funcsigs Documentation

Release 0.1

Aaron Iles

January 06, 2013

CONTENTS

CHAPTER

ONE

THE FUNCSIGS PACKAGE

funcsigs is a backport of the PEP 362 function signature features from Python 3.3's inspect module. The backport is compatible with Python 2.6, 2.7 as well as 3.2 and up.

1.1 Compatability

The *funcsigs* backport has been tested against:

- CPython 2.6
- CPython 2.7
- CPython 3.2

Continuous integration testing is provided by Travis CI.

There is one known compatability issue with Python 2.x when a function is assigned to the `__wrapped__` property of a class after it has been constructed. Otherwise the functionality is believed to be uniform between both Python2 and Python3.

1.2 Issues

Source code for *funcsigs* is hosted on GitHub. Any bug reports or feature requests can be made using GitHub's issues system.

CHAPTER

TWO

INTROSPECTING CALLABLES WITH THE SIGNATURE OBJECT

Note: This section of documentation is a direct repoduction of the Python standard library documentation for the inspect module.

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

signature(*callable*)

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.

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

class Signature

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.

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

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'"
```

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

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

name
The name of the parameter as a string. Must be a valid python identifier name (with the exception of `POSITIONAL_ONLY` parameters, which can have it set to `None`).

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. 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, default=42)
>>> str(param)
'foo=42'

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

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

class 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, it is easy to include them.

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

>>> sig = signature(foo)
>>> ba = sig.bind(5)

>>> ba.args, ba.kwargs
((5,), {})

>>> for param in sig.parameters.values():
...     if param.name not in ba.arguments:
...         ba.arguments[param.name] = param.default

>>> ba.args, ba.kwargs
((5, 10), {})
```

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.

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.

CHAPTER

THREE

COPYRIGHT

funcsigs is a derived work of CPython under the terms of the PSF License Agreement. The original CPython inspect module, its unit tests and documentation are the copyright of the Python Software Foundation. The derived work is distributed under the Apache License Version 2.0.