sigtools Documentation

Release 2.0.2

Yann Kaiser

Jun 05, 2018

Contents

`sigtools` is a Python package that improves on introspection tools available for determining function signatures. This is useful for libraries or tooling that want to know how a function can be called: Documentation generators, IDEs, and tools that adapt themselves to functions they are given.

`sigtools` also provides a backport of Python 3's keyword-only parameters and function annotations.

CHAPTER 1

Keyword-only parameters

`sigtools` provides two decorators to emulate keyword-only parameters:

`sigtools.modifiers.`**`autokwoargs`**(*func=None*, *, *exceptions=()*)

Converts all parameters with default values to keyword-only parameters:

```
from sigtools.modifiers import autokwoargs


@autokwoargs
def func(arg, opt1=False, opt2=False, *rest):
    print(arg, rest, opt1, opt2)


func(1, 2, 3)
# 1 (2, 3) False False

func(1, 2, opt2=True)
# -> 1 (2,) False True
```

In the example above, the `opt1` and `opt2` parameters are 'skipped' by positional arguments and can only be set using named arguments (`opt2=True`).

If you wish to prevent parameters that have a default from becoming keyword-only, you can use the `exceptions=` parameter:

```
@autokwoargs(exceptions=['arg2'])
def func(arg1, arg2=42, opt1=False, opt2=False):
    print(arg1, arg2, opt1, opt2)

func(1, 2, opt1=False)
# 1 2 False False
```

If you wish to pick individual parameters to convert, use *`sigtools.modifiers.kwoargs`*. This module also allows you to add function annotations using the *`annotate`* decorator.

CHAPTER 2

Improved signature reporting

Python 3.3's `inspect` module introduces signature objects, which represent how a function can be called. Their textual representation roughly matches the parameter list part of a function definition:

```
import inspect


def func(abc, *args, **kwargs):
    ...


print(inspect.signature(func))
# (abc, *args, **kwargs)
```

`sigtools.`**`signature`**(*obj*, *auto=True*, *args=()*, *kwargs={}*)

Improved version of `inspect.signature`. Takes into account decorators from *`sigtools.specifiers`* if present or tries to determine a full signature automatically.

For instance, consider this example of a decorator being defined and applied:

```
import inspect

from sigtools import specifiers


def decorator(param):
    def _decorate(wrapped):
        def _wrapper(*args, **kwargs):
            wrapped(param, *args, **kwargs)
        return _wrapper
    return _decorate


@decorator('eggs')
def func(ham, spam):
```

(continues on next page)

(continued from previous page)

```
    return ham, spam


print(inspect.signature(func))
# (*args, **kwargs)

print(specifiers.signature(func))
# (spam)
```

Where `inspect.signature` simply sees `(*args, **kwargs)` from `_wrapper`, *`sigtools.signature`* returns the correct signature for the `*args, **kwargs` portion of `wrapped(param, *args, **params)`.

CHAPTER 3

Improved signatures in `sphinx.ext.autodoc` documentation

Sphinx, the documentation tool, comes with an extension, `sphinx.ext.autodoc`, which lets you source some of your documentation from your code and its docstrings. `sigtools.sphinxext`, if activated, automatically improves signatures like explained above.

To activate it, add `'sigtools.sphinxext'` to the `extensions` list in your Sphinx's `conf.py`:

```
extensions = [
    'sphinx.ext.autodoc', ...
    'sigtools.sphinxext']
```

If you want to use the automatic signature gathering while ignoring the docstring in order to supply your own explanations, you can use this directive instead of `autofunction`:

.. autosignature:: `object`

Documents a Python object while ignoring the source docstring. *`sigtools.specifiers.signature`* is used to retrieve the object's call signature

CHAPTER 4

Reference

4.1 API Reference

4.1.1 `sigtools.modifiers`: Modify the effective signature of the decorated callable

The functions in this module can be used as decorators to mark and enforce some parameters to be keyword-only (*`kwoargs`*) or annotate (*`annotate`*) them, just like you can using Python 3 syntax. You can also mark and enforce parameters to be positional-only (*`posoargs`*). *`autokwoargs`* helps you quickly make your parameters with default values become keyword-only.

class sigtools.modifiers.**annotate**(*_annotate__return_annotation=<unset>*, ***annotations*)

Annotates a function, avoiding the use of python3 syntax

These two functions are equivalent:

```
def py3_func(spam: 'ham', eggs: 'chicken'=False) -> 'return':
    return spam, eggs

@annotate('return', spam='ham', eggs='chicken')
def py23_func(spam, eggs=False):
    return spam, eggs
```

Parameters

- **_annotate__return_annotation** – The annotation to attach for return value
- **annotations** – The annotations to attach for each parameter

Raises `ValueError` if a parameter to be annotated does not exist on the function

sigtools.modifiers.**kwoargs**(**kwoarg_names*, *start=None*)

Marks the given parameters as keyword-only, avoiding the use of python3 syntax.

These two functions are equivalent:

```
def py3_func(spam, *, ham, eggs='chicken'):
    return spam, ham, eggs

@kwoargs('ham', 'eggs')
def py23_func(spam, ham, eggs='chichen'):
    return spam, ham, eggs
```

Parameters

- **start** (*str*) – If given and is the name of a parameter, it and all parameters after it are made keyword-only
- **kwoarg_names** (*str*) – Names of the parameters to convert

Raises ValueError if end or one of posoarg_names isn't in the decorated function's signature.

sigtools.modifiers.**autokwoargs**(*func=None*, *, *exceptions=()*)

Marks all arguments with default values as keyword-only.

Parameters **exceptions** (*sequence*) – names of parameters not to convert

```
>>> from sigtools.modifiers import autokwoargs
>>> @autokwoargs(exceptions=['c'])
... def func(a, b, c=3, d=4, e=5):
...     pass
...
>>> from inspect import signature
>>> print(signature(func))
(a, b, c=3, *, d=4, e=5)
```

sigtools.modifiers.**posoargs**(**posoarg_names*, *end=None*)

Marks the given parameters as positional-only.

If the resulting function is passed any named arguments that references a positional parameter, TypeError will be raised.

```
>>> from sigtools.modifiers import posoargs
>>> @posoargs('ham')
... def func(ham, spam):
...     pass
...
>>> func('ham', 'spam')
>>> func('ham', spam='spam')
>>> func(ham='ham', spam='spam')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "./sigtools/modifiers.py", line 94, in __call__
    .format(' '.join(repr(name) for name in intersect))
TypeError: Named arguments refer to positional-only parameters: 'ham'
```

Parameters

- **end** (*str*) – If given and is the name of a parameter, it and all parameters leading to it are made positional-only.
- **posoarg_names** (*str*) – Names of the parameters to convert

Raises ValueError if end or one of posoarg_names isn't in the decorated function's signature.

4.1.2 `sigtools.specifiers`: Decorators to enhance a callable's signature

The `forwards_to_*` decorators from this module will leave a "note" on the decorated object for *`sigtools.specifiers.signature`* to pick up. These "notes" tell *`signature`* in which way the signature of the examinated object should be crafted. The `forwards_to_*` decorators here will help you tell introspection or documentation tools what the `*args` and `**kwargs` parameters stand for in your function if it forwards them to another callable. This should cover most use cases, but you can use *`forger_function`* or *`set_signature_forger`* to create your own.

`sigtools.specifiers.`**`signature`**(*obj*, *auto=True*, *args=()*, *kwargs={}*)

Retrieves the full signature of `obj`, either by taking note of decorators from this module, or by performing automatic signature discovery.

If `auto` is true, the signature will be automatically refined based on how `*args` and `**kwargs` are used throughout the function.

If `args` and/or `kwargs` are specified, they are used by automatic signature discovery as arguments passed into the function. This is useful if the function calls something passed in as a parameter.

You can use `emulate=True` as an argument to the decorators from this module if you wish them to work with `inspect.signature` or its `funcsigs` backport directly.

```
>>> from sigtools import specifiers
>>> import inspect
>>> def inner(a, b):
...     return a + b
...
>>> # Relying on automatic discovery
>>> def outer(c, *args, **kwargs):
...     return c * inner(*args, **kwargs)
>>> print(inspect.signature(outer))
(c, *args, **kwargs)
>>> print(specifiers.signature(outer, auto=False))
(c, *args, **kwargs)
>>> print(specifiers.signature(outer))
(c, a, b)
>>>
>>> # Using a decorator from this module
>>> @specifiers.forwards_to_function(inner)
... def outer(c, *args, **kwargs):
...     return c * inner(*args, **kwargs)
...
>>> print(inspect.signature(outer))
(c, *args, **kwargs)
>>> print(specifiers.signature(outer), auto=False)
(c, a, b)
>>> print(specifiers.signature(outer))
(c, a, b)
>>>
>>> # Using the emulate argument for compatibility with inspect
>>> @specifiers.forwards_to_function(inner, emulate=True)
... def outer(c, *args, **kwargs):
...     return c * inner(*args, **kwargs)
>>> print(inspect.signature(outer))
(c, a, b)
>>> print(specifiers.signature(outer), auto=False)
(c, a, b)
>>> print(specifiers.signature(outer))
(c, a, b)
```

Parameters

- **auto** (*bool*) – Enable automatic signature discovery.
- **args** (*sequence*) – Positional arguments passed to the function.
- **mapping** – Named arguments passed to the function.

sigtools.specifiers.**forwards_to_function**(*wrapped*, *num_args=0*, **named_args*, *emulate=None*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Wraps the decorated function to give it the effective signature it has when it forwards its `*args` and `**kwargs` to the static callable `wrapped`.

```
>>> from sigtools.specifiers import forwards_to_function
>>> def wrapped(x, y):
...     return x * y
...
>>> @forwards_to_function(wrapped)
... def wrapper(a, *args, **kwargs):
...     return a + wrapped(*args, **kwargs)
...
>>> from inspect import signature
>>> print(signature(wrapper))
(a, x, y)
```

See *Picking the appropriate arguments for forwards* for more information on the parameters.

sigtools.specifiers.**forwards_to_method**(*wrapped_name*, *num_args=0*, **named_args*, *emulate=None*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Wraps the decorated method to give it the effective signature it has when it forwards its `*args` and `**kwargs` to the method or attribute named by `wrapped_name`.

Parameters **wrapped_name** (*str*) – The name of the wrapped method or attribute. Passing a name with dots(`.`) will do a deep attribute search.

See *Picking the appropriate arguments for forwards* for more information on the parameters.

```
>>> from sigtools.specifiers import signature, forwards_to_method
>>> class Ham(object):
...     def egg(self, a, b):
...         return a + b
...     @forwards_to_method('egg')
...     def spam(self, c, *args, **kwargs):
...         return c * self.egg(*args, **kwargs)
...
>>> h = Ham()
>>> print(signature(h.spam))
(c, a, b)
```

sigtools.specifiers.**forwards_to_super**(*num_args=0*, **named_args*, *emulate=None*, *cls=None*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Wraps the decorated method to give it the effective signature it has when it forwards its `*args` and `**kwargs` to the same method on the super object for the class it belongs in.

You can only use this decorator directly in Python versions 3.3 and up, and the wrapped function must make use of the arg-less form of super:

```
>>> from sigtools.specifiers import forwards_to_super
>>> class Base:
...     def func(self, x, y):
...         return x * y
..
>>> class Subclass(Base):
...     @forwards_to_super()
...     def func(self, a, *args, **kwargs):
...         return a + super().func(*args, **kwargs)
...
>>> from inspect import signature
>>> print(signature(Subclass.func))
(self, a, x, y)
>>> print(signature(Subclass().func))
(a, x, y)
```

If you need to use similar functionality in older python versions, use *apply_forwards_to_super* instead.

See *Picking the appropriate arguments for forwards* for more information on the parameters.

sigtools.specifiers.**apply_forwards_to_super**(**member_names*, *num_args=0*, *named_args=()*, ***kwargs*)

Applies the *forwards_to_super* decorator on `member_names` in the decorated class, in a way which works in Python 2.6 and up.

```
>>> from sigtools.specifiers import apply_forwards_to_super
>>> class Base:
...     def func(self, x, y):
...         return x * y
...
>>> @apply_forwards_to_super('func')
... class Subclass(Base):
...     def func(self, a, *args, **kwargs):
...         return a + super(Subclass, self).func(*args, **kwargs)
...
>>> from inspect import signature
>>> print(signature(Subclass.func))
(self, a, x, y)
>>> print(signature(Subclass().func))
(a, x, y)
```

See *Picking the appropriate arguments for forwards* for more information on the parameters.

sigtools.specifiers.**forwards**(*wrapper*, *wrapped*, *num_args=0*, **named_args*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Returns an effective signature of `wrapper` when it forwards its `*args` and `**kwargs` to `wrapped`.

Parameters

- **wrapper** (*callable*) – The outer callable
- **wrapped** (*callable*) – The callable `wrapper`'s extra arguments are passed to.

Returns a `inspect.Signature` object

See *Picking the appropriate arguments for forwards* for more information on the parameters.

sigtools.specifiers.**forger_function**(*func*)

Creates a decorator factory which, when applied will set `func` as the forger function of the decorated object.

Parameters **func** (*callable*) – Must return a fake signature for the object passed as the named argument `obj`. Any arguments supplied during decoration are also passed.

The decorator produced by this function also accepts an `emulate` parameter. See *set_signature_forger* for information on it.

This function can be used as a decorator:

```
>>> from sigtools import specifiers, modifiers, support
>>> @specifiers.forger_function
... @modifiers.kwoargs('obj')
... def static_signature(obj, sig):
...     return sig
...
>>> @static_signature(support.s('a, b, /'))
... def my_func(d, e):
...     pass
...
>>> print(specifiers.signature(my_func))
(a, b, /)
```

sigtools.specifiers.**set_signature_forger**(*obj*, *forger*, *emulate=None*)

Attempts to set the given signature forger on the supplied object.

This function first tries to set an attribute on `obj` and returns it. If that fails, it wraps the object that advertises the correct signature (even to `inspect.signature`) and forwards calls.

Parameters **emulate** – If supplied, forces the function to adhere to one strategy: either set the attribute or fail(`False`), or always wrap the object(`True`). If something else is passed, it is called with `(obj, forger)` and the return value is used.

sigtools.specifiers.**as_forged**

Descriptor that returns the computer signature for the object it is an attribute of. Most useful as `__signature__`.

Allows `inspect.signature` to read forged signatures from your own objects.

```
>>> from sigtools import specifiers
>>> import inspect
>>> class MyClass(object):
...     __signature__ = specifiers.as_forged
...     @specifiers.forwards_to_method('method')
...     def __call__(self, x, *args, **kwargs):
...         pass
...     def method(self, a, b, c):
...         pass
...
>>> print(inspect.signature(MyClass()))
(x, a, b, c)
```

4.1.3 `sigtools.wrappers`: Combine multiple functions

The functions here help you combine multiple functions into a new callable which will automatically advertise the correct signature.

class sigtools.wrappers.**Combination**(**functions*)

Bases: object

Creates a callable that passes the first argument through each callable, using the result of each pass as the argument to the next

get_signature(*obj*)

sigtools.wrappers.**decorator**(*func*)

Turns a function into a decorator.

The function received the decorated function as first argument.

```
from sigtools import wrappers

@wrappers.decorator
def my_decorator(func, *args, deco_param=False, **kwargs):
    ... # Do stuff with deco_param
    return func(*args, **kwargs)

@my_decorator
def my_function(func_param):
    ...

my_function('value for func_param', deco_param=True)

from sigtools import specifiers
print(specifiers.signature(my_function))
# (func_param, *, deco_param=False)
```

Unlike *wrapper_decorator*, `decorator` does not require you to specify how your function uses `*args, **kwargs` and lets automatic signature discovery figure it out.

Note: Signature reporting will not work in interactive sessions, as per *Limitations of automatic signature discovery*.

---

sigtools.wrappers.**wrapper_decorator**(*num_args=0*, **named_args*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Turns a function into a decorator that wraps callables with that function.

Consult *signatures.forwards*'s documentation for *help picking the correct values for the parameters*.

The wrapped function is passed as first argument to the wrapper.

As an example, here we create a `@print_call` decorator which wraps the decorated function and prints a line everytime the function is called:

```
>>> from sigtools import modifiers, wrappers
>>> @wrappers.wrapper_decorator
... @modifiers.autokwoargs
... def print_call(func, _show_return=True, *args, **kwargs):
...     print('Calling {0.__name__}(*{1}, **{2})'.format(func, args, kwargs))
...     ret = func(*args, **kwargs)
...     if _show_return:
...         print('Return: {0!r}'.format(ret))
...     return ret
...
```

(continues on next page)

(continued from previous page)

```
>>> print_call
<decorate with <<function print_call at 0x7f28d721a950> with signature print_cal
l(func, *args, _show_return=True, **kwargs)>>
>>> @print_call
... def as_list(obj):
...     return [obj]
...
>>> as_list
<<function as_list at 0x7f28d721ad40> decorated with <<function print_call at 0x
7f28d721a950> with signature print_call(func, *args, _show_return=True, **kwargs
)>>
>>> from inspect import signature
>>> print(signature(as_list))
(obj, *, _show_return=True)
>>> as_list('ham')
Calling as_list(*('ham',), **{})
Return: ['ham']
['ham']
>>> as_list('spam', _show_return=False)
Calling as_list(*('spam',), **{})
['spam']
```

sigtools.wrappers.**wrappers**(*obj*)

For introspection purposes, returns an iterable that yields each wrapping function of obj(as done through *wrapper_decorator*, outermost wrapper first.

Continuing from the *wrapper_decorator* example:

```
>>> list(wrappers.wrappers(as_list))
[<<function print_call at 0x7f28d721a950> with signature print_call(func, *args,
 _show_return=True, **kwargs)>]
```

4.1.4 sigtools.signatures: Signature object manipulation

The functions here are high-level operations that produce a signature from other signature objects, as opposed to dealing with each parameter individually. They are most notably used by the decorators from *sigtools.specifiers* to compute combined signatures.

sigtools.signatures.**signature**(*obj*)

Retrieves to unmodified signature from obj, without taking *sigtools.specifiers* decorators into account or attempting automatic signature discovery.

sigtools.signatures.**merge**(**signatures*)

Tries to compute a signature for which a valid call would also validate the given signatures.

It guarantees any call that conforms to the merged signature will conform to all the given signatures. However, some calls that don't conform to the merged signature may actually work on all the given ones regardless.

Parameters **signatures** (*inspect.Signature*) – The signatures to merge together.

Returns a inspect.Signature object

Raises *IncompatibleSignatures*

```
>>> from sigtools import signatures, support
>>> print(signatures.merge(
...     support.s('one, two, *args, **kwargs'),
```

(continues on next page)

(continued from previous page)

```
...     support.s('one, two, three, *, alpha, **kwargs'),
...     support.s('one, *args, beta, **kwargs')
...     ))
(one, two, three, *, alpha, beta, **kwargs)
```

The resulting signature does not necessarily validate all ways of conforming to the underlying signatures:

```
>>> from sigtools import signatures
>>> from inspect import signature
>>>
>>> def left(alpha, *args, **kwargs):
...     return alpha
...
>>> def right(beta, *args, **kwargs):
...     return beta
...
>>> sig_left = signature(left)
>>> sig_right = signature(right)
>>> sig_merged = signatures.merge(sig_left, sig_right)
>>>
>>> print(sig_merged)
(alpha, /, *args, **kwargs)
>>>
>>> kwargs = {'alpha': 'a', 'beta': 'b'}
>>> left(**kwargs), right(**kwargs) # both functions accept the call
('a', 'b')
>>>
>>> sig_merged.bind(**kwargs) # the merged signature doesn't
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/usr/lib64/python3.4/inspect.py", line 2642, in bind
    return args[0]._bind(args[1:], kwargs)
  File "/usr/lib64/python3.4/inspect.py", line 2542, in _bind
    raise TypeError(msg) from None
TypeError: 'alpha' parameter is positional only, but was passed as a keyword
```

sigtools.signatures.**embed**(**signatures*, *use_varargs=True*, *use_varkwargs=True*)

Embeds a signature within another's `*args` and `**kwargs` parameters, as if a function with the outer signature called a function with the inner signature with just `f(*args, **kwargs)`.

Parameters

- **signatures** (*inspect.Signature*) – The signatures to embed within one-another, outermost first.
- **use_varargs** (*bool*) – Make use of the `*args`-like parameter.
- **use_varkwargs** (*bool*) – Make use of the `*kwargs`-like parameter.

Returns a `inspect.Signature` object

Raises *IncompatibleSignatures*

```
>>> from sigtools import signatures, support
>>> print(signatures.embed(
...     support.s('one, *args, **kwargs'),
...     support.s('two, *args, kw, **kwargs'),
...     support.s('last'),
```

(continues on next page)

(continued from previous page)

```
...     ))
(one, two, last, *, kw)
>>> # use signatures.mask() to remove self-like parameters
>>> print(signatures.embed(
...     support.s('self, *args, **kwargs'),
...     signatures.mask(
...         support.s('self, *args, keyword, **kwargs'), 1),
...     ))
(self, *args, keyword, **kwargs)
```

sigtools.signatures.**mask**(*sig*, *num_args=0*, **named_args*, *hide_args=False*, *hide_kwargs=False*, *hide_varargs=False*, *hide_varkwargs=False*)

Removes the given amount of positional parameters and the given named parameters from `sig`.

Parameters

- **sig** (*inspect.Signature*) – The signature to operate on
- **num_args** (*int*) – The amount of positional arguments passed
- **named_args** (*str*) – The names of named arguments passed
- **hide_args** – If true, mask all positional parameters
- **hide_kwargs** – If true, mask all keyword parameters
- **hide_varargs** – If true, mask the `*args`-like parameter completely if present.
- **hide_varkwargs** – If true, mask the `*kwargs`-like parameter completely if present.

Returns a `inspect.Signature` object

Raises `ValueError` if the signature cannot handle the arguments to be passed.

```
>>> from sigtools import signatures, support
>>> print(signatures.mask(support.s('a, b, *, c, d'), 1, 'd'))
(b, *, c)
>>> print(signatures.mask(support.s('a, b, *args, c, d'), 3, 'd'))
(*args, c)
>>> print(signatures.mask(support.s('*args, c, d'), 2, 'd', hide_varargs=True))
(*, c)
```

sigtools.signatures.**forwards**(*outer*, *inner*, *num_args=0*, **named_args*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

Calls *mask* on `inner`, then returns the result of calling *embed* with `outer` and the result of *mask*.

Parameters

- **outer** (*inspect.Signature*) – The outermost signature.
- **inner** (*inspect.Signature*) – The inner signature.
- **partial** (*bool*) – Set to `True` if the arguments are passed to `partial(func_with_inner, *args, **kwargs)` rather than `func_with_inner`.

`use_varargs` and `use_varkwargs` are the same parameters as in *embed*, and `num_args`, `named_args`, `hide_args` and `hide_kwargs` are parameters of *mask*.

Returns the resulting `inspect.Signature` object

Raises *IncompatibleSignatures*

```
>>> from sigtools import support, signatures
>>> print(signatures.forwards(
...     support.s('a, *args, x, **kwargs'),
...     support.s('b, c, *, y, z'),
...     1, 'y'))
(a, c, *, x, z)
```

See also:

Picking the appropriate arguments for forwards

exception sigtools.signatures.**IncompatibleSignatures**(*sig*, *others*)

Bases: ValueError

Raised when two or more signatures are incompatible for the requested operation.

Variables

- **sig** (*inspect.Signature*) – The signature at which point the incompatibility was discovered
- **others** – The signatures up until sig

sigtools.signatures.**sort_params**(*sig*, *sources=False*)

Classifies the parameters from sig.

Parameters **sig** (*inspect.Signature*) – The signature to operate on

Returns A tuple (posargs, pokargs, varargs, kwoargs, varkwas)

Return type (list, list, Parameter or None, dict, Parameter or None)

```
>>> from sigtools import signatures, support
>>> from pprint import pprint
>>> pprint(signatures.sort_params(support.s('a, /, b, *args, c, d')))
([<Parameter at 0x7fdda4e89418 'a'>],
 [<Parameter at 0x7fdda4e89470 'b'>],
 <Parameter at 0x7fdda4e89c58 'args'>,
 {'c': <Parameter at 0x7fdda4e89c00 'c'>,
  'd': <Parameter at 0x7fdda4e89db8 'd'>},
 None)
```

sigtools.signatures.**apply_params**(*sig*, *posargs*, *pokargs*, *varargs*, *kwoargs*, *varkwargs*, *sources=None*)

Reverses *sort_params*'s operation.

Returns A new inspect.Signature object based off sig, with the given parameters.

4.1.5 sigtools.support: Utilities for use in interactive sessions and unit tests

sigtools.support.**s**(*sig_str*, *ret=None*, ***, *pre=''*, *locals=None*, *name='func'*)

Creates a signature from the given string representation of one.

> **Warning:** The contents of the arguments are eventually passed to exec. Do not use with untrusted input.

```
>>> from sigtools.support import s
>>> sig = s('a, b=2, *args, c:"annotation", **kwargs')
```

(continues on next page)

(continued from previous page)

```
>>> sig
<inspect.Signature object at 0x7f15e6055550>
>>> print(sig)
(a, b=2, *args, c:'annotation', **kwargs)
```

sigtools.support.**f**(*sig_str*, *ret=None*, ***, *pre=''*, *locals=None*, *name='func'*)

Creates a dummy function that has the signature represented by `sig_str` and returns a tuple containing the arguments passed, in order.

> **Warning:** The contents of the arguments are eventually passed to `exec`. Do not use with untrusted input.

```
>>> from sigtools.support import f
>>> import inspect
>>> func = f('a, b=2, *args, c:"annotation", **kwargs')
>>> print(inspect.signature(func))
(a, b=2, *args, c:'annotation', **kwargs)
>>> func(1, c=3)
{'b': 2, 'a': 1, 'kwargs': {}, 'args': ()}
>>> func(1, 2, 3, 4, c=5, d=6)
{'b': 2, 'a': 1, 'kwargs': {'d': 6}, 'args': (3, 4)}
```

sigtools.support.**read_sig**(*sig_str*, *ret=None*)

Reads a string representation of a signature and returns a tuple *func_code* can understand.

sigtools.support.**func_code**(*names*, *return_annotation*, *annotations*, *posoarg_n*, *kwoarg_n*, *params*, *pre=''*, *name='func'*)

Formats the code to construct a function to *read_sig*'s design.

sigtools.support.**make_func**(*code*, *locals=None*, *name='func'*)

Executes the given code and returns the object named func from the resulting namespace.

sigtools.support.**func_from_sig**(*sig*)

Creates a dummy function from the given signature object

> **Warning:** The contents of the arguments are eventually passed to `exec`. Do not use with untrusted input.

sigtools.support.**make_up_callsigs**(*sig*, *extra=2*)

Figures out reasonably as many ways as possible to call a callable with the given signature.

sigtools.support.**bind_callsig**(*sig*, *args*, *kwargs*)

Returns a dict with each parameter name from `sig` mapped to values from `args`, `kwargs` as if a function with `sig` was called with `(*args, **kwargs)`.

Similar to `inspect.Signature.bind`.

sigtools.support.**sort_callsigs**(*sig*, *callsigs*)

Determines which ways to call `sig` in `callsigs` are valid or not.

Returns

Two lists: `(valid, invalid)`.

valid `(args, kwargs, bound)` in which `bound` is the dict returned by *bind_callsig*. It will be equal to the return value of a function with `sig` returned by *f*

invalid `(args, kwargs)`

sigtools.support.**test_func_sig_coherent**(*func*, *check_return=True*, *check_invalid=True*)
Tests if a function is coherent with its signature.

Parameters

- **check_return** (*bool*) – Check if the return value is correct (see *sort_callsigs*)
- **check_invalid** (*bool*) – Make sure call signatures invalid for the signature are also invalid for the passed callable.

Raises AssertionError

4.1.6 sigtools.sphinxext: Extension to make Sphinx use signature objects

sphinx.ext.autodoc can only automatically discover the signatures of basic callables. This extension makes it use *sigtools.specifiers.signature* on the callable instead.

Enable it by appending 'sigtools.sphinxext' to the extensions list in your Sphinx conf.py

4.2 Picking the appropriate arguments for forwards

If automatic signature reporting doesn't work for your use case and you still want to specify how a function's *args, **kwargs is being used, you may either use *sigtools.specifiers.forger_function* and *sigtools.support.s* to override its signature completely, or you can use the forwards_to_* functions from *sigtools.specifiers*.

For forwards_to_* decorators, you only need to specify what function *args, **kwargs are forwarded to and what other arguments are passed in. Here's a primer with common examples on how to use them.

4.2.1 Picking the appropriate forwards_to_* decorator

Several forwards_to_* decorators exist. You must pick one depending on what you are forwarding your parameters to:

forwards_to_function When forwarding to a plain function:

```
def inner(a, b, c):
    ...

@specifiers.forwards_to_function(inner)
def outer(*args, **kwargs):
    inner(*args, **kwargs)
```

Provide the inner function as the first argument to *forwards_to_function*.

forwards_to_method When forwarding to an attribute of the current object, usually to a method:

```
class Spam:
    def inner(self, a, b, c):
        ...

    @specifiers.forwards_to_method('inner')
    def outer(self, *args, **kwargs):
        self.inner(*args, **kwargs)
```

Provide the inner function's *name* to *forwards_to_method*.

You can also specify a "deep" attribute: `'attribute.method'` should be specified if a call is made like this:

```
self.attribute.method(*args, **kwargs)
```

forwards_to_super When forwarding to the superclass's method of the same name:

```
class Spam:
    def method(self):
        pass

class Ham(Spam):
    @specifiers.forwards_to_super()
    def method(self, *args, ham, **kwargs):
        super().method(*args, **kwargs)
```

This only works when using the short form of `super()` introduced in Python 3.3. If you are targetting earlier versions, use *apply_forwards_to_super* on the class, while specifying which methods need to receive the decorator.

For the following examples, we will be using the `forwards_to_function` decorator, although these will work with other `forwards_to_*` decorators.

4.2.2 `*args` and `**kwargs` are forwarded directly if present

You do not need to signal anything about the wrapper function's parameters:

```
@specifiers.forwards_to_function(wrapped)
def outer(arg, *args, **kwargs):
    inner(*args, **kwargs)
```

This holds true even if you omit one of `*args` and `**kwargs`:

```
@specifiers.forwards_to_function(wrapped)
def outer(**kwargs):
    inner(**kwargs)
```

4.2.3 Passing positional arguments to the wrapped function

Indicate the number of arguments you are passing to the wrapped function:

```
@specifiers.forwards_to_function(wrapped, 1)
def outer(*args, **kwargs):
    inner('abc', *args, **kwargs)
```

This applies even if the argument comes from the wrapper:

```
@specifiers.forwards_to_function(wrapped, 1)
def outer(arg, *args, **kwargs):
    inner(arg, *args, **kwargs)
```

4.2.4 Passing named arguments to from the wrapper

Pass the names of the arguments after `num_args`:

```
@specifiers.forwards_to_function(wrapped, 0, 'arg')
def outer(*args, **kwargs):
    inner(*args, arg='abc', **kwargs)
```

Once again, the same goes for if that argument comes from the outer function's:

```
@specifiers.forwards_to_function(wrapped, 0, 'arg')
def outer(*args, arg, **kwargs): # py 3
    inner(*args, arg=arg, **kwargs)
```

If you combine positional and named arguments, follow the previous advice as well:

```
@specifiers.forwards_to_function(wrapped, 2, 'alpha', 'beta')
def outer(two, *args, beta, **kwargs):
    inner(one, two=two, *args, alpha='abc', beta=beta, **kwargs)
```

4.2.5 When the outer function uses `*args` or `**kwargs` but doesn't forward them to the inner function

Pass `use_varargs=False` if your outer function has an `*args`-like parameter but doesn't use it on the inner function directly:

```
@specifiers.forwards_to_function(wrapped, use_varargs=False)
def outer(*args, **kwargs):
    inner(**kwargs)
```

Pass `use_varkwargs=False` if you outer function has a `**kwargs`-like parameter but doesn't use it on the inner function directly:

```
@specifiers.forwards_to_function(wrapped, use_varkwargs=False)
def outer(*args, **kwargs):
    inner(*args)
```

4.2.6 When the outer function passes an arbitrary `*args` or `**kwargs` to the inner function

Pass `hide_args=True` if your outer function uses an arbitrary `*args` when calling the inner function (whether one exists or not in the outer function):

```
@specifiers.forwards_to_function(wrapped, hide_args=True)
def outer(**kwargs):
    args = other_function(...)
    inner(*args, **kwargs)
```

If you know exactly how many items `args` will have, specify the amount of items in `args` instead, as in *Passing positional arguments to the wrapped function*.

Conversely, pass `hide_kwargs=True` if your outer function uses an arbitrary `*kwargs` when calling the inner function (whether one exists or not in the outer function):

```
@specifiers.forwards_to_function(wrapped, hide_args=True)
def outer(*args):
    kwargs = other_function(...)
    inner(*args, **kwargs)
```

If you know exactly which keys `kwargs` will potentially have, specify all possible named keys it might have, as in *Passing named arguments to from the wrapper*.

Note: Neither are needed if the outer function hasn't got an `*args` nor `**kwargs` parameter

4.2.7 Summary

Finally, here's an overview of all parameters from `forwards_to_*` functions

`sigtools.specifiers.`**`forwards_to_function`**(*wrapped*, *num_args=0*, **named_args*, *emulate=None*, *hide_args=False*, *hide_kwargs=False*, *use_varargs=True*, *use_varkwargs=True*, *partial=False*)

num_args The number of arguments you pass by position, excluding `*args`.

***named_args** The names of the arguments you pass by name, excluding `**kwargs`.

use_varargs= Tells if the wrapper's `*args` is being passed to the wrapped function.

use_varkwargs= Tells if the wrapper's `**kwargs` is being passed to the wrapped function.

hide_args= Tells if the wrapped function is given an `*args` parameter (other than the wrapper function's) in such a way that all positional parameters are consumed.

hide_varargs= Tells if the wrapped function is given a `**kargs` parameter (other than the wrapper function's) in such a way that all keyword parameters are consumed.

4.3 Limitations of automatic signature discovery

`sigtools.specifiers.signature` is able to examine a function to determine how its `*args, **kwargs` parameters are being used, even when no information is otherwise provided.

This is very useful for documentation or introspection tools, because it means authors of documented or introspected code don't have to worry about providing this meta-information.

It should handle almost all instances of decorator code, though more unusual code could go beyond its ability to understand it. If this happens it will fall back to a generic signature.

Here is a list of the current limitations:

- It requires the source code to be available. This means automatic introspection of functions that were defined in missing `.py` files, in code passed to `eval()` or in an interactive session will fail.
- It doesn't handle transformations or resetting of `args` and `kwargs`
- It doesn't handle Python 3.5's multiple `*args` and `**kwargs` support
- It doesn't handle calls to the superclass

In some other instances, the signature genuinely can't be determined in advance. For instance, if you call one function or another depending on a parameter, and these functions have incompatible signatures, there wouldn't be *one* common signature for the outer function.

If you still need accurate signature reporting when automatic discovery fails, use the decorators from the *specifiers* module:

See also:

Picking the appropriate arguments for forwards

4.4 Building signature-inspecting tools with sigtools

Whether it is for building documentation or minimizing repetition for the users of your library or framework, inspecting callables might be something that will help you achieve this.

Python has long provided tools to help you do this. The `inspect` module in version 2.1 brought `getargspec` which made use of attributes dating back to Python 1.3. Python 3.3 brought `inspect.signature` which improved upon the concept and made of an object-oriented approach to describing function signatures.

sigtools.signature is a drop-in replacement for `inspect.signature`, with a few key improvements:

- It is available on Python 2, thanks to `funcsigs`
- It can automatically traverse through decorators, while keeping track of which functions owns each parameter
- It supports a mechanism for functions to dynamically alter their reported signature

Separately, *sigtools.modifiers* brings keyword-only parameters and annotations to Python 2, allowing you to rely on these features without having to dismiss Python 2 compatility.

4.4.1 Using `sigtools.signature`

As with `inspect.signature`, simply call *sigtools.signature* with an object to inspect:

```
signature(myfunc)
# <Signature (param, *, decorator_param)>
```

The objects *sigtools.signature* returns are `inspect.Signature` objects. You can get an orderred dict containing all parameters using the `parameter` attribute, and so forth:

```
sig = signature(myfunc)
for param in sig.parameters.values():
    print(param.name, param.kind)
# param POSITIONAL_OR_KEYWORD
# decorator_param KEYWORD_ONLY
```

sigtools.signature adds a `sources` attribute to the signature object. For each parameter, it lists all functions that will receive this parameter:

```
for param in sig.parameters.values():
    print(param.name, sig.sources[param.name])
# param [<function myfunc at ...>]
# decorator_param [<function decorator.<locals>._wrapper at ...>]
```

Additionally, this attribute contains the *depth* of each function, if you need a reliable order for them:

```
print(sig.sources['+depths'])
# {<function decorator.<locals>._wrapper at 0x7f354829c6a8>: 0,
#  <function myfunc at 0x7f354829c730>: 1}
```

4.4.2 Getting more help

If there is anything you wish to discuss more thoroughly, feel free to come by the sigtools gitter chat.

CHAPTER 5

Installing

You can install *sigtools* using pip. If in an activated virtualenv, type:

```
pip install sigtools
```

If you wish to do a user-wide install:

```
pip install --user sigtools
```

Python Module Index

S

Index