signature_dispatch

Release 1.0.0

unknown

Mar 02, 2022

CONTENTS

1 Installation 3

2 Usage 5

3 Details 7

4 Applications 9

5 Alternatives 11

`signature_dispatch` is a simple python library for overloading functions based on their call signature and type annotations.

CHAPTER

ONE

INSTALLATION

Install from PyPI:

```
$ pip install signature_dispatch
```

Version numbers follow semantic versioning.

CHAPTER

TWO

USAGE

Use the module itself to decorate multiple functions (or methods) that all have the same name:

```
>>> import signature_dispatch
>>> @signature_dispatch
... def f1(x):
...     return x
...
>>> @signature_dispatch
... def f1(x, y):
...     return x, y
...
```

When called, all of the decorated functions will be tested in order to see if they match the given arguments. The first one that does will be invoked:

```
>>> f1(1)
1
>>> f1(1, 2)
(1, 2)
```

A `TypeError` will be raised if no matches are found:

```
>>> f1(1, 2, 3)
Traceback (most recent call last):
    ...
TypeError: can't dispatch the given arguments to any of the candidate functions:
arguments: 1, 2, 3
candidates:
(x): too many positional arguments
(x, y): too many positional arguments
```

Type annotations are taken into account when choosing which function to invoke:

```
>>> from typing import List
>>> @signature_dispatch
... def f2(x: int):
...     return 'int', x
...
>>> @signature_dispatch
... def f2(x: List[int]):
```

(continues on next page)

(continued from previous page)

```
...     return 'list', x
...
```

```
>>> f2(1)
('int', 1)
>>> f2([1, 2])
('list', [1, 2])
>>> f2('a')
Traceback (most recent call last):
    ...
TypeError: can't dispatch the given arguments to any of the candidate functions:
arguments: 'a'
candidates:
(x: int): type of x must be int; got str instead
(x: List[int]): type of x must be a list; got str instead
>>> f2(['a'])
Traceback (most recent call last):
    ...
TypeError: can't dispatch the given arguments to any of the candidate functions:
arguments: ['a']
candidates:
(x: int): type of x must be int; got list instead
(x: List[int]): type of x[0] must be int; got str instead
```

CHAPTER

THREE

DETAILS

- When using the module directly as a decorator, every decorated function must have the same name and must be defined in the same local scope. If this is not possible (e.g. the implementations are in different modules), every function decorated with `@signature_dispatch` provides an `overload()` method that can be used to add implementations defined elsewhere:

```
>>> @signature_dispatch
... def f3(x):
...     return x
...
>>> @f3.overload
... def _(x, y):
...     return x, y
...
>>> f3(1)
1
>>> f3(1, 2)
(1, 2)
```

- By default, the decorated functions are tried in the order they were defined. If for some reason this order is undesirable, both `@signature_dispatch` and `@*.overload` accept an optional numeric *priority* argument that can be used to specify a custom order. Functions with higher priorities will be tried before those with lower priorities. Functions with the same priority will be tried in the order they were defined. The default priority is 0:

```
>>> @signature_dispatch
... def f4():
...     return 'first'
...
>>> @signature_dispatch(priority=1)
... def f4():
...     return 'second'
...
>>> f4()
'second'
```

- The docstring will be taken from the first decorated function. All other docstrings will be ignored.
- It's possible to use `@signature_dispatch` with class/static methods, but doing so is a bit of a special case. Basically, the class/static method must be applied after all of the overloaded implementations have been defined:

```
>>> class C:
...
```

(continues on next page)

(continued from previous page)

```
...     @signature_dispatch
...     def m(cls, x):
...         return cls, x
...
...     @signature_dispatch
...     def m(cls, x, y):
...         return cls, x, y
...
...     m = classmethod(m)
...
>>> obj = C()
>>> obj.m(1)
(<class '__main__.C'>, 1)
>>> obj.m(1, 2)
(<class '__main__.C'>, 1, 2)
```

Let me know if you find this too annoying. It would probably be possible to special-case class/static methods so that you could just apply both decorators to all the same functions, but that could be complicated and this work-around seems fine for now.

- Calling `@signature_dispatch` may be more expensive than you think, because it has to find the scope that it was called from. This is fast enough that it shouldn't matter in most practical settings, but it does mean that you should take care to not write your code in such a way that, e.g., the `@signature_dispatch` decorator is called every time the function is invoked. Instead, decorate your functions once and then call the resulting function as often as you'd like.
- You can get direct access to the core dispatching functionality provided by this library via the `signature_dispatch.dispatch()` function. This will allow you to call one of several functions based on a given set of arguments, without the need to use any decorators:

```
>>> import signature_dispatch
>>> candidates = [
...         lambda x: x,
...         lambda x, y: (x, y),
... ]
>>> signature_dispatch.dispatch(candidates, args=(1,), kwargs={})
1
>>> signature_dispatch.dispatch(candidates, args=(1, 2), kwargs={})
(1, 2)
```

CHAPTER

FOUR

APPLICATIONS

Writing decorators that can *optionally* be given arguments is tricky to get right, but `signature_dispatch` makes it easy. For example, here is a decorator that prints a message to the terminal every time a function is called and optionally accepts an extra message to print:

```
>>> import signature_dispatch, functools
>>> from typing import Optional

>>> @signature_dispatch
... def log(msg: Optional[str]=None):
...     def decorator(f):
...         @functools.wraps(f)
...         def wrapper(*args, **kwargs):
...             print("Calling:", f.__name__)
...             if msg: print(msg)
...             return f(*args, **kwargs)
...         return wrapper
...     return decorator
...
>>> @signature_dispatch
... def log(f):
...     return log()(f)
```

Using `@log` without an argument:

```
>>> @log
... def foo():
...     pass
>>> foo()
Calling: foo
```

Using `@log` with an argument:

```
>>> @log("Hello world!")
... def bar():
...     pass
>>> bar()
Calling: bar
Hello world!
```

CHAPTER

FIVE

ALTERNATIVES

The dispatching library does almost the same thing as this one, with a few small differences:

- More boilerplate.
- Subscripted generic types (e.g. `List[int]`) are not supported.
- Annotations can be arbitrary functions.

PEP 3124 proposes to add something similar to `@signature_dispatch` to the python standard library, but appears to have been stalled for over a decade.