Make Your Python Prettier With Decorators

Originalartikel

Backup

<html> Many Pythonistas are familiar with using decorators, but far fewer understand what’s happening under the hood and can write their own. It takes a little effort to learn their subtleties but, once grasped, they’re a great tool for writing concise, elegant Python. This post will briefly introduce the concept, start with a basic decorator implementation, then walk through a few more involved examples one by one. <h1>What is a decorator</h1> Decorators are most commonly used with the

```
@decorator
```

 syntax. You may have seen Python that looks something like these examples. <pre class=„brush: python; title: ; notranslate“ title=„“> @app.route(„/home“) def home():

```
  return render_template("index.html")
```

@performance_analysis def foo():

```
  pass
```

@property def total_requests(self):

```
  return self._total_requests
```

</pre> To understand what a decorator does, we first have to take a step back and look at some of the things we can do with functions in Python. <pre class=„brush: python; title: ; notranslate“ title=„“> def get_hello_function(punctuation):

```
  """Returns a hello world function, with or without punctuation."""
  def hello_world():
      print("hello world")
  def hello_world_punctuated():
      print("Hello, world!")
  if punctuation:
      return hello_world_punctuated
  else:
      return hello_world
```

if name == 'main':

```
  ready_to_call = get_hello_function(punctuation=True)
  ready_to_call()
  # "Hello, world!" is printed
```

</pre> In the above snippet,

```
get_hello_function
```

 returns a function. The returned function gets assigned and then called. This flexibility in the way functions can be used and manipulated is key to the operation of decorators. As well as returning functions, we can also pass functions as arguments. In the example below, we wrap a function, adding a delay before it’s called. <pre class=„brush: python; title: ; notranslate“ title=„“> from time import sleep def delayed_func(func):

```
  """Return a wrapper which delays `func` by 10 seconds."""
  def wrapper():
      print("Waiting for ten seconds...")
      sleep(10)
      # Call the function that was passed in
      func()
  return wrapper
```

def print_phrase():

```
  print("Fresh Hacks Every Day")
```

if name == 'main':

```
  delayed_print_function = delayed_func(print_phrase)
  delayed_print_function()
```

</pre> This can feel a bit confusing at first, but we’re just defining a new function

```
wrapper
```

, which sleeps before calling

```
func
```

. It’s important to note that we haven’t changed the behaviour of

```
func
```

 itself, we’ve only returned a different function which calls

```
func
```

 after a delay. When the code above is run, the following output is produced: <pre class=„brush: plain; title: ; notranslate“ title=„“> $ python decorator_test.py Waiting for ten seconds... Fresh Hacks Every Day </pre> <h2>Let’s make it pretty</h2> If you rummage around the internet for information on decorators, the phrase you’ll see again and again is “syntactic sugar”. This does a good job of explaining what decorators are: simply a shortcut to save typing and improve readability. The

```
@decorator
```

 syntax makes it very easy to apply our wrapper to any function. We could re-write our delaying code above like this: <pre class=„brush: python; title: ; notranslate“ title=„“> from time import sleep def delayed_func(func):

```
  """Return `func`, delayed by 10 seconds."""
  def wrapper():
      print("Waiting for ten seconds...")
      sleep(10)
      # Call the function that was passed in
      func()
  return wrapper
```

@delayed_func def print_phrase():

```
  print("Fresh Hacks Every Day")
```

if name == 'main':

```
  print_phrase()
```

</pre> Decorating

```
print_phrase
```

 with

```
@delayed_func
```

 automatically does the wrapping, meaning that whenever

```
print_phrase
```

 is called we get the delayed wrapper instead of the original function;

```
print_phrase
```

 has been replaced by

```
wrapper
```

. <h1>Why is this useful?</h1> Decorators can’t change a function, but they can extend its behaviour, modify and validate inputs and outputs, and implement any other external logic. The benefit of writing decorators comes from their ease of use once written. In the example above we could easily add

```
@delayed_func
```

 to any function of our choice. This ease of application is useful for debug code as well as program code. One of the most common applications for decorators is to provide debug information on the performance of a function. Let’s write a simple decorator which logs the datetime the function was called at, and the time taken to run. <pre class=„brush: python; title: ; notranslate“ title=„“> import datetime import time from app_config import log def log_performance(func):

```
  def wrapper():
      datetime_now = datetime.datetime.now()
      log.debug(f"Function {func.__name__} being called at {datetime_now}")
      start_time = time.time()
      func()
      log.debug(f"Took {time.time() - start_time} seconds")
  return wrapper
```

@log_performance def calculate_squares():

```
  for i in range(10_000_000):
      i_squared = i**2
```

if name == 'main':

```
  calculate_squares()
```

</pre> In the code above we use our

```
log_performance
```

 decorator on a function which calculates the squares of the numbers 0 to 10,000,000. This is the output when run: <pre class=„brush: plain; title: ; notranslate“ title=„“> $ python decorator_test.py Function calculate_squares being called at 2018-08-23 12:39:02.112904 Took 2.5019338130950928 seconds </pre> <h1>Dealing with parameters</h1> In the example above, the

```
calculate_squares
```

 function didn’t need any parameters, but what if we wanted to make our

```
log_performance
```

 decorator work with any function that takes any parameters? The solution is simple: allow

```
wrapper
```

 to accept arguments, and pass those arguments directly into

```
func
```

. To allow for any number of arguments and keyword arguments, we’ve used

```
*args, **kwargs
```

, passing all of the arguments to the wrapped function. <pre class=„brush: python; title: ; notranslate“ title=„“> import datetime import time from app_config import log def log_performance(func):

```
  def wrapper(*args, **kwargs):
      datetime_now = datetime.datetime.now()
      log.debug(f"Function {func.__name__} being called at {datetime_now}")
      start_time = time.time()
      result = func(*args, **kwargs)
      log.debug(f"Took {time.time() - start_time} seconds")
      return result
  return wrapper
```

@log_performance def calculate_squares(n):

```
  """Calculate the squares of the numbers 0 to n."""
  for i in range(n):
      i_squared = i**2
```

if name == 'main':

```
  calculate_squares(10_000_000) # Python 3!
```

</pre> Note that we also capture the result of the

```
func
```

call and use it as the return value of the wrapper. <h1>Validation</h1> Another common use case of decorators is to validate function arguments and return values. Here’s an example where we’re dealing with multiple functions which return an IP address and port in the same format. <pre class=„brush: python; title: ; notranslate“ title=„“> def get_server_addr():

```
  """Return IP address and port of server."""
  ...
  return ('192.168.1.0', 8080)
```

def get_proxy_addr():

```
  """Return IP address and port of proxy."""
  ...
  return ('127.0.0.1', 12253)
```

</pre> If we wanted to do some basic validation on the returned port, we could write a decorator like so: <pre class=„brush: python; title: ; notranslate“ title=„“> PORTS_IN_USE = [1500, 1834, 7777] def validate_port(func):

```
  def wrapper(*args, **kwargs):
      # Call `func` and store the result
      result = func(*args, **kwargs)
      ip_addr, port = result
      if port &lt; 1024:
          raise ValueError("Cannot use priviledged ports below 1024")
      elif port in PORTS_IN_USE:
          raise RuntimeError(f"Port {port} is already in use")
      # If there were no errors, return the result
      return result
  return wrapper
```

</pre> Now it’s easy to ensure our ports are validated, we simply decorate any appropriate function with

```
@validate_port
```

. <pre class=„brush: python; title: ; notranslate“ title=„“> @validate_port def get_server_addr():

```
  """Return IP address and port of server."""
  ...
  return ('192.168.1.0', 8080)
```

@validate_port def get_proxy_addr():

```
  """Return IP address and port of proxy."""
  ...
  return ('127.0.0.1', 12253)
```

</pre> The advantage of this approach is that validation is done externally to the function – there’s no risk that changes to the internal function logic or order will affect validation. <h2>Dealing with function attributes</h2> Let’s say we now want to access some of the metadata of the

```
get_server_addr
```

 function above, like the name and docstring. <pre class=„brush: python; title: ; notranslate“ title=„“> >>> get_server_addr.name 'wrapper' >>> get_server_addr.doc >>> </pre> Disaster! Since our

```
validate_port
```

 decorator essentially replaces the functions it decorates with our wrapper, all of the function attributes are those of

```
wrapper
```

, not the original function. Fortunately, this problem is common, and the

```
functools
```

 module in the standard library has a solution:

```
wraps
```

. Let’s use it in our

```
validate_port
```

 decorator, which now looks like this: <pre class=„brush: python; title: ; notranslate“ title=„“> from functools import wraps def validate_port(func):

```
  @wraps(func)
  def wrapper(*args, **kwargs):
      # Call `func` and store the result
      result = func(*args, **kwargs)
      ip_addr, port = result
      if port &lt; 1024:
          raise ValueError("Cannot use priviledged ports below 1024")
      elif port in PORTS_IN_USE:
          raise RuntimeError(f"Port {port} is already in use")
      # If there were no errors, return the result
      return result
  return wrapper
```

</pre> Line 4 indicates that

```
wrapper
```

should preserve the metadata of

```
func
```

, which is exactly what we want. Now when we try and access metadata, we get what we expect. <pre class=„brush: python; title: ; notranslate“ title=„“> >>> get_server_addr.name 'get_server_addr' >>> get_server_addr.doc 'Return IP address and port of server.' >>> </pre> <h1>Summary</h1> Decorators are a great way to make your codebase more flexible and easy to maintain. They provide a simple way to do runtime validation on functions and are handy for debugging as well. Even if writing custom decorators isn’t your thing, an understanding of what makes them tick will be a significant asset when understanding third-party code and for utilising decorators which are already written. </html>