29.12. `inspect` — Inspect live objects

Source code: Lib/inspect.py

The `inspect` module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

29.12.1. Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The functions whose names begin with “is” are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:

Type	Attribute	Description
module	__doc__	documentation string
	__file__	filename (missing for built-in modules)
class	__doc__	documentation string
	__name__	name with which this class was defined
	__qualname__	qualified name
	__module__	name of module in which this class was defined
method	__doc__	documentation string
	__name__	name with which this method was defined
	__qualname__	qualified name
	__func__	function object containing implementation of method
	__self__	instance to which this method is bound, or `None`

Type	Attribute	Description
function	__doc__	documentation string
	__name__	name with which this function was defined
	__qualname__	qualified name
	__code__	code object containing compiled function bytecode
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__annotations__	mapping of parameters names to annotations; `"return"` key is reserved for return annotations.
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_restricted	0 or 1 if frame is in restricted execution mode
	f_trace	tracing function for this frame, or `None`
code	co_argcount	

Type	Attribute	Description
		number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more here
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_names	tuple of names of local variables
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
generator	__name__	name
	__qualname__	qualified name
	gi_frame	frame
	gi_running	is the generator running?
	gi_code	code
	gi_yieldfrom	object being iterated by `yield from`, or `None`
coroutine	__name__	name
	__qualname__	qualified name
	cr_await	object being awaited on, or `None`
	cr_frame	frame
	cr_running	is the coroutine running?

Type	Attribute	Description
	cr_code	code
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	qualified name
	__self__	instance to which a method is bound, or None

Changed in version 3.5: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

`inspect.`**`getmembers`**(*object*[, *predicate*])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

> **Note:** `getmembers()` will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmodulename`**(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in `importlib.machinery.all_suffixes()`. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

Changed in version 3.3: The function is based directly on `importlib`.

`inspect.`**`ismodule`**(*object*)

Return true if the object is a module.

`inspect.`**`isclass`**(*object*)

Return true if the object is a class, whether built-in or created in Python code.

`inspect.`**`ismethod`**(*object*)

Return true if the object is a bound method written in Python.

`inspect.`**`isfunction`**(*object*)
Return true if the object is a Python function, which includes functions created by a lambda expression.

`inspect.`**`isgeneratorfunction`**(*object*)
Return true if the object is a Python generator function.

`inspect.`**`isgenerator`**(*object*)
Return true if the object is a generator.

`inspect.`**`iscoroutinefunction`**(*object*)
Return true if the object is a coroutine function (a function defined with an `async def` syntax).

New in version 3.5.

`inspect.`**`iscoroutine`**(*object*)
Return true if the object is a coroutine created by an `async def` function.

New in version 3.5.

`inspect.`**`isawaitable`**(*object*)
Return true if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

New in version 3.5.

`inspect.`**`isasyncgenfunction`**(*object*)
Return true if the object is an asynchronous generator function, for example:

```
>>> async def agen():
...     yield 1
...
```

```
>>> inspect.isasyncgenfunction(agen)
True
```

New in version 3.6.

inspect.**isasyncgen**(*object*)

Return true if the object is an asynchronous generator iterator created by an asynchronous generator function.

New in version 3.6.

inspect.**istraceback**(*object*)

Return true if the object is a traceback.

inspect.**isframe**(*object*)

Return true if the object is a frame.

inspect.**iscode**(*object*)

Return true if the object is a code.

inspect.**isbuiltin**(*object*)

Return true if the object is a built-in function or a bound built-in method.

inspect.**isroutine**(*object*)

Return true if the object is a user-defined or built-in function or method.

inspect.**isabstract**(*object*)

Return true if the object is an abstract base class.

inspect.**ismethoddescriptor**(*object*)

Return true if the object is a method descriptor, but not if `ismethod()`, `isclass()`, `isfunction()` or `isbuiltin()` are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method but not a `__set__()` method, but beyond that the set of attributes varies. A `__name__` attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the `ismethoddescriptor()` test, simply because the other tests promise more – you can, e.g., count on having the `__func__` attribute (etc) when an object passes `ismethod()`.

inspect.**isdatadescriptor**(*object*)

Return true if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have `__name__` and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

`inspect.`**`isgetsetdescriptor`**(*object*)

Return true if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

`inspect.`**`ismemberdescriptor`**(*object*)

Return true if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

29.12.2. Retrieving source code

`inspect.`**`getdoc`**(*object*)

Get the documentation string for an object, cleaned up with `cleandoc()`. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

Changed in version 3.5: Documentation strings are now inherited if not overridden.

`inspect.`**`getcomments`**(*object*)

Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return `None`. This could happen if the object has been defined in C or the interactive shell.

`inspect.`**`getfile`**(*object*)

Return the name of the (text or binary) file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`inspect.`**`getmodule`**(*object*)

Try to guess which module an object was defined in.

`inspect.`**`getsourcefile`**(*object*)
Return the name of the Python source file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

`inspect.`**`getsourcelines`**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An `OSError` is raised if the source code cannot be retrieved.

Changed in version 3.3: `OSError` is raised instead of `IOError`, now an alias of the former.

`inspect.`**`getsource`**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `OSError` is raised if the source code cannot be retrieved.

Changed in version 3.3: `OSError` is raised instead of `IOError`, now an alias of the former.

`inspect.`**`cleandoc`**(*doc*)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

29.12.3. Introspecting callables with the Signature object

New in version 3.3.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the `signature()` function.

`inspect.`**`signature`**(*callable*, *, *follow_wrapped=True*)
Return a `Signature` object for the given `callable`:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of python callables, from plain functions and classes to `functools.partial()` objects.

Raises `ValueError` if no signature can be provided, and `TypeError` if that type of object is not supported.

New in version 3.5: `follow_wrapped` parameter. Pass `False` to get a signature of `callable` specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

> **Note:** Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

class `inspect.`**`Signature`**(*parameters=None*, *, *return_annotation=Signature.empty*)

A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a `Parameter` object in its `parameters` collection.

The optional *parameters* argument is a sequence of `Parameter` objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument, can be an arbitrary Python object, is the “return” annotation of the callable.

Signature objects are *immutable*. Use `Signature.replace()` to make a modified copy.

Changed in version 3.5: Signature objects are picklable and hashable.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters' names to the corresponding `Parameter` objects.

return_annotation

The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to `Signature.empty`.

bind(**args*, ***kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns `BoundArguments` if `*args` and `**kwargs` match the signature, or raises a `TypeError`.

bind_partial(**args*, ***kwargs*)

Works the same way as `Signature.bind()`, but allows the omission of some required arguments (mimics `functools.partial()` behavior.) Returns `BoundArguments`, or raises a `TypeError` if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)

Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different `parameters` and/or `return_annotation` to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in `Signature.empty`.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod **from_callable**(*obj*, ***, *follow_wrapped=True*)

Return a `Signature` (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

This method simplifies subclassing of `Signature`:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

New in version 3.5.

class `inspect.`**`Parameter`**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

Parameter objects are *immutable*. Instead of modifying a Parameter object, you can use `Parameter.replace()` to create a modified copy.

Changed in version 3.5: Parameter objects are picklable and hashable.

`empty`

A special class-level marker to specify absence of default values and annotations.

`name`

The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.
Changed in version 3.6: These parameter names are exposed by this module as names like `implicit0`.

`default`

The default value for the parameter. If the parameter has no default value, this attribute is set to `Parameter.empty`.

`annotation`

The annotation for the parameter. If the parameter has no annotation, this attribute is set to `Parameter.empty`.

`kind`

Describes how argument values are bound to the parameter. Possible values (accessible via `Parameter`, like `Parameter.KEYWORD_ONLY`):

Name	Meaning
POSITIONAL_ONLY	Value must be supplied as a positional argument.

Name	Meaning
	Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`replace`(**[, name][, kind][, default][, annotation]*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a `Parameter` attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass `Parameter.empty`.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY,
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy
'foo=42'

>>> str(param.replace(default=Parameter.empty, annota
"foo:'spam'"
```

Changed in version 3.4: In Python 3.3 Parameter objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class `inspect.`**BoundArguments**

Result of a `Signature.bind()` or `Signature.bind_partial()` call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (`collections.OrderedDict`) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in `arguments` will reflect in `args` and `kwargs`.

Should be used in conjunction with `Signature.parameters` for any argument processing purposes.

> **Note:** Arguments for which `Signature.bind()` or `Signature.bind_partial()` relied on a default value are skipped. However, if needed, use `BoundArguments.apply_defaults()` to add them.

args

A tuple of positional arguments values. Dynamically computed from the `arguments` attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the `arguments` attribute.

signature

A reference to the parent `Signature` object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

New in version 3.5.

The `args` and `kwargs` properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

See also:

PEP 362 - Function Signature Object.
The detailed specification, implementation details and examples.

29.12.4. Classes and functions

`inspect.`**`getclasstree`**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.`**`getargspec`**(*func*)

Get the names and default values of a Python function's parameters. A named tuple `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

Deprecated since version 3.0: Use `getfullargspec()` for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use `signature()` and Signature Object, which provide a more structured introspection API for callables.

`inspect.`**`getfullargspec`**(*func*)

Get the names and default values of a Python function's parameters. A named tuple is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs,
kwonlydefaults, annotations)
```

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that `signature()` and Signature Object provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

Changed in version 3.4: This function is now based on `signature()`, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

Changed in version 3.6: This method was previously documented as deprecated in favour of `signature()` in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy `getargspec()` API.

`inspect.`**`getargvalues`**(*frame*)

Get information about arguments passed into a particular frame. A named tuple `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

> **Note:** This function was inadvertently marked as deprecated in Python 3.5.

`inspect.`**`formatargspec`**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Format a pretty argument spec from the values returned by `getfullargspec()`.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwonlyargs`, `kwonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.

For example:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

Deprecated since version 3.5: Use `signature()` and Signature Object, which provide a better introspecting API for callables.

`inspect.`**`formatargvalues`**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by `getargvalues()`. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

> **Note:** This function was inadvertently marked as deprecated in Python 3.5.

`inspect.`**`getmro`**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

`inspect.`**`getcallargs`**(*func*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned,

mapping the argument names (including the names of the * and ** arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos'
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

New in version 3.2.

Deprecated since version 3.5: Use `Signature.bind()` and `Signature.bind_partial()` instead.

`inspect.`**`getclosurevars`**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A named tuple `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

`TypeError` is raised if *func* is not a Python function or method.

New in version 3.3.

`inspect.`**`unwrap`**(*func*, *, *stop=None*)

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, `signature()` uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

`ValueError` is raised if a cycle is encountered.

New in version 3.4.

29.12.5. The interpreter stack

When the following functions return "frame records," each record is a named tuple `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

Changed in version 3.5: Return a named tuple instead of a tuple.

Note: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

Get information about a frame or traceback object. A named tuple `Traceback(filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getouterframes`**(*frame*, *context=1*)

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`currentframe`**()

Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

`inspect.`**`stack`**(*context=1*)

Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`trace`**(*context=1*)

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

29.12.6. Fetching attributes statically

Both `getattr()` and `hasattr()` can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. `getattr_static()` has the same signature as `getattr()` but avoids executing code when it fetches attributes.

`inspect.`**`getattr_static`**(*obj*, *attr*, *default=None*)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance `__dict__` is shadowed by another member (for example a property) then this function will be unable to find instance members.

New in version 3.2.

`getattr_static()` does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descri

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
```

```
        # in which case the descriptor itself will
        # have to do
        pass
```

29.12.7. Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. `getgeneratorstate()` allows the current state of a generator to be determined easily.

`inspect.`**`getgeneratorstate`**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

New in version 3.2.

`inspect.`**`getcoroutinestate`**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

New in version 3.5.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

`inspect.`**`getgeneratorlocals`**(*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling `locals()` in the body of the generator, and all the same caveats apply.

If *generator* is a generator with no currently associated frame, then an empty dictionary is returned. `TypeError` is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.
New in version 3.3.

`inspect.`**`getcoroutinelocals`**(*coroutine*)
This function is analogous to `getgeneratorlocals()`, but works for coroutine objects created by `async def` functions.

New in version 3.5.

29.12.8. Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

`inspect.`**`CO_OPTIMIZED`**
The code object is optimized, using fast locals.

`inspect.`**`CO_NEWLOCALS`**
If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

`inspect.`**`CO_VARARGS`**
The code object has a variable positional parameter (`*args`-like).

`inspect.`**`CO_VARKEYWORDS`**
The code object has a variable keyword parameter (`**kwargs`-like).

`inspect.`**`CO_NESTED`**
The flag is set when the code object is a nested function.

`inspect.`**`CO_GENERATOR`**
The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

`inspect.`**`CO_NOFREE`**
The flag is set if there are no free or cell variables.

`inspect.`**`CO_COROUTINE`**

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

New in version 3.5.

`inspect.`**`CO_ITERABLE_COROUTINE`**

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

New in version 3.5.

`inspect.`**`CO_ASYNC_GENERATOR`**

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

New in version 3.6.

Note: The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the `inspect` module for any introspection needs.

29.12.9. Command Line Interface

The `inspect` module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

`--details`

Print information about the specified object rather than the source code