Real-world examples

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Director of Software Engineering @ American Efficient

datacamp

Time a function

It helps to figure out where the computational bottle-necks are

```
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.
  """
```

```
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 5.00509500503545s
```

```
sleep_n_seconds(10)
```

```
sleep_n_seconds took 10.010067701339722s
```

```
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
```

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

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

```
Sleeping...
7
```

```
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
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Decorators and metadata

WRITING FUNCTIONS IN PYTHON

Metadata is "data that provides information about other data". In other words, it is "data about data"

Shayne Miel
Director of Software Engineering @ American Efficient

```
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.
```

```
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,)
```

```
@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
```

```
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
```

```
@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

```
@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

datacamp

```
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()

```
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

```
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)
```

```
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)
```

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!
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Timeout(): a real world example

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Director of Software Engineering @ American Efficient

datacamp

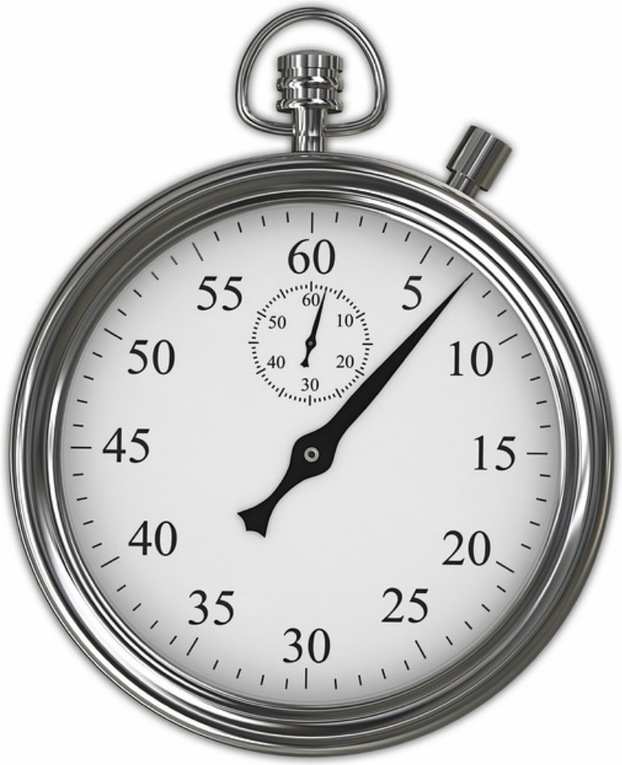
Timeout

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

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

Timeout

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



Timeout - background info

```
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)
```

```
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
```

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

```
foo()
```

```
TimeoutError
```

```
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
```

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

```
TimeoutError
```

```
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