
glom Documentation

Release 20.11.0

Mahmoud Hashemi

Nov 04, 2020

1 Installation	3
Python Module Index	57
Index	59

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 *debugging* features
- Plus, *deep assignment, streaming, data validation, and more!*

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

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 no time!

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

Go beyond basic with 10 minutes or less, and even further if you can spare a half hour.

Contents

- *Dealing with Data*
- *Accessing Nested Data*
 - *Interactive Deep Get*
- *Going Beyond Access*

- *Handling Nested Lists*
- *Changing Requirements*
- *Data-Driven Assignment*
- *True Python Native*
 - *Interactive Planetary Templating*
- *Practical Production Use*
- *Understanding the Specification*
 - *Interactive Contact Management*
- *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. Or [try it in your browser here](#) or in the embedded REPLs below!

1.1.1 Dealing with Data

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

What does nested data look 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 Accessing Nested Data

AKA “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.

See also:

For more on glom's error messages, see *Exceptions & Debugging*.

Interactive Deep Get

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 Handling Nested Lists

In the example above we introduced a new wrinkle: the target for `planets` has multiple entries stored in a list. Previously our targets were all nested dictionaries.

To handle this we use a new *spec* pattern: `(path, [subpath])`. In this pattern `path` is the path to the list, and `subpath` is the path within each element of the list. What's that? You need to handle lists within lists (within lists ...)? Then just repeat the pattern, replacing `subpath` with another `(path, [subpath])` tuple. For example, say we have information about each planet's moons like so:

```

>>> target = {
...     'system': {
...         'planets': [
...             {
...                 'name': 'earth',
...                 'moons': [
...                     {'name': 'luna'}
...                 ]
...             },
...             {
...                 'name': 'jupiter',
...                 'moons': [
...                     {'name': 'io'},
...                     {'name': 'europa'}
...                 ]
...             }
...         ]
...     }
... }

```

We can get the names of each moon from our nested lists by nesting our subpath specs:

```

>>> spec = {
...     'planet_names': ('system.planets', ['name']),
...     'moon_names': ('system.planets', [('moons', ['name'])])
... }
>>> pprint(glom(target, spec))
{'moon_names': [['luna'], ['io', 'europa']], 'planet_names': ['earth', 'jupiter']}

```

1.1.5 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. Subspects 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.

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

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

1.1.6 Data-Driven Assignment

Quite often APIs deliver data in dictionaries without constant key values. They use parts of the data itself as a key. This we call data-driven assignment.

The following example shows you a way to handle this situation. It extracts the moon count from a dictionary that has the planet names as a key.

```
>>> from glom import glom, T, Merge, Iter, Coalesce
>>> target = {
...     "pluto": {"moons": 6, "population": None},
...     "venus": {"population": {"aliens": 5}},
...     "earth": {"moons": 1, "population": {"humans": 7700000000, "aliens": 1}},
```

(continues on next page)

(continued from previous page)

```

... }
>>> spec = {
...     "moons": (
...         T.items(),
...         Iter({T[0]: (T[1], Coalesce("moons", default=0))}),
...         Merge(),
...     )
... }
>>> pprint(glom(target, spec))
{'moons': {'earth': 1, 'pluto': 6, 'venus': 0}}

```

Don't worry if you do not fully understand how this works at this point. If you would like to learn more, look up [Iter\(\)](#), [T](#), or [Merge](#) in the glom API reference.

1.1.7 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.

Interactive Planetary Templating

1.1.8 Practical Production Use

AKA “*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: pri

```

>>> from glom.tutorial import * # import the tutorial module members
>>> 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')

```

(continues on next page)

(continued from previous page)

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
        "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.

1.1.9 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>

Interactive Contact Management

1.1.10 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 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?*

1.2.1 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!**

1.2.2 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.

1.2.3 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` which goes 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!](#)

1.2.4 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 *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.

1.2.5 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)
```

(continues on next page)

(continued from previous page)

```

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

```

1.2.6 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.

1.3 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.

1.3.1 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 square brackets like `[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 `SKIP`, which can exclude items from a list.

1.3.2 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](#).

1.3.3 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.

1.3.4 Similarity to validation (jsonschema, schema, cerberus)

`glom` is a generalized form of intake libraries including validation. 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.

1.3.5 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*.

1.3.6 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.

1.3.7 Others

Beyond what's listed above, several other packages and language features exist in glom's ballpark, including:

- Specter (for Clojure)
- Lenses (for Haskell)
- Dig (for Ruby Hashmaps)

If you know of other useful comparisons, [let us know!](#)

1.4 Examples & 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*
- *Filter Iterable*
- *Lisp-style If Extension*
- *Parallel Evaluation of Sub-Specs*
- *Clamp Values*
- *Transform Tree*
- *Fix Up Strings in Parsed JSON*
- *Store and Retrieve Current Target*

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

1.4.1 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])
```

1.4.2 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))
```

1.4.3 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)
```

1.4.4 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))
```

1.4.5 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 SKIP])
```

The second approach demonstrates the use of `glom.SKIP` 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=SKIP)])
```

1.4.6 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.

1.4.7 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.

1.4.8 Filter Iterable

An iteration specifier can filter items out by using *SKIP* as the default of a *Check* object.

```
glom(['cat', 1, 'dog', 2], [Check(types=str, default=SKIP)])
# ['cat', 'dog']
```

You can also truncate the list at the first failing check by using *STOP*.

1.4.9 Lisp-style If Extension

Any class with a `glomit` method will be treated as a spec by glom. As an example, here is a lisp-style *If* expression custom spec type:

```
class If(object):
    def __init__(self, cond, if_, else_=None):
        self.cond, self.if_, self.else_ = cond, if_, else_

    def glomit(self, target, scope):
        g = lambda spec: scope[glom](target, spec, scope)
        if g(self.cond):
            return g(self.if_)
        elif self.else_:
            return g(self.else_)
        else:
            return None
```

(continues on next page)

(continued from previous page)

```
glom(1, If(bool, {'yes': T}, {'no': T}))
# {'yes': 1}
glom(0, If(bool, {'yes': T}, {'no': T}))
# {'no': 0}
```

1.4.10 Parallel Evaluation of Sub-Specs

This is another example of a simple glom extension. Sometimes it is convenient to execute multiple glom-specs in parallel against a target, and get a sequence of their results.

```
class Seq(object):
    def __init__(self, *subspecs):
        self.subspecs = subspecs

    def glomit(self, target, scope):
        return [scope[glom](target, spec, scope) for spec in self.subspecs]

glom('1', Seq(float, int))
# [1.0, 1]
```

Without this extension, the simplest way to achieve the same result is with a dict:

```
glom('1', ({1: float, 2: int}, T.values()))
```

1.4.11 Clamp Values

A common numerical operation is to clamp values – if they are above or below a certain value, assign them to that value.

Using a pattern-matching glom idiom, this can be implemented simply:

```
glom(range(10), [(M < 7) | Val(7)])
# [0, 1, 2, 3, 4, 5, 6, 7, 7, 7]
```

What if you want to drop rather than clamp out-of-range values?

```
glom(range(10), [(M < 7) | Val(SKIP)])
# [0, 1, 2, 3, 4, 5, 6]
```

1.4.12 Transform Tree

With an arbitrary depth tree, *Ref* can be used to express a recursive spec.

For example, this *etree2dicts* spec will recursively walk an *ElementTree* instance and transform it from nested objects to nested dicts.

```
etree2dicts = Ref('ElementTree',
    {"tag": "tag", "text": "text", "attrib": "attrib", "children": (iter, [Ref(
        ↪'ElementTree')])})
```

Alternatively, say we only wanted to generate tuples of tag and children:

```
etree2tuples = Fill(Ref('ElementTree', (T.tag, Iter(Ref('ElementTree')).all())))
```

(Note also the use of `Fill` mode to easily construct a tuple.)

```
<html>
  <head>
    <title>the title</title>
  </head>
  <body id="the-body">
    <p>A paragraph</p>
  </body>
</html>
```

Will translate to the following tuples:

```
>>> etree = ElementTree.fromstring(html_text)
>>> glom(etree, etree2tuples)
('html', [('head', [('title', [])]), ('body', [('p', [])])])
```

1.4.13 Fix Up Strings in Parsed JSON

Tree-walking with `Ref()` combines powerfully with pattern matching from `Match()`.

In this case, consider that we want to transform parsed JSON recursively, such that all unicodes are converted to native strings.

```
glom(json.loads(data),
     Ref('json',
        Match(Switch({
            dict: {Ref('json'): Ref('json')},
            list: [Ref('json')],
            type(u''): Auto(str),
            object: T}))
    )
)
```

`Match()` above splits the `Ref()` evaluation into 4 cases:

- on `dict`, use `Ref()` to recurse for all keys and values
- on `list`, use `Ref()` to recurse on each item
- on text objects (`type(u'')`) – py3 `str` or py2 `unicode` – transform the target with `str`
- for all other values (`object`), pass them through

As motivation for why this might come up: attributes, class names, function names, and identifiers must be the native string type for a given Python, i.e., bytestrings in Python 2 and unicode in Python 3.

1.4.14 Store and Retrieve Current Target

The `A` scope assignment helper makes it extremely convenient to hold on to the current target and then reset it.

The $(A.t, \dots, S.t)$ “sandwich” is a convenient idiom for these cases.

For example, we could use this to update a `dict`:

```
glom({}, (A.t, T.update({1: 1}), S.t))
```

1.5 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",
```

(continues on next page)

(continued from previous page)

```

    "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.

1.6 Core glom API

glom gets results.

The `glom` package has one central entrypoint, `glom.glom()`. Everything else in the package revolves around that one function. Sometimes, big things come in small packages.

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.

See also:

As the `glom` API grows, we've refactored the docs into separate domains. The core API is below. More specialized types can also be found in the following docs:

- [Assignment & Mutation](#)
- [Streaming & Iteration](#)
- [Reduction & Grouping](#)
- [Matching & Validation](#)

Longtime `glom` docs readers: thanks in advance for reporting/fixing any broken links you may find.

Contents

- [The `glom` Function](#)
- [Basic Specifiers](#)
- [Object-Oriented Access and Method Calls with `T`](#)

- *Defaults with Coalesce*
- *Calling Callables with Invoke*
 - *Alternative approach to functions: Call*
- *Self-Referential Specs*
- *The glom Scope*
 - *Basic usage*
 - *Updating the scope - S() & A*
 - *Sensible saving - Vars & S.globals*
- *Core Exceptions*
- *Setup and Registration*

1.6.1 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.
- **scope** (*dict*) – Additional data that can be accessed via *S* inside the glom-spec. Read more: *The glom Scope*.

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

1.6.2 Basic Specifiers

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'
```

Paths can be used to join together other Path objects, as well as *T* objects:

```
>>> Path(T['a'], T['b'])
T['a']['b']
>>> Path(Path('a', 'b'), Path('c', 'd'))
Path('a', 'b', 'c', 'd')
```

Paths also support indexing and slicing, with each access returning a new Path object:

```
>>> path = Path('a', 'b', 1, 2)
>>> path[0]
Path('a')
>>> path[-2:]
Path(1, 2)
```

class `glom.Val` (*value*)

Val objects are specs which evaluate to the wrapped *value*.

```
>>> target = {'a': {'b': 'c'}}
>>> spec = {'a': 'a.b', 'readability': Val('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 Val and included the value.

Val takes one argument, the value to be returned.

Note: *Val* was named `Literal` in versions of glom before 20.7.0. An alias has been preserved for backwards compatibility, but reprs have changed.

class `glom.Spec` (*spec*, *scope=None*)

Spec objects serve three purposes, here they are, roughly ordered by utility:

1. As a form of compiled or “curried” glom call, similar to Python’s built-in `re.compile()`.
2. A marker as an object as representing a spec rather than a literal value in certain cases where that might be ambiguous.
3. A way to update the scope within another Spec.

In the second usage, Spec objects are the complement to *Val*, wrapping a value and marking that it should be interpreted as a glom spec, rather than a literal value. This is useful in places where it would be interpreted as a value by default. (Such as `T[key]`, `Call(func)` where key and func are assumed to be literal values and not specs.)

Parameters

- **spec** – The glom spec.
- **scope** (*dict*) – additional values to add to the scope when evaluating this Spec

See also:

Note that many of the Specifier types previously mentioned here have moved into their own docs, among them:

- *Assignment & Mutation*
- *Streaming & Iteration*
- *Reduction & Grouping*
- *Matching & Validation*

1.6.3 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
...
>>> 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')
```

T-related access errors raise a *PathAccessError* during the *glom()* call.

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.

Note: T attributes starting with `__` are reserved to avoid colliding with many built-in Python behaviors, current and future. The `T.__()` method is available for cases where they are needed. For example, `T.__('class__')` is equivalent to accessing the `__class__` attribute.

1.6.4 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 familiar 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`.

Note: Coalesce is a *branching* specifier type, so as of v20.7.0, its exception messages feature an error tree. See *Reading Branched Exceptions* for details on how to interpret these exceptions.

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
- **default_factory** – A callable whose result will be returned as a default
- **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.**SKIP** = **Sentinel**('SKIP')

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

```
>>> target = {'a': 'b'}
>>> spec = {'a': lambda t: t['a'] if t['a'] == 'a' else SKIP}
>>> 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.

Note: SKIP was known as OMIT in versions 18.3.1 and prior. Versions 19+ will remove the OMIT alias entirely.

```
glom.STOP = Sentinel('STOP')
```

The STOP singleton can be used to halt iteration of a list or execution of a tuple of subspecs.

```
>>> target = range(10)
>>> spec = [lambda x: x if x < 5 else STOP]
>>> glom(target, spec)
[0, 1, 2, 3, 4]
```

1.6.5 Calling Callables with Invoke

New in version 19.10.0.

From calling functions to constructing objects, it's hardly Python if you're not invoking callables. By default, single-argument functions work great on their own in glom specs. The function gets passed the target and it just works:

```
>>> glom(['1', '3', '5'], [int])
[1, 3, 5]
```

Zero-argument and multi-argument functions get a lot trickier, especially when more than one of those arguments comes from the target, thus the Invoke spec.

```
class glom.Invoke(func)
```

Specifier type designed for easy invocation of callables from glom.

Parameters *func* (*callable*) – A function or other callable object.

Invoke is similar to `functools.partial()`, but with the ability to set up a “templated” call which interleaves constants and glom specs.

For example, the following creates a spec which can be used to check if targets are integers:

```
>>> is_int = Invoke(isinstance).specs(T).constants(int)
>>> glom(5, is_int)
True
```

And this composes like any other glom spec:

```
>>> target = [7, object(), 9]
>>> glom(target, [is_int])
[True, False, True]
```

Another example, mixing positional and keyword arguments:

```
>>> spec = Invoke(sorted).specs(T).constants(key=int, reverse=True)
>>> target = ['10', '5', '20', '1']
>>> glom(target, spec)
['20', '10', '5', '1']
```

Invoke also helps with evaluating zero-argument functions:

```
>>> glom(target={}, spec=Invoke(int))
0
```

(A trivial example, but from timestamps to UUIDs, zero-arg calls do come up!)

Note: `Invoke` is mostly for functions, object construction, and callable objects. For calling methods, consider the `T` object.

constants (*a, **kw)

Returns a new `Invoke` spec, with the provided positional and keyword argument values stored for passing to the underlying function.

```
>>> spec = Invoke(T).constants(5)
>>> glom(range, (spec, list))
[0, 1, 2, 3, 4]
```

Subsequent positional arguments are appended:

```
>>> spec = Invoke(T).constants(2).constants(10, 2)
>>> glom(range, (spec, list))
[2, 4, 6, 8]
```

Keyword arguments also work as one might expect:

```
>>> round_2 = Invoke(round).constants(ndigits=2).specs(T)
>>> glom(3.14159, round_2)
3.14
```

`constants()` and other `Invoke` methods may be called multiple times, just remember that every call returns a new spec.

classmethod specfunc (spec)

Creates an `Invoke` instance where the function is indicated by a spec.

```
>>> spec = Invoke.specfunc('func').constants(5)
>>> glom({'func': range}, (spec, list))
[0, 1, 2, 3, 4]
```

specs (*a, **kw)

Returns a new `Invoke` spec, with the provided positional and keyword arguments stored to be interpreted as specs, with the results passed to the underlying function.

```
>>> spec = Invoke(range).specs('value')
>>> glom({'value': 5}, (spec, list))
[0, 1, 2, 3, 4]
```

Subsequent positional arguments are appended:

```
>>> spec = Invoke(range).specs('start').specs('end', 'step')
>>> target = {'start': 2, 'end': 10, 'step': 2}
>>> glom(target, (spec, list))
[2, 4, 6, 8]
```

Keyword arguments also work as one might expect:

```
>>> multiply = lambda x, y: x * y
>>> times_3 = Invoke(multiply).constants(y=3).specs(x='value')
>>> glom({'value': 5}, times_3)
15
```

`specs()` and other *Invoke* methods may be called multiple times, just remember that every call returns a new spec.

star (*args=None, kwargs=None*)

Returns a new *Invoke* spec, with *args* and/or *kwargs* specs set to be “starred” or “star-starred” (respectively)

```
>>> import os.path
>>> spec = Invoke(os.path.join).star(args='path')
>>> target = {'path': ['path', 'to', 'dir']}
>>> glom(target, spec)
'path/to/dir'
```

Parameters

- **args** (*spec*) – A spec to be evaluated and “starred” into the underlying function.
- **kwargs** (*spec*) – A spec to be evaluated and “star-starred” into the underlying function.

One or both of the above arguments should be set.

The *star()*, like other *Invoke* methods, may be called multiple times. The *args* and *kwargs* will be stacked in the order in which they are provided.

Alternative approach to functions: Call

An earlier, more primitive approach to callables in glom was the *Call* specifier type.

Warning: Given superiority of its successor, *Invoke*, the *Call* type may be deprecated in a future release.

class `glom.Call` (*func=None, 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.

Warning: `Call` has a successor with a fuller-featured API, new in 19.10.0: the `Invoke` specifier type.

1.6.6 Self-Referential Specs

Sometimes nested data repeats itself, either recursive structure or just through redundancy.

class `glom.Ref` (*name*, *subspec*=`Sentinel('_MISSING')`)

Name a part of a spec and refer to it elsewhere in the same spec, useful for trees and other self-similar data structures.

Parameters

- **name** (*str*) – The name of the spec to reference.
- **subspec** – Pass a spec to name it *name*, or leave unset to refer to an already-named spec.

1.6.7 The `glom` Scope

Sometimes data transformation involves more than a single target and spec. For those times, `glom` has a *scope* system designed to manage additional state.

Basic usage

On its surface, the `glom` scope is a dictionary of extra values that can be passed in to the top-level `glom` call. These values can then be addressed with the `S` object, which behaves similarly to the `T` object.

Here's an example case, counting the occurrences of a value in the target, using the scope:

```
>>> count_spec = T.count(S.search)
>>> glom(['a', 'c', 'a', 'b'], count_spec, scope={'search': 'a'})
2
```

Note how `S` supports attribute-style dot-access for its keys. For keys which are not valid attribute names, key-style access is also supported.

Note: `glom` itself uses certain keys in the scope to manage internal state. Consider the namespace of strings, integers, builtin types, and other common Python objects open for your usage. Read *the custom spec doc* to learn about more advanced, reserved cases.

Updating the scope - `S()` & `A`

`glom`'s scope isn't only set once when the top-level `glom()` function is called. It's dynamic and updatable.

If your use case requires saving a value from one part of the target for usage elsewhere, then `S` will allow you to save values to the scope:

```
>>> target = {'data': {'val': 9}}
>>> spec = (S(value=T['data']['val']), {'val': S['value']})
>>> glom(target, spec)
{'val': 9}
```

Any keyword arguments to the **S** will have their values evaluated as a spec, with the result being saved to the keyword argument name in the scope.

When only the target is being assigned, you can use the **A** as a shortcut:

```
>>> target = {'data': {'val': 9}}
>>> spec = ('data.val', A.value, {'val': S.value})
>>> glom(target, spec)
{'val': 9}
```

A enables a shorthand which assigns the current target to a location in the scope.

Sensible saving - Vars & S.globals

Of course, glom's scopes do not last forever. Much like function calls in Python, new child scopes can see and read values in parent scopes. When a child spec saves a new value to the scope, it's lost when the child spec completes.

If you need values to be saved beyond a spec's local scope, the best way to do that is to create a *Vars* object in a common ancestor scope. *Vars* acts as a mutable namespace where child scopes can store state and have it persist beyond their local scope. Choose a location in the spec such that all involved child scopes can see and share the value.

Note: glom precreates a *global Vars* object at `S.globals`. Any values saved there will be accessible throughout that given `glom()` call:

```
>>> last_spec = ([A.globals.last], S.globals.last)
>>> glom([3, 1, 4, 1, 5], last_spec)
5
```

While not shared across calls, most of the same care prescribed about using global state still applies.

class `glom.Vars` (*base=()*, ***kw*)

Vars is a helper that can be used with **S** in order to store shared mutable state.

Takes the same arguments as `dict()`.

Arguments here should be thought of the same way as default arguments to a function. Each time the spec is evaluated, the same arguments will be referenced; so, think carefully about mutable data structures.

1.6.8 Core Exceptions

Not all data is going to match specifications. Luckily, glom errors are designed to be as readable and actionable as possible.

All glom exceptions inherit from `GlomError`, described below, along with other core exception types. For more details about handling and debugging exceptions, see "[Exceptions & Debugging](#)".

class `glom.PathAccessError` (*exc*, *path*, *part_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.
- **part_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', part 2 of Path('a', 'b', 'c'), got error: .
↪...
```

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) ...
```

Note: `Coalesce` is a *branching* specifier type, so as of v20.7.0, its exception messages feature an error tree. See [Reading Branched Exceptions](#) for details on how to interpret these exceptions.

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 a 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.BadSpec`

Raised when a spec structure is malformed, e.g., when a specifier type is invalid for the current mode.

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`.

1.6.9 Setup and Registration

When it comes to targets, `glom()` will operate on the vast majority of objects out there in Python-land. However, for that very special remainder, `glom` is readily extensible!

`glom.register` (*target_type*, ***kwargs*)

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.Glomer` (***kwargs*)

The `Glommer` type mostly serves to encapsulate type registration context so that advanced uses of `glom` don't need to worry about stepping on each other.

`Glommer` objects are lightweight and, once instantiated, provide a `glom()` method:

```
>>> 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`.

1.7 Assignment & Mutation

By default, glom aims to safely return a transformed copy of your data. But sometimes you really need to transform an existing object.

When you already have a large or complex bit of nested data that you are sure you want to modify in-place, glom has you covered, with the `assign()` function, and the `Assign()` specifier type.

Contents

- [Assignment](#)
- [Deletion](#)
- [Exceptions](#)

1.7.1 Assignment

Deeply assign within an existing structure, given a path and a value.

`glom.assign(obj, path, val, missing=None)`
New in glom 18.3.0

The `assign()` function provides convenient “deep set” functionality, modifying nested data structures in-place:

```
>>> target = {'a': [{'b': 'c'}, {'d': None}]}
>>> _ = assign(target, 'a.1.d', 'e') # let's give 'd' a value of 'e'
>>> pprint(target)
{'a': [{'b': 'c'}, {'d': 'e'}]}
```

Missing structures can also be automatically created with the `missing` parameter. For more information and examples, see the `Assign` specifier type, which this function wraps.

`class glom.Assign(path, val, missing=None)`
New in glom 18.3.0

The `Assign` specifier type enables glom to modify the target, performing a “deep-set” to mirror glom’s original deep-get use case.

`Assign` can be used to perform spot modifications of large data structures when making a copy is not desired:

```
# deep assignment into a nested dictionary
>>> target = {'a': {}}
>>> spec = Assign('a.b', 'value')
>>> _ = glom(target, spec)
>>> pprint(target)
{'a': {'b': 'value'}}
```

The value to be assigned can also be a `Spec`, which is useful for copying values around within the data structure:

```
# copying one nested value to another
>>> _ = glom(target, Assign('a.c', Spec('a.b')))
>>> pprint(target)
{'a': {'b': 'value', 'c': 'value'}}
```

Another handy use of `Assign` is to deep-apply a function:

```
# sort a deep nested list
>>> target={'a':{'b':[3,1,2]}}
>>> _ = glom(target, Assign('a.b', Spec(('a.b',sorted))))
>>> pprint(target)
{'a': {'b': [1, 2, 3]}}
```

Like many other specifier types, `Assign`'s destination path can be a `T` expression, for maximum control:

```
# changing the error message of an exception in an error list
>>> err = ValueError('initial message')
>>> target = {'errors': [err]}
>>> _ = glom(target, Assign(T['errors'][0].args, ('new message',)))
>>> str(err)
'new message'
```

`Assign` has built-in support for assigning to attributes of objects, keys of mappings (like dicts), and indexes of sequences (like lists). Additional types can be registered through `register()` using the "assign" operation name.

Attempting to assign to an immutable structure, like a `tuple`, will result in a `PathAssignError`. Attempting to assign to a path that doesn't exist will raise a `PathAccessError`.

To automatically backfill missing structures, you can pass a callable to the `missing` argument. This callable will be called for each path segment along the assignment which is not present.

```
>>> target = {}
>>> assign(target, 'a.b.c', 'hi', missing=dict)
{'a': {'b': {'c': 'hi'}}
```

1.7.2 Deletion

Delete attributes from objects and keys from containers.

`glom.delete(obj, path, ignore_missing=False)`

The `delete()` function provides “deep del” functionality, modifying nested data structures in-place:

```
>>> target = {'a': [{'b': 'c'}, {'d': None}]}
>>> delete(target, 'a.0.b')
{'a': [{}, {'d': None}]}
```

Attempting to delete missing keys, attributes, and indexes will raise a `PathDeleteError`. To ignore these errors, use the `ignore_missing` argument:

```
>>> delete(target, 'does_not_exist', ignore_missing=True)
{'a': [{}, {'d': None}]}
```

For more information and examples, see the `Delete` specifier type, which this convenience function wraps.

New in version 20.5.0.

class `glom.Delete(path, ignore_missing=False)`

In addition to `glom`'s core “deep-get” and `Assign`'s “deep-set”, the `Delete` specifier type performs a “deep-del”, which can remove items from larger data structures by key, attribute, and index.

```
>>> target = {'dict': {'x': [5, 6, 7]}}
>>> glom(target, Delete('dict.x.1'))
```

(continues on next page)

(continued from previous page)

```
{'dict': {'x': [5, 7]}}
>>> glom(target, Delete('dict.x'))
{'dict': {}}
```

If a target path is missing, a `PathDeleteError` will be raised. To ignore missing targets, use the `ignore_missing` flag:

```
>>> glom(target, Delete('does_not_exist', ignore_missing=True))
{'dict': {}}
```

Delete has built-in support for deleting attributes of objects, keys of dicts, and indexes of sequences (like lists). Additional types can be registered through `register()` using the "delete" operation name.

New in version 20.5.0.

1.7.3 Exceptions

class `glom.PathAssignError` (*exc, path, dest_name*)

This `GlomError` subtype is raised when an assignment fails, stemming from an `assign()` call or other `Assign` usage.

One example would be assigning to an out-of-range position in a list:

```
>>> assign(["short", "list"], Path(5), 'too far')
Traceback (most recent call last):
...
PathAssignError: could not assign 5 on object at Path(), got error: IndexError(...)
```

Other assignment failures could be due to assigning to an `@property` or exception being raised inside a `__setattr__()`.

class `glom.PathDeleteError` (*exc, path, dest_name*)

This `GlomError` subtype is raised when an assignment fails, stemming from a `delete()` call or other `Delete` usage.

One example would be deleting an out-of-range position in a list:

```
>>> delete(["short", "list"], Path(5))
Traceback (most recent call last):
...
PathDeleteError: could not delete 5 on object at Path(), got error: IndexError(...)
```

Other assignment failures could be due to deleting a read-only `@property` or exception being raised inside a `__delattr__()`.

1.8 Streaming & Iteration

New in version 19.10.0. `glom`'s helpers for streaming use cases.

Specifier types which yield their results incrementally so that they can be applied to targets which are themselves streaming (e.g. chunks of rows from a database, lines from a file) without excessive memory usage.

`glom`'s streaming functionality revolves around a single `Iter` Specifier type, which has methods to transform the target stream.

class `glom.Iter` (*subspec=T, **kwargs*)

`Iter()` is glom's counterpart to Python's built-in `iter()` function. Given an iterable target, `Iter()` yields the result of applying the passed spec to each element of the target, similar to the built-in `[]` spec, but streaming.

The following turns a list of strings into integers using `Iter()`, before deduplicating and converting it to a tuple:

```
>>> glom(['1', '2', '1', '3'], (Iter(int), set, tuple))
(1, 2, 3)
```

`Iter()` also has many useful methods which can be chained to compose a stream processing pipeline. The above can also be written as:

```
>>> glom(['1', '2', '1', '3'], (Iter().map(int).unique(), tuple))
(1, 2, 3)
```

`Iter()` also respects glom's `SKIP` and `STOP` singletons for filtering and breaking iteration.

Parameters

- **subspec** – A subspec to be applied on each element from the iterable.
- **sentinel** – Keyword-only argument, which, when found in the iterable stream, causes the iteration to stop. Same as with the built-in `iter()`.

map (*subspec*)

Return a new `Iter()` spec which will apply the provided *subspec* to each element of the iterable.

```
>>> glom(range(5), Iter().map(lambda x: x * 2).all())
[0, 2, 4, 6, 8]
```

Because a spec can be a callable, `Iter.map()` does everything the built-in `map()` does, but with the full power of glom specs.

```
>>> glom(['a', 'B', 'C'], Iter().map(T.islower()).all())
[True, False, False]
```

filter (*key=T*)

Return a new `Iter()` spec which will include only elements matching the given *key*.

```
>>> glom(range(6), Iter().filter(lambda x: x % 2).all())
[1, 3, 5]
```

Because a spec can be a callable, `Iter.filter()` does everything the built-in `filter()` does, but with the full power of glom specs. For even more power, combine, `Iter.filter()` with `Check()`.

```
>>> # PROTIP: Python's ints know how many binary digits they require, using
↳ the bit_length method
>>> glom(range(9), Iter().filter(Check(T.bit_length(), one_of=(2, 4),
↳ default=SKIP)).all())
[2, 3, 8]
```

chunked (*size, fill=Sentinel('_MISSING')*)

Return a new `Iter()` spec which groups elements in the iterable into lists of length *size*.

If the optional *fill* argument is provided, iterables not evenly divisible by *size* will be padded out by the *fill* constant. Otherwise, the final chunk will be shorter than *size*.

```
>>> list(glom(range(10), Iter().chunked(3)))
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

(continues on next page)

(continued from previous page)

```
>>> list(glom(range(10), Iter().chunked(3, fill=None)))
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, None, None]]
```

split (*sep=None, maxsplit=None*)

Return a new *Iter()* spec which will lazily split an iterable based on a separator (or list of separators), *sep*. Like `str.split()`, but for all iterables.

`split_iter()` yields lists of non-separator values. A separator will never appear in the output.

```
>>> target = [1, 2, None, None, 3, None, 4, None]
>>> list(glom(target, Iter().split()))
[[1, 2], [3], [4]]
```

Note that `split_iter` is based on `str.split()`, so if *sep* is `None`, `split()` **groups** separators. If empty lists are desired between two contiguous `None` values, simply use `sep=[None]`:

```
>>> list(glom(target, Iter().split(sep=[None])))
[[1, 2], [], [3], [4], []]
```

A max number of splits may also be set:

```
>>> list(glom(target, Iter().split(maxsplit=2)))
[[1, 2], [3], [4, None]]
```

flatten ()

Returns a new *Iter()* instance which combines iterables into a single iterable.

```
>>> target = [[1, 2], [3, 4], [5]]
>>> list(glom(target, Iter().flatten()))
[1, 2, 3, 4, 5]
```

unique (*key=T*)

Return a new *Iter()* spec which lazily filters out duplicate values, i.e., only the first appearance of a value in a stream will be yielded.

```
>>> target = list('glOMlIcious')
>>> out = list(glom(target, Iter().unique(T.lower())))
>>> print(''.join(out))
glOMIcus
```

limit (*count*)

A convenient alias for `slice()`, which takes a single argument, *count*, the max number of items to yield.

slice (**args*)

Returns a new *Iter()* spec which trims iterables in the same manner as `itertools.islice()`.

```
>>> target = [0, 1, 2, 3, 4, 5]
>>> glom(target, Iter().slice(3).all())
[0, 1, 2]
>>> glom(target, Iter().slice(2, 4).all())
[2, 3]
```

This method accepts only positional arguments.

takewhile (*key=T*)

Returns a new *Iter()* spec which stops the stream once *key* becomes falsy.

```
>>> glom([3, 2, 0, 1], Iter().takewhile().all())
[3, 2]
```

`itertools.takewhile()` for more details.

dropwhile (*key=T*)

Returns a new `Iter()` spec which drops stream items until *key* becomes falsy.

```
>>> glom([0, 0, 3, 2, 0], Iter().dropwhile(lambda t: t < 1).all())
[3, 2, 0]
```

Note that while similar to `Iter.filter()`, the filter only applies to the beginning of the stream. In a way, `Iter.dropwhile()` can be thought of as `lstrip()` for streams. See `itertools.dropwhile()` for more details.

all ()

A convenience method which returns a new spec which turns an iterable into a list.

```
>>> glom(range(5), Iter(lambda t: t * 2).all())
[0, 2, 4, 6, 8]
```

Note that this spec will always consume the whole iterable, and as such, the spec returned is *not* an `Iter()` instance.

first (*key=T, default=None*)

A convenience method for lazily yielding a single truthy item from an iterable.

```
>>> target = [False, 1, 2, 3]
>>> glom(target, Iter().first())
1
```

This method takes a condition, *key*, which can also be a glomspec, as well as a *default*, in case nothing matches the condition.

As this spec yields at most one item, and not an iterable, the spec returned from this method is not an `Iter()` instance.

1.9 Reduction & Grouping

This document contains glom techniques for transforming a collection of data to a smaller set, otherwise known as “grouping” or “reduction”.

1.9.1 Combining iterables with Flatten and Merge

New in version 19.1.0.

Got lists of lists? Sets of tuples? A sequence of dicts (but only want one)? Do you find yourself reaching for Python’s builtin `sum()` and `reduce()`? To handle these situations and more, glom has five specifier types and two convenience functions:

glom.**flatten** (*target, **kwargs*)

At its most basic, `flatten()` turns an iterable of iterables into a single list. But it has a few arguments which give it more power:

Parameters

- **init** (*callable*) – A function or type which gives the initial value of the return. The value must support addition. Common values might be `list` (the default), `tuple`, or even `int`. You can also pass `init="lazy"` to get a generator.
- **levels** (*int*) – A positive integer representing the number of nested levels to flatten. Defaults to 1.
- **spec** – The glomspec to fetch before flattening. This defaults to the the root level of the object.

Usage is straightforward.

```
>>> target = [[1, 2], [3], [4]]
>>> flatten(target)
[1, 2, 3, 4]
```

Because integers themselves support addition, we actually have two levels of flattening possible, to get back a single integer sum:

```
>>> flatten(target, init=int, levels=2)
10
```

However, flattening a non-iterable like an integer will raise an exception:

```
>>> target = 10
>>> flatten(target)
Traceback (most recent call last):
...
FoldError: can only Flatten on iterable targets, not int type (...)
```

By default, `flatten()` will add a mix of iterables together, making it a more-robust alternative to the built-in `sum(list_of_lists, list())` trick most experienced Python programmers are familiar with using:

```
>>> list_of_iterables = [range(2), [2, 3], (4, 5)]
>>> sum(list_of_iterables, [])
Traceback (most recent call last):
...
TypeError: can only concatenate list (not "tuple") to list
```

Whereas `flatten()` handles this just fine:

```
>>> flatten(list_of_iterables)
[0, 1, 2, 3, 4, 5]
```

The `flatten()` function is a convenient wrapper around the `Flatten` specifier type. For embedding in larger specs, and more involved flattening, see `Flatten` and its base, `Fold`.

class `glom.Flatten` (*subspec=T, init=<type 'list'>*)

The `Flatten` specifier type is used to combine iterables. By default it flattens an iterable of iterables into a single list containing items from all iterables.

```
>>> target = [[1], [2, 3]]
>>> glom(target, Flatten())
[1, 2, 3]
```

You can also set `init` to `"lazy"`, which returns a generator instead of a list. Use this to avoid making extra lists and other collections during intermediate processing steps.

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

By default, `merge()` turns an iterable of mappings into a single, merged `dict`, leveraging the behavior of the `update()` method. A new mapping is created and none of the passed mappings are modified.

```
>>> target = [{'a': 'alpha'}, {'b': 'B'}, {'a': 'A'}]
>>> res = merge(target)
>>> pprint(res)
{'a': 'A', 'b': 'B'}
```

Parameters `target` – The list of dicts, or some other iterable of mappings.

The start state can be customized with the `init` keyword argument, as well as the update operation, with the `op` keyword argument. For more on those customizations, see the [Merge](#) spec.

class `glom.Merge(subspec=T, init=<type 'dict'>, op=None)`

By default, `Merge` turns an iterable of mappings into a single, merged `dict`, leveraging the behavior of the `update()` method. The start state can be customized with `init`, as well as the update operation, with `op`.

Parameters

- **subspec** – The location of the iterable of mappings. Defaults to `T`.
- **init** (*callable*) – A type or callable which returns a base instance into which all other values will be merged.
- **op** (*callable*) – A callable, which takes two arguments, and performs a merge of the second into the first. Can also be the string name of a method to fetch on the instance created from `init`. Defaults to `"update"`.

Note: Besides the differing defaults, the primary difference between [Merge](#) and other [Fold](#) subtypes is that its `op` argument is assumed to be a two-argument function which has no return value and modifies the left parameter in-place. Because the initial state is a new object created with the `init` parameter, none of the target values are modified.

class `glom.Sum(subspec=T, init=<type 'int'>)`

The `Sum` specifier type is used to aggregate integers and other numericals using addition, much like the `sum()` builtin.

```
>>> glom(range(5), Sum())
10
```

Note that this specifier takes a callable `init` parameter like its friends, so to change the start value, be sure to wrap it in a callable:

```
>>> glom(range(5), Sum(init=lambda: 5.0))
15.0
```

To “sum” lists and other iterables, see the [Flatten](#) spec. For other objects, see the [Fold](#) specifier type.

class `glom.Fold(subspec, init, op=<built-in function iadd>)`

The `Fold` specifier type is `glom`’s building block for reducing iterables in data, implementing the classic `fold` from functional programming, similar to Python’s built-in `reduce()`.

Parameters

- **subspec** – A spec representing the target to fold, which must be an iterable, or otherwise registered to ‘iterate’ (with `register()`).

- **init** (*callable*) – A function or type which will be invoked to initialize the accumulator value.
- **op** (*callable*) – A function to call on the accumulator value and every value, the result of which will become the new accumulator value. Defaults to `operator.iadd()`.

Usage is as follows:

```
>>> target = [set([1, 2]), set([3]), set([2, 4])]
>>> result = glom(target, Fold(T, init=frozenset, op=frozenset.union))
>>> result == frozenset([1, 2, 3, 4])
True
```

Note the required `spec` and `init` arguments. `op` is optional, but here must be used because the `set` and `frozenset` types do not work with addition.

While `Fold` is powerful, `Flatten` and `Sum` are subtypes with more convenient defaults for day-to-day use.

1.9.2 Exceptions

class `glom.FoldError`

Error raised when `Fold()` is called on non-iterable targets, and possibly other uses in the future.

1.10 Matching & Validation

New in version 20.7.0.

Sometimes you want to confirm that your target data matches your code's assumptions. With `glom`, you don't need a separate validation step, you can do these checks inline with your `glom` spec, using `Match` and friends.

Contents

- *Validation with Match*
- *Optional and required dict key matching*
- *M Expressions*
- *Boolean operators and matching*
- *String matching*
- *Control flow with Switch*
- *Exceptions*
- *Validation with Check*

1.10.1 Validation with Match

For matching whole data structures, use a `Match` spec.

class `glom.Match` (*spec, default=Sentinel('_MISSING')*)

`glom`'s `Match` specifier type enables a new mode of `glom` usage: pattern matching. In particular, this mode has been designed for nested data validation.

Pattern specs are evaluated as follows:

1. Spec instances are always evaluated first
2. Types match instances of that type
3. Instances of `dict`, `list`, `tuple`, `set`, and `frozenset` are matched recursively
4. Any other values are compared for equality to the target with `==`

By itself, this allows to assert that structures match certain patterns, and may be especially familiar to users of the `schema` library.

For example, let's load some data:

```
>>> target = [
... {'id': 1, 'email': 'alice@example.com'},
... {'id': 2, 'email': 'bob@example.com'}]
```

A `Match` pattern can be used to ensure this data is in its expected form:

```
>>> spec = Match({'id': int, 'email': str})
```

This spec succinctly describes our data structure's pattern. Specifically, a `list` of `dict` objects, each of which has exactly two keys, 'id' and 'email', whose values are an `int` and `str`, respectively. Now, `glom()` will ensure our `target` matches our pattern `spec`:

```
>>> result = glom(target, spec)
>>> assert result == \
... [{'id': 1, 'email': 'alice@example.com'}, {'id': 2, 'email': 'bob@example.com'}]
```

With a more complex `Match` spec, we can be more precise:

```
>>> spec = Match({'id': And(M > 0, int), 'email': Regex('[^@]+@[^@]+')})
```

`And` allows multiple conditions to be applied. `Regex` evaluates the regular expression against the target value under the 'email' key. In this case, we take a simple approach: an email has exactly one @, with at least one character before and after.

Finally, `M` is our stand-in for the current target we're matching against, allowing us to perform in-line comparisons using Python's native greater-than operator (as well as others). We apply our `Match` pattern as before:

```
>>> assert glom(target, spec) == \
... [{'id': 1, 'email': 'alice@example.com'}, {'id': 2, 'email': 'bob@example.com'}]
```

And as usual, upon a successful match, we get the matched result.

Note: For Python 3.6+ where dictionaries are ordered, keys in the target are matched against keys in the spec in their insertion order.

Parameters

- **spec** – The glomspec representing the pattern to match data against.
- **default** – The default value to be returned if a match fails. If not set, a match failure will raise a `MatchError`.

matches (*target*)

A convenience method on a *Match* instance, returns `True` if the *target* matches, `False` if not.

```
>>> Match(int).matches(-1.0)
False
```

Parameters *target* – Target value or data structure to match against.

verify (*target*)

A convenience function a *Match* instance which returns the matched value when *target* matches, or raises a *MatchError* when it does not.

Parameters *target* – Target value or data structure to match against.

Raises *glom.MatchError*

1.10.2 Optional and required dict key matching

Note that our four *Match* rules above imply that *object* is a match-anything pattern. Because `isinstance(val, object)` is true for all values in Python, *object* is a useful stopping case. For instance, if we wanted to extend an example above to allow additional keys and values in the user dict above we could add *object* as a generic pass through:

```
>>> target = [{'id': 1, 'email': 'alice@example.com', 'extra': 'val'}]
>>> spec = Match({'id': int, 'email': str, object: object})
>>> assert glom(target, spec) == \
... [{'id': 1, 'email': 'alice@example.com', 'extra': 'val'}]
True
```

The fact that `{object: object}` will match any dictionary exposes the subtlety in *Match* dictionary evaluation.

By default, value match keys are required, and other keys are optional. For example, 'id' and 'email' above are required because they are matched via `==`. If either was not present, it would raise `class:~glom.MatchError`. `class:object` however is matched with `func:isinstance()`. Since it is not an value-match comparison, it is not required.

This default behavior can be modified with *Required* and *Optional*.

class *glom.Optional* (*key*, *default*=*Sentinel('_MISSING')*)

Used as a *dict* key in a *Match()* spec, marks that a value match key which would otherwise be required is optional and should not raise *MatchError* even if no keys match.

For example:

```
>>> spec = Match({Optional("name"): str})
>>> glom({"name": "alice"}, spec)
{'name': 'alice'}
>>> glom({}, spec)
{}
>>> spec = Match({Optional("name", default=""): str})
>>> glom({}, spec)
{'name': ''}
```

class *glom.Required* (*key*)

Used as a *dict* key in *Match()* mode, marks that a key which might otherwise not be required should raise *MatchError* if the key in the target does not match.

For example:

```
>>> spec = Match({object: object})
```

This spec will match any dict, because `object` is the base type of every object:

```
>>> glom({}, spec)
{}
```

`{}` will also match because match mode does not require at least one match by default. If we want to require that a key matches, we can use `Required`:

```
>>> spec = Match({Required(object): object})
>>> glom({}, spec)
Traceback (most recent call last):
...
MatchError: error raised while processing.
  Target-spec trace, with error detail (most recent last):
  - Target: {}
  - Spec: Match({Required(object): <type 'object'>})
  - Spec: {Required(object): <type 'object'>}
MatchError: target missing expected keys Required(object)
```

Now our spec requires at least one key of any type. You can refine the spec by putting more specific subpatterns inside of `Required`.

1.10.3 `M` Expressions

The most concise way to express validation and guards.

`glom.M = M`

`M` is similar to `T`, a stand-in for the current target, but where `T` allows for attribute and key access and method calls, `M` allows for comparison operators.

If a comparison succeeds, the target is returned unchanged. If a comparison fails, `MatchError` is thrown.

Some examples:

```
>>> glom(1, M > 0)
1
>>> glom(0, M == 0)
0
>>> glom('a', M != 'b') == 'a'
True
```

`M` by itself evaluates the current target for truthiness. For example, `M | Val(None)` is a simple idiom for normalizing all falsey values to `None`:

```
>>> from glom import Val
>>> glom([0, False, "", None], [M | Val(None)])
[None, None, None, None]
```

For convenience, `&` and `|` operators are overloaded to construct `And` and `Or` instances.

```
>>> glom(1.0, (M > 0) & float)
1.0
```

Note: Python's operator overloading may make for concise code, but it has its limits.

Because bitwise operators (& and |) have higher precedence than comparison operators (>, <, etc.), expressions must be parenthesized.

```
>>> M > 0 & float
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for &: 'int' and 'type'
```

Similarly, because of special handling around ternary comparisons ($1 < M < 5$) are implemented via short-circuiting evaluation, they also cannot be captured by *M*.

1.10.4 Boolean operators and matching

While *M* is an easy way to construct expressions, sometimes a more object-oriented approach can be more suitable.

class `glom.Or` (*children, **kw)

Tries to apply the first child spec to the target, and return the result. If *GlomError* is raised, try the next child spec until there are no all child specs have been tried, then raise *MatchError*.

class `glom.And` (*children, **kw)

Applies child specs one after the other to the target; if none of the specs raises *GlomError*, returns the last result.

class `glom.Not` (child)

Inverts the *child*. Child spec will be expected to raise *GlomError* (or subtype), in which case the target will be returned.

If the child spec does not raise *GlomError*, *MatchError* will be raised.

1.10.5 String matching

class `glom.Regex` (pattern, flags=0, func=None)

checks that target is a string which matches the passed regex pattern

raises *MatchError* if there isn't a match; returns Target if match

variables captures in regex are added to the scope so they can be used by downstream processes

1.10.6 Control flow with Switch

Match becomes even more powerful when combined with the ability to branch spec execution.

class `glom.Switch` (cases, default=Sentinel('_MISSING'))

The *Switch* specifier type routes data processing based on matching keys, much like the classic switch statement.

Here is a spec which differentiates between lowercase English vowel and consonant characters:

```
>>> switch_spec = Match(Switch([(Or('a', 'e', 'i', 'o', 'u'), Val('vowel')),
...                               (And(str, M, M(T[2:]) == ''), Val('consonant'))]))
```

The constructor accepts a dict of {keyspec: valspec} or a list of items, [(keyspec, valspec)]. Keys are tried against the current target in order. If a keyspec raises *GlomError*, the next keyspec is tried. Once a keyspec succeeds, the corresponding valspec is evaluated and returned. Let's try it out:

```
>>> glom('a', switch_spec)
'vowel'
>>> glom('z', switch_spec)
'consonant'
```

If no keyspec succeeds, a `MatchError` is raised. Our spec only works on characters (strings of length 1). Let's try a non-character, the integer 3:

```
>>> glom(3, switch_spec)
Traceback (most recent call last):
...
glom.matching.MatchError: error raised while processing, details below.
  Target-spec trace (most recent last):
  - Target: 3
  - Spec: Match(Switch([(Or('a', 'e', 'i', 'o', 'u'), Val('vowel'))), (And(str, M,
  ↳ (M(T[2:]) == '')), Val('...
  + Spec: Switch([(Or('a', 'e', 'i', 'o', 'u'), Val('vowel'))), (And(str, M,
  ↳ (M(T[2:]) == '')), Val('conson...
  ↳ \ Spec: Or('a', 'e', 'i', 'o', 'u')
  ↳ \ \ Spec: 'a'
  ↳ \ \ X glom.matching.MatchError: 3 does not match 'a'
  ↳ \ \ Spec: 'e'
  ↳ \ \ X glom.matching.MatchError: 3 does not match 'e'
  ↳ \ \ Spec: 'i'
  ↳ \ \ X glom.matching.MatchError: 3 does not match 'i'
  ↳ \ \ Spec: 'o'
  ↳ \ \ X glom.matching.MatchError: 3 does not match 'o'
  ↳ \ \ Spec: 'u'
  ↳ \ \ X glom.matching.MatchError: 3 does not match 'u'
  ↳ \ X glom.matching.MatchError: 3 does not match 'u'
  ↳ \ \ Spec: And(str, M, (M(T[2:]) == ''))
  ↳ \ \ Spec: str
  ↳ \ X glom.matching.TypeMatchError: expected type str, not int
glom.matching.MatchError: no matches for target in Switch
```

Note: `Switch` is one of several *branching* specifier types in glom. See “[Reading Branched Exceptions](#)” for details on interpreting its exception messages.

A *default* value can be passed to the spec to be returned instead of raising a `MatchError`.

Note: `Switch` implements control flow similar to the switch statement proposed in [PEP622](#).

1.10.7 Exceptions

class `glom.MatchError` (*fmt*, **args*)
 Raised when a `Match` or `M` check fails.

```
>>> glom({123: 'a'}, Match({'id': int}))
Traceback (most recent call last):
...
MatchError: key 123 didn't match any of ['id']
```

class `glom.TypeMatchError` (*actual, expected*)
MatchError subtype raised when a *Match* fails a type check.

```
>>> glom({'id': 'a'}, Match({'id': int}))
Traceback (most recent call last):
...
TypeMatchError: error raised while processing.
Target-spec trace, with error detail (most recent last):
- Target: {'id': 'a'}
- Spec: Match({'id': <type 'int'>})
- Spec: {'id': <type 'int'>}
- Target: 'a'
- Spec: int
TypeMatchError: expected type int, not str
```

1.10.8 Validation with Check

Warning: Given the suite of tools introduced with *Match*, the *Check* specifier type may be deprecated in a future release.

class `glom.Check` (*spec=T, **kwargs*)

Check objects are used to make assertions about the target data, and either pass through the data or raise exceptions if there is a problem.

If any check condition fails, a *CheckError* is raised.

Parameters

- **spec** – a sub-spec to extract the data to which other assertions will be checked (defaults to applying checks to the target itself)
- **type** – a type or sequence of types to be checked for exact match
- **equal_to** – a value to be checked for equality match (“==”)
- **validate** – a callable or list of callables, each representing a check condition. If one or more return False or raise an exception, the Check will fail.
- **instance_of** – a type or sequence of types to be checked with `isinstance()`
- **one_of** – an iterable of values, any of which can match the target (“in”)
- **default** – an optional default value to replace the value when the check fails (if default is not specified, *GlomCheckError* will be raised)

Aside from *spec*, all arguments are keyword arguments. Each argument, except for *default*, represent a check condition. Multiple checks can be passed, and if all check conditions are left unset, Check defaults to performing a basic truthy check on the value.

class `glom.CheckError` (*msgs, check, path*)

This *GlomError* subtype is raised when target data fails to pass a *Check*’s specified validation.

An uncaught *CheckError* looks like this:

```
>>> target = {'a': {'b': 'c'}}
>>> glom(target, {'b': ('a.b', Check(type=int))})
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
CheckError: target at path ['a.b'] failed check, got error: "expected type to be
↳ 'int', found type 'str'"
```

If the `Check` contains more than one condition, there may be more than one error message. The string rendition of the `CheckError` will include all messages.

You can also catch the `CheckError` and programmatically access messages through the `msgs` attribute on the `CheckError` instance.

1.11 Exceptions & Debugging

While `glom` works well when all goes as intended, it even shines when data doesn't match expectations. `glom`'s error messages and exception hierarchy have been designed to maximize readability and debuggability. Read on for a listing of `glom`'s exceptions and how to debug them.

Contents

- [Exceptions](#)
- [Reading a glom Exception](#)
- [Reading Branched Exceptions](#)
- [Debugging](#)

1.11.1 Exceptions

`glom` introduces a several 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.

Here is a short list of links to all public exception types in `glom`.

- [GlomError](#)
- [PathAccessError](#)
- [PathAssignError](#)
- [PathDeleteError](#)
- [CoalesceError](#)
- [FoldError](#)
- [MatchError](#)
- [TypeMatchError](#)
- [CheckError](#)
- [UnregisteredTarget](#)
- [BadSpec](#)

1.11.2 Reading a glom Exception

`glom` errors are regular Python exceptions, but may look a little different from other Python errors. Because `glom` is a data manipulation library, `glom` errors include a data traceback, interleaving `spec` and `target` data.

For example, let's raise an error by glomming up some data that doesn't exist:

```
1 >>> target = {'planets': [{'name': 'earth', 'moons': 1}]}
2 >>> glom(target, ('planets', ['rings']))
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "/home/mahmoud/projects/glom/glom/core.py", line 1787, in glom
6     raise err
7 glom.core.PathAccessError: error raised while processing, details below.
8   Target-spec trace (most recent last):
9   - Target: {'planets': [{'name': 'earth', 'moons': 1}]}
10  - Spec: ('planets', ['rings'])
11  - Spec: 'planets'
12  - Target: [{'name': 'earth', 'moons': 1}]
13  - Spec: ['rings']
14  - Target: {'name': 'earth', 'moons': 1}
15  - Spec: 'rings'
16 glom.core.PathAccessError: could not access 'rings', part 0 of Path('rings'), got_
   ↪error: KeyError('rings')
```

Let's step through this output:

- Line 1: We created a planet registry, similar to the one in the *glom Tutorial*.
- Line 2-3: We try to get a listing of rings of all the planets. Instead, we get a Python traceback.
- Line 7: We see we have a *PathAccessError*.
- Line 8-9: The “target-spec trace”, our data stack, begins. It always starts with the target data as it was passed in.
- Line 10: Next is the top-level spec, as passed in: ('planets', ['rings'])
- Line 11: glom takes the first part of the spec from line 9, 'planets', to get the next target.
- Line 12: Because the spec on line 11 updated the current target, glom outputs it. When a spec is evaluated but the target value is unchanged, the target is skipped in the trace.
- Line 14-15: We get to the last two lines, which include the culprit target and spec
- Line 16: Finally, our familiar *PathAccessError* message, with more details about the error, including the original *KeyError*('rings').

This view of glom evaluation answers many of the questions a developer or user would ask upon encountering the error:

- What was the data?
- Which part of the spec failed?
- What was the original error?

The data trace does this by peeling away at the target and spec until it hones in on the failure. Both targets and specs in traces are truncated to terminal width to maximize readability.

Note: If for some reason you need the full Python stack instead of the glom data traceback, pass `glom_debug=True` to the top-level glom call.

1.11.3 Reading Branched Exceptions

Some glom spec types, like *Coalesce* and *Switch*, can try multiple specs in succession. These “branching” specs can also get multiple exceptions.

Initially, debugging data for these branching specs was limited. But in v20.7.0, branching error trees were introduced, exposing information about every spec and target attempted before raising the final exception.

All the exception reading advice in the “*Reading a glom Exception*” section applies, but there’s a bit of extra formatting to visualize the error tree in the target-spec trace.

Let’s step line by line through a *Coalesce* error tree:

```

1 >>> target = {'n': 'nope', 'xxx': {'z': {'v': 0}}}
2 >>> glom(target, Coalesce(('xxx', 'z', 'n'), 'yyy'))
3 Traceback (most recent call last):
4   File "tmp.py", line 9, in _make_stack
5     glom(target, spec)
6   File "/home/mahmoud/projects/glom/glom/core.py", line 2029, in glom
7     raise err
8 glom.core.CoalesceError: error raised while processing, details below.
9   Target-spec trace (most recent last):
10  - Target: {'n': 'nope', 'xxx': {'z': {'v': 0}}}
11  + Spec: Coalesce(('xxx', 'z', 'n'), 'yyy')
12  | \ Spec: ('xxx', 'z', 'n')
13  || Spec: 'xxx'
14  || Target: {'z': {'v': 0}}
15  || Spec: 'z'
16  || Target: {'v': 0}
17  || Spec: 'n'
18  |X glom.core.PathAccessError: could not access 'n', part 0 of Path('n'), got error:
↳KeyError('n')
19  | \ Spec: 'yyy'
20  |X glom.core.PathAccessError: could not access 'yyy', part 0 of Path('yyy'), got
↳error: KeyError('yyy')
21  glom.core.CoalesceError: no valid values found. Tried (('xxx', 'z', 'n'), 'yyy') and
↳got (PathAccessError, PathAccessError) (at path ['xxx', 'z'])

```

- Line 1-10: Standard fare for glom use and error behavior, see “*Reading a glom Exception*”
- Line 11: We see a “+” when starting a branching spec. Each level of branch adds a “|” on the left to help track nesting level.
- Line 12: We see a “\” indicating a new branch of the root branching spec.
- Line 13-17: Traversing downward as usual until...
- Line 18: We see an “X” indicating our first exception, causing the failure of this branch.
- Line 19: We see a “\” which starts our next branch.
- Line 20: We see an “X” indicating our second and last exception, causing the failure of this branch.
- Line 21: The last line is our root level exception, dedented, same as any other glom error.

Apart from the formatting, error branching doesn’t change any other semantics of the glom exception being raised.

1.11.4 Debugging

Good error messages are great when the data has a problem, but what about when a spec is incorrect?

Even the most carefully-constructed specifications 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.

1.12 Writing a custom Specifier Type

While glom comes with a lot of built-in features, no library can ever encompass all data manipulation operations.

To cover every case out there, glom provides a way to extend its functionality with your own data handling hooks. This document explains glom's execution model and how to integrate with it when writing a custom Specifier Type.

1.12.1 When to write a Specifier Type

glom has always supported arbitrary callables, like so:

```
glom({'nums': range(5)}, ('nums', sum))
# 10
```

With this built-in extensibility, what does a glom specifier type add?

Custom specifier types are useful when you want to:

1. Perform validation at spec construction time
2. Enable users to interact with new target types and operations
3. Improve readability and reusability of your data transformations
4. Temporarily change the glom runtime behavior

If you're just building a one-off spec for transforming your own data, there's no reason to reach for an extension. glom's extension API is easy, but a good old Python `lambda` is even easier.

1.12.2 Building your Specifier Type

Any object instance with a `glomit` method can participate in a glom call. By way of example, here is a programming cliché implemented as a glom specifier type, with comments referencing notes below.

```
class HelloWorldSpec(object): # 1
    def glomit(self, target, scope): # 2
        print("Hello, world!")
        return target
```

And now let's put it to use!

```
from glom import glom

target = {'example': 'object'}

glom(target, HelloWorldSpec()) # 3
# prints "Hello, world!" and returns target
```

There are a few things to note from this example:

1. Specifier types do not need to inherit from any type. Just implement the `glomit` method.
2. The `glomit` signature takes two parameters, `target` and `scope`. The `target` should be familiar from using `glom()`, and it's the `scope` that makes glom really tick.
3. By convention, instances are used in specs passed to `glom()` calls, not the types themselves.

1.12.3 The glom Scope

The *glom scope* is also used to expose runtime state to the specifier type. Let's take a look inside a scope:

```
from glom import glom
from pprint import pprint

class ScopeInspectorSpec(object):
    def glomit(self, target, scope):
        pprint(dict(scope))
        return target

glom(target, ScopeInspectorSpec())
```

Which gives us:

```
{T: {'example': 'object'},
 <function glom at 0x7f208984d140>: <function _glom at 0x7f208984d5f0>,
 <class 'glom.core.Path'>: [],
 <class 'glom.core.Spec'>: <__main__.ScopeInspectorSpec object at 0x7f208bf58690>,
 <class 'glom.core.Inspect'>: None,
 <class 'glom.core.TargetRegistry'>: <glom.core.TargetRegistry object at 0x7f208984b4d0>}
```

As you can see, all glom’s core workings are present, all under familiar keys:

- The current *target*, accessible using *T* as a scope key.
- The current *spec*, accessible under *Spec*.
- The current *path*, accessible under *Path*.
- The TargetRegistry, used to *register new operations and target types*.
- Even the `glom()` function itself, filed under `glom()`.

To learn how to use the scope’s powerful features idiomatically, let’s reimplement at one of glom’s standard specifier types.

1.12.4 Specifiers by example

While we’ve technically created a couple of extensions above, let’s really dig into the features of the scope using an example.

`Sum` is a standard extension that ships with glom, and it works like this:

```
from glom import glom, Sum

glom([1, 2, 3], Sum())
# 6
```

The version below does not have as much error handling, but reproduces all the same basic principles. This version of `Sum()` code also contains comments with references to explanatory notes below.

```
from glom import glom, Path, T
from glom.core import TargetRegistry, UnregisteredTarget # 1

class Sum(object):
    def __init__(self, subspec=T, init=int): # 2
        self.subspec = subspec
        self.init = init

    def glomit(self, target, scope):
        if self.subspec is not T:
            target = scope[glom](target, self.subspec, scope) # 3

        try:
            # 4
            iterate = scope[TargetRegistry].get_handler('iterate', target,
↳path=scope[Path])
        except UnregisteredTarget as ut:
            # 5
            raise TypeError('can only %s on iterable targets, not %s type (%s)'
                            % (self.__class__.__name__, type(target).__name__, ut))
```

(continues on next page)

(continued from previous page)

```

    try:
        iterator = iterate(target)
    except Exception as e:
        raise TypeError('failed to iterate on instance of type %r at %r (got %r)'
                        % (target.__class__.__name__, Path(*scope[Path]), e))

    return self._sum(iterator)

def _sum(self, iterator): # 6
    ret = self.init()

    for v in iterator:
        ret += v

    return ret

```

Now, let's take a look at the interesting parts, referencing the comments above:

1. Specifier types often reference the `TargetRegistry`, which is not part of the top-level `glom` API, and must be imported from `glom.core`. More on this in #4.
2. Specifier type `__init__` methods may take as many or as few arguments as desired, but many `glom` specifier types take a first parameter of a *subspec*, meant to be fetched right before the actual specifier's operation. This helps readability of `glomspecs`. See *Coalesce* for an example of this idiom.
3. Specifier types should not reference the `glom()` function directly, instead use the `glom()` function as a key to the `scope` map to get the currently active `glom()`. This ensures that the extension type is compatible with advanced specifier types which override the `glom()` function.
4. To maximize compatibility with new target types, `glom` allows *new types and operations to be registered* with the `TargetRegistry`. Specifier types should respect this by contextually fetching these standard operators as demonstrated above. At the time of writing, three primary operators are used by `glom` itself, "get", "iterate", and "assign".
5. In the event that the current target does not support your Specifier type's desired operation, it's customary to raise a helpful error. Consider creating your own exception type and inheriting from `GlomError`.
6. Specifier types may have other methods and members in addition to the primary `glomit()` method. This `_sum()` method implements most of the core of our custom specifier type.

Check out the implementation of the real `glom.Sum()` specifier for more details.

1.12.5 Summing up

`glom` Specifier Types are more than just add-ons; the extension architecture is how most of `glom` itself is implemented. Build knowing that the paradigm is as powerful as anything built-in.

If you need more examples, another simple one can be found in *this snippet*. `glom`'s source code itself contains many specifiers more advanced than the above. Simply search the codebase for `glomit()` methods and you will find no shortage.

Happy extending!

1.13 glom Modes

Note: Be sure to read “*Writing a custom Specifier Type*” before diving into the deep details below.

A glom “mode” determines how Python built-in data structures are evaluated. Think of it like a dialect for how `dict`, `tuple`, `list`, etc., are interpreted in a spec. Modes do not change the behavior of `T`, or many other core specifiers. Modes are one of the keys to keeping glom specs short and readable.

A mode is used similar to a spec: whatever Python data structure is passed to the mode type constructor will be evaluated under that mode. Once set, the mode remains in place until it is overridden by another mode.

glom only has a few modes:

1. `Auto` - The default glom behavior, used for data transformation, with the spec acting as a template.
2. `Fill` - A variant of the default transformation behavior; preferring to “fill” containers instead of iterating, chaining, etc.
3. `Match` - Treats the spec as a pattern, checking that the target matches.

Adding a new mode is relatively rare, but when it comes up this document includes relevant details.

1.13.1 Writing custom Modes

A mode is a spec which sets `scope[MODE]` to a function which accepts `target`, `spec`, and `scope` and returns a result, a signature very similar to the top-level `glom()` method itself.

For example, here is an abbreviated version of the `Fill` mode:

```
class Fill(object):
    def __init__(self, spec):
        self.spec = spec

    def glomit(self, target, scope):
        scope[MODE] = _fill
        return scope[glom](target, self.spec, scope)

def _fill(target, spec, scope):
    recurse = lambda val: scope[glom](target, val, scope)
    if type(spec) is dict:
        return {recurse(key): recurse(val)
                for key, val in spec.items()}
    if type(spec) in (list, tuple, set, frozenset):
        result = [recurse(val) for val in spec]
        if type(spec) is list:
            return result
        return type(spec)(result)
    if callable(spec):
        return spec(target)
    return spec
```

Like any other *Specifier Type*, `Fill` has a `glomit()` method, and this method sets the `MODE` key in the *glom scope* to our `_fill` function. The name itself doesn’t matter, but the signature must match exactly: `(target, spec, scope)`.

As mentioned above, custom modes are relatively rare for glom. If you write one, [let us know!](#)

g

`glom.core`, 21
`glom.matching`, 42
`glom.mutation`, 34
`glom.streaming`, 36
`glom.tutorial`, 4

A

all() (glom.Iter method), 39
And (class in glom), 46
Assign (class in glom), 34
assign() (in module glom), 34

B

BadSpec (class in glom), 33

C

Call (class in glom), 29
Check (class in glom), 48
CheckError (class in glom), 48
chunked() (glom.Iter method), 37
Coalesce (class in glom), 25
CoalesceError (class in glom), 32
constants() (glom.Invoke method), 28

D

Delete (class in glom), 35
delete() (in module glom), 35
dropwhile() (glom.Iter method), 39

F

filter() (glom.Iter method), 37
first() (glom.Iter method), 39
Flatten (class in glom), 40
flatten() (glom.Iter method), 38
flatten() (in module glom), 39
Fold (class in glom), 41
FoldError (class in glom), 42

G

glom() (in module glom), 22
glom.core (module), 21
glom.matching (module), 42
glom.mutation (module), 34
glom.streaming (module), 36
glom.tutorial (module), 4

GlomError (class in glom), 33
Glommer (class in glom), 33

I

Inspect (class in glom), 52
Invoke (class in glom), 27
Iter (class in glom), 36

L

limit() (glom.Iter method), 38

M

M (in module glom), 45
map() (glom.Iter method), 37
Match (class in glom), 42
MatchError (class in glom), 47
matches() (glom.Match method), 43
Merge (class in glom), 41
merge() (in module glom), 40

N

Not (class in glom), 46

O

Optional (class in glom), 44
Or (class in glom), 46

P

Path (class in glom), 23
PathAccessError (class in glom), 31
PathAssignError (class in glom), 36
PathDeleteError (class in glom), 36

R

Ref (class in glom), 30
Regex (class in glom), 46
register() (in module glom), 33
Required (class in glom), 44

S

SKIP (in module glom), 26
slice() (glom.Iter method), 38
Spec (class in glom), 24
specfunc() (glom.Invoke class method), 28
specs() (glom.Invoke method), 28
split() (glom.Iter method), 38
star() (glom.Invoke method), 29
STOP (in module glom), 27
Sum (class in glom), 41
Switch (class in glom), 46

T

T (in module glom), 24
takewhile() (glom.Iter method), 38
TypeMatchError (class in glom), 47

U

unique() (glom.Iter method), 38
UnregisteredTarget (class in glom), 32

V

Val (class in glom), 23
Vars (class in glom), 31
verify() (glom.Match method), 44