Writing Functions in Python
by datacamp

Anatomy of a docstring
*always defined with """ """
1. what the function does
2. description of arguments, if any
3. description of the return value(s), if any
4. description of errors raised, if any
5. optional extra notes or examples of usage

Multiple docstring formats
- Google style
- Numpydoc
- reStructuredText
- EpyText

Google Style and Numpydoc are two of the most popular

Google Style
description - concise description of what the function does
should be in imperative language
example
"Split the data frame and stack the columns" instead of
"This function will split the data frame and stack the columns"

example
```
def function(arg_1, arg_2=42):
#concise description
    """Description of what the function does.
#list argument name, state each argument 'type', then what its role is in the function
#if argument has a default value, mark it as "optional" when describing the type
#can leave this section out if the function does not take in any parameters
    Args:
        arg_1 (str): Description of arg_1 that can break ont the next line if needed.
        arg_2 (int, optional): Write optional when an argument has a default value.
#list the expected type or types of what gets returned
    Returns:
        bool: Optional description of the return value
```

Extra lines are not indented.
#if you function intentionally raises any errors
Raises:
ValueError: Include any error types that the function intentionally raises.
#now include any necessary notes or examples
Notes:
See www.datacamp.com for more info.
"""

Numpydoc style
most common format in the science community
example
def function(ar_1, arg_2=42):
"""
Description of what the function does.

Parameters
————————-
arg_1 : expected type of arg_1
Description of arg_1.
arg_2 : int, optional
Write optional when an argument has a default value.
Default=42.

Returns
———-
The type of the return value
Can inclued a description of the return value.
Replace "Returns" with "Yields" if this function is a generator.
"""

Retrieving docstrings
it can be useful for your code to access the contents of your function's docstring
*every function in Pythoncomes with a __doc__ attribute that holds this information
__doc__ includes the raw docstring including tabs or spaces

example
def the_answer():
"""Return the answer to life,
to everything

Returns:
int

```
    """
    return 42
print(the_answer.__doc__)
output >
Return the answer to life,
    to everything

    Returns:
        int
```

To retrieve this output without tabs and in a cleaner fashion we can use the ‘inspect’ package

```
import inspect
print(inspect.getdoc(the_answer))
```

Another example

```
def count_letter(content, letter):
  """Count the number of times `letter` appears in `content`.

  Args:
    content (str): The string to search.
    letter (str): The letter to search for.

  Returns:
    int

  # Add a section detailing what errors might be raised
  Raises:
    ValueError: If `letter` is not a one-character string.
  """
  if (not isinstance(letter, str)) or len(letter) != 1:
    raise ValueError('`letter` must be a single character string.')
  return len([char for char in content if char == letter])
```

Don’t repeat yourself (DRY)
use functions to avoid repetition

Do One Thing
example - insteat of one big function that loads and plots data
make two separt functions
one for loading and
one for plotting
makes for more flexible code in the future

more easily understood
simpler to test
simpler to debug
easier to change

example of a load function

```
def load_data(path):
    """Load a dataset.

    Args:
        path (str): The location of a CSV file.

    Returns:
        tuple of ndarray: (features, labels)
    """
    data = pd.read_csv(path)
    y = data['labels'].values
    X = data{col for col in data.columns
                if col != 'labels'].values
    return X, y

def plot_data(X):
    """Plot the first two principal components of a matrix.

    Args:
        X (numpy.ndarray): The data to plot.
    """
    pca = PCA(n_components=2).fit_transform(X)
    plt.scatter(pca[;,0], pca[:,1])
```

Repeated code and functions that do more than one thing are examples of 'code smells',
which are indications that you may need to refactor
Refactoring is the process of improving code by changing it a little bit at a time

Getting z-scores example

```
# Standardize the GPAs for each year
df['y1_z'] = (df.y1_gpa - df.y1_gpa.mean()) / df.y1_gpa.std()
df['y2_z'] = (df.y2_gpa - df.y2_gpa.mean()) / df.y2_gpa.std()
df['y3_z'] = (df.y3_gpa - df.y3_gpa.mean()) / df.y3_gpa.std()
df['y4_z'] = (df.y4_gpa - df.y4_gpa.mean()) / df.y4_gpa.std()

def standardize(column):
```

```
  """Standardize the values in a column.

  Args:
    column (pandas Series): The data to standardize.

  Returns:
    pandas Series: the values as z-scores
  """
  # Finish the function so that it returns the z-scores
  z_score = (column - column.mean()) / column.std()
  return z_score

# Use the standardize() function to calculate the z-scores
df['y1_z'] = standardize(df['y1_gpa'])
df['y2_z'] = standardize(df['y2_gpa'])
df['y3_z'] = standardize(df['y3_gpa'])
df['y4_z'] = standardize(df['y4_gpa'])

Writing mean() and median() functions
def mean_and_median(values):
  """Get the mean and median of a sorted list of `values`

  Args:
    values (iterable of float): A list of numbers

  Returns:
    tuple (float, float): The mean and median
  """
  mean = sum(values) / len(values)
  midpoint = int(len(values) / 2)
  if len(values) % 2 == 0:
    median = (values[midpoint - 1] + values[midpoint]) / 2
  else:
    median = values[midpoint]

  return mean, median

Median
def median(values):
  """Get the median of a sorted list of values

  Args:
    values (iterable of float): A list of numbers
```

```
  Returns:
    float
  """
  # Write the median() function
  midpoint = int(len(values) / 2)
  if len(values) % 2 == 0:
    median = (values[midpoint -1] + values[midpoint]) / 2
  else:
    median = values[midpoint]
  return median
```

Pass by assignment

```
def foo(x):
        x[0] = 99
my_list = [1,2,3]
foo(my_list)
print(my_list)
```

output > [99, 2, 3]

**Lists in Python are mutable objects

```
def bar(x):
        x = x+90
my_var = 3
bar(my_var)
print(my_var)
```

output > 3

**in Python integers are immutable

Another example

```
a = [1,2,3]
b = a
a.append(4)
print(b)
```

output > [1,2,3,4]

```
b.append(5)
print(a)
```

output > [1,2,3,4,5]

***why is this? > because the computer saves a & b in this situation like this - - ->
a - -> 1 <- - b

3

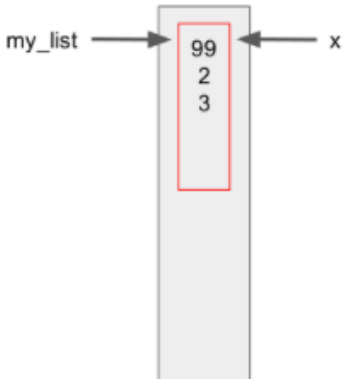
4

5

Pass by assignment

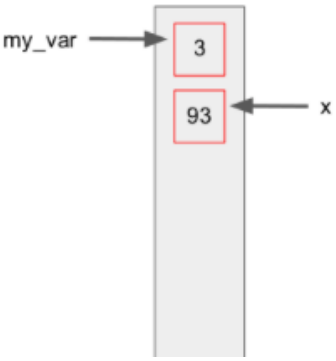
```
def foo(x):
  x[0] = 99
my_list = [1, 2, 3]
foo(my_list)
print(my_list)
```

```
[99, 2, 3]
```




```
def bar(x):
  x = x + 90
my_var = 3
bar(my_var)
my_var
```

```
3
```



Here the bar() function assigns ‘x’ to a new value, so the ‘my_var’ variable isnt’ touched
**This is because there is no way in Python to have changed ‘x’ to ‘my_var’ directly, because integers are immutable variables

Immutable

1. int
2. float
3. bool
4. string
5. bytes
6. tuple
7. frozenset

8. None

Mutable
1. list
2. dict
3. set
4. bytearray
5. objects
6. functions
7. almost everything else!

Adding a column example

```
# Use an immutable variable for the default argument
def better_add_column(values, df=None):
  """Add a column of `values` to a DataFrame `df`.
  The column will be named "col_<n>" where "n" is
  the numerical index of the column.

  Args:
    values (iterable): The values of the new column
    df (DataFrame, optional): The DataFrame to update.
      If no DataFrame is passed, one is created by default.

  Returns:
    DataFrame
  """
  # Update the function to create a default DataFrame
  if df is None:
    df = pandas.DataFrame()
  df['col_{}'.format(len(df.columns))] = values
  return df
```

Using context managers
sets up a context
runse your code
removes the context

example

```
with open('my_file.txt') as my_file:
        text = my_file.read()
        length = len(text)

print('The file is {} characters long'.format(length))
```

open() does three things:
1. sets up a context by opening a file
2. lets you run any code you want on that file
3. removes the context by closing the file

*the above print statement happens outside of the context

with <context-manager>(<args>) as variable-name:
    compound statement

'with' lets Python know that you are trying to enter a context
then you call a function (ie any context manager function)
any normal function arguments
"as" to return a value that you can use inside the context, you can assign the returned value to the variable name
end with a colon as if you were writing a for loop or an if statement
"compound statements" are an indented block after for loops, if/else statement, function definitions, with statement

How to create a context manager
1. define a function
2. optional - add any set up code your contex needs
3. use the 'yield' keyword
4. optional - add any teardown code you context needs
5. add the '@contextlib.contextmanager' decorator

```
@contextlib.contextmanager
def my_context():
    #add any set up code you need
    yield
    #add any teardown code you need
```

When you write 'yield', it means that you are going to return a value
but you expect to finish the rest of the function at some point in the future
the yield value can be assigned to the 'as variable-name'

example

```
@contextlib.contextmanager
def my_context():
    print('hello)
    yield
    print('goodbye')
```

```
with my_context() as foo:
        print('foo is {}'.format(foo))
output >
hello
foo is 42
goodbye
```

**a context manager function is technically a generator that yields a single value
**the ability for a function to yield control and know that it will get to finish running later is what makes context managers so useful

Another example

```
@contextlib.contextmanager
def database(url):
        #set up database connection
        db = postgres.connect(url)

        yield db

        #tear down database connection
        db.disconnect()

url = 'http://datacamp.com/data'
with database(url) as my_db:
        course_list = my_db.execute('SELECT * FROM courses')
```

this setup/teardown behavior allows a context manager to hide things like connecting and disconnecting from a database
so that a programmer using the context manager can just perform operations on the database without worrying about the underlying details

```python
@contextlib.contextmanager
def in_dir(path):
  # save current working directory
  old_dir = os.getcwd()

  # switch to new working directory
  os.chdir(path)

  yield

  # change back to previous
  # working directory
  os.chdir(old_dir)
```

```python
with in_dir('/data/project_1/'):
  project_files = os.listdir()
```

Another example

```
# Add a decorator that will make timer() a context manager
@contextlib.contextmanager
def timer():
  """Time the execution of a context block.

  Yields:
    None
  """
  start = time.time()
  # Send control back to the context block
  yield
  end = time.time()
  print('Elapsed: {:.2f}s'.format(end - start))

with timer():
  print('This should take approximately 0.25 seconds')
  time.sleep(0.25)
```

The regular open() context manager:

- takes a filename and a mode ('r' for read, 'w' for write, or 'a' for append)

- opens the file for reading, writing, or appending
- yields control back to the context, along with a reference to the file
- waits for the context to finish
- and then closes the file before exiting

Example - context manager that acts like ‘open()’ but operates in read-only

```
@contextlib.contextmanager
def open_read_only(filename):
  """Open a file in read-only mode.

  Args:
    filename (str): The location of the file to read

  Yields:
    file object
  """
  read_only_file = open(filename, mode='r')
  # Yield read_only_file so it can be assigned to my_file
  yield read_only_file
  # Close read_only_file
  read_only_file.close()

with open_read_only('my_file.txt') as my_file:
  print(my_file.read())
```

Nested contexts

example - we want to open two files and copy one file over one line at a time

open() context manager returns can be iterated over in a for loop

```
with open(‘my_file.txt’) as my_file:
        for line in my_file:
```

**this will let us copy the file without worrying about how big it is

```
def copy(src, dst):
  """Copy the contents of one file to another.

  Args:
    src (str): File name of the file to be copied.
    dst (str): Where to write the new file.
  """
  # Open both files
  with open(src) as f_src:
    with open(dst, 'w') as f_dst:
      # Read and write each line, one at a time
      for line in f_src:
        f_dst.write(line)
```

Handling errors
try/except/finally technique
allows you to write code that might raise an error inside the ‘try’ block
and catch that error inside the ‘except’ block
*you can choose to ignore the error or re-raise it
in the ‘finally’ block - this is code that runs no matter what, wheter an exception occurred or not

```
def get_printer(ip):
  p = connect_to_printer(ip)

  try:
    yield
  finally:
    p.disconnect()
    print('disconnected from printer')

doc = {'text': 'This is my text.'}

with get_printer('10.0.34.111') as printer:
  printer.print_page(doc['txt'])
```

```
disconnected from printer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    printer.print_page(doc['txt'])
KeyError: 'txt'
```

Another example
Use the "stock('NVDA')" context manager
and assign the result to the variable "nvda"
with stock('NVDA') as nvda:
  # Open "NVDA.txt" for writing as f_out
  with open('NVDA.txt', 'w') as f_out:
    for _ in range(10):
      value = nvda.price()
      print('Logging ${:.2f} for NVDA'.format(value))
      f_out.write('{:.2f}\n'.format(value))

Functions are objects
functions are just like any other object in Python
they are no different from lists, dictionaries, DataFrames, strings, integers, floats, modules, or anything else in Python
reminder
def x():
        pass
x= [1,2,3]
x={'foo':42}
x=pandas.DataFrame()
x='This is a sentence.'
x=3
x-71.2
import x

you can take a function and assign it to a variable like 'x'
def my_function():

```
    print('Hello')
x = my_function
type(x)
output <type 'function'>
you can then call x, and it would be the same as calling my_function
x()
output> Hello

**also doesn't have to be a function you defined
example - assign the print() function to PrintyMcPrintface
PrintyMcPrintface = print
PrintyMcPrintface('Python is awesome!')
output > Python is awesome!

Can add functions to lists and dictionaries
**and call elements from that list and pass it arguments
example
list_of_functions = [my_function, open, print]
list_of_functions[2]('I am printing with an element of a list!')
output > I am printing with an element of a list!
**reminder 'print' is a function
we indexed it out of the list and voila

Can do the same with a dictionary
dict_of_functions = {'func1': my_function, 'funce2': open, 'func3': print}
dict_of_functions['func3']('I am printing with a value of a dict!')
output > I am printing with a value of a dict!
again we indexed out the print function and used it as if we were calling it directly

Referencing a function
**Remember when you assign a function to a variable you do not include the
parentheses
*when you type the 'function()' with parentheses, you are calling that function
it evaluates to the value that the function returns
*when you type the 'function' without the parentheses, you are referencing the
function itself
it evaluates to a function object
example
def my_function():
    return 42

x = my_function
my_function()
```

output 42

my_function
output <function my_function> #ie I am object function

Functions as arguments
you can pass a function just like any other object as an argument to another function

Defining a function inside another function
these kind of functions are called:
nested functions or inner functions or helper functions or child functions
they can make you code easier to read

Functions as return values
example
```
def get_function():
    def print_me(s):
        print(s)

    return print_me

new_func = get_function()
new_func('This is a senetence.')
```
output > This is a sentence.
**we assigned the result of calling get_function() to the variable new_func, in turn we are assigning the return value print_me to new_func.
we can then call new_func() as if it were the print_me() function

Scope
determines which variables can be accessed at different points in your code
scope is like Python making inferences
if we defined x within the function and then we ask to print x Python infers that we want to print the x that we just defined
but say we asked Python to print x and y but didn't define y, Python will look outside the function for a definition for y
how does Python make these inferences?
based off of rules
'LEGB'
first local - inside a function (arguments and defined variables)
then enclosing or nonlocal - this is for nested function, Python will look to see if there is a parent function; **enclosing scope entails non-modifiable variables within the child (or inner) function; nonlocal scope allows you to access and

modify variables in an outer (or enclosing) function's scope from within the nested function > used when you want to reassign or modify a variable that is in an enclosing scope, rather than creating a new variable with the same name within the inner function's local scope
then global - these are things defined outside the function, things that are accessible to any part of the program
then built-in

Closures
a tuple of variables that are no longer in scope, but that a function needs in order to run
*ie attaching nonlocal variables to nested functions
example

```
def foo():
    a = 5
    def bar():
        print(a)
    return bar

func = foo()

func()
```

output > 5
*so here we the function foo to func which means that by calling func() the function bar runs
but wait we called func which called foo which means that it was a scope outside of bar() so how does it know about variable 'a'?
this is where closures come in
when foo() returned bar() Python helpfully attached any nonlocal variable that bar() was going to need to the function object
these variables get stored in a tuple in the ''__closure__'" attribute of the function

```
type(func.__closure__)
```

output > <class 'tuple'>

**key point if we were to create a global variable that we then called it would get placed into this closure
here is where it gets interesting
if we were than to delete that global variable and call the function we would still get the same output as we did with the global variable
reason being is that it was stored in the closure
when Python could not find anything in the scope it moved to the closure

Example

```
def my_special_function():
  print('You are running my_special_function()')

def get_new_func(func):
  def call_func():
    func()
  return call_func

# Overwrite `my_special_function` with the new function
my_special_function = get_new_func(my_special_function)

my_special_function()
```

Modifying, deleting, overwriting none of these changed the output of my_special_function()
this is because the nested function can still access those values because they are stored safely in the function’s closure

Decorators
a wrapper that you can place around a function that changes that function’s behavior
such as modify inputs, outputs, or even the function itself
‘@‘ defines that you are using a decorator
easy example

```
@double_args
def multiply(a,b):
        return a*b
multiply(1,5)
```

output > 20
@double_args multplies each argument by 2 before passing them into the function
so instead of getting 1*5
we get 2*10

Creating double_args

```
def multiply(a, b):
  return a * b
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper
multiply = double_args(multiply)
multiply(1, 5)
```

```
20
```

Here we overwrite the original multiply function with double_args(multiply)
we can do this because Python stores the original multiply fucntion in the new function's closure
**the @ sign essentially removes the renaming - see below

```
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper


def multiply(a, b):
  return a * b

multiply = double_args(multiply)

multiply(1, 5)
```

```
20
```

```
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper

@double_args
def multiply(a, b):
  return a * b



multiply(1, 5)
```

```
20
```

Time a 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
        start_time = 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.
        execution_time = time.time() - start_time
        print(f"Execution time of {func.__name__}: {execution_time} seconds")
        return result
    return wrapper
```

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

```
def memoize(func):
#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
```

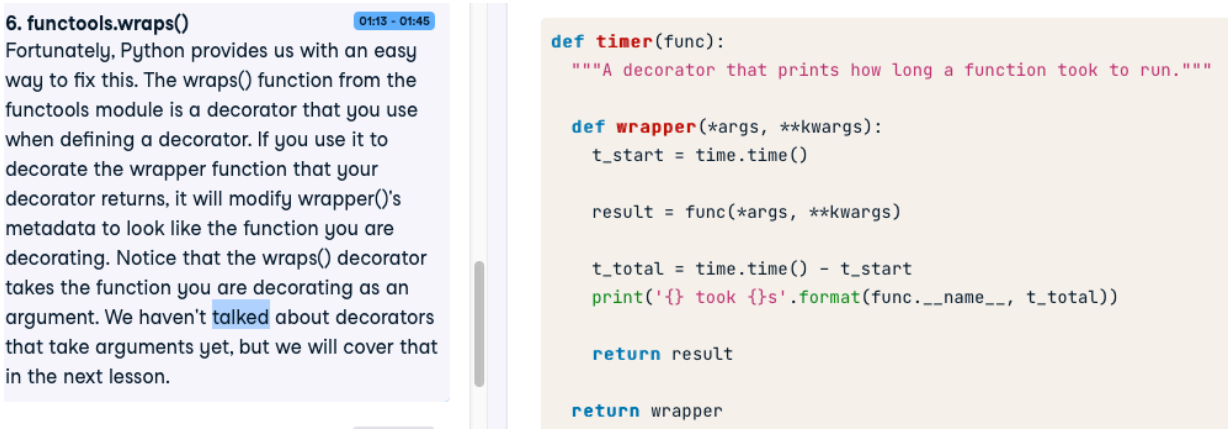
When to use decorators?
consider using a decorator when you want to add some common bit of code to multiple functions

Example - a function that returns dtype, helpful for debuggin

```
def print_return_type(func):
  # Define wrapper(), the decorated function
  def wrapper(*args, **kwargs):
    # Call the function being decorated
    result = func(*args, **kwargs)
    print('{}() returned type {}'.format(
      func.__name__, type(result)
    ))
    return result
  # Return the decorated function
  return wrapper
```

```
@print_return_type
def foo(value):
  return value

print(foo(42))
print(foo([1, 2, 3]))
print(foo({'a': 42}))

example - counter decorator
def counter(func):
  def wrapper(*args, **kwargs):
    wrapper.count += 1
    # Call the function being decorated and return the result
    return func(*args, **kwargs)
  wrapper.count = 0
  # Return the new decorated function
  return wrapper

# Decorate foo() with the counter() decorator
@counter
def foo():
  print('calling foo()')

foo()
foo()

print('foo() was called {} times.'.format(foo.count))
```

Decorators and metadata
in Python metadata refers to additional infor about an object, module, package, or other entity within the code



example

```
from functools import wraps

def add_hello(func):
  # Decorate wrapper() so that it keeps func()'s metadata
  @wraps(func)
  def wrapper(*args, **kwargs):
    """Print 'hello' and then call the decorated function."""
    print('Hello')
    return func(*args, **kwargs)
  return wrapper

@add_hello
def print_sum(a, b):
  """Adds two numbers and prints the sum"""
  print(a + b)

print_sum(10, 20)
print_sum_docstring = print_sum.__doc__
print(print_sum_docstring)

@check_everything
def duplicate(my_list):
  """Return a new list that repeats the input twice"""
  return my_list + my_list

t_start = time.time()
duplicated_list = duplicate(list(range(50)))
t_end = time.time()
decorated_time = t_end - t_start

t_start = time.time()
# Call the original function instead of the decorated one
duplicated_list = duplicate.__wrapped__(list(range(50)))
t_end = time.time()
undecorated_time = t_end - t_start

print('Decorated time: {:.5f}s'.format(decorated_time))
print('Undecorated time: {:.5f}s'.format(undecorated_time))
```

Decorators that take arguments
the key step is creating a function that returns a decorator, rather than a function

that is a decorator
what this means
function capabilitites can get limited because a decorator can only take in one argument at a time
example - a decorator that runs a funciton 3 times

```
def run_n_times(n):
    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)
```

Timeout() decorator
example - using Python's signal module

```
import signal
def raise_timeout(*args, **kwargs):
    raint TimeoutError ()
#when an 'alarm' signal goes off, call raise_timeout()
signal.signal(signalnum=signal.SIGALRM, handler=raise_timeout)
#when you see the signal whose number is signalnum, call the handler function
#alarm function lets us sett of an alarm in 5 seconds
signal.alarm(5)
#passing 0 to the alarm function cancels the alarm
```

Timeout function

```
def timeout_in_5s(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        signal.alarm(5)
        try:
        #call the decorated func
            return func(*args, **kwargs)
        finally:
        #cancel alarm
            signal.alarm(0)
    return wrapper
```

Solid example

```
def returns(return_type):
```

```
  # Complete the returns() decorator
  def decorator(func):
    def wrapper(*args, **kwargs):
      result = func(*args, **kwargs)
      assert type(result) == return_type
      return result
    return wrapper
  return decorator

@returns(dict)
def foo(value):
  return value

try:
  print(foo([1,2,3]))
except AssertionError:
  print('foo() did not return a dict!')
```