Real Python Cheat Sheet

realpython.com

Python Decorators Examples

Learn more about decorators in Python in our in-depth tutorial at realpython.com/primer-on-python-decorators/

Using decorators

The normal way of using a decorator is by specifying it just before the definition of the function you want to decorate:

```
@decorator
def f(arg_1, arg_2):
    ...
```

If you want to decorate an already existing function you can use the following syntax:

```
f = decorator(f)
```

Decorator not changing the decorated function

If you don't want to change the decorated function, a decorator is simply a function taking in and returning a function:

```
def name(func):
    # Do something with func
    return func
```

Example: Register a list of decorated functions.

```
def register(func):
    """Register a function as a plug-in"""
    PLUGINS[func.__name__] = func
    return func
```

Basic decorator

Template for basic decorator that can modify the decorated function:

```
import functools

def name(func):
    @functools.wraps(func)
    def wrapper_name(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_name
```

Example: A `timer` decorator that prints the runtime of a function.

```
import functools
import time

def timer(func):
    """Print the runtime of the function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Run time: {run_time:.4f} secs")
        return value
    return wrapper_timer
```

Decorator with arguments

If you want your decorator to take arguments, create a decorator factory that can create decorators:

```
import functools

def name(arg_1, ...):
    def decorator_name(func):
        @functools.wraps(func)
        def wrapper_name(*args, **kwargs):
            # Do something before using arg_1, ...
            value = func(*args, **kwargs)
            # Do something after using arg_1, ...
            return value
        return wrapper_name
    return decorator_name
```

Example: Rate limit your code by sleeping a given amount of seconds before calling the function.

```
import functools
import time

def slow_down(rate):
    """Sleep before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down
    return decorator_slow_down
```

Decorators that can optionally take arguments

If you want your decorator to be able to be called with or without arguments, you need a dummy argument, _func, that is set automatically if the decorator is called without arguments:

```
import functools

def name(_func=None, *, arg_1=val_1, ...):
    def decorator_name(func):
        @functools.wraps(func)
        def wrapper_name(*args, **kwargs):
            # Do something before using arg_1, ...
            value = func(*args, **kwargs)
            # Do something after using arg_1,  ...
            return value
        return wrapper_name

    if _func is None:
        return decorator_name
    else:
        return decorator_name(_func)
```

Example: Rate limit your code by sleeping an optionally given amount of seconds before calling the function.

```
import functools
import time

def slow_down(_func=None, *, rate=1):
    """Sleep before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down

    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
```

Decorators that keep state

If you need your decorator to maintain state, use a class as a decorator:

```
import functools

class Name:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        # Initialize state attributes

    def __call__(self, *args, **kwargs):
        # Update state attributes
        return self.func(*args, **kwargs)
```

Example: Count the number of times the decorated function is called.

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls}")
        return self.func(*args, **kwargs)
```