example":

```
component = Component(
    "nested-object", dotted_name="module.Example",
    factory_name="Nested")
```

Or using XML configuration:

```
<component id="nested-object" dotted-name="module.Example"
    factory-name="Nested" />
```

factory_name may also be a dot-separated name to specify an arbitrarily-nested callable. The following example is equivalent to the above:

```
component = Component(
    "nested-object", dotted_name="module",
    factory_name="Example.Nested")
```

Or again using XML configuration:

```
<component id="nested-object" dotted-name="module"
    factory-name="Example.Nested" />
```

Note: The important thing to remember is that *dotted_name* must be **importable**, and factory_name must be accessible from the imported class or module via attribute access.

member_name

The name of any member of *dotted_name* (*read-only*).

member_name can be used to obtain an object *directly* from an importable module or class. The named member is simply accessed and returned (it is **not** called, even if it is callable).

Consider the following:

```
# module.py
class Example:
    class Nested:
        pass
```

The following example shows how to define a component that will produce the module.Example.Nested class *itself* when assembled:

```
component = Component(
    "nested-class", dotted_name="module.Example",
    member_name="Nested")
```

Or using XML configuration:

```
<component id="nested-class" dotted-name="module.Example"
    member-name="Nested" />
```

member_name may also be a dot-separated name to specify an arbitrarily-nested member. The following example is equivalent to the above:

```
component = Component(
    "nested-class", dotted_name="module",
    member_name="Example.Nested")
```

Or again using XML configuration:

```
<component id="nested-class" dotted-name="module"
    member-name="Example.Nested" />
```

Note: The important thing to remember is that *dotted_name* must be **importable**, and member_name must be accessible from the imported class or module via attribute access.

> **Warning:** When a component specifies `member_name`, initialization is assumed. In other words, Aglyph **will not** attempt to initialize the member, and will **ignore** any `args` and `keywords`.
>
> On assembly, if any initialization arguments and/or keyword arguments have been defined for such a component, they are discarded and a WARNING-level log record is emitted to the "aglyph.assembler.Assembler" channel.
>
> (Any `attributes` that have been specified for the component will still be processed as setter injection dependencies, however.)

`strategy`
The component assembly strategy *(read-only)*.

1.4.4 `aglyph.context` — Defining component contexts

Release 3.0.0.post1

The classes in this module are used to define collections ("contexts") of related components and templates.

A context can be created in pure Python using the following API classes:

- *`aglyph.component.Template`*
- *`aglyph.component.Component`*
- *`aglyph.component.Reference`* (used to indicate that one component depends on another component)
- *`aglyph.component.Evaluator`* (used like a partial function to lazily evaluate component initialization arguments and attributes)
- *`aglyph.context.Context`* or a subclass

New in version 3.0.0: For easier programmatic configuration, refer to *The Aglyph Context fluent API*.

Alternatively, a context can be defined using a declarative XML syntax that conforms to the `Aglyph context DTD` (included in the *resources/* directory of the distribution). This approach requires only the *`aglyph.context.XMLContext`* class, which parses the XML document and then uses the API classes mentioned above to populate the context.

class `aglyph.context.`**`Context`**(*context_id*, *after_inject=None*, *before_clear=None*)
Bases: `dict`, `aglyph.context._ContextBuilder`

A mapping of unique IDs to `Component` and `Template` objects.

Parameters

- **`context_id`** (*`str`*) – an identifier for this context
- **`after_inject`** (*`str`*) – specifies the name of the method that will be called (if it exists) on **all** component objects after all of their dependencies have been injected
- **`before_clear`** (*`str`*) – specifies the name of the method that will be called (if it exists) on **all** singleton, borg, and weakref objects immediately before they are cleared from cache

`register`(*definition*)
Add a component or template *definition* to this context.

Parameters **`definition`** – a `Component` or `Template` object

Raises ***`AglyphError`*** – if a component or template with the same unique ID is already registered in this context

Note: To **replace** an already-registered component or template with the same unique ID, use `dict.__setitem__()` directly.

`get_component`(*component_id*)
Return the `Component` identified by *component_id*.

Parameters **`component_id`** (`str`) – a unique ID that identifies a `Component`

Returns the `Component` identified by *component_id*

Return type `Component` if *component_id* is mapped, else `None`

`iter_components`(*strategy=None*)
Yield all definitions in this context that are instances of `Component`, optionally filtered by *strategy*.

Parameters **`strategy`** (`str`) – only yield component definitions that use this assembly strategy (by default, **all** component definitions are yielded)

Returns a `Component` generator

`borg`(*component_id_spec*, *parent=None*)
Return a `borg` `Component` builder for a component identified by *component_id_spec*.

Parameters

- **`component_id_spec`** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **`parent`** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`component`(*component_id_spec*, *parent=None*)
Return a fluent `Component` builder for *component_id_spec*.

Parameters

- **`component_id_spec`** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **`parent`** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`prototype`(*component_id_spec*, *parent=None*)
Return a `prototype` `Component` builder for a component identified by *component_id_spec*.

Parameters

- **`component_id_spec`** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **`parent`** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

singleton(*component_id_spec*, *parent=None*)

Return a *singleton* Component builder for a component identified by *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

template(*template_id_spec*, *parent=None*)

Return a `Template` builder for a template identified by *template_spec*.

Parameters

- **template_id_spec** – a context-unique identifier for this template; or the object whose dotted name will identify this template
- **parent** – the context-unique identifier for this template's parent template or component definition; or the object whose dotted name identifies this template's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

weakref(*component_id_spec*, *parent=None*)

Return a *weakref* Component builder for a component identified by *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

class `aglyph.context.`**XMLContext**(*source*, *parser=None*, *default_encoding='ascii'*)

Bases: *aglyph.context.Context*

A mapping of unique IDs to `Component` and `Template` objects.

Components and templates are declared in an XML document that conforms to the `Aglyph context DTD` (included in the *resources/* directory of the distribution).

Parameters

- **source** – a filename or stream from which XML data is read
- **parser** (*xml.etree.ElementTree.XMLParser*) – the ElementTree parser to use (instead of Aglyph's default)
- **default_encoding** (*str*) – the default character set used to encode certain element content

Raises ***AglyphError*** – if unexpected elements are encountered, or if expected elements are *not* encountered, in the document structure

In most cases, *parser* should be left unspecified. Aglyph's default parser will be sufficient for all but extreme edge cases.

default_encoding is the character set used to encode `<bytes>` (or `<str>` under Python 2) element content when an **@encoding** attribute is *not* specified on those elements. It defaults to the system-dependent value of `sys.getdefaultencoding()`. **This is not related to the document encoding!**

Note: Aglyph uses a non-validating XML parser by default, so DTD conformance is **not** enforced at runtime. It is recommended that XML contexts be validated at least once (manually) during testing.

An `AglyphError` *will* be raised under certain conditions (an unexpected element is encounted, or an expected element is *not* encountered), but Aglyph does not "reinvent the wheel" by implementing strict validation in the parsing logic.

Warning: Although Aglyph contexts are `dict` types, `XMLContext` does not permit the same unique ID to be (re-)mapped multiple times.

Attempting to define more than one `<component>` or `<template>` with the same ID will raise `AglyphError` when the document is parsed.

After an Aglyph `<context>` document has been successfully parsed, a unique component or template ID can be re-mapped using standard `dict` protocols.

See also:

Validity constraint: ID https://www.w3.org/TR/REC-xml/#id

`default_encoding`

The default encoding of `<bytes>` (or `<str>` under Python 2) element content when an **@encoding** attribute is *not* specified.

Note: This is **unrelated** to the document encoding!

1.4.5 `aglyph.integration.cherrypy` — Integrating Aglyph with CherryPy

Release 3.0.0.post1

1.5 The Aglyph Context fluent API

Release 3.0.0.post1

1.5.1 Introduction to the Context fluent API

The easiest way to configure Aglyph programmatically is to use the "fluent" API exposed by `aglyph.context.Context`.

The Context fluent API consists of a set of chained-call-style methods (exposed on `Context` or its subclasses) that can be used to define components of any kind (prototype, singleton, borg, weakref) as well as templates.

There are six (6) "entry points" into the Context fluent API:

1. `aglyph.context.Context.prototype()` to define a prototype component
2. `aglyph.context.Context.singleton()` to define a singleton component
3. `aglyph.context.Context.borg()` to define a borg component
4. `aglyph.context.Context.weakref()` to define a weakref component
5. `aglyph.context.Context.template()` to define a template
6. `aglyph.context.Context.component()` to define any kind of component *(more on this method later)*

Each entry point returns a "builder" which can be used to further define the component or template.

For the sake of demonstration, assume we begin with the following initialization code:

```
from aglyph.assembler import Assembler
from aglyph.component import Reference as ref
from aglyph.context import Context

context = Context("example-context")
```

Let's define a singleton component:

```
context.singleton("my-singleton")
```

Here we have provided a unique ID for the component ("my-singleton"). Since this is not a dotted name, though, Aglyph doesn't know which class to use to create the object. We will provide the class name with a chained call to the `create()` method:

```
context.singleton("my-singleton").create("module.ClassName")
```

We could also have used the dotted name as the unique ID, in which case calling `create()` would not be necessary.

Now we wish to define the initialization arguments and keywords. We add another chained call to the `init()` method to do so. We can also break the chained call sequence onto multiple lines for readability:

```
(context.singleton("my-singleton").
    create("module.ClassName").
    init("argument", keyword="keyword"))
```

Finally, we terminate the fluent sequence by calling the `register()` method, which actually creates an `aglyph.component.Component` and adds it to the context:

```
(context.singleton("my-singleton").
    create("module.ClassName").
    init("argument", keyword="keyword").
    register())
```

Note: Terminating the fluent builder with a call to `register()` is crucial. The component definition is not actually created or stored in the context unless/until this method is called!

Now this context can be given to an assembler, and we can create an object:

```
assembler = Assembler(context)
my_singleton = assembler.assemble("my-singleton")
```

You can find many other examples of using the Context fluent API in the *Aglyph cookbook*.

A note regarding the `component()` entry point method

The strategy-specific entry point methods (`prototype()`, `singleton()`, `borg()` and `weakref()`) are implemented in terms of `component()`.

For example, calling `singleton("package.ClassName")` is the shorthand equivalent of calling `component("package.ClassName").create(strategy="singleton")`.

If `component("my-id")` is used to define a component and no explicit strategy is specified, then the default strategy (“prototype”) is assumed.

Warning: There is one “special case” that warrants explanation:

When specifying a *member* using the fluent API (i.e. objects of the component are “created” by attribute access on the object identified by a dotted name), then the creation strategy is implicitly set to “_imported” and SHOULD NOT be set explicitly. In short - consider *member* and *strategy* to be mutually exclusive.

1.5.2 Using component references with the Context fluent API

Let’s add another component to the context. This new component needs to be injected with an object of “my-singleton” via a property. We will use an *`aglyph.component.Reference`* to provide the dependency:

```
(context.prototype("example-object").
    create("module2.ExampleObject").
    set(thing=ref("my-singleton")).
    register())
```

Assembling an object of “example-object” will resolve “my-singleton” to the same object we assembled earlier, and will set that object as the `module2.ExampleObject.thing` property.

1.5.3 Using the Context fluent API to define templates

Defining templates via the Context fluent API is identical to defining components, with the exception that there is no `create()` method for template builders. Remember, templates cannot be assembled - they serve only as a basis for further defining other templates or components.

Here is an example:

```
context.template("my-template").init("arg", keyword="kw").register()
(context.singleton("my-singleton", parent="my-template").
    create("module.ClassName").
    register())
```

Notice that we have added `parent="my-template"` to the singleton method call.

Now when we assemble "my-singleton" its initializer will be called with the positional argument "arg" and the keyword argument keyword="kw".

1.5.4 Letting Aglyph introspect dotted names

If preferred, classes (or other callables) that will be defined as components or templates can be introspected by Aglyph. Let's revisit the earlier example, slightly modified, to demonstrate this:

```
from module import ClassName
from module2 import ExampleObject

from aglyph.assembler import Assembler
from aglyph.component import Reference as ref
from aglyph.context import Context

context = Context("example-context")

(context.singleton(ClassName).
    init("argument", keyword="keyword").
    register())

(context.prototype("example-object").
    create(ExampleObject).
    set(thing=ref(ClassName)).
    register())

assembler = Assembler(context)
example = assembler.assemble("example-object")
```

Notice that we call `singleton(ClassName)` as the entry point. Aglyph will automatically convert ClassName into its dotted name "module.ClassName" and also use that value as the unique ID. This means that the expicit call to `create()` is now unnecessary, and so it has been removed.

Next, notice the call to `create(ExampleObject)`. We used "example-object" as the component ID, so we must still tell Aglyph the dotted name of the class. But instead of passing the dotted name as a string, we again pass the class and let Aglyph determine the value.

Finally, notice that we use `ref(ClassName)` when setting the *thing* property. Like the fluent API entry point methods and `create()`, *`aglyph.component.Reference`* is also capable of introspecting a dotted name.

Warning: Callable object dotted-name introspection for **nested** callables (e.g. nested classes) does not work in Python versions prior to 3.3 because the introspection depends on the *Qualified name for classes and functions* (i.e. `__qualname__`, specified in **PEP 3155**).

1.5.5 Overview of the Context fluent API methods

The Context fluent API is made up of a number of "mixin" classes that are combined in different ways to support describing templates and components.

These classes are never instantiated directly; rather, the "entry point" methods are exposed as members of *`aglyph.context.Context`* that return either a `aglyph.context._ComponentBuilder` or a `aglyph.context._TemplateBuilder`; and those builder classes inherit relevant methods from the mixin classes.

Note: For reference, here is the class hierarchy for the Context fluent API (all classes are defined in the `aglyph.context` namespace):

class Context (*_ContextBuilder*)

class _TemplateBuilder (*_InjectionBuilderMixin*, *_LifecycleBuilderMixin*, *_RegistrationMixin*)

class _ComponentBuilder (*_CreationBuilderMixin*, *_TemplateBuilder*)

"Entry point" methods to create component and template builders

`_ContextBuilder.`**component**(*component_id_spec*, *parent=None*)

Return a fluent `Component` builder for *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`_ContextBuilder.`**prototype**(*component_id_spec*, *parent=None*)

Return a *`prototype`* `Component` builder for a component identified by *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`_ContextBuilder.`**singleton**(*component_id_spec*, *parent=None*)

Return a *`singleton`* `Component` builder for a component identified by *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`_ContextBuilder.`**borg**(*component_id_spec*, *parent=None*)

Return a *`borg`* `Component` builder for a component identified by *component_id_spec*.

Parameters

- **component_id_spec** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **parent** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`_ContextBuilder.`**`weakref`**(*component_id_spec*, *parent=None*)

Return a *`weakref`* `Component` builder for a component identified by *component_id_spec*.

Parameters

- **`component_id_spec`** – a context-unique identifier for this component; or the object whose dotted name will identify this component
- **`parent`** – the context-unique identifier for this component's parent template or component definition; or the object whose dotted name identifies this component's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

`_ContextBuilder.`**`template`**(*template_id_spec*, *parent=None*)

Return a `Template` builder for a template identified by *template_spec*.

Parameters

- **`template_id_spec`** – a context-unique identifier for this template; or the object whose dotted name will identify this template
- **`parent`** – the context-unique identifier for this template's parent template or component definition; or the object whose dotted name identifies this template's parent definition

New in version 3.0.0: This method is an entry point into *The Aglyph Context fluent API*.

Describing object creation for component builders

`_CreationBuilderMixin.`**`create`**(*dotted_name=None*, *factory=None*, *member=None*, *strategy=None*)

Specify the object creation aspects of a component being defined.

Parameters

- **`dotted_name`** – an **importable** dotted name or an object whose dotted name will be introspected
- **`factory`** – names a `callable` member of the object represented by the dotted name
- **`member`** – names **any** member of the object represented by the dotted name
- **`strategy`** – specifies the component assembly strategy

Returns *self* (to support chained calls)

Any keyword whose value is `None` will be ignored (i.e. `None` values are not explicitly set).

Note: The *member* and *strategy* keywords should be treated as mutually exclusive. Any component definition that specifies a *member* is implicitly assigned the special strategy "_imported".

Describing dependencies and lifecycle methods for template and component builders

`_InjectionBuilderMixin.`**`init`**(**args*, ***keywords*)

Specify the initialization arguments (positional and keyword) for templates and/or components.

Parameters

- **`args`** (*`tuple`*) – the positional initialization arguments
- **`keywords`** (*`dict`*) – the keyword initialization arguments

Returns *self* (to support chained calls)

Note: Successive calls to this method on the same instance have a cumulative effect; the list of positional arguments is extended, and the dictionary of keyword arguments is updated.

`_InjectionBuilderMixin.`**`set`**`(`**pairs*, ***attributes*`)`

Specify the setter (method/attribute/property) depdendencies for tempaltes and/or components.

Parameters

- **`pairs`** – a sequence of (name, value) 2-tuples (optional)
- **`attributes`** – a mapping of name->value dependencies

Returns *self* (to support chained calls)

Note: Successive calls to this method on the same instance have a cumulative effect (i.e. the attributes mapping is updated).

`_LifecycleBuilderMixin.`**`call`**`(`*after_inject=None*, *before_clear=None*`)`

Specify the names of lifecycle methods to be called for templates and/or components.

Parameters

- **`after_inject`** – the name of the method to call after a component has been assembled but before it is returned to the caller
- **`before_clear`** – the name of the method to call immediately before a *singleton*, *borg*, or *weakref* object is evicted from the internal cache

Returns *self* (to support chained calls)

Any keyword whose value is `None` will be ignored (i.e. `None` values are not explicitly set).

Registering the template or component described by a fluent builder

`_RegistrationMixin.`**`register`**`()`

Add a component or template definition to a context.

Returns `None` (terminates the fluent call sequence)

1.6 Aglyph 3.0.0.post1 testing summary

Note: Changed in version 3.0.0.post1.

The testing results below **supercede** *Aglyph 3.0.0 testing summary*.

Note: Prior to each Aglyph release being committed, tagged, and packaged, its test suite is executed on several Python implementations, versions, and platforms.

When a Python *version* reaches end-of-life, the Aglyph test suite is no longer "officially" executed against that version. This does not necessarily imply that Aglyph unit tests *fail* under that Python version; it simply means that I no longer publish or withhold a release based on the test suite results from that particular Python version.

1.6.1 CPython

Version	Platform
3.7.0	[GCC 8.1.1 20180502 (Red Hat 8.1.1-1)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[MSC v.1914 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
	[Clang 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
3.6.6	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
3.5.6	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
3.5.4	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
3.4.9	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
3.4.4	[MSC v.1600 64 bit (AMD64)] on Windows-10-10.0.17134
2.7.15	[GCC 8.1.1 20180502 (Red Hat 8.1.1-1)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[MSC v.1500 64 bit (AMD64)] on Windows-10-10.0.17134
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.1)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)

1.6.2 PyPy

Version	Platform
3.5.3	[PyPy 6.0.0 with GCC 7.2.0] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[PyPy 5.10.1 with MSC v.1500 32 bit] on Windows-8-6.2.9200-SP0
	[PyPy 6.0.0 with GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
2.7.13	[PyPy 6.0.0 with GCC 7.2.0] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	[PyPy 6.0.0 with MSC v.1500 32 bit] on Windows-10-10.0.17134
	[PyPy 6.0.0 with GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)

1.6.3 Stackless Python

Version	Platform
3.6.4	Stackless 3.1b3 060516 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	Stackless 3.1b3 060516 (v3.6.4-slp:9557b2e530, Dec 21 2017, 15:23:10) [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
	Stackless 3.1b3 060516 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)
3.5.4	Stackless 3.1b3 060516 (v3.5.4-slp-1-g40e5316803:40e5316803, Oct 1 2017, 23:23:05) [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
2.7.14	Stackless 3.1b3 060516 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux-4.17.18-200.fc28.x86_64-x86_64-with-fedora-28-Twenty_Eight
	Stackless 3.1b3 060516 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.7.0-x86_64-i386-64bit (Mac OS X 10.13.6)

1.6.4 Jython

Version	Platform
2.7.1	[OpenJDK 64-Bit Server VM (Oracle Corporation)] on Java-9.0.4-OpenJDK_64-Bit_Server_VM,_9.0.4+11,_Oracle_Corporation-on-Linux-4.17.18-200.fc28.x86_64-amd64
	[Java HotSpot(TM) 64-Bit Server VM (“Oracle Corporation”)] on Java-10.0.2-Java_HotSpot-TM-_64-Bit_Server_VM,_10.0.2+13,_-Oracle_Corporation-on-Windows_10-10.0-amd64
	[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on Java-9.0.4-Java_HotSpot-TM-_64-Bit_Server_VM,_9.0.4+11,_Oracle_Corporation-on-Mac_OS_X-10.13.6-x86_64

1.6.5 IronPython

Version	Platform
2.7.8	(IronPython 2.7.8 (2.7.8.0) on .NET 4.0.30319.42000 (64-bit)) on Windows-10-10.0.17134
	(IronPython 2.7.8 (2.7.8.0) on .NET 4.0.30319.42000 (32-bit)) on Windows-10-10.0.17134

1.6.6 Previous releases of Aglyph

Aglyph 3.0.0 testing summary

Warning: Changed in version 3.0.0.post1.

The testing results below are **superceded by** *Aglyph 3.0.0.post1 testing summary*.

Warning: Changed in version 3.0.0.

The Python 3.3 lifespan has ended; the Aglyph test suite is no longer executed under this version.

Note: Prior to each Aglyph release being committed, tagged, and packaged, its test suite is executed on several Python implementations, versions, and platforms.

When a Python *version* reaches end-of-life, the Aglyph test suite is no longer "officially" executed against that version. This does not necessarily imply that Aglyph unit tests *fail* under that Python version; it simply means that I no longer publish or withhold a release based on the test suite results from that particular Python version.

CPython

Version	Platform
3.7.0b2	[GCC 7.3.1 20180130 (Red Hat 7.3.1-2)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	[Clang 6.0 (clang-600.0.57)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS 10.13.3)
	[MSC v.1912 64 bit (AMD64)] on Windows-10-10.0.16299-SP0
3.6.4	[GCC 7.3.1 20180130 (Red Hat 7.3.1-2)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.38)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.16299-SP0
3.5.5	[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.16299-SP0
3.5.4	[GCC 7.2.1 20170915 (Red Hat 7.2.1-2)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
3.4.8	[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
3.4.7	[GCC 7.2.1 20170915 (Red Hat 7.2.1-2)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
3.4.4	[MSC v.1600 64 bit (AMD64)] on Windows-10-10.0.16299
2.7.14	[GCC 7.3.1 20180130 (Red Hat 7.3.1-2)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	[MSC v.1500 64 bit (AMD64)] on Windows-10-10.0.16299
	[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.37)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)

PyPy

Version	Platform
3.5.3	[PyPy 5.10.1 with GCC 7.2.0] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	[PyPy 5.10.1 with MSC v.1500 32 bit] on Windows-8-6.2.9200-SP0
2.7.13	[PyPy 5.10.0 with GCC 7.2.0] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	[PyPy 5.10.0 with GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
	[PyPy 5.9.0 with MSC v.1500 32 bit] on Windows-10-10.0.16299

Stackless Python

Version	Platform
3.6.4	Stackless 3.1b3 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	Stackless 3.1b3 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
	Stackless 3.1b3 [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.16299-SP0
3.5.4	Stackless 3.1b3 [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.16299-SP0
2.7.14	Stackless 3.1b3 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux-4.15.8-300.fc27.x86_64-x86_64-with-fedora-27-Twenty_Seven
	Stackless 3.1b3 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin-17.4.0-x86_64-i386-64bit (Mac OS X 10.13.3)
	Stackless 3.1b3 [MSC v.1500 64 bit (AMD64)] on Windows-10-10.0.16299

Jython

Version	Platform
2.7.1	[OpenJDK 64-Bit Server VM (Oracle Corporation)] on Java-9.0.4-OpenJDK_64-Bit_Server_VM,_9.0.4+11,_Oracle_Corporation-on-Linux-4.15.8-300.fc27.x86_64-amd64
	[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java9.0.4 (==darwin for targets) (Mac OS X 10.13.3)
	[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.8.0_162 (==darwin for targets) (Mac OS X 10.13.3)
	[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on Java-9.0.4-Java_HotSpot-TM-_64-Bit_Server_VM,_9.0.4+11,_Oracle_Corporation-on-Windows_10-10.0-amd64
2.7.0	[OpenJDK 64-Bit Server VM (Oracle Corporation)] on Java-1.8.0_161-OpenJDK_64-Bit_Server_VM,_25.161-b14,_Oracle_Corporation-on-Linux-4.15.8-300.fc27.x86_64-amd64

IronPython

Version	Platform
2.7.7	cli-10-AMD64-Intel64_Family_6_Model_94_Stepping_3,_GenuineIntel-64bit [.NET 4.0.30319.42000 (64-bit)] on Windows 10 Home v1709 build 16299.251
	cli-10-AMD64-Intel64_Family_6_Model_94_Stepping_3,_GenuineIntel-32bit [.NET 4.0.30319.42000 (32-bit)] on Windows 10 Home v1709 build 16299.251

Aglyph 2.1.1 testing summary

Warning: Changed in version 2.1.1.

The Python 3.2 lifespan has ended; the Aglyph test suite is no longer executed under this version.

Note: Prior to each Aglyph release being committed, tagged, and packaged, its test suite is executed on several Python implementations, versions, and platforms.

When a Python *version* reaches end-of-life, the Aglyph test suite is no longer "officially" executed against that version. This does not necessarily imply that Aglyph unit tests *fail* under that Python version; it simply means that I no longer publish or withhold a release based on the test suite results from that particular Python version.

CPython

Version	Platform
2.7.12	[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
	[GCC 6.1.1 20160621 (Red Hat 6.1.1-3)] on Linux-4.6.5-300.fc24.x86_64-x86_64-with-fedora-24-Twenty_Four
2.7.11	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.3.6	[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
3.3.5	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit (Intel)
3.4.5	[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
3.4.4	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit (Intel)
3.5.2	[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
3.5.1	Windows-7-6.1.7601-SP1 MSC v.1900 32 bit (Intel)
	[GCC 6.1.1 20160621 (Red Hat 6.1.1-3)] on Linux-4.6.5-300.fc24.x86_64-x86_64-with-fedora-24-Twenty_Four

PyPy

Version	Platform
5.3.1	(CPython 2.7.10) [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
2.6.0	(CPython 2.7.9) Windows-7-6.1.7601-SP1
2.4.0	(CPython 3.2.5) Windows-7-6.1.7601-SP1

Stackless Python

Version	Platform
3.1b3	(CPython 2.7.9) [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
	(CPython 3.3.6) [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit
	(CPython 3.4.2) [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on Darwin-15.6.0-x86_64-i386-64bit

IronPython

Version	Platform
2.7.5	cli-32bit .NET 4.0.30319.42000 (32-bit) on Windows-7-6.1.7601-SP1

Jython

Version	Platform
2.7.0	Java-1.8.0_51-Java_HotSpot-TM-_64-Bit_Server_VM,_25.51-b03,_Oracle_Corporation-on-Mac_OS_X-10.11.6-x86_64
	Java-1.8.0_51-Java_HotSpot-TM-_Client_VM,_25.51-b03,_Oracle_Corporation-on-Windows_7-6.1-x86

Aglyph 2.1.0 testing summary

Warning: Changed in version 2.1.0.

The Python 2.6 lifespan has ended; the Aglyph test suite is no longer executed under this version.

Note: Prior to each Aglyph release being committed, tagged, and packaged, its test suite is executed on several Python implementations, versions, and platforms.

When a Python *version* reaches end-of-life, the Aglyph test suite is no longer "officially" executed against that version. This does not necessarily imply that Aglyph unit tests *fail* under that Python version; it simply means that I no longer publish or withhold a release based on the test suite results from that particular Python version.

CPython

Version	Platform
2.7.6	Linux-3.11.10-25-desktop-x86_64-with-SuSE-13.1-x86_64 GCC
2.7.9	Darwin-14.0.0-x86_64-i386-64bit GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)
	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.1.4	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.1.5	Darwin-14.0.0-x86_64-i386-64bit GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)
3.2.5	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.2.6	Darwin-14.0.0-x86_64-i386-64bit GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)
3.3.5	Linux-3.11.10-25-desktop-x86_64-with-SuSE-13.1-x86_64 GCC
	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit (Intel)
3.3.6	Darwin-14.0.0-x86_64-i386-64bit GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)
3.4.2	Darwin-14.0.0-x86_64-i386-64bit GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)
	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit (Intel)

PyPy

Version	Platform
2.4.0 (CPython 2.7.8)	Darwin-14.0.0-x86_64-i386-64bit
	Linux-3.11.10-25-desktop-x86_64-with-SuSE-13.1-x86_64 GCC
	Windows-7-6.1.7601-SP1
2.4.0 (CPython 3.2.5)	Linux-3.11.10-25-desktop-x86_64-with-SuSE-13.1-x86_64 GCC
	Windows-7-6.1.7601-SP1

Stackless Python

Version	Platform
3.1b3 060516 (CPython 2.7.8)	(v2.7.8-slp:ff29dd4f67de, Jan 8 2015, 23:41:02) Darwin-14.0.0-x86_64-i386-64bit
	(default, Jul 11 2014, 19:43:19) [MSC v.1500 32 bit (Intel)] Windows-7-6.1.7601-SP1
3.1b3 060516 (CPython 3.3.5)	(v3.3.5-slp:c856cc204b1c, Jan 9 2015, 00:11:34) Darwin-14.0.0-x86_64-i386-64bit
	(3.3-slp:cea13d5e707f+, Apr 10 2014, 10:33:27) [MSC v.1600 32 bit (Intel)] Windows-7-6.1.7601-SP1

IronPython

Version	Platform
2.7.5	Windows-7-6.1.7601-SP1 cli-32bit .NET 4.0.30319.18408 (32-bit)

Jython

Version	Platform
2.5.3	Java-1.8.0_05-Java_HotSpot-TM-_64-Bit_Server_VM,_25.5-b02,_Oracle_Corporation-on-Mac_OS_X-10.10.1-x86_64 Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)
	Java-1.7.0_55-OpenJDK_64-Bit_Server_VM,_24.55-b03,_Oracle_Corporation-on-Linux-3.11.10-25-desktop-amd64 OpenJDK 64-Bit Server VM (Oracle Corporation)
	Java-1.8.0_25-Java_HotSpot-TM-_Client_VM,_25.25-b02,_Oracle_Corporation-on-Windows_7-6.1-x86 Java HotSpot(TM) Client VM (Oracle Corporation)
2.7.0	Java-1.8.0_05-Java_HotSpot-TM-_64-Bit_Server_VM,_25.5-b02,_Oracle_Corporation-on-Mac_OS_X-10.10.1-x86_64 java1.8.0_05
	Java-1.7.0_55-OpenJDK_64-Bit_Server_VM,_24.55-b03,_Oracle_Corporation-on-Linux-3.11.10-25-desktop-amd64 java1.7.0_55
	Java-1.8.0_25-Java_HotSpot-TM-_Client_VM,_25.25-b02,_Oracle_Corporation-on-Windows_7-6.1-x86 java1.8.0_25

Aglyph 2.0.0 testing summary

Warning: Changed in version 2.0.0.

The Python 2.5 lifespan has ended; the Aglyph test suite is no longer executed under this version.

Note: Prior to each Aglyph release being committed, tagged, and packaged, its test suite is executed on several Python implementations, versions, and platforms.

When a Python *version* reaches end-of-life, the Aglyph test suite is no longer "officially" executed against that version. This does not necessarily imply that Aglyph unit tests *fail* under that Python version; it simply means that I no longer publish or withhold a release based on the test suite results from that particular Python version.

CPython

Version	Platform
2.6.6	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
2.6.9	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
2.7.6	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.1.4	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.1.5	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
3.2.5	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit (Intel)
3.3.3	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit (Intel)

PyPy

Version	Platform
2.2.1	CPython 2.7.3 Darwin-10.8.0-i386-64bit
	CPython 2.7.3 Windows-7-6.1.7601-SP1

Stackless Python

Version	Platform
3.1b3	CPython 2.7.5 Stackless 3.1b3 060516 (default, Jan 5 2014, 23:58:02)
	CPython 2.7.5 Stackless 3.1b3 060516 (default, May 21 2013, 17:59:42) [MSC v.1500 32 bit (Intel)] Windows-7-6.1.7601-SP1
	CPython 3.2.2 Stackless 3.1b3 060516 (default, Mar 20 2013, 23:10:27)
	CPython 3.2.2 Stackless 3.1b3 060516 (default, Feb 20 2012, 13:36:12) [MSC v.1500 32 bit (Intel)] Windows-7-6.1.7601-SP1

IronPython

Version	Platform
2.7.4	Windows-7-6.1.7601-SP1 cli-32bit .NET 4.0.30319.1008 (32-bit)

Jython

Version	Platform
2.5.3	Java-1.6.0_65-Java_HotSpot-TM-_64-Bit_Server_VM,_20.65-b04-462,_Apple_Inc.-on-Mac_OS_X-10.6.8-x86_64 Java HotSpot(TM) 64-Bit Server VM (Apple Inc.)
	Java-1.7.0_17-Java_HotSpot-TM-_Client_VM,_23.7-b01,_Oracle_Corporation-on-Windows_7-6.1-x86 Java HotSpot(TM) Client VM (Oracle Corporation)

Aglyph 1.1.1 testing summary

Warning: Changed in version 1.1.1.

The Python 3.0 lifespan has ended; the Aglyph test suite is no longer executed under this version.

CPython

Version	Platform
2.5.4	Windows-Vista-6.1.7601 MSC v.1310 32 bit
2.5.6	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
2.6.6	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
2.6.8	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
2.7.3	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
3.1.4	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
3.1.5	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
3.2.3	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
3.3.0	Windows-7-6.1.7601-SP1 MSC v.1600 32 bit
	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5666) (dot 3)

PyPy

Version	Platform
1.9 (2.7.2)	Windows-7-6.1.7601-SP1 32 bit
	Darwin-10.8.0-i386-64bit

Stackless Python

Version	Platform
2.7.2 (3.1b3 060516)	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)
3.2.2 (3.1b3 060516)	Windows-7-6.1.7601-SP1 MSC v.1500 32 bit
	Darwin-10.8.0-i386-64bit GCC 4.2.1 (Apple Inc. build 5646) (dot 1)

IronPython

Version	Platform
2.7.3	Windows-7-6.1.7601-SP1 cli-32bit .NET 4.0.30319.296 (32-bit)

Jython

Version	Platform
2.5.3	Java-1.7.0_17-Java_HotSpot-TM-_Client_VM,_23.7-b01,_Oracle_Corporation-on-Windows_7-6.1-x86 Java HotSpot(TM) Client VM (Oracle Corporation)
	Java-1.6.0_43-Java_HotSpot-TM-_64-Bit_Server_VM,_20.14-b01-447,_Apple_Inc.-on-Mac_OS_X-10.6.8-x86_64 Java HotSpot(TM) 64-Bit Server VM (Apple Inc.)

Aglyph 1.1.0 testing summary

CPython

Version	Platform
2.5.4	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
2.5.6	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)
2.6.6	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
2.6.7	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)
2.7.2	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)
	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
3.0.1	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
3.1.4	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)
	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
3.2.1	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)

PyPy

Version	Platform
1.6 (2.7.1)	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit)

Stackless Python

Version	Platform
2.7.2	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)
3.2.1	Windows 7 Ultimate (Windows-7-6.1.7601-SP1)

IronPython

Version	Platform
2.7	Windows 7 Ultimate (Windows-7-6.1.7601-SP1; .NET 4.0.30319.237)

Jython

Version	Platform
2.5.2	Mac OS X 10.6.8 (Darwin-10.8.0-i386-64bit; 1.6.0_26)
	Windows 7 Ultimate (Windows-7-6.1.7601-SP1; Java 1.6.0_27)

1.7 Roadmap for future releases

Warning: Nothing here is set in stone. This list is simply a bucket to record my thoughts about the **possible** future of Aglyph.

The next release may or may not include any of items listed here, and might even include items *not* listed here.

- There's quite a bit of disdain for XML in the Python community. I personally do not agree with it, but that's the reality. Regardless of whether or not the anti-XML sentiments are warranted, though, it does at least *suggest* that there might be a place for an alternative non-programmatic (i.e. text-based) configuration option for Aglyph. I'm mulling over `json`, `configparser`, and YAML at the moment.
- I have encountered several real-world scenarios which suggest that support for **bound** factory methods as an object creation strategy is needed. (These scenarios come almost exclusively from the Java world, but Aglyph is committed to supporting Jython, so. . .)

See also:

Inversion of Control Containers and the Dependency Injection pattern The definitive introduction to Dependency Injection

Python Dependency Injection [PDF] Alex Martelli's introduction to Dependency Injection (and alternatives) in Python

CHAPTER 2

Aglyph versioning

Aglyph follows **PEP 440** for versioning and maintains Semantic Versioning (SemVer) compatibility.

The Aglyph version is always defined as the `__version__` member of the `aglyph/__init__.py` module:

```
>>> import aglyph
>>> aglyph.__version__
'3.0.0.post1'
```

The `Aglyph context DTD` includes the Aglyph version in the filename and in a header comment.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index