# behold

***Release 0.2.0***

December 23, 2016

Contents

CHAPTER 1

# Behold: Python debugging for large projects

Behold is a package that let's you perform contextual debugging. You can use the state inside one module to either trigger a step debugger or trigger print statements in a completely different module. Given the stateful nature of many large, multi-file applications (I'm looking at you, Django), this capability provides valuable control over your debugging work flow.

Behold is written in pure Python with no dependencies. It is compatible with both Python2 and Python3.

See the Github project page. for examples of how to use *behold*.

CHAPTER 2

# API Documentation

This is the API documentation for the *behold* package. To see examples of how to use *behold*, visit the Github project page.

## 2.1 Managing Context

**class** behold.logger.**in_context**(***context_vars*)

**Parameters** **context_vars** (*key-work arguments*) – Key-word arguments specifying the context variables you would like to set.

You can define arbitrary context in which to perform your debugging. A common use case for this is when you have a piece of code that is called from many different places in your code base, but you are only interested in what happens when it's called from a particular location. You can just wrap that location in a context and only debug when in that context. Here is an example.

```
from behold import BB  # this is an alias for Behold
from behold import in_context

# A function that can get called from anywhere
def my_function():
    for nn in range(5):
        x, y = nn, 2 * nn

        # this will only print for testing
        BB().when_context(what='testing').show('x')

        # this will only print for prodution
        BB().when_context(what='production').show('y')

# Set a a testing context using a decorator
@in_context(what='testing')
def test_x():
   my_function()

# Now run the function under a test
test_x()

# Set a production context using a context-manager and call the function
with in_context(what='production'):
   my_function()
```

behold.logger.**set_context**(**kwargs)

**Parameters** **context_vars** (*key-work arguments*) – Key-word arguments specifying the context variables you would like to set.

This function lets you manually set context variables without using decorators or with statements.

```
from behold import Behold
from behold import set_context, unset_context


# manually set a context
set_context(what='my_context')

# print some variables in that context
Behold().when_context(what='my_context').show(x='hello')

# manually unset the context
unset_context('what')
```

behold.logger.**unset_context**(*keys)

**Parameters** **keys** (*string arguments*) – Arguments specifying the names of context variables you would like to unset.

See the `set_context()` method for an example of how to use this.

behold.logger.**get_stash**(*name*)

**Parameters** **name** (*str*) – The name of the stash you want to retrieve

**Return type** list

**Returns** A list of dictionaries holding stashed records for each time the `behold.stash()` method was called.

For examples, see documentation for `Behold.stash()` as well as the stash examples on Github.

behold.logger.**clear_stash**(*names)

**Parameters** **name** – The names of stashes you would like to clear.

This method removes all global data associated with a particular stash name.

## 2.2 Printing / Debugging

**class** behold.logger.**Behold**(*tag=None*, *strict=False*, *stream=None*)

**Parameters**

- **tag** (*str*) – A tag with which to label all output (default: None)
- **strict** (*Bool*) – When set to true, will only only allow existing keys to be used in the `when_contex()` and `when_values()` methods.
- **stream** (*FileObject*) – Any write-enabled python FileObject (default: sys.stdout)

**Variables**

- **stream** – sys.stdout: The stream that will be written to
- **tag** – None: A string with which to tag output

- **strict** – False: A Bool that sets whether or not only existing keys allowed in `when_contex()` and `when_values()` methods.

`Behold` objects are used to probe state within your code base. They can be used to log output to the console or to trigger entry points for step debugging.

Because it is used so frequently, the behold class has a couple of aliases. The following three statements are equivalent

```
from behold import Behold  # Import using the name of the class

from behold import B       # If you really hate typing

from behold import BB      # If you really hate typing but would
                           # rather use a name that's easier to
                           # search for in your editor.

from behold import *       # Although bad practice in general, since
                           # you'll usually be using behold just for
                           # debugging, this is pretty convenient.
```

`Behold.`**show**(**values*, ***data*)

**Parameters**

- **values** (*str arguments*) – A list of variable or attribute names you want to print. At most one argument can be something other than a string. Strings are interpreted as the variable/attribute names you want to print. If a single non-string argument is provided, it must be an object having attributes named in the string variables. If no object is provided, the strings must be the names of variables in the local scope.
- **data** (*keyword args*) – A set of keyword arguments. The key provided will be the name of the printed variables. The value associated with that key will have its str() representation printed. You can think of these keyword args as attatching additional attributes to any object that was passed in args. If no object was passed, then these kwargs will be used to create an object.

This method will return `True` if all the filters passed, otherwise it will return `False`. This allows you to perform additional logic in your debugging code if you wish. Here are some examples.

```
from behold import Behold, Item
a, b = 1, 2
my_list = [a, b]

# show arguments from local scope
Behold().show('a', 'b')

# show values from local scope using keyword arguments
Behold.show(a=my_list[0], b=my_list[1])

# show values from local scope using keyword arguments, but
# force them to be printed in a specified order
Behold.show('b', 'a', a=my_list[0], b=my_list[1])

# show attributes on an object
item = Item(a=1, b=2)
Behold.show(item, 'a', 'b')

# use the boolean return by show to control more debugging
a = 1
```

```
    if Behold.when(a > 1).show('a'):
        import pdb; pdb.set_trace()
```

Behold.**when**(**bools*)

**Parameters** **bools** (*bool*) – Boolean arguments

All boolean arguments passed to this method must evaluate to *True* for printing to be enabled.

So for example, the following code would print `x: 1`

```
    for x in range(10):
        Behold().when(x == 1).show('x')
```

Behold.**when_values**(***criteria*)

By default, `Behold` objects call `str()` on all variables before sending them to the output stream. This method enables you to filter on those extracted string representations. The syntax is exactly like that of the `when_context()` method. Here is an example.

```
    from behold import Behold, Item

    items = [
       Item(a=1, b=2),
       Item(c=3, d=4),
    ]

    for item in items:
       # You can filter on the string representation
       Behold(tag='first').when_values(a='1').show(item)

       # Behold is smart enough to transform your criteria to strings
       # so this also works
       Behold(tag='second').when_values(a=1).show(item)

       # Because the string representation is not present in the local
       # scope, you must use Django-query-like syntax for logical
       # operations.
       Behold(tag='third').when_values(a__gte=1).show(item)
```

Behold.**when_context**(***criteria*)

**Parameters** **criteria** (*kwargs*) – Key word arguments of var_name=var_value

The key-word arguments passed to this method specify the context constraints that must be met in order for printing to occur. The syntax of these constraints is reminiscent of that used in Django querysets. All specified criteria must be met for printing to occur.

The following syntax is supported.

- `x__lt=1` means `x < 1`
- `x__lte=1` means `x <= 1`
- `x__le=1` means `x <= 1`
- `x__gt=1` means `x > 1`
- `x__gte=1` means `x >= 1`
- `x__ge=1` means `x >= 1`
- `x__ne=1` means `x != 1`
- `x__in=[1, 2, 3]` means `x in [1, 2, 3]`

The reason this syntax is needed is that the context values being compared are not available in the local scope. This renders the normal Python comparison operators useless.

Behold.**view_context**(**context_keys*)

**Parameters** **context_keys** (*string arguments*) – Strings with context keys

Supply this method with any context keys you would like to show.

Behold.**stash**(**values*, ***data*)

The stash method allows you to stash values for later analysis. The arguments are identical to the `show()` method. Instead of writing outpout, however, the `stash()` method populates a global list with the values that would have been printed. This allows them to be accessed later in the debugging process.

Here is an example.

```
from behold import Behold, get_stash

for nn in range(10):
    # You can only invoke ``stash()`` on behold objects that were
    # created with tag.  The tag becomes the global key for the stash
    # list.
    behold = Behold(tag='my_stash_key')
    two_nn = 2 * nn

    behold.stash('nn' 'two_nn')

# You can then run this in a completely different file of your code
# base.
my_stashed_list = get_stash('my_stash_key')
```

Behold.**extract**(*item*, *name*)

You should never need to call this method when you are debugging. It is an internal method that is nevertheless exposed to allow you to implement custom extraction logic for variables/attributes.

This method is responsible for turning attributes into string for printing. The default implementation is shown below, but for custom situations, you inherit from *Behold* and override this method to obtain custom behavior you might find useful. A common strategy is to load up class-level state to help you make the necessary transformation.

**Parameters**

- **item** (*Object*) – The object from which to print attributes. If you didn't explicitly provide an object to the *.show()* method, then *Behold* will attach the local variables you specified as attributes to an *Item* object.
- **name** (*str*) – The attribute name to extract from item

Here is the default implementation.

```
def extract(self, item, name):
    val = ''
    if hasattr(item, name):
        val = getattr(item, name)
    return str(val)
```

Here is an example of transforming Django model ids to names.

```
class CustomBehold(Behold):
    def load_state(self):
        # Put logic here to load your lookup dict.
        self.lookup = your_lookup_code()
```

```
        def extract(self, item, name):
            if hasattr(item, name):
                val = getattr(item, name)
                if isinstance(item, Model) and name == 'client_id':
                    return self.lookup.get(val, '')
                else:
                    return super(CustomBehold, self).extract(name, item)
             else:
                 return ''
```

## 2.3 Items

**class** behold.logger.**Item**(*_item_self*, ***kwargs*)

Item is a simple container class that sets its attributes from constructor kwargs. It supports both object and dictionary access to its attributes. So, for example, all of the following statements are supported.

```
item = Item(a=1, b=2)
item['c'] = 2
a = item['a']
```

An instance of this class is created when you ask to show local variables with a *Behold* object. The local variables you want to show are attached as attributes to an *Item* object.

## B



## C



## E



## G



## I



## S



## U



## V



## W