glom Documentation

Release 18.1.1

Mahmoud Hashemi

May 19, 2018

Contents

1 Installation **3**
- 1.1 glom Tutorial . . . 3
- 1.2 glom API reference . . . 8
- 1.3 glom Command-Line Interface . . . 16
- 1.4 Frequently Asked Questions . . . 17
- 1.5 glom by Analogy . . . 20
- 1.6 Snippets . . . 21

Python Module Index **25**

Restructuring data, the Python way

glom is a new approach to working with data in Python, featuring:

- *Path-based access* for nested structures
- *Declarative data transformation* using lightweight, Pythonic specifications
- Readable, meaningful *error messages*
- Built-in *data exploration and debugging* features
- And *more*!

While it may sound like a lot, glom's straightforward approach becomes second-nature very quickly. *Get started with the five-minute tutorial!*

CHAPTER 1

Installation

glom is pure Python, and tested on Python 2.7-3.7, as well as PyPy. Installation is easy:

```
pip install glom
```

Then you're ready to get glomming!

```
from glom import glom

target = {'a': {'b': {'c': 'd'}}}
glom(target, 'a.b.c')  # returns 'd'
```

There's much, much more to glom, check out the *glom Tutorial* and *API reference*!

Just glom it!

1.1 glom Tutorial

Learn to use glom in 10 minutes or less!

Basic use of glom requires only a glance, not a whole tutorial. The case study below takes a wider look at day-to-day data and object manipulation, helping you develop an eye for writing robust, declarative data transformations.

Contents

- *Dealing with Data*
- *Access Granted*
- *Going Beyond Access*
- *Changing Requirements*
- *True Python Native*

- *Point of Contact*
- *Understanding the Specification*
- *Conclusion*

Note: glom's tutorial is a runnable module, feel free to run `pip install glom` and `from glom.tutorial import *` in the Python REPL to glom along.

1.1.1 Dealing with Data

Every application deals with data, and these days, even the simplest applications deals with rich, heavily-nested data.

What does nested data looks like? In its most basic form:

```
>>> data = {'a': {'b': {'c': 'd'}}}
>>> data['a']['b']['c']
'd'
```

Pretty simple right? On a good day, it certainly can be. But other days, a value might not be set:

```
>>> data2 = {'a': {'b': None}}
>>> data2['a']['b']['c']
Traceback (most recent call last):
...
TypeError: 'NoneType' object is not subscriptable
```

Well that's no good. We didn't get our value. We got a TypeError, a type of error that doesn't help us at all. The error message doesn't even tell us which access failed. If `data2` had been passed to us, we wouldn't know if `'a'`, `'b'`, or `'c'` had been set to `None`.

If only there were a more semantically powerful accessor.

1.1.2 Access Granted

After years of research and countless iterations, the glom team landed on this simple construct:

```
>>> glom(data, 'a.b.c')
'd'
```

Well that's short, and reads fine, but what about in the error case?

```
>>> glom(data2, 'a.b.c')
Traceback (most recent call last):
...
PathAccessError: could not access 'c', index 2 in path Path('a', 'b', 'c'), got␣
↪error: ...
```

That's more like it! We have a function that can give us our data, or give us an error message we can read, understand, and act upon.

And would you believe this "deep access" example doesn't even scratch the surface of the tip of the iceberg? Welcome to glom.

1.1.3 Going Beyond Access

To start out, let's introduce some basic terminology:

- *target* is our data, be it a dict, list, or any other object
- *spec* is what we want *output* to be

With `output = glom(target, spec)` committed to memory, we're ready for some new requirements.

Let's follow some astronomers on their journey exploring the solar system.

```
>>> target = {'galaxy': {'system': {'planet': 'jupiter'}}}
>>> spec = 'galaxy.system.planet'
>>> glom(target, spec)
'jupiter'
```

Our astronomers want to focus in on the Solar system, and represent planets as a list. Let's restructure the data to make a list of names:

```
>>> target = {'system': {'planets': [{'name': 'earth'}, {'name': 'jupiter'}]}}
>>> glom(target, ('system.planets', ['name']))
['earth', 'jupiter']
```

And let's say we want to capture a parallel list of moon counts with the names as well:

```
>>> target = {'system': {'planets': [{'name': 'earth', 'moons': 1},
...                                  {'name': 'jupiter', 'moons': 69}]}}
>>> spec = {'names': ('system.planets', ['name']),
...         'moons': ('system.planets', ['moons'])}
>>> pprint(glom(target, spec))
{'moons': [1, 69], 'names': ['earth', 'jupiter']}
```

We can react to changing data requirements as fast as the data itself can change, naturally restructuring our results, despite the input's nested nature. Like a list comprehension, but for nested data, our code mirrors our output.

1.1.4 Changing Requirements

Unfortunately, data in the real world is messy. You might be expecting a certain format and end up getting something completely different. No worries, glom to the rescue.

Coalesce is a glom construct that allows you to specify fallback behavior for a list of subspecs. Subspecs are passed as positional arguments, while defaults can be set using keyword arguments.

Let's say our astronomers recently got a new update in their systems, and sometimes `system` will contain `dwarf_planets` instead of `planets`.

To handle this, we can define the `dwarf_planets` subspec as a Coalesce fallback.

```
>>> target = {'system': {'planets': [{'name': 'earth', 'moons': 1},
...                                  {'name': 'jupiter', 'moons': 69}]}}
>>> spec = {'names': (Coalesce('system.planets', 'system.dwarf_planets'), ['name']),
...         'moons': (Coalesce('system.planets', 'system.dwarf_planets'), ['moons'])}
>>> pprint(glom(target, spec))
{'moons': [1, 69], 'names': ['earth', 'jupiter']}
```

You can see here we get the expected results, but say our target changes...

```
>>> target = {'system': {'dwarf_planets': [{'name': 'pluto', 'moons': 5},
...                                         {'name': 'ceres', 'moons': 0}]}}
>>> pprint(glom(target, spec))
{'moons': [5, 0], 'names': ['pluto', 'ceres']}
```

Voila, the target can still be parsed and we can elegantly handle changes in our data formats.

1.1.5 True Python Native

Most other implementations are limited to a particular data format or pure model, be it jmespath or XPath/XSLT. glom makes no such sacrifices of practicality, harnessing the full power of Python itself.

Going back to our example, let's say we wanted to get an aggregate moon count:

```
>>> target = {'system': {'planets': [{'name': 'earth', 'moons': 1},
...                                   {'name': 'jupiter', 'moons': 69}]}}
>>> pprint(glom(target, {'moon_count': ('system.planets', ['moons'], sum)}))
{'moon_count': 70}
```

With glom, you have full access to Python at any given moment. Pass values to functions, whether built-in, imported, or defined inline with lambda.

1.1.6 Point of Contact

glom is a practical tool for production use. To best demonstrate how you can use it, we'll be building an API response. We're implementing a Contacts web service, like an address book, but backed by an ORM/database and compatible with web and mobile frontends.

Let's create a Contact to familiarize ourselves with our test data:

```
>>> contact = Contact('Julian',
...                   emails=[Email(email='jlahey@svtp.info')],
...                   location='Canada')
>>> contact.save()
>>> contact.primary_email
Email(id=5, email='jlahey@svtp.info', email_type='personal')
>>> contact.add_date
datetime.datetime(...)
>>> contact.id
5
```

As you can see, the Contact object has fields for `primary_email`, defaulting to the first email in the email list, and `add_date`, to track the date the contact was added. And as the unique, autoincrementing `id` suggests, there appear to be a few other contacts already in our system.

```
>>> len(Contact.objects.all())
5
```

Sure enough, we've got a little address book going here. But right now it consists of plain Python objects, not very API friendly:

```
>>> json.dumps(Contact.objects.all())
Traceback (most recent call last):
...
TypeError: Contact(id=1, name='Kurt', ...) ... is not JSON serializable
```

But at least we know our data, so let's get to building the API response with glom.

First, let's set our source object, conventionally named *target*:

```
>>> target = Contact.objects.all()  # here we could do filtering, etc.
```

Next, let's specify the format of our result. Remember, the processing is not happening here, this is just declaring the format. We'll be going over the specifics of what each line does after we get our results.

```
>>> spec = {'results': [{'id': 'id',
...                      'name': 'name',
...                      'add_date': ('add_date', str),
...                      'emails': ('emails', [{'id': 'id',
...                                             'email': 'email',
...                                             'type': 'email_type'}]),
...                      'primary_email': Coalesce('primary_email.email',␣
↪default=None),
...                      'pref_name': Coalesce('pref_name', 'name', skip='', default='
↪'),
...                      'detail': Coalesce('company',
...                                         'location',
...                                         ('add_date.year', str),
...                                         skip='', default='')}]}
```

With *target* and *spec* in hand, we're ready to glom, build our response, and take a look the final json-serialized form:

```
>>> resp = glom(target, spec)
>>> print(json.dumps(resp, indent=2, sort_keys=True))
{
  "results": [
    {
      "add_date": "20...",
      "detail": "Mountain View",
      "emails": [
        {
          "email": "kurt@example.com",
          "id": 1,
          "type": "personal"
        }
      ],
      "id": 1,
      "name": "Kurt",
      "pref_name": "Kurt",
      "primary_email": "kurt@example.com"
    },
...
}
```

As we can see, our response looks a lot like our glom specification. This type of WYSIWYG code is one of glom's most important features. After we've appreciated that simple fact, let's look at it line by line.

1.1.7 Understanding the Specification

For `id` and `name`, we're just doing simple copy-overs. For `add_date`, we use a tuple to denote repeated gloms; we access `add_date` and pass the result to `str` to convert it to a string.

For emails we need to serialize a list of subobjects. Good news, glom subgloms just fine, too. We use a tuple to access `emails`, iterate over that list, and from each we copy over `id` and `email`. Note how `email_type` is easily remapped to simply `type`.

For `primary_email` we see our first usage of glom's `Coalesce` feature. Much like SQL's keyword of the same name, `Coalesce` returns the result of the first spec that returns a valid value. In our case, `primary_email` can be None, so a further access of `primary_email.email` would, outside of glom, result in an AttributeError or TypeError like the one we described before the Contact example. Inside of a glom `Coalesce`, exceptions are caught and we move on to the next spec. glom raises a `CoalesceError` when no specs match, so we use `default` to tell it to return None instead.

Some Contacts have nicknames or other names they prefer to go by, so for `pref_name`, we want to return the stored `pref_name`, or fall back to the normal name. Again, we use `Coalesce`, but this time we tell it not only to ignore the default `GlomError` exceptions, but also ignore empty string values, and finally default to empty string if all specs result in empty strings or *`GlomError`*.

And finally, for our last field, `detail`, we want to conjure up a bit of info that'll help jog the user's memory. We're going to include the location, or company, or year the contact was added. You can see an example of this feature as implemented by GitHub, here: https://github.com/mahmoud/glom/stargazers

1.1.8 Conclusion

We've seen a crash course in how glom can tame your data and act as a powerful source of code coherency. glom transforms not only your data, but also your code, bringing it in line with the data itself.

glom tamed our nested data, avoiding tedious, bug-prone lines, replacing what would have been large sections with code that was declarative, but flexible, an ideal balance for maintainability.

1.2 `glom` API reference

glom gets results.

If there was ever a Python example of "big things come in small packages", `glom` might be it.

The `glom` package has one central entrypoint, *`glom.glom()`*. Everything else in the package revolves around that one function.

A couple of conventional terms you'll see repeated many times below:

- **target** - glom is built to work on any data, so we simply refer to the object being accessed as the *"target"*
- **spec** - *(aka "glomspec", short for specification)* The accompanying template used to specify the structure of the return value.

Now that you know the terms, let's take a look around glom's powerful semantics.

Contents

- *The `glom` Function*
- *Specifier Types*
- *Advanced Specifiers*
  - *Conditional access and defaults with Coalesce*
  - *Reducing lambda usage with Call*
  - *Object-oriented access and method calls with `T`*

- *Exceptions*
- *Debugging*
- *Setup and Registration*

1.2.1 The `glom` Function

Where it all happens. The reason for the season. The eponymous function, `glom`.

`glom.glom(target, spec, **kwargs)`

Access or construct a value from a given *target* based on the specification declared by *spec*.

Accessing nested data, aka deep-get:

```
>>> target = {'a': {'b': 'c'}}
>>> glom(target, 'a.b')
'c'
```

Here the *spec* was just a string denoting a path, `'a.b.`. As simple as it should be. The next example shows how to use nested data to access many fields at once, and make a new nested structure.

Constructing, or restructuring more-complicated nested data:

```
>>> target = {'a': {'b': 'c', 'd': 'e'}, 'f': 'g', 'h': [0, 1, 2]}
>>> spec = {'a': 'a.b', 'd': 'a.d', 'h': ('h', [lambda x: x * 2])}
>>> output = glom(target, spec)
>>> pprint(output)
{'a': 'c', 'd': 'e', 'h': [0, 2, 4]}
```

`glom` also takes a keyword-argument, *default*. When set, if a `glom` operation fails with a `GlomError`, the *default* will be returned, very much like `dict.get()`:

```
>>> glom(target, 'a.xx', default='nada')
'nada'
```

The *skip_exc* keyword argument controls which errors should be ignored.

```
>>> glom({}, lambda x: 100.0 / len(x), default=0.0, skip_exc=ZeroDivisionError)
0.0
```

Parameters

- **target** (`object`) – the object on which the glom will operate.
- **spec** (`object`) – Specification of the output object in the form of a dict, list, tuple, string, other glom construct, or any composition of these.
- **default** (`object`) – An optional default to return in the case an exception, specified by *skip_exc*, is raised.
- **skip_exc** (`Exception`) – An optional exception or tuple of exceptions to ignore and return *default* (None if omitted). If *skip_exc* and *default* are both not set, glom raises errors through.

It's a small API with big functionality, and glom's power is only surpassed by its intuitiveness. Give it a whirl!

1.2.2 Specifier Types

Basic glom specifications consist of `dict`, `list`, `tuple`, `str`, and `callable` objects. However, as data calls for more complicated interactions, `glom` provides specialized specifier types that can be used with the basic set of Python builtins.

class glom.**Path**(**path_parts*)

Path objects specify explicit paths when the default `'a.b.c'`-style general access syntax won't work or isn't desirable.

Use this to wrap ints, datetimes, and other valid keys, as well as strings with dots that shouldn't be expanded.

```
>>> target = {'a': {'b': 'c', 'd.e': 'f', 2: 3}}
>>> glom(target, Path('a', 2))
3
>>> glom(target, Path('a', 'd.e'))
'f'
```

class glom.**Literal**(*value*)

Literal objects specify literal values in rare cases when part of the spec should not be interpreted as a glommable subspec. Wherever a Literal object is encountered in a spec, it is replaced with its wrapped *value* in the output.

```
>>> target = {'a': {'b': 'c'}}
>>> spec = {'a': 'a.b', 'readability': Literal('counts')}
>>> pprint(glom(target, spec))
{'a': 'c', 'readability': 'counts'}
```

Instead of accessing `'counts'` as a key like it did with `'a.b'`, `glom()` just unwrapped the literal and included the value.

`Literal` takes one argument, the literal value that should appear in the glom output.

This could also be achieved with a callable, e.g., `lambda x: 'literal_string'` in the spec, but using a `Literal` object adds explicitness, code clarity, and a clean `repr()`.

1.2.3 Advanced Specifiers

The specification techniques detailed above allow you to do pretty much everything glom is designed to do. After all, you can always define and insert a function or `lambda` into almost anywhere in the spec?

Still, for even more specification readability and improved error reporting, glom has a few more tricks up its sleeve.

Conditional access and defaults with Coalesce

Data isn't always where or what you want it to be. Use these specifiers to declare away overly branchy procedural code.

class glom.**Coalesce**(**subspecs*, ***kwargs*)

Coalesce objects specify fallback behavior for a list of subspecs.

Subspecs are passed as positional arguments, and keyword arguments control defaults. Each subspec is evaluated in turn, and if none match, a `CoalesceError` is raised, or a default is returned, depending on the options used.

Note: This operation may seem very familar if you have experience with SQL or even C# and others.

In practice, this fallback behavior's simplicity is only surpassed by its utility:

```
>>> target = {'c': 'd'}
>>> glom(target, Coalesce('a', 'b', 'c'))
'd'
```

glom tries to get `'a'` from `target`, but gets a KeyError. Rather than raise a `PathAccessError` as usual, glom *coalesces* into the next subspec, `'b'`. The process repeats until it gets to `'c'`, which returns our value, `'d'`. If our value weren't present, we'd see:

```
>>> target = {}
>>> glom(target, Coalesce('a', 'b'))
Traceback (most recent call last):
...
CoalesceError: no valid values found. Tried ('a', 'b') and got (PathAccessError,␣
↪PathAccessError) ...
```

Same process, but because `target` is empty, we get a `CoalesceError`. If we want to avoid an exception, and we know which value we want by default, we can set *default*:

```
>>> target = {}
>>> glom(target, Coalesce('a', 'b', 'c'), default='d-fault')
'd-fault'
```

`'a'`, `'b'`, and `'c'` weren't present so we got `'d-fault'`.

Parameters

- **`subspecs`** – One or more glommable subspecs
- **`default`** – A value to return if no subspec results in a valid value
- **`skip`** – A value, tuple of values, or predicate function representing values to ignore
- **`skip_exc`** – An exception or tuple of exception types to catch and move on to the next subspec. Defaults to `GlomError`, the parent type of all glom runtime exceptions.

If all subspecs produce skipped values or exceptions, a `CoalesceError` will be raised. For more examples, check out the *glom Tutorial*, which makes extensive use of Coalesce.

`glom.`**`OMIT = Sentinel('OMIT')`**

The `OMIT` singleton can be returned from a function or included via a `Literal` to cancel assignment into the output object.

```
>>> target = {'a': 'b'}
>>> spec = {'a': lambda t: t['a'] if t['a'] == 'a' else OMIT}
>>> glom(target, spec)
{}
>>> target = {'a': 'a'}
>>> glom(target, spec)
{'a': 'a'}
```

Mostly used to drop keys from dicts (as above) or filter objects from lists.

Reducing lambda usage with Call

There's nothing quite like inserting a quick lambda into a glom spec to get the job done. But once a spec works, how can it be cleaned up?

`glom.`**`Call`**(*func*, *args=None*, *kwargs=None*)

`Call` specifies when a target should be passed to a function, *func*.

Call is similar to partial() in that it is no more powerful than lambda or other functions, but it is designed to be more readable, with a better repr.

Parameters func (*callable*) – a function or other callable to be called with the target

Call combines well with *T* to construct objects. For instance, to generate a dict and then pass it to a constructor:

```
>>> class ExampleClass(object):
...     def __init__(self, attr):
...         self.attr = attr
...
>>> target = {'attr': 3.14}
>>> glom(target, Call(ExampleClass, kwargs=T)).attr
3.14
```

This does the same as glom(target, lambda target: ExampleClass(**target)), but it's easy to see which one reads better.

Note: Call is mostly for functions. Use a *T* object if you need to call a method.

Object-oriented access and method calls with T

glom's shortest-named feature may be its most powerful.

glom.**T = T**

T, short for "target". A singleton object that enables object-oriented expression of a glom specification.

Note: T is a singleton, and does not need to be constructed.

Basically, think of T as your data's stunt double. Everything that you do to T will be recorded and executed during the *glom()* call. Take this example:

```
>>> spec = T['a']['b']['c']
>>> target = {'a': {'b': {'c': 'd'}}}
>>> glom(target, spec)
'd'
```

So far, we've relied on the 'a.b.c'-style shorthand for access, or used the *Path* objects, but if you want to explicitly do attribute and key lookups, look no further than T.

But T doesn't stop with unambiguous access. You can also call methods and perform almost any action you would with a normal object:

```
>>> spec = ('a', (T['b'].items(), list))  # reviewed below
>>> glom(target, spec)
[('c', 'd')]
```

A T object can go anywhere in the spec. As seen in the example above, we access 'a', use a T to get 'b' and iterate over its items, turning them into a list.

You can even use T with *Call* to construct objects:

```
>>> class ExampleClass(object):
...     def __init__(self, attr):
...         self.attr = attr
```

(continues on next page)

(continued from previous page)

```
...
>>> target = {'attr': 3.14}
>>> glom(target, Call(ExampleClass, kwargs=T)).attr
3.14
```

On a further note, while `lambda` works great in glom specs, and can be very handy at times, `T` and *`Call`* eliminate the need for the vast majority of `lambda` usage with glom.

Unlike `lambda` and other functions, `T` roundtrips beautifully and transparently:

```
>>> T['a'].b['c']('success')
T['a'].b['c']('success')
```

Note: While `T` is clearly useful, powerful, and here to stay, its semantics are still being refined. Currently, operations beyond method calls and attribute/item access are considered experimental and should not be relied upon.

Error types and messages are also being rationalized to match those of *`glom.Path`*.

1.2.4 Exceptions

glom introduces a few new exception types designed to maximize readability and debuggability. Note that all these errors derive from `GlomError`, and are only raised from `glom()` calls, not from spec construction or glom type registration. Those declarative and setup operations raise `ValueError`, `TypeError`, and other standard Python exceptions as appropriate.

class glom.**PathAccessError**(*exc*, *path*, *path_idx*)

This *`GlomError`* subtype represents a failure to access an attribute as dictated by the spec. The most commonly-seen error when using glom, it maintains a copy of the original exception and produces a readable error message for easy debugging.

If you see this error, you may want to:

- Check the target data is accurate using *`Inspect`*
- Catch the exception and return a semantically meaningful error message
- Use *`glom.Coalesce`* to specify a default
- Use the top-level `default` kwarg on *`glom()`*

In any case, be glad you got this error and not the one it was wrapping!

Parameters

- **exc** (`Exception`) – The error that arose when we tried to access *path*. Typically an instance of KeyError, AttributeError, IndexError, or TypeError, and sometimes others.
- **path** (`Path`) – The full Path glom was in the middle of accessing when the error occurred.
- **path_idx** (`int`) – The index of the part of the *path* that caused the error.

```
>>> target = {'a': {'b': None}}
>>> glom(target, 'a.b.c')
Traceback (most recent call last):
...
PathAccessError: could not access 'c', index 2 in path Path('a', 'b', 'c'), got␣
↪error: ...
```

(continues on next page)

(continued from previous page)

```
```

class glom.**CoalesceError**(*coal_obj*, *skipped*, *path*)

This *GlomError* subtype is raised from within a *Coalesce* spec's processing, when none of the subspecs match and no default is provided.

The exception object itself keeps track of several values which may be useful for processing:

Parameters

- **coal_obj** (*Coalesce*) – The original failing spec, see *Coalesce*'s docs for details.
- **skipped** (*list*) – A list of ignored values and exceptions, in the order that their respective subspecs appear in the original *coal_obj*.
- **path** – Like many GlomErrors, this exception knows the path at which it occurred.

```
>>> target = {}
>>> glom(target, Coalesce('a', 'b'))
Traceback (most recent call last):
...
CoalesceError: no valid values found. Tried ('a', 'b') and got (PathAccessError,
↪PathAccessError) ...
```

class glom.**UnregisteredTarget**(*op*, *target_type*, *type_map*, *path*)

This *GlomError* subtype is raised when a spec calls for an unsupported action on a target type. For instance, trying to iterate on an non-iterable target:

```
>>> glom(object(), ['a.b.c'])
Traceback (most recent call last):
...
UnregisteredTarget: target type 'object' not registered for 'iterate', expected
↪one of registered types: (...)
```

It should be noted that this is a pretty uncommon occurrence in production glom usage. See the *Setup and Registration* section for details on how to avoid this error.

An UnregisteredTarget takes and tracks a few values:

Parameters

- **op** (*str*) – The name of the operation being performed ('get' or 'iterate')
- **target_type** (*type*) – The type of the target being processed.
- **type_map** (*dict*) – A mapping of target types that do support this operation
- **path** – The path at which the error occurred.

class glom.**GlomError**

The base exception for all the errors that might be raised from *glom()* processing logic.

By default, exceptions raised from within functions passed to glom (e.g., `len`, `sum`, any `lambda`) will not be wrapped in a GlomError.

1.2.5 Debugging

Even the most carefully-constructed specfications eventually need debugging. If the error message isn't enough to fix your glom issues, that's where **Inspect** comes in.

class glom.**Inspect**(**a*, ***kw*)

The *Inspect* specifier type provides a way to get visibility into glom's evaluation of a specification, enabling debugging of those tricky problems that may arise with unexpected data.

Inspect can be inserted into an existing spec in one of two ways. First, as a wrapper around the spec in question, or second, as an argument-less placeholder wherever a spec could be.

Inspect supports several modes, controlled by keyword arguments. Its default, no-argument mode, simply echos the state of the glom at the point where it appears:

```
>>> target = {'a': {'b': {}}}
>>> val = glom(target, Inspect('a.b'))  # wrapping a spec
---
path:   ['a.b']
target: {'a': {'b': {}}}
output: {}
---
```

Debugging behavior aside, *Inspect* has no effect on values in the target, spec, or result.

Parameters

- **echo** (*bool*) – Whether to print the path, target, and output of each inspected glom. Defaults to True.
- **recursive** (*bool*) – Whether or not the Inspect should be applied at every level, at or below the spec that it wraps. Defaults to False.
- **breakpoint** (*bool*) – This flag controls whether a debugging prompt should appear before evaluating each inspected spec. Can also take a callable. Defaults to False.
- **post_mortem** (*bool*) – This flag controls whether exceptions should be caught and interactively debugged with pdb on inspected specs.

All arguments above are keyword-only to avoid overlap with a wrapped spec.

Note: Just like `pdb.set_trace()`, be careful about leaving stray `Inspect()` instances in production glom specs.

1.2.6 Setup and Registration

For the vast majority of objects and types out there in Python-land, *glom()* will just work. However, for that very special remainder, glom is ready and extensible!

glom.**register**(*target_type*, *get=None*, *iterate=None*, *exact=False*)

Register *target_type* so `glom()` will know how to handle instances of that type as targets.

Parameters

- **target_type** (*type*) – A type expected to appear in a glom() call target
- **get** (*callable*) – A function which takes a target object and a name, acting as a default accessor. Defaults to `getattr()`.
- **iterate** (*callable*) – A function which takes a target object and returns an iterator. Defaults to `iter()` if *target_type* appears to be iterable.
- **exact** (*bool*) – Whether or not to match instances of subtypes of *target_type*.

Note: The module-level `register()` function affects the module-level `glom()` function's behavior. If this global effect is undesirable for your application, or you're implementing a library, consider instantiating a `Glommer` instance, and using the `register()` and `Glommer.glom()` methods instead.

class glom.**Glommer**(*register_default_types=True*)

All the wholesome goodness that it takes to make glom work. This type mostly serves to encapsulate the type registration context so that advanced uses of glom don't need to worry about stepping on each other's toes.

Glommer objects are lightweight and, once instantiated, provide the `glom()` method we know and love:

```
>>> glommer = Glommer()
>>> glommer.glom({}, 'a.b.c', default='d')
'd'
>>> Glommer().glom({'vals': list(range(3))}, ('vals', len))
3
```

Instances also provide `register()` method for localized control over type handling.

Parameters **register_default_types** (*bool*) – Whether or not to enable the handling behaviors of the default `glom()`. These default actions include dict access, list and iterable iteration, and generic object attribute access. Defaults to True.

1.3 `glom` Command-Line Interface

Note: glom's CLI is still under construction. Definitely usable and useful, but glom is a library *first*, and if you're reading this, the CLI should not be considered stable.

All the power of `glom`, without even opening your text editor!

```
$ glom --help
Usage: /home/mahmoud/virtualenvs/glom/bin/glom [FLAGS] [spec [target]]

Command-line interface to the glom library, providing nested data
access and data restructuring with the power of Python.

Flags:

--help / -h                    show this help message and exit
--target-file TARGET_FILE      path to target data source (optional)
--target-format TARGET_FORMAT  format of the source data (json or python)
                                 (defaults to 'json')
--spec-file SPEC_FILE          path to glom spec definition (optional)
--spec-format SPEC_FORMAT      format of the glom spec definition (json, python,
                                 python-full) (defaults to 'python')
--indent INDENT                number of spaces to indent the result, 0 to disable
                                 pretty-printing (defaults to 2)
--debug                        interactively debug any errors that come up
--inspect                      interactively explore the data
```

The `glom` command will also read from standard input (stdin) and process that data as the *target*.

Here's an example, filtering a GitHub API example to something much more flat and readable:

```
$ pip install glom
$ curl -s https://api.github.com/repos/mahmoud/glom/events \
    | glom '[{"type": "type", "date": "created_at", "user": "actor.login"}]'
```

This yields:

```
[
  {
    "date": "2018-05-09T03:39:44Z",
    "type": "WatchEvent",
    "user": "asapzacy"
  },
  {
    "date": "2018-05-08T22:51:46Z",
    "type": "WatchEvent",
    "user": "CameronCairns"
  },
  {
    "date": "2018-05-08T03:27:27Z",
    "type": "PushEvent",
    "user": "mahmoud"
  },
  {
    "date": "2018-05-08T03:27:27Z",
    "type": "PullRequestEvent",
    "user": "mahmoud"
  }
  ...
]
```

By default the CLI *target* is JSON and the *spec* is a Python literal.

Note: Because the default CLI spec is a Python literal, there are no lambdas and other Python/glom constructs available. These features are gated behind the `--spec-format python-full` option to avoid code injection and other unwanted consequences.

The `--debug` and `--inspect` flags are useful for exploring data. Note that they are not available when piping data through stdin. Save that API response to a file and use `--target-file` to do your interactive experimenting.

1.4 Frequently Asked Questions

Paradigm shifts always raise a question or two.

Contents

- *What does "glom" mean?*
- *Any other glom terminology worth knowing?*
- *Other glom tips?*
- *Why not just write more Python?*
- *How does glom work?*

- *Does Python need a null-coalescing operator?*

1.4.1 What does "glom" mean?

"glom" is short for "conglomerate", which means "gather into a compact form", coming from the Latin "glom-" meaning *ball*, like *globe*.

glom can be used as a noun or verb. A developer might say, "I glommed together this API response." An astronomer might say, "these gloms of space dust are forming planets and comets."

Got some data you need to transform? **glom it!**

1.4.2 Any other glom terminology worth knowing?

A couple of conventional terms that help navigate around glom's semantics:

- **target** - `glom` operates on a variety of inputs, so we simply refer to the object being accessed (i.e., the first argument to `glom()`) as the "target"
- **spec** - *(aka "glomspec")* The accompanying template used to specify the structure and sources of the output.
- **output** - The value retrieved or created and returned by `glom()`.

All of these can be seen in the conventional call to `glom()`:

```
output = glom(target, spec)
```

Nothing too wild, but these standard terms really do help clarify the complex situations `glom` was built to handle.

1.4.3 Other glom tips?

Just a few (for now):

- Specs don't have to live in the glom call. You can put them anywhere. Commonly-used specs work as class attributes and globals.
- Using glom's declarative approach does wonders for code coverage, much like attrs and schema, both of which go great with `glom`.
- **Advanced tips**
  - glom is designed to support all of Python's built-ins as targets, and is readily extensible to other types and special handling, through `register()`.
  - If you're trying to minimize global state, consider instantiating your own `Glommer` object to encapsulate any type registration changes.

If you've got more tips or patterns, send them our way!

1.4.4 Why not just write more Python?

The answer is more than just DRY ("Don't Repeat Yourself").

Here on the glom team, we're big fans of Python. Have been for years. In fact, Python is one of a tiny handful of languages that could support something as powerful as glom.

But not all Python code is the same. We built glom to replace the kind of Python that is about as un-Pythonic as code gets: simultaneously fluffy, but also fragile. Simple transformations requiring countless lines.

Before glom, the "right" way to write this transformation code was verbose. Whether trying to fetch values nested within objects that may contain attributes set to `None`, or performing a list comprehension which may raise an exception, the *correct* code was many lines of repetitious `try-except` blocks with a lot of hand-written exception messages.

Written any more compactly, this Python would produce failures expressed in errors too low-level to associate with the higher-level transformation.

So the glom-less code was hard to change, hard to debug, or both. `glom` specifications are none of the above, thanks to meaningful, high-level error messages, a *a built-in debugging facility*, and a compact, composable design.

In short, thanks to Python, glom can provide a Pythonic solution for those times when pure Python wasn't Pythonic enough.

1.4.5 How does glom work?

The core conceptual engine of glom is a very simple recursive loop. It could fit on a business card. OK maybe a postcard.

In fact, here it is, in literate form, modified from this early point in glom history:

```
def glom(target, spec):

    # if the spec is a string or a Path, perform a deep-get on the target
    if isinstance(spec, (basestring, Path)):
        return _get_path(target, spec)

    # if the spec is callable, call it on the target
    elif callable(spec):
        return spec(target)

    # if the spec is a dict, assign the result of
    # the glom on the right to the field key on the left
    elif isinstance(spec, dict):
        ret = {}
        for field, subspec in spec.items():
           ret[field] = glom(target, subspec)
        return ret

    # if the spec is a list, run the spec inside the list on every
    # element in the list and return the new list
    elif isinstance(spec, list):
        subspec = spec[0]
        iterator = _get_iterator(target)
        return [glom(t, subspec) for t in iterator]

    # if the spec is a tuple of specs, chain the specs by running the
    # first spec on the target, then running the second spec on the
    # result of the first, and so on.
    elif isinstance(spec, tuple):
        res = target
        for subspec in spec:
            res = glom(res, subspec)
        return res
```

(continues on next page)

(continued from previous page)

```
    else:
        raise TypeError('expected one of the above types')
```

1.4.6 Does Python need a null-coalescing operator?

Not technically a glom question, but it is frequently asked!

Null coalescing operators traverse nested objects and return null (or `None` for us) on the first null or non-traversable object, depending on implementation.

It's basically a compact way of doing a deep `getattr()` with a default set to `None`.

Suffice to say that `glom(target, T.a.b.c, default=None)` achieves this with ease, but I still want to revisit the question, since it's part of what got me thinking about `glom` in the first place.

First off, working in PayPal's SOA environment, my team dealt with literally tens of thousands of service objects, with object definitions (from other teams) nested so deep as to make an 80-character line length laughable.

But null coalescing wouldn't have helped, because in most of those cases `None` wasn't what we needed. We needed a good, automatically generated error message when a deeply-nested field wasn't accessible. Not `NoneType has no attribute 'x'`, but not plain old `None` either.

To solve this, I wrote my share of deep-gets before `glom`, including the open-source boltons.iterutils.get_path(). For whatever reason, it took me years of usage to realize just how often the deep-gets were coupled with the other transformations that `glom` enables. Now, I can never go back to a simple deep-get.

Another years-in-the-making observation, from my time doing JavaScript then PHP then Django templates: all were much more lax on typing than Python. Not because of a fierce belief in weak types, though. More because when you're templating, it's inherently safer to return a blank value on lookup failures. You're so close to text formats that this default achieves a pretty desirable result. While implicitly doing this isn't my cup of tea, and `glom` opts for explicit *`Coalesce`* specifiers, this connection contributed to the concept of `glom` as an "object templating" system.

1.5 `glom` by Analogy

`glom` is pure Python, and you don't need to know anything but Python to use it effectively.

Still, most everyone who encounters `glom` for the first time finds analogies to tools they already know. Whether SQL, list comprehensions, or HTML templates, there seems to be no end to the similarities. Many of them intentional!

While `glom` is none of those tools, and none of those tools are `glom`, a little comparison doesn't hurt. This document collects analogies to help guide understanding along.

1.5.1 Similarity to list comprehensions

One of the key inspirations for `glom` was the humble list comprehension, one of my favorite Python features.

List comprehensions make your code look like its output, and that goes a long way in readability. `glom` itself does list processing with `[lambda x:  x % 2]`, which actually makes it more like a list comp and the old `filter()` function.

`glom`'s list processing differs in two ways:

- Required use of a callable or other `glom` spec, to enable deferred processing.
- Ability to return *`OMIT`*, which can exclude items from a list.

1.5.2 Similarity to templating (Jinja, Django, Mustache)

`glom` is a lot like templating engines, including modern formatters like gofmt, but with all the format affordances distilled out. glom doesn't just work on HTML, XML, JSON, or even just strings.

`glom` works on objects, including functions, dicts, and all other primitives. In fact, it would be safe to call `glom` an "object templating" system.

A lot of insights for `glom` came (and continue to come) from writing ashes.

1.5.3 Similarity to SQL and GraphQL

In some ways, `glom` is a Python query language for Python objects. But thanks to its restructuring capabilities, it's much more than SQL or GraphQL.

With SQL the primary abstraction is an table, or table-like resultset. With GraphQL, the analogous answer to this is, of course, the graph.

glom goes further, not only offering the Python object tree as a graph, but also allowing you to change the shape of the data, restructuring it while fetching and transforming values, which GraphQL only minimally supports, and SQL barely supports at all. Table targets get you table outputs.

1.5.4 Similiarity to validation (jsonschema, schema, cerberus)

`glom` is a generalized form of intake libraries, and will have explicit validation support soon. We definitely took schema becoming successful as a sign that others shared our appetite for succinct, declarative Python datastructure manipulation.

More importantly, these libraries seem to excel at structuring and parsing data, and don't solve much on the other end. Translating valid, structured objects like database models to JSON serializable objects is glom's forté.

1.5.5 Similarity to jq

The CLI that `glom` packs is very similar in function to jq, except it uses Python as its query language, instead of making its own. Most importantly glom gives you a programmatic way forward.

1.5.6 Similarity to XPath/XSLT

These hallowed technologies of yore, they were way ahead of the game in many ways. glom intentionally avoids their purity and verbosity, while trying to take as much inspiration as possible from their function.

1.6 Snippets

glom can do a lot of things, in the right hands. This doc makes those hands yours, through sample code of useful building blocks and common glom tasks.

Contents

- *Reversing a Target*

- *Iteration Result as Tuple*
- *Data-Driven Assignment*
- *Construct Instance*
- *Filtered Iteration*
- *Preserve Type*
- *Automatic Django ORM type handling*

Note: All samples below assume `from glom import glom, T, Call` and any other dependencies.

1.6.1 Reversing a Target

Here are a couple ways to reverse the current target. The first uses basic Python builtins, the second uses the *T* object.

```
glom([1, 2, 3], (reversed, list))
glom([1, 2, 3], T[::-1])
```

1.6.2 Iteration Result as Tuple

The default glom iteration specifier returns a list, but it's easy to turn that list into a tuple. The following returns a tuple of absolute-valued integers:

```
glom([-1, 2, -3], ([abs], tuple))
```

1.6.3 Data-Driven Assignment

glom's dict specifier interprets the keys as constants. A different technique is required if the dict keys are part of the target data rather than spec.

```
glom({1:2, 2:3}, Call(dict, args=T.items())
glom({1:2, 2:3}, lambda t: dict(t.items()))
glom({1:2, 2:3}, dict)
```

1.6.4 Construct Instance

A common use case is to construct an instance. In the most basic case, the default behavior on callable will suffice.

The following converts a list of ints to a list of `decimal.Decimal` objects.

```
glom([1, 2, 3], [Decimal])
```

If additional arguments are required, *Call* or `lambda` are good options.

This converts a list to a collection.deque, while specifying a max size of 10.

```
glom([1, 2, 3], Call(deque, args=[T, 10]))
glom([1, 2, 3], lambda t: deque(t, 10))
```

1.6.5 Filtered Iteration

Sometimes in addition to stepping through an iterable, you'd like to omit some of the items from the result set all together. Here are two ways to filter the odd numbers from a list.

```
glom([1, 2, 3, 4, 5, 6], lambda t: [i for i in t if i % 2])
glom([1, 2, 3, 4, 5, 6], [lambda i: i if i % 2 else OMIT])
```

The second approach demonstrates the use of `glom.OMIT` to back out of an execution.

This can also be combined with *`Coalesce`* to filter items which are missing sub-attributes.

Here is an example of extracting the primary email from a group of contacts, skipping where the email is empty string, None, or the attribute is missing.

```
glom(contacts, [Coalesce('primary_email.email', skip=('', None), default=OMIT)])
```

1.6.6 Preserve Type

The iteration specifier will walk lists and tuples. In some cases it would be convenient to preserve the target type in the result type.

This glomspec iterates over a tuple or list, adding one to each element, and uses *`T`* to return a tuple or list depending on the target input's type.

```
glom((1, 2, 3), (
    {
        "type": type,
        "result": [lambda v: v + 1]  # arbitrary operation
    }, T['type'](T['result'])))
```

This demonstrates an advanced technique – just as a tuple can be used to process sub-specs "in series", a dict can be used to store intermediate results while processing sub-specs "in parallel" so they can then be recombined later on.

1.6.7 Automatic Django ORM type handling

In day-to-day Django ORM usage, Managers and QuerySets are everywhere. They work great with glom, too, but they work even better when you don't have to call `.all()` all the time. Enable automatic iteration using the following *`register()`* technique:

```
import glom
import django.db.models

glom.register(django.db.models.Manager, iterate=lambda m: m.all())
glom.register(django.db.models.QuerySet, iterate=lambda qs: qs.all())
```

Call this in `settings` or somewhere similarly early in your application setup for the best results.

Python Module Index

g

glom.core, 8
glom.tutorial, 4

Index

C

Call() (in module glom), 11
Coalesce (class in glom), 10
CoalesceError (class in glom), 14

G

glom() (in module glom), 9
glom.core (module), 8
glom.tutorial (module), 4
GlomError (class in glom), 14
Glommer (class in glom), 16

I

Inspect (class in glom), 14

L

Literal (class in glom), 10

O

OMIT (in module glom), 11

P

Path (class in glom), 10
PathAccessError (class in glom), 13

R

register() (in module glom), 15

T

T (in module glom), 12

U

UnregisteredTarget (class in glom), 14