inspect — Inspect Live Objects

Purpose: The inspect module provides functions for introspecting on live objects and their source code.

The `inspect` module provides functions for learning about live objects, including modules, classes, instances, functions, and methods. The functions in this module can be used to retrieve the original source code for a function, look at the arguments to a method on the stack, and extract the sort of information useful for producing library documentation for source code.

Example Module

The rest of the examples for this section use this example file, `example.py`.

```
# example.py

# This comment appears first
# and spans 2 lines.

# This comment does not show up in the output of getcomments().

"""Sample file to serve as the basis for inspect examples.
"""


def module_level_function(arg1, arg2='default', *args, **kwargs):
    """This function is declared in the module."""
    local_variable = arg1 * 2
    return local_variable


class A(object):
    """The A class."""

    def __init__(self, name):
        self.name = name

    def get_name(self):
        "Returns the name of the instance."
        return self.name


instance_of_a = A('sample_instance')


class B(A):
    """This is the B class.
    It is derived from A.
    """

    # This method is not part of A.
    def do_something(self):
        """Does some work"""

    def get_name(self):
        "Overrides version from A"
        return 'B(' + self.name + ')'
```

Inspecting Modules

The first kind of introspection probes live objects to learn about them. Use `getmembers()` to discover the member attributes of object. The types of members that might be returned depend on the type of object scanned. Modules can contain classes and functions; classes can contain methods and attributes; and so on.

The arguments to `getmembers()` are an object to scan (a module, class, or instance) and an optional predicate function that is

used to filter the objects returned. The return value is a list of tuples with two values: the name of the member, and the type of the member. The `inspect` module includes several such predicate functions with names like `ismodule()`, `isclass()`, etc.

```
# inspect_getmembers_module.py

import inspect

import example

for name, data in inspect.getmembers(example):
    if name.startswith('__'):
        continue
    print('{} : {!r}'.format(name, data))
```

This sample prints the members of the `example` module. Modules have several private attributes that are used as part of the import implementation as well as a set of `__builtins__`. All of these are ignored in the output for this example because they are not actually part of the module and the list is long.

```
$ python3 inspect_getmembers_module.py

A : <class 'example.A'>
B : <class 'example.B'>
instance_of_a : <example.A object at 0x1045a6978>
module_level_function : <function module_level_function at
0x1045be8c8>
```

The `predicate` argument can be used to filter the types of objects returned.

```
# inspect_getmembers_module_class.py

import inspect

import example

for name, data in inspect.getmembers(example, inspect.isclass):
    print('{} : {!r}'.format(name, data))
```

Only classes are included in the output, now.

```
$ python3 inspect_getmembers_module_class.py

A : <class 'example.A'>
B : <class 'example.B'>
```

Inspecting Classes

Classes are scanned using `getmembers()` in the same way as modules, though the types of members are different.

```
# inspect_getmembers_class.py

import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.A), width=65)
```

Because no filtering is applied, the output shows the attributes, methods, slots, and other members of the class.

```
$ python3 inspect_getmembers_class.py

[('__class__', <class 'type'>),
 ('__delattr__',
  <slot wrapper '__delattr__' of 'object' objects>),
 ('__dict__',
  mappingproxy({'__dict__': <attribute '__dict__' of 'A'
objects>,
```

```
                '__doc__': 'The A class.',
                '__init__': <function A.__init__ at
0x1045b3158>,
                '__module__': 'example',
                '__weakref__': <attribute '__weakref__' of 'A'
objects>,
                'get_name': <function A.get_name at
0x1045b31e0>})),
 ('__dir__', <method '__dir__' of 'object' objects>),
 ('__doc__', 'The A class.'),
 ('__eq__', <slot wrapper '__eq__' of 'object' objects>),
 ('__format__', <method '__format__' of 'object' objects>),
 ('__ge__', <slot wrapper '__ge__' of 'object' objects>),
 ('__getattribute__',
  <slot wrapper '__getattribute__' of 'object' objects>),
 ('__gt__', <slot wrapper '__gt__' of 'object' objects>),
 ('__hash__', <slot wrapper '__hash__' of 'object' objects>),
 ('__init__', <function A.__init__ at 0x1045b3158>),
 ('__init_subclass__',
  <built-in method __init_subclass__ of type object at
0x101d12d58>),
 ('__le__', <slot wrapper '__le__' of 'object' objects>),
 ('__lt__', <slot wrapper '__lt__' of 'object' objects>),
 ('__module__', 'example'),
 ('__ne__', <slot wrapper '__ne__' of 'object' objects>),
 ('__new__',
  <built-in method __new__ of type object at 0x100996700>),
 ('__reduce__', <method '__reduce__' of 'object' objects>),
 ('__reduce_ex__', <method '__reduce_ex__' of 'object'
objects>),
 ('__repr__', <slot wrapper '__repr__' of 'object' objects>),
 ('__setattr__',
  <slot wrapper '__setattr__' of 'object' objects>),
 ('__sizeof__', <method '__sizeof__' of 'object' objects>),
 ('__str__', <slot wrapper '__str__' of 'object' objects>),
 ('__subclasshook__',
  <built-in method __subclasshook__ of type object at
0x101d12d58>),
 ('__weakref__', <attribute '__weakref__' of 'A' objects>),
 ('get_name', <function A.get_name at 0x1045b31e0>)]
```

To find the methods of a class, use the `isfunction()` predicate. The `ismethod()` predicate only recognizes bound methods of instances.

```
# inspect_getmembers_class_methods.py

import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.A, inspect.isfunction))
```

Only unbound methods are returned now.

```
$ python3 inspect_getmembers_class_methods.py

[('__init__', <function A.__init__ at 0x1045b2158>),
 ('get_name', <function A.get_name at 0x1045b21e0>)]
```

The output for B includes the override for `get_name()` as well as the new method, and the inherited `__init__()` method implemented in A.

```
# inspect_getmembers_class_methods_b.py

import inspect
from pprint import pprint

import example
```

```
pprint(inspect.getmembers(example.B, inspect.isfunction))
```

Methods inherited from `A`, such as `__init__()`, are identified as being methods of `B`.

```
$ python3 inspect_getmembers_class_methods_b.py

[('__init__', <function A.__init__ at 0x103dc5158>),
 ('do_something', <function B.do_something at 0x103dc5268>),
 ('get_name', <function B.get_name at 0x103dc52f0>)]
```

Inspecting Instances

Introspecting instances works in the same way as other objects.

```
# inspect_getmembers_instance.py

import inspect
from pprint import pprint

import example

a = example.A(name='inspect_getmembers')
pprint(inspect.getmembers(a, inspect.ismethod))
```

The predicate `ismethod()` recognizes two bound methods from `A` in the example instance.

```
$ python3 inspect_getmembers_instance.py

[('__init__', <bound method A.__init__ of <example.A object at 0
x101d9c0f0>>),
 ('get_name', <bound method A.get_name of <example.A object at 0
x101d9c0f0>>)]
```

Documentation Strings

The docstring for an object can be retrieved with `getdoc()`. The return value is the `__doc__` attribute with tabs expanded to spaces and with indentation made uniform.

```
# inspect_getdoc.py

import inspect
import example

print('B.__doc__:')
print(example.B.__doc__)
print()
print('getdoc(B):')
print(inspect.getdoc(example.B))
```

The second line of the docstring is indented when it is retrieved through the attribute directly, but moved to the left margin by `getdoc()`.

```
$ python3 inspect_getdoc.py

B.__doc__:
This is the B class.
    It is derived from A.


getdoc(B):
This is the B class.
It is derived from A.
```

In addition to the actual docstring, it is possible to retrieve the comments from the source file where an object is implemented, if the source is available. The `getcomments()` function looks at the source of the object and finds comments on lines preceding the implementation.

```
# inspect_getcomments_method.py

import inspect
import example

print(inspect.getcomments(example.B.do_something))
```

The lines returned include the comment prefix with any whitespace prefix stripped off.

```
$ python3 inspect_getcomments_method.py

# This method is not part of A.
```

When a module is passed to `getcomments()`, the return value is always the first comment in the module.

```
# inspect_getcomments_module.py

import inspect
import example

print(inspect.getcomments(example))
```

Contiguous lines from the example file are included as a single comment, but as soon as a blank line appears the comment is stopped.

```
$ python3 inspect_getcomments_module.py

# This comment appears first
# and spans 2 lines.
```

Retrieving Source

If the `.py` file is available for a module, the original source code for the class or method can be retrieved using `getsource()` and `getsourcelines()`.

```
# inspect_getsource_class.py

import inspect
import example

print(inspect.getsource(example.A))
```

When a class is passed in, all of the methods for the class are included in the output.

```
$ python3 inspect_getsource_class.py

class A(object):
    """The A class."""

    def __init__(self, name):
        self.name = name

    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

To retrieve the source for a single method, pass the method reference to `getsource()`.

```
# inspect_getsource_method.py

import inspect
import example

print(inspect.getsource(example.A.get_name))
```

The original indent level is retained in this case.

```
$ python3 inspect_getsource_method.py

    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

Use `getsourcelines()` instead of `getsource()` to retrieve the lines of source split into individual strings.

```
# inspect_getsourcelines_method.py

import inspect
import pprint
import example

pprint.pprint(inspect.getsourcelines(example.A.get_name))
```

The return value from `getsourcelines()` is a `tuple` containing a list of strings (the lines from the source file), and a starting line number in the file where the source appears.

```
$ python3 inspect_getsourcelines_method.py

(['    def get_name(self):\n',
  '        "Returns the name of the instance."\n',
  '        return self.name\n'],
 23)
```

If the source file is not available, `getsource()` and `getsourcelines()` raise an `IOError`.

Method and Function Signatures

In addition to the documentation for a function or method, it is possible to ask for a complete specification of the arguments the callable takes, including default values. The `signature()` function returns a `Signature` instance containing information about the arguments to the function.

```
# inspect_signature_function.py

import inspect
import example

sig = inspect.signature(example.module_level_function)
print('module_level_function{}'.format(sig))

print('\nParameter details:')
for name, param in sig.parameters.items():
    if param.kind == inspect.Parameter.POSITIONAL_ONLY:
        print('  {} (positional-only)'.format(name))
    elif param.kind == inspect.Parameter.POSITIONAL_OR_KEYWORD:
        if param.default != inspect.Parameter.empty:
            print('  {}={!r}'.format(name, param.default))
        else:
            print('  {}'.format(name))
    elif param.kind == inspect.Parameter.VAR_POSITIONAL:
        print('  *{}'.format(name))
    elif param.kind == inspect.Parameter.KEYWORD_ONLY:
        if param.default != inspect.Parameter.empty:
            print('  {}={!r} (keyword-only)'.format(
                name, param.default))
        else:
            print('  {} (keyword-only)'.format(name))
    elif param.kind == inspect.Parameter.VAR_KEYWORD:
        print('  **{}'.format(name))
```

The function arguments are available through the `parameters` attribute of the `Signature`. `parameters` is an ordered dictionary mapping the parameter names to `Parameter` instances describing the argument. In this example, the first argument to the function, `arg1`, does not have a default value, while `arg2` does.

```
$ python3 inspect_signature_function.py

module_level_function(arg1, arg2='default', *args, **kwargs)

Parameter details:
  arg1
  arg2='default'
  *args
  **kwargs
```

The `Signature` for a function can be used by decorators or other functions to validate inputs, provide different defaults, etc. Writing a suitably generic and reusable validation decorator has one special challenge, though, because it can be complicated to match up incoming arguments with their names for functions that accept a combination of named and positional arguments. The `bind()` and `bind_partial()` methods provide the necessary logic to handle the mapping. They return a `BoundArguments` instance populated with the arguments associated with the names of the arguments of a specified function.

```
# inspect_signature_bind.py

import inspect
import example

sig = inspect.signature(example.module_level_function)

bound = sig.bind(
    'this is arg1',
    'this is arg2',
    'this is an extra positional argument',
    extra_named_arg='value',
)

print('Arguments:')
for name, value in bound.arguments.items():
    print('{} = {!r}'.format(name, value))

print('\nCalling:')
print(example.module_level_function(*bound.args, **bound.kwargs))
```

The `BoundArguments` instance has attributes `args` and `kwargs` that can be used to call the function using the syntax to expand the tuple and dictionary onto the stack as the arguments.

```
$ python3 inspect_signature_bind.py

Arguments:
arg1 = 'this is arg1'
arg2 = 'this is arg2'
args = ('this is an extra positional argument',)
kwargs = {'extra_named_arg': 'value'}

Calling:
this is arg1this is arg1
```

If only some arguments are available, `bind_partial()` will still create a `BoundArguments` instance. It may not be fully usable until the remaining arguments are added.

```
# inspect_signature_bind_partial.py

import inspect
import example

sig = inspect.signature(example.module_level_function)

partial = sig.bind_partial(
    'this is arg1',
)

print('Without defaults:')
for name, value in partial.arguments.items():
    print('{} = {!r}'.format(name, value))

print('\nWith defaults:')
```

```
print('Without defaults:')
partial.apply_defaults()
for name, value in partial.arguments.items():
    print('{} = {!r}'.format(name, value))
```

`apply_defaults()` will add any values from the parameter defaults.

```
$ python3 inspect_signature_bind_partial.py

Without defaults:
arg1 = 'this is arg1'

With defaults:
arg1 = 'this is arg1'
arg2 = 'default'
args = ()
kwargs = {}
```

Class Hierarchies

`inspect` includes two methods for working directly with class hierarchies. The first, `getclasstree()`, creates a tree-like data structure based on the classes it is given and their base classes. Each element in the list returned is either a tuple with a class and its base classes, or another list containing tuples for subclasses.

```
# inspect_getclasstree.py
```

```
import inspect
import example


class C(example.B):
    pass


class D(C, example.A):
    pass


def print_class_tree(tree, indent=-1):
    if isinstance(tree, list):
        for node in tree:
            print_class_tree(node, indent + 1)
    else:
        print('   ' * indent, tree[0].__name__)
    return


if __name__ == '__main__':
    print('A, B, C, D:')
    print_class_tree(inspect.getclasstree(
        [example.A, example.B, C, D])
    )
```

The output from this example is the tree of inheritance for the `A`, `B`, `C`, and `D` classes. `D` appears twice, since it inherits from both `C` and `A`.

```
$ python3 inspect_getclasstree.py

A, B, C, D:
 object
   A
     D
     B
       C
         D
```

If `getclasstree()` is called with `unique` set to a true value, the output is different.

```
# inspect_getclasstree_unique.py
```

```
import inspect
import example
from inspect_getclasstree import *

print_class_tree(inspect.getclasstree(
    [example.A, example.B, C, D],
    unique=True,
))
```

This time, `D` only appears in the output once:

```
$ python3 inspect_getclasstree_unique.py

 object
   A
     B
       C
         D
```

Method Resolution Order

The other function for working with class hierarchies is `getmro()`, which returns a `tuple` of classes in the order they should be scanned when resolving an attribute that might be inherited from a base class using the *Method Resolution Order* (MRO). Each class in the sequence appears only once.

```
# inspect_getmro.py

import inspect
import example


class C(object):
    pass


class C_First(C, example.B):
    pass


class B_First(example.B, C):
    pass


print('B_First:')
for c in inspect.getmro(B_First):
    print('  {}'.format(c.__name__))
print()
print('C_First:')
for c in inspect.getmro(C_First):
    print('  {}'.format(c.__name__))
```

This output demonstrates the “depth-first” nature of the MRO search. For `B_First`, `A` also comes before `C` in the search order, because `B` is derived from `A`.

```
$ python3 inspect_getmro.py

B_First:
  B_First
  B
  A
  C
  object

C_First:
  C_First
  C
  B
  A
```

```
  object
```

The Stack and Frames

In addition to introspection of code objects, `inspect` includes functions for inspecting the runtime environment while a program is being executed. Most of these functions work with the call stack, and operate on *call frames*. Frame objects hold the current execution context, including references to the code being run, the operation being executed, as well as the values of local and global variables. Typically such information is used to build tracebacks when exceptions are raised. It can also be useful for logging or when debugging programs, since the stack frames can be interrogated to discover the argument values passed into the functions.

`currentframe()` returns the frame at the top of the stack (for the current function).

```
# inspect_currentframe.py

import inspect
import pprint


def recurse(limit, keyword='default', *, kwonly='must be named'):
    local_variable = '.' * limit
    keyword = 'changed value of argument'
    frame = inspect.currentframe()
    print('line {} of {}'.format(frame.f_lineno,
                                 frame.f_code.co_filename))
    print('locals:')
    pprint.pprint(frame.f_locals)
    print()
    if limit <= 0:
        return
    recurse(limit - 1)
    return local_variable

if __name__ == '__main__':
    recurse(2)
```

The values of the arguments to `recurse()` are included in the frame's dictionary of local variables.

```
$ python3 inspect_currentframe.py

line 14 of inspect_currentframe.py
locals:
{'frame': <frame object at 0x10458d408>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 2,
 'local_variable': '..'}

line 14 of inspect_currentframe.py
locals:
{'frame': <frame object at 0x101b1ba18>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 1,
 'local_variable': '.'}

line 14 of inspect_currentframe.py
locals:
{'frame': <frame object at 0x101b2cdc8>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 0,
 'local_variable': ''}
```

Using `stack()`, it is also possible to access all of the stack frames from the current frame to the first caller. This example is similar to the one shown earlier, except it waits until reaching the end of the recursion to print the stack information.

```
# inspect_stack.py
```

```
import inspect
import pprint


def show_stack():
    for level in inspect.stack():
        print('{}[{}]\n  -> {}'.format(
            level.frame.f_code.co_filename,
            level.lineno,
            level.code_context[level.index].strip(),
        ))
        pprint.pprint(level.frame.f_locals)
        print()


def recurse(limit):
    local_variable = '.' * limit
    if limit <= 0:
        show_stack()
        return
    recurse(limit - 1)
    return local_variable


if __name__ == '__main__':
    recurse(2)
```

The last part of the output represents the main program, outside of the `recurse()` function.

```
$ python3 inspect_stack.py

inspect_stack.py[11]
  -> for level in inspect.stack():
{'level': FrameInfo(frame=<frame object at 0x1045823f8>,
filename='inspect_stack.py', lineno=11, function='show_stack',
code_context=['    for level in inspect.stack():\n'], index=0)}

inspect_stack.py[24]
  -> show_stack()
{'limit': 0, 'local_variable': ''}

inspect_stack.py[26]
  -> recurse(limit - 1)
{'limit': 1, 'local_variable': '.'}

inspect_stack.py[26]
  -> recurse(limit - 1)
{'limit': 2, 'local_variable': '..'}

inspect_stack.py[31]
  -> recurse(2)
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': 'Inspecting the call stack.\n',
 '__file__': 'inspect_stack.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader
object at 0x101f9c080>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'inspect': <module 'inspect' from
'.../lib/python3.6/inspect.py'>,
 'pprint': <module 'pprint' from '.../lib/python3.6/pprint.py'>,
 'recurse': <function recurse at 0x1045b9f28>,
 'show_stack': <function show_stack at 0x101f21e18>}
```

There are other functions for building lists of frames in different contexts, such as when an exception is being processed. See the documentation for `trace()`, `getouterframes()`, and `getinnerframes()` for more details.

Command Line Interface

The `inspect` module also includes a command line interface for getting details about objects without having to write out the calls in a separate Python program. The input is a module name and optional object from within the module. The default output is the source code for the named object. Using the `--details` argument causes metadata to be printed instead of the source.

```
$ python3 -m inspect -d example

Target: example
Origin: .../example.py
Cached: .../__pycache__/example.cpython-36.pyc
Loader: <_frozen_importlib_external.SourceFileLoader object at 0
x103e16fd0>



$ python3 -m inspect -d example:A

Target: example:A
Origin: .../example.py
Cached: .../__pycache__/example.cpython-36.pyc
Line: 16



$ python3 -m inspect example:A.get_name

    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

See also

- Standard library documentation for inspect
- Python 2 to 3 porting notes for inspect
- Python 2.3 Method Resolution Order – Documentation for the C3 Method Resolution order used by Python 2.3 and later.
- `pyclbr` – The `pyclbr` module provides access to some of the same information as `inspect` by parsing the module without importing it.
- **PEP 362** – Function Signature Object

dis — Python Bytecode Disassembler

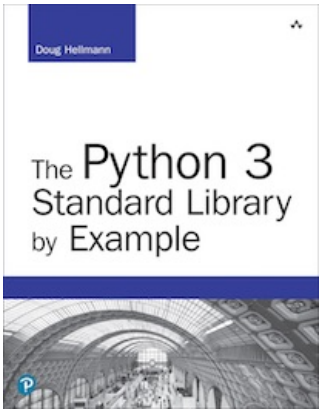
Modules and Packages

Quick Links

This page was last updated 2018-03-18.

Navigation

dis — Python Bytecode Disassembler
Modules and Packages



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example