spy

Release v0.3.3

Mar 15, 2019

Contents

spy is a CLI and API for processing streams in Python.

CHAPTER 1

Contents

1.1 Introduction

spy is a Python CLI. It's quite powerful, as you'll see below, but let's start with the basics: you feed it a Python expression, it spits out the result.

```
$ spy '3*4'
12
```

There's no need to import modules—just use them and spy will make sure they're available:

```
$ spy 'math.pi'
3.141592653589793
```

1.1.1 I/O

Standard input is exposed as a file-like object called `pipe`:

```
$ cat test.txt
this
file
has
five
lines
$ spy 'pipe.readline()' < test.txt
this
```

It's a `io.TextIOBase`, with a couple of extra features: You can index into it, or convert all of stdin into a string with `str()`.

```
$ spy 'pipe[1]' < test.txt
file
```

(continues on next page)

(continued from previous page)

```
$ spy 'pipe[1::2]' < test.txt
['file', 'five']
$ spy 'str(pipe).replace("\n", " ")' < test.txt
this file has five lines
```

Passing *-l* (or *--each-line*) to spy will iterate through stdin instead, so your expressions will run once per line of input:

```
$ spy -l '"-%s-" % pipe' < test.txt
-this-
-file-
-has-
-five-
-lines-
```

spy helpfully removes the terminating newlines from these strings. If you don't want that, you can pass *--raw* to get `stdin` unadulterated.

```
$ spy -lrc repr < test.txt
'this\n'
'file\n'
'has\n'
'five\n'
'lines\n'
```

1.1.2 Piping

Much like the standard assortment of unix utilities, which expect to have their inputs and outputs wired up to each other in order to do useful things, each fragment processes some data then passes it on to the next one.

Data passes from left to right. Fragments can return the special constant *spy.DROP* to prevent further processing of the current datum and continue to the next.

```
$ spy '3' 'pipe * 2' 'pipe * "!"'
!!!!!!
$ spy -l 'if pipe.startswith("f"): pipe = spy.DROP' < test.txt
this
has
lines
```

1.1.3 Limiting output

--start=<integer>, **-s** <integer>
: Start printing output at this zero-based index.

--end=<integer>, **-e** <integer>
: Stop processing at this zero-based index.

`-s` and `-e` mirror Python's slice semantics, so `-s 1 -e 3` will show results 1 and 2. This means `-e` on its own is equivalent to a limit on the number of results.

Once the result specified by `-e` has been hit, no more data will be processed.

1.1.4 Data flow

Before we construct anything more complex, a brief discourse into how data moves around in spy: Each fragment in spy tries to consume data from the fragment to its left. It processes it, then yields to the fragment to its right, which will do the same thing. To run the program, spy just tries to pump as much data out of the rightmost fragment as it can—everything else is handled by the fragment mechanic.

In the examples I've given above, each fragment has consumed and yielded data on a one-to-one basis, but there's no inherent reason for that restriction. Fragments can yield or consume (or both) multiple values using *spy.many* and *spy.collect*, respectively.

1.1.5 Decorators

In one example above, we used an `if` statement to filter by a predicate. That's far from elegant—by my rough guess, about half the characters in the fragment are boilerplate. spy provides some function decorators to avoid repeating this and a few other common constructs—they're available as flags from the CLI:

--accumulate <fragment>, **-a** <fragment>
: passes the the result of *spy.collect()* to the fragment.

--callable <fragment>, **-c** <fragment>
: calls whatever the following fragment returns, with a single argument: the input value to the fragment.

--filter <fragment>, **-f** <fragment>
: filters the data stream, using the fragment as a predicate: if it returns any true value, the data passes through, but if it returns a false value *spy.DROP* is returned instead.

--many <fragment>, **-m** <fragment>
: calls *spy.many()* with the return value of the fragment (which must be iterable).

1.1.6 Deferred application

spy overloads callable objects (when they're builtins or autoimported) to add implementations of most Python operators. These return a function that calls the original function and then applies the specified operation. They take a single argument only, and are essentially just a shortcut that lets you avoid typing `(pipe)` in some cases:

```
$ spy '[1,2,3]' -c 'sum/2'
3.0
$ spy '[1,2,3]' -c 'sum/len'
2.0
```

1.1.7 Doing stuff

Nothing here is particularly useful in isolation. Let's throw it all together by pretending we're `jq`:

```
$ spy -lc json.loads -f 'pipe["state"] == "Outbreak"' 'pipe["name"]' -e 10 < stations.
↪jsonl
Burbank Gateway
Stefanyshyn-Piper Port
Cheli Station
Buckell Ring
Bolotov Port
Schumacher Hub
Huss Station
```

(continues on next page)

(continued from previous page)

```
Wilhelm von Struve Port
Wedge Terminal
Orsini Mining Platform
```

Note how `-l` trivially gives us newline-delimited JSON, a job which was previously so hard it required its own top-2000 PyPI package!

1.1.8 Exception handling

If your code raises an uncaught exception, spy will try to intercept and reformat the traceback, omitting the frames from spy's own machinery. Special frames will be inserted where appropriate describing the fragment's position, source code, and input data at the time the exception was raised:

```
$ spy 'None + 2'
Traceback (most recent call last):
  Fragment 1
    None + 2
    input to fragment was <SpyFile stream='<stdin>'>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

If an exception is raised in a decorator outside the call to the fragment body, the fragment is mentioned anyway. This is not strictly true, given that none of the code in the fragment takes part in the call stack in this case, but this particular lie is almost universally more useful:

```
$ spy -c None
Traceback (most recent call last):
  Fragment 1, in decorator spy.decorators.callable
    --callable 'None'
    input to fragment was <SpyFile stream='<stdin>'>
TypeError: 'NoneType' object is not callable
```

The philosophy here is that what made it go wrong is more interesting than *exactly how* it went wrong, so that's what spy gives you by default. You can get the real traceback by passing `--no-exception-handling` to spy.

1.2 CLI reference

Contents

1.2.1 Regular options

--each-line, -l
Process each line as its own string (rather than stdin as a file at once).

Equivalent to starting with a fragment of `spy.many(pipe)`, but more efficient since we don't need save the contents of the input stream for indexing.

--no-default-fragments
Don't add any fragments to the chain that weren't explicitly specified in the command line.

--no-exception-handling
Disable spy's exception handling and reformatting. This is mostly only useful for debugging changes to spy itself.

--pipe-name=<name>
Name the magic pipe variable <name> instead of `pipe`.

--prelude=<statement>, **-p** <statement>
Run some Python before processing starts.

--raw, -r
Don't wrap `stdin` before passing it to the first fragment.

Output limiting options

The index arguments for these options refer to results, not input. If a single piece of input data results in 4 separate pieces of output, they'll all count.

--start=<index>, **-s** <index>
Start printing results at this zero-based index.

--end=<index>, **-e** <index>
Stop processing data at this zero-based index.

1.2.2 Decorators

Decorator options must precede a code step. Multiple decorators can stack together. They have exactly the same effect as decorating a function in Python.

See the decorator *API docs* for a list of them.

1.2.3 Alternative actions

--help, -h
Show usage and option descriptions.

--show-fragments
Print out a list of string representations of the complete fragment chain that would be executed.

1.3 spy from Python

The *introduction* showed how to use spy from the command line. That's not the only way: spy works just as well from other Python code. The CLI is just a wrapper around spy's public API to make it easier to get to.

I don't think it is useful in very many cases as a Python library, but if you want to create an alternative command-line interface for example, this may be of interest.

API documentation is available. What follows is a (very) brief guide, which I hope to expand on in the future.

1.3.1 Basic usage

As with the CLI, you create fragments and then pass data through them. And, as with the CLI, creating fragments is easy. You decorate a regular function with *spy.fragment()*:

```
import spy

@spy.fragment
def add_five(v):
    return v + 5
```

So, on to the feeding data part. You don't feed data to fragments on their own, but to chains, so let's create one:

```
chain = spy.chain([add_five])
```

In order to feed data into it, call the chain object with an iterable to feed into the chain. The call will return an iterable of the results:

```
data = [1, 2, 3, 4]
print(list(chain(data)))  # [6, 7, 8, 9]
```

These iterators don't interfere with each other, even if they're created by the same chain object, so one chain can be used to process multiple independent sets of input data.

1.3.2 Differences from the CLI

As documented, *collect()* takes a `context` argument. It can be omitted when using the CLI because it's automatically filled in (it has to be, since there's no way to access the context object from CLI fragments). There is no equivalent mechanism outside the CLI, so if you want to use *collect()*, you must provide `context`. You can get the context object by accepting a `context` argument in your fragment function:

```
@spy.fragment
def foo(v, context):
   c = spy.collect(context)
   # do stuff with c
```

1.4 API

Detailed documentation for spy's API.

1.4.1 spy

This module exposes spy's core API.

See also:

spy.decorators Function decorators for use with spy fragments

Constants

spy.**DROP**
A signaling object: when returned from a *fragment*, the fragment will not yield any value.

Exceptions

exception spy.**CaughtException**

print_traceback()
Print the (formatted) traceback captured when the exception was raised.

Functions

class spy.**catch**
A context manager. Exceptions raised in the context will be subject to spy's traceback formatting and wrapped in a *CaughtException*. If these are not caught, spy uses an exception hook to force them to be formatted properly. If you opt to catch *CaughtException* instead, you can use its *print_traceback()* method to print the formatted traceback without exiting.

class spy.**chain**(*seq*)
Construct a chain of *fragments* from seq.

Parameters **seq** (sequence) – Fragments to chain together

__call__(*data*)
Alias for *apply()*.

apply(*data*)
Feed data into the fragment chain, and return an iterator over the resulting data.

classmethod auto_fragments(*seq*)
Like the regular constructor, but for each element in seq, apply *fragment()* to it if it isn't already a fragment.

Items in seq must be either regular functions (not generators) or *fragments*.

run_to_exhaustion(*data*)
Call *apply()*, then iterate until the chain runs out of data.

spy.**collect**(*context*)
Return an iterator of the elements being processed by the current fragment. Can be used to write a fragment that consumes multiple items.

@spy.**fragment**
Given a callable func, return a *fragment* that calls func to process data. func must take at least one positional argument, a single value to process and return.

Optionally it can take another argument, called context. If it does, a context object will be passed to it on each invocation. This object has no documented public functionality; its purpose is to be passed to spy API functions that require it (namely *collect()*).

spy.**many**(*ita*)
Return a signaling object that instructs spy to yield values from ita from the current fragment, instead of yielding only one value.

1.4.2 spy.decorators

This module contains various function decorators for use in spy fragments.

`@spy.decorators.`**accumulate**

--accumulate, -a

Accumulate values into an iterator by calling *spy.collect()*, and pass that to the fragment.

This can be used to write a fragment which executes at most once while passing data through:

```
-ma 'x = y;'
```

`@spy.decorators.`**callable**

--callable, -c

Call the result of the decorated fragment

`@spy.decorators.`**filter**

--filter, -f

Use the decorated fragment as a predicate—only elements for which the fragment returns a true value will be passed through.

`@spy.decorators.`**many**

--many, -m

Call *spy.many()* with the result of the fragment.

1.4.3 spy.prelude

Utilities that are automatically imported during spy runs and of little use elsewhere.

`spy.prelude.`**id**(*x*)
Return `x`.

`spy.prelude.`**ft**(**args*)
Return a tuple that, when called, calls all its members with the same argument.

1.5 Examples

1.5.1 Sort

```
$ spy -mc sorted < test.txt
file
five
has
lines
this
```

Similarly:

```
$ spy -mc reversed < test.txt
lines
five
has
file
this
```

1.5.2 Filter

```
$ spy -l -f 'len(pipe) == 4' < test.txt
this
file
five
```

1.5.3 Enumerate

Naively:

```
$ spy -m "['{}: {}'.format(n, v) for n, v in enumerate(pipe, 1)]" < test.txt
1: this
2: file
3: has
4: five
5: lines
```

Taking advantage of spy piping:

```
$ spy -m 'enumerate(pipe, 1)' "'{}: {}'.format(*pipe)" < test.txt
1: this
2: file
3: has
4: five
5: lines
```

1.5.4 Convert CSV to JSON

```
$ spy -c csv.DictReader -c list -c json.dumps < thing.csv > thing.json
```

1.6 Glossary

fragment An object which can be used by *spy.chain()* to create chained iterators.

The following kinds of object *only* are considered fragments:

- The return value of a successful call to *spy.fragment()*
- A generator taking exactly one argument, the iterable to get input values from.

Note: In any given version of spy, it's possible that other objects may work as fragments. This is **not part of the API**, and any accidental support for using other objects may go away at any time.

Python Module Index

S

Index

Symbols

A

C

D

F

I

M

P

R

S