# variants Documentation

*Release 0.2.1.dev7+g1c2bf41*

Paul Ganssle

Jun 18, 2020

# Contents

CHAPTER 1

# variants

variants is a library that provides syntactic sugar for creating alternate forms of functions and other callables, in the same way that alternate constructors are class methods that provide alternate forms of the constructor function.

To create a function with variants, simply decorate the primary form with @variants.primary, which then adds the .variant decorator to the original function, which can be used to register new variants. Here is a simple example of a function that prints text, with variants that specify the source of the text to print:

```
import variants

@variants.primary
def print_text(txt):
    print(txt)

@print_text.variant('from_file')
def print_text(fobj):
    print_text(fobj.read())

@print_text.variant('from_filepath')
def print_text(fpath):
    with open(fpath, 'r') as f:
        print_text.from_file(f)

@print_text.variant('from_url')
def print_text(url):
    import requests
    r = requests.get(url)
    print_text(r.text)
```

print_text and its variants can be used as such:

```
print_text('Hello, world!')                    # Hello, world!

# Create a text file
with open('hello_world.txt', 'w') as f:
    f.write('Hello, world (from file)')

# Print from an open file object
with open('hello_world.txt', 'r') as f:
    print_text.from_file(f)                    # Hello, world (from file)

# Print from the path to a file object
print_text.from_filepath('hello_world.txt') # Hello, world (from file)

# Print from a URL
hw_url = 'https://ganssle.io/files/hello_world.txt'
print_text.from_url(hw_url)                    # Hello, world! (from url)
```

## 1.1 Differences from singledispatch

While `variants` and `singledispatch` are both intended to provide alternative implementations to a primary function, the overall aims are slightly different. `singledispatch` transparently dispatches to variant functions based on the *type* of the argument, whereas `variants` provides *explicit* alternative forms of the function. Note that in the above example, both `print_text.from_filepath` and `print_text.from_url` take a string, one representing a file path and one representing a URL.

Additionally, the `variants` is compatible with `singledispatch`, so you can have the best of both worlds; an example that uses both:

```
@variants.primary
@singledispatch
def add(x, y):
    return x + y

@add.variant('from_list')
@add.register(list)
def add(x, y):
    return x + [y]
```

Which then automatically dispatches between named variants based on type:

```
>>> add(1, 2)
3
>>> add([1], 2)
[1, 2]
```

But also exposes the explicit variant functions:

```
>>> add.from_list([1], 2)
[1, 2]
>>> add.from_list()
      7 @add.register(list)
      8 def add(x, y):
----> 9     return x + [y]

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

It is important to note that the `variants` decorators **must be the outer decorators**.

## 1.2 Installation

To install variants, run this command in your terminal:

```
$ pip install variants
```

## 1.3 Requirements

This is a library for Python, with support for versions 2.7 and 3.4+.

# CHAPTER 2

# Documentation

Contents:

## 2.1 variants

`variants` is a library that provides syntactic sugar for creating alternate forms of functions and other callables, in the same way that alternate constructors are class methods that provide alternate forms of the constructor function.

To create a function with variants, simply decorate the primary form with `@variants.primary`, which then adds the `.variant` decorator to the original function, which can be used to register new variants. Here is a simple example of a function that prints text, with variants that specify the source of the text to print:

```
import variants

@variants.primary
def print_text(txt):
    print(txt)

@print_text.variant('from_file')
def print_text(fobj):
    print_text(fobj.read())

@print_text.variant('from_filepath')
def print_text(fpath):
    with open(fpath, 'r') as f:
        print_text.from_file(f)

@print_text.variant('from_url')
```

(continues on next page)

(continued from previous page)

```
def print_text(url):
    import requests
    r = requests.get(url)
    print_text(r.text)
```

`print_text` and its variants can be used as such:

```
print_text('Hello, world!')                    # Hello, world!

# Create a text file
with open('hello_world.txt', 'w') as f:
    f.write('Hello, world (from file)')

# Print from an open file object
with open('hello_world.txt', 'r') as f:
    print_text.from_file(f)                    # Hello, world (from file)

# Print from the path to a file object
print_text.from_filepath('hello_world.txt') # Hello, world (from file)

# Print from a URL
hw_url = 'https://ganssle.io/files/hello_world.txt'
print_text.from_url(hw_url)                    # Hello, world! (from url)
```

### 2.1.1 Differences from singledispatch

While `variants` and `singledispatch` are both intended to provide alternative implementations to a primary function, the overall aims are slightly different. `singledispatch` transparently dispatches to variant functions based on the *type* of the argument, whereas `variants` provides *explicit* alternative forms of the function. Note that in the above example, both `print_text.from_filepath` and `print_text.from_url` take a string, one representing a file path and one representing a URL.

Additionally, the `variants` is compatible with `singledispatch`, so you can have the best of both worlds; an example that uses both:

```
@variants.primary
@singledispatch
def add(x, y):
    return x + y

@add.variant('from_list')
@add.register(list)
def add(x, y):
    return x + [y]
```

Which then automatically dispatches between named variants based on type:

```
>>> add(1, 2)
3
>>> add([1], 2)
[1, 2]
```

But also exposes the explicit variant functions:

```
>>> add.from_list([1], 2)
[1, 2]
>>> add.from_list()
      7 @add.register(list)
      8 def add(x, y):
----> 9     return x + [y]

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

It is important to note that the `variants` decorators **must be the outer decorators**.

### 2.1.2 Installation

To install variants, run this command in your terminal:

```
$ pip install variants
```

### 2.1.3 Requirements

This is a library for Python, with support for versions 2.7 and 3.4+.

## 2.2 Installation

### 2.2.1 Stable release

To install variants, run this command in your terminal:

```
$ pip install variants
```

This is the preferred method to install variants, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

### 2.2.2 From sources

The sources for variants can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/python-variants/variants
```

Or download the tarball:

```
$ curl  -OL https://github.com/python-variants/variants/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

## 2.3 Usage

To use `variants` in a project, decorate the primary function with the `@variants.primary` decorator, which will add the `variant` method to your original function.

`variants.`**`primary`**`(`*f*`)`

Decorator to register a function that has variant forms.

Decorate the main form of the function with this decorator, and then subsequent variants should be declared with the same name as the original function[1]:

```
import variants

@variants.primary
def myfunc(fpath):
    with open(fpath, 'r') as f:
        do_something(f.read())

@myfunc.variant('from_url') as f:
def myfunc(url):
    r = requests.get(url)
    do_something(r.text)
```

The `primary` decorator returns an object that attempts to transparently proxy the original methods of the original callable, but variants added to the primary function will shadow the original methods and attributes. Other than this, any valid python identifier is a valid name for a variant.

## 2.4 Documentation example

Using the sphinx_autodoc_variants extension, primary and variant functions can be grouped together as a single function group.

### 2.4.1 Functions

With the `sphinx_autodoc_variants` extension enabled in `conf.py`, function groups at the module level will automatically be grouped together by the `automodule` directive. To explicitly document a function group, use the `autoprimary_function` directive:

```
.. autoprimary_function:: examples.example_variants.primary_func
    :members:
```

Which generates:

`examples.example_variants.`**`primary_func`**`(`*x, y*`)`

This is the primary function. Its docstring should be shown under the primary function documentation.

`primary_func.`**`onearg`**`(`*x*`)`

This is the `onearg` variant, it only takes one argument.

`primary_func.`**`threearg`**`(`*x, y, z*`)`

This is the `threearg` variant, it takes three arguments.

[1] Declaring subsequent variants with the same name as the original function is a stylistic convention, not a requirement. Decorating any function with the `.variant` decorator will mutate the primary function object, no matter the name of the variant function. However, whatever function you use for the variant function declaration will become an alias for the primary function.

### 2.4.2 Methods

For a class containing function group methods, the `autoclass` directive works, so:

```
.. autoclass:: examples.example_variants.VariantMethodsClass
    :members:
```

Resulting in:

**class** `examples.example_variants.`**`VariantMethodsClass`**

**`normal_method`**(*a*, *b*)
This is a normal method, it has no variants

**`primary_method`**(*x*, *y*)
This is the primary method, its docstring should be shown under the primary method documentation.

`primary_method.`**`onearg`**(*x*)
This is the `onearg` variant, it takes one argument.

As with functions, individual method groups can be documented using the `autoprimary_method` directive:

```
.. autoprimary_method:: examples.example_variants.VariantMethodsClass.primary_method
    :members:
```

Which generates:

`VariantMethodsClass.`**`primary_method`**(*x*, *y*)
This is the primary method, its docstring should be shown under the primary method documentation.

`primary_method.`**`onearg`**(*x*)
This is the `onearg` variant, it takes one argument.

## 2.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

### 2.5.1 Types of Contributions

#### Report Bugs

Report bugs at https://github.com/python-variants/variants/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

#### Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

#### Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

#### Write Documentation

variants could always use more documentation, whether as part of the official variants docs, in docstrings, or even on the web in blog posts, articles, and such.

#### Submit Feedback

The best way to send feedback is to file an issue at https://github.com/python-variants/variants/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

### 2.5.2 Get Started!

Ready to contribute? Here's how to set up *variants* for local development.

1. Fork the *variants* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/variants.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv variants
   $ cd variants/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 variants tests
   $ python setup.py test or py.test
   $ tox
   ```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

### 2.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, 3.6 and for PyPy. Check https://travis-ci.org/python-variants/variants/pull_requests and make sure that the tests pass for all supported Python versions.

## 2.6 Credits

### 2.6.1 Development Lead

- Paul Ganssle <paul@ganssle.io>

### 2.6.2 Contributors

None yet. Why not be the first?

## 2.7 Changelog

### 2.7.1 Version 0.2.0 (2018-05-03)

#### Features

- Added the `variants.inspect` module, which provides a public API for inspecting variant function groups. It currently provides two functions: `is_primary` and `is_primary_method`. (GH #29)
- Added python_requires to setup, indicating Python version support. (GH #34)

#### Bugfixes

- Updated MANIFEST.in to include NEWS.rst (GH #34)

### Improved Documentation

- Updated permanent redirect links in documentation. (GH #28)
- Flattened TOC entry for changelog in documentation. (GH #32)
- Added section for API documentation, including documentation for the inspect submodule. (GH #33)

## 2.8 Variants API Documentation

### 2.8.1 variants

`variants.`**`primary`**`(`*f*`)`

Decorator to register a function that has variant forms.

Decorate the main form of the function with this decorator, and then subsequent variants should be declared with the same name as the original function[1]:

```
import variants

@variants.primary
def myfunc(fpath):
    with open(fpath, 'r') as f:
        do_something(f.read())

@myfunc.variant('from_url') as f:
def myfunc(url):
    r = requests.get(url)
    do_something(r.text)
```

The `primary` decorator returns an object that attempts to transparently proxy the original methods of the original callable, but variants added to the primary function will shadow the original methods and attributes. Other than this, any valid python identifier is a valid name for a variant.

### 2.8.2 inspect

Provides inspection tools for extracting metadata from function groups.

`variants.inspect.`**`is_primary`**`(`*f*`)`

Detect if a function is a primary function in a variant group

`variants.inspect.`**`is_primary_method`**`(`*f*`)`

Detect if a function is a primary method in a variant group

[1] Declaring subsequent variants with the same name as the original function is a stylistic convention, not a requirement. Decorating any function with the `.variant` decorator will mutate the primary function object, no matter the name of the variant function. However, whatever function you use for the variant function declaration will become an alias for the primary function.

# CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index