Real-world examples

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @
American Efficient

Time a function

```python
import time

def timer(func):
  """A decorator that prints how long a function took to run.

  Args:
    func (callable): The function being decorated.

  Returns:
    callable: The decorated function.
  """
```

```python
import time

def timer(func):
  """A decorator that prints how long a function took to run."""
  # Define the wrapper function to return.
  def wrapper(*args, **kwargs):
    # When wrapper() is called, get the current time.
    t_start = time.time()
    # Call the decorated function and store the result.
    result = func(*args, **kwargs)
    # Get the total time it took to run, and print it.
    t_total = time.time() - t_start
    print('{} took {}s'.format(func.__name__, t_total))
    return result
  return wrapper
```

Using timer()

```
@timer
def sleep_n_seconds(n):
  time.sleep(n)
```

```
sleep_n_seconds(5)
```

```
sleep_n_seconds took 5.0050950050354s
```

```
sleep_n_seconds(10)
```

```
sleep_n_seconds took 10.010067701339722s
```

```python
def memoize(func):
  """Store the results of the decorated function for fast lookup
  """
  # Store results in a dict that maps arguments to results
  cache = {}
  # Define the wrapper function to return.
  def wrapper(*args, **kwargs):
    # If these arguments haven't been seen before,
    if (args, kwargs) not in cache:
      # Call func() and store the result.
      cache[(args, kwargs)] = func(*args, **kwargs)
    return cache[(args, kwargs)]
  return wrapper
```

5. Memoizing

Memoizing is the process of storing the results of a function so that the next time the function is called with the same arguments; you can just look up the answer.

We start by setting up a dictionary that will map arguments to results. Then, as usual, we create wrapper() to be the new decorated function that this decorator returns. When the new function gets called, we check to see whether we've ever seen these arguments before. If we haven't, we send them to the decorated function and store the result in the "cache" dictionary. Now we can look up the return value quickly in a dictionary of results. The next time we call this function with those same arguments, the return value will already be in the dictionary.

6. Using memoize()

Here we are memoizing slow_function(). slow_function() simply returns the sum of its arguments. In order to simulate a slow function, we have it sleep for 5 seconds before returning. If we call slow_function() with the arguments 3 and 4, it will sleep for 5 seconds and then return 7. But if we call slow_function() with the arguments 3 and 4 again, it will immediately return 7. Because we've stored the answer in the cache, the decorated function doesn't even have to call the original slow_function() function.

```python
@memoize
def slow_function(a, b):
  print('Sleeping...')
  time.sleep(5)
  return a + b
```

```python
slow_function(3, 4)
```

```
Sleeping...
7
```

```python
slow_function(3, 4)
```

```
7
```

When to use decorators

- Add common behavior to multiple functions

```
@timer
def foo():
  # do some computation

@timer
def bar():
  # do some other computation

@timer
def baz():
  # do something else
```

7. When to use decorators
So when is it appropriate to use a decorator? You should consider using a decorator when you want to add some common bit of code to multiple functions. We could have added timing code in the body of all three of these functions, but that would violate the principle of Don't Repeat Yourself. Adding a decorator is a better choice.

Let's practice!

WRITING FUNCTIONS IN PYTHON

Decorators and metadata

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

```python
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
print(sleep_n_seconds.__doc__)
```

```
Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
```

```python
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
print(sleep_n_seconds.__name__)
```

```
sleep_n_seconds
```

```python
print(sleep_n_seconds.__defaults__)
```

```
(10,)
```

```
@timer
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
print(sleep_n_seconds.__doc__)
```

```
print(sleep_n_seconds.__name__)
```

```
wrapper
```

The timer decorator

```
def timer(func):
  """A decorator that prints how long a function took to run."""

  def wrapper(*args, **kwargs):
    t_start = time.time()

    result = func(*args, **kwargs)

    t_total = time.time() - t_start
    print('{} took {}s'.format(func.__name__, t_total))

    return result

  return wrapper
```

5. The timer decorator
To understand why we have to examine the timer() decorator. Remember that when we write decorators, we almost always define a nested function to return. Because the decorator overwrites the sleep_n_seconds() function, when you ask for sleep_n_seconds()'s docstring or name, you are actually referencing the nested function that was returned by the decorator. In this case, the nested function was called wrapper() and it didn't have a docstring.

```python
from functools import wraps
def timer(func):
  """A decorator that prints how long a function took to run."""

  @wraps(func)
  def wrapper(*args, **kwargs):
    t_start = time.time()

    result = func(*args, **kwargs)

    t_total = time.time() - t_start
    print('{} took {}s'.format(func.__name__, t_total))

    return result
  return wrapper
```

```python
@timer
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
print(sleep_n_seconds.__doc__)
```

```
Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
```

```
@timer
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
print(sleep_n_seconds.__name__)
```

```
sleep_n_seconds
```

```
print(sleep_n_seconds.__defaults__)
```

```
(10,)
```

Access to the original function

```python
@timer
def sleep_n_seconds(n=10):
  """Pause processing for n seconds.

  Args:
    n (int): The number of seconds to pause for.
  """
  time.sleep(n)
sleep_n_seconds.__wrapped__
```

```
<function sleep_n_seconds at 0x7f52cab44ae8>
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Decorators that take arguments

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @
American Efficient

```python
def run_three_times(func):
  def wrapper(*args, **kwargs):
    for i in range(3):
      func(*args, **kwargs)
  return wrapper
@run_three_times
def print_sum(a, b):
  print(a + b)
print_sum(3, 5)
```

```
8
8
8
```

run_n_times()

```python
def run_n_times(func):
  def wrapper(*args, **kwargs):
    # How do we pass "n" into this function?
    for i in range(???):
      func(*args, **kwargs)
  return wrapper
@run_n_times(3)
def print_sum(a, b):
  print(a + b)
@run_n_times(5)
def print_hello():
  print('Hello!')
```

A decorator factory

```python
def run_n_times(n):
  """Define and return a decorator"""
  def decorator(func):
    def wrapper(*args, **kwargs):
      for i in range(n):
        func(*args, **kwargs)
    return wrapper
  return decorator
@run_n_times(3)
def print_sum(a, b):
  print(a + b)
```

4. A decorator factory

To make run_n_times() work, we have to turn it into a function that returns a decorator, rather than a function that is a decorator. So let's start by redefining run_n_times() so that it takes n as an argument, instead of func. Then, inside of run_n_times(), we'll define a new decorator function. This function takes "func" as an argument because it is the function that will be acting as our decorator. We start our new decorator with a nested wrapper() function, as usual. Now, since we are still inside the run_n_times() function, we have access to the n parameter that was passed to run_n_times(). We can use that to control how many times we repeat the loop that calls our decorated function. As usual for any decorator, we return the new wrapper() function. And, if run_n_times() returns the decorator() function we just defined, then we can use that return value as a decorator. Notice how when we decorate print_sum() with run_n_times(), we use parentheses after @run_n_times. This indicates that we are actually calling run_n_times() and decorating print_sum() with the result of that function call. Since the return value from run_n_times() is a decorator function, we can use it to decorate print_sum().

```python
def run_n_times(n):
  """Define and return a decorator"""
  def decorator(func):
    def wrapper(*args, **kwargs):
      for i in range(n):
        func(*args, **kwargs)
    return wrapper
  return decorator
run_three_times = run_n_times(3)
@run_three_times
def print_sum(a, b):
  print(a + b)
@run_n_times(3)
def print_sum(a, b):
  print(a + b)
```

5. Expanded code

This is a little bit confusing, so let me show you how this works without using decorator syntax. Like before, we have a function, run_n_times() that returns a decorator function when you call it. If we call run_n_times() with the argument 3, it will return a decorator. In fact, it returns the decorator that we defined at the beginning of this lesson, run_three_times(). We could decorate print_sum() with this new decorator using decorator syntax. Python makes it convenient to do both of those in a single step though. When we use decorator syntax, the thing that comes after the @ symbol must be a reference to a decorator function. We can use the name of a specific decorator, or we can call a function that returns a decorator.

Using run_n_times()

```
@run_n_times(3)
def print_sum(a, b):
  print(a + b)
print_sum(3, 5)
```

```
8
8
8
```

```
@run_n_times(5)
def print_hello():
  print('Hello!')
print_hello()
```

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

6. Using run_n_times()
To prove to you that it works the way we expect here is print_sum() decorated with run_n_times(3). When we call print_sum() with the arguments 3 and 5, it prints 8 three times. And we can just as easily decorate print_hello(), which prints a hello message, with run_n_times(5). When we call print_hello(), we get five hello messages, as expected.

Let's practice!

WRITING FUNCTIONS IN PYTHON

Timeout(): a real world example

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Timeout

```
def function1():
  # This function sometimes
  # runs for a loooong time
  ...

def function2():
  # This function sometimes
  # hangs and doesn't return
  ...
```

Timeout

```
@timeout
def function1():
  # This function sometimes
  # runs for a loooong time
  ...
@timeout
def function2():
  # This function sometimes
  # hangs and doesn't return
  ...
```

Timeout - background info

```python
import signal
def raise_timeout(*args, **kwargs):
  raise TimeoutError()
# When an "alarm" signal goes off, call raise_timeout()
signal.signal(signalnum=signal.SIGALRM, handler=raise_timeout)
# Set off an alarm in 5 seconds
signal.alarm(5)
# Cancel the alarm
signal.alarm(0)
```

```python
def timeout_in_5s(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    # Set an alarm for 5 seconds
    signal.alarm(5)
    try:
      # Call the decorated func
      return func(*args, **kwargs)
    finally:
      # Cancel alarm
      signal.alarm(0)
  return wrapper
```

```python
@timeout_in_5s
def foo():
  time.sleep(10)
  print('foo!')
```

```python
foo()
```

```
TimeoutError
```

```python
def timeout(n_seconds):
  def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
      # Set an alarm for n seconds
      signal.alarm(n_seconds)
      try:
        # Call the decorated func
        return func(*args, **kwargs)
      finally:
        # Cancel alarm
        signal.alarm(0)
    return wrapper
  return decorator
```

```python
@timeout(5)
def foo():
  time.sleep(10)
  print('foo!')
@timeout(20)
def bar():
  time.sleep(10)
  print('bar!')
foo()
```

```
TimeoutError
```

```python
bar()
```

```
bar!
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Great job!

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @
American Efficient

Chapter 1 - Best Practices

- Docstrings
- DRY and Do One Thing
- Pass by assignment (mutable vs immutable)

Chapter 2 - Context Managers

```
with my_context_manager() as value:
  # do something
```

```
@contextlib.contextmanager
def my_function():
  # this function can be used in a "with" statement now
```

Chapter 3 - Decorators

```
@my_decorator
def my_decorated_function():
  # do something
```

```
def my_decorator(func):
  def wrapper(*ars, **kwargs):
    return func(*args, **kwargs)
  return wrapper
```

Chapter 4 - More on Decorators

```
def my_decorator(func):
  @functools.wraps(func)
  def wrapper(*ars, **kwargs):
    return func(*args, **kwargs)
  return wrapper
```

Chapter 4 - More on Decorators

```
def decorator_that_takes_args(a, b, c):
  def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
      return func(*args, **kwargs)
    return wrapper
  return decorator
```

Thank you!

WRITING FUNCTIONS IN PYTHON