Writing Functions in Python

August 10, 2020

We've done our analysis, built the report, and trained a model. What's next? Well, if you want to deploy your model into production, your code will need to be more reliable than exploratory scripts in a Jupyter notebook. Writing Functions in Python will give you a strong foundation in writing complex and beautiful functions so that you can contribute research and engineering skills to your team. I will work on tricks, like how to write context managers and decorators. I will also work on best practices around how to write maintainable reusable functions with good documentation. They say that people who can do good research and write high-quality code are unicorns.

Best Practices

Here I will cover best practices when writing functions, docstrings and why they matter and how to know when you need to turn a chunk of code into a function. I will also work on the details of how Python passes arguments to functions, as well as some common gotchas that can cause debugging headaches when calling functions.

Crafting a docstring

You've decided to write the world's greatest open-source natural language processing Python package. It will revolutionize working with free-form text, the way numpy did for arrays, pandas did for tabular data, and scikit-learn did for machine learning.

The first function you write is count_letter(). It takes a string and a single letter and returns the number of times the letter appears in the string. You want the users of your open-source package to be able to understand how this function works easily, so you will need to give it a docstring. Build up a Google Style docstring for this function by following these steps.

[1]:
```python
# Copy the following string and add it as the docstring for the function: Count␣
↪the number of times `letter` appears in `content`.

# Adding a docstring to count_letter()
def count_letter(content, letter):
  """Count the number of times `letter` appears in `content`"""
  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])

# Now adding the arguments section, using the Google style for docstrings. Use␣
↪str to indicate a string.

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

    # Add a Google-style arguments section
    Args:
      content (str): The string to search.
      letter (str): The letter to search for.
    """
    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])

# Adding a returns section that informs the user the return value is an int.

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.

    # Add a returns section
    Returns:
      int
    """
    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])

# Finally, add some information about the ValueError that gets raised when the␣
 ↪arguments aren't correct.
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])
```

What a delightful docstring! While it does require a bit more typing, the information presented here will make it very easy for others to use this code in the future. Remember that even though computers execute it, code is actually written for humans to read (otherwise you'd just be writing the 1s and 0s that the computer operates on).

Retrieving docstrings

You and a group of friends are working on building an amazing new Python IDE (integrated development environment – like PyCharm, Spyder, Eclipse, Visual Studio, etc.). The team wants to add a feature that displays a tooltip with a function's docstring whenever the user starts typing the function name. That way, the user doesn't have to go elsewhere to look up the documentation for the function they are trying to use. You've been asked to complete the build_tooltip() function that retrieves a docstring from an arbitrary function.

[2]:
```python
import inspect
```

[3]:
```python
dir(inspect)
```

[3]:
```
['ArgInfo',
 'ArgSpec',
 'Arguments',
 'Attribute',
 'BlockFinder',
 'BoundArguments',
 'CORO_CLOSED',
 'CORO_CREATED',
 'CORO_RUNNING',
 'CORO_SUSPENDED',
 'CO_ASYNC_GENERATOR',
 'CO_COROUTINE',
 'CO_GENERATOR',
 'CO_ITERABLE_COROUTINE',
 'CO_NESTED',
 'CO_NEWLOCALS',
 'CO_NOFREE',
 'CO_OPTIMIZED',
 'CO_VARARGS',
 'CO_VARKEYWORDS',
 'ClosureVars',
 'EndOfBlock',
 'FrameInfo',
 'FullArgSpec',
 'GEN_CLOSED',
 'GEN_CREATED',
 'GEN_RUNNING',
 'GEN_SUSPENDED',
 'OrderedDict',
 'Parameter',
 'Signature',
 'TPFLAGS_IS_ABSTRACT',
 'Traceback',
 '_ClassMethodWrapper',
 '_KEYWORD_ONLY',
 '_MethodWrapper',
 '_NonUserDefinedCallables',
 '_PARAM_NAME_MAPPING',
 '_POSITIONAL_ONLY',
 '_POSITIONAL_OR_KEYWORD',
 '_ParameterKind',
 '_VAR_KEYWORD',
 '_VAR_POSITIONAL',
 '_WrapperDescriptor',
 '__author__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_check_class',
 '_check_instance',
 '_empty',
 '_filesbymodname',
 '_findclass',
 '_finddoc',
 '_get_paramkind_descr',
 '_getfullargs',
 '_is_type',
 '_main',
 '_missing_arguments',
 '_sentinel',
 '_shadowed_dict',
 '_signature_bound_method',
 '_signature_from_builtin',
 '_signature_from_callable',
 '_signature_from_function',
 '_signature_fromstr',
 '_signature_get_bound_param',
 '_signature_get_partial',
 '_signature_get_user_defined_method',
 '_signature_is_builtin',
 '_signature_is_functionlike',
 '_signature_strip_non_python_syntax',
 '_static_getmro',
 '_too_many',
 '_void',
 'abc',
 'attrgetter',
 'builtins',
 'classify_class_attrs',
 'cleandoc',
 'collections',
 'currentframe',
 'dis',
 'enum',
 'findsource',
 'formatannotation',
 'formatannotationrelativeto',
 'formatargspec',
 'formatargvalues',
 'functools',
 'getabsfile',
 'getargs',
 'getargspec',
 'getargvalues',
 'getattr_static',
 'getblock',
 'getcallargs',
 'getclasstree',
 'getclosurevars',
 'getcomments',
 'getcoroutinelocals',
 'getcoroutinestate',
 'getdoc',
 'getfile',
 'getframeinfo',
 'getfullargspec',
 'getgeneratorlocals',
 'getgeneratorstate',
 'getinnerframes',
 'getlineno',
 'getmembers',
 'getmodule',
 'getmodulename',
 'getmro',
 'getouterframes',
 'getsource',
 'getsourcefile',
 'getsourcelines',
 'importlib',
 'indentsize',
 'isabstract',
 'isasyncgen',
 'isasyncgenfunction',
 'isawaitable',
 'isbuiltin',
 'isclass',
 'iscode',
 'iscoroutine',
 'iscoroutinefunction',
 'isdatadescriptor',
 'isframe',
 'isfunction',
 'isgenerator',
 'isgeneratorfunction',
 'isgetsetdescriptor',
 'ismemberdescriptor',
 'ismethod',
 'ismethoddescriptor',
 'ismodule',
 'isroutine',
 'istraceback',
 'itertools',
 'k',
 'linecache',
 'mod_dict',
 'modulesbyfile',
 'namedtuple',
 'os',
 're',
 'signature',
 'stack',
 'sys',
 'token',
 'tokenize',
 'trace',
 'types',
 'unwrap',
 'v',
 'walktree',
 'warnings']
```

[ ]:
```
# Getting the docstring with an attribute of count_letter()
docstring = count_letter.__doc__

border = '#' * 28
print('{}\n{}\n{}'.format(border, docstring, border))
```

```python
# Now using a function from the inspect module to get a better-formatted␣
↪version of count_letter()'s docstring.

import inspect

# Getting the docstring with a function from the inspect module
docstring = inspect.getdoc(count_letter)

border = '#' * 28
print('{}\n{}\n{}'.format(border, docstring, border))

# Using the inspect module again to get the docstring for any function being␣
↪passed to the build_tooltip() function.

def build_tooltip(function):
  """Create a tooltip for any function that shows the
  function's docstring.

  Args:
    function (callable): The function we want a tooltip for.

  Returns:
    str
  """
  # Using 'inspect' to get the docstring
  docstring = inspect.getdoc(function)
  border = '#' * 28
  return '{}\n{}\n{}'.format(border, docstring, border)

print(build_tooltip(count_letter))
print(build_tooltip(range))
print(build_tooltip(print))
```

This IDE is going to be an incredibly delightful experience for your users now! Notice how the count_letter.__doc__ version of the docstring had strange whitespace at the beginning of all but the first line. That's because the docstring is indented to line up visually when reading the code. But when we want to print the docstring, removing those leading spaces with inspect.getdoc() will look much better.

Docstrings to the rescue!

If you want to explore docstrings of a function below is the code.

inspect.getdoc()

just insert function name in the brackets.

Extracting a function

While you were developing a model to predict the likelihood of a student graduating from college, you wrote this bit of code to get the z-scores of students' yearly GPAs. Now you're ready to turn it into a production-quality system, so you need to do something about the repetition. Writing a function to calculate the z-scores would improve this code.

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
  """
  # Finishing the function so that it returns the z-scores
  z_score = (column - column.mean()) / column.std()
  return z_score

# Using 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)
```

[ ]:
```
# Splitting up a function

Another engineer on your team has written this function to calculate the mean␣
↪and median of a list. You want to show them how to split it into two simpler␣
↪functions: mean() and median()

def mean_and_median(values):
  """Get the mean and median of a 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
```

[14]:
```
def mean(values):
  """Get the mean of a list of values

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

  Returns:
    float
  """
  # Write the mean() function
  mean = sum(values)/len(values)
  return mean
```

[15]:
```
mean(values)
```

[15]:
```
3.5
```

[16]:
```
values = [1,2,3,4,5,6,7]
```

[17]:
```
def median(values):
  """Get the median of a 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
```

[18]:
```
median(values)
```

[18]:
```
4
```

A perfect split! Each function does one thing and does it well. Using, testing, and maintaining these will be a breeze (although you'll probably just use numpy.mean() and numpy.median() for this in real life).

[ ]:
```
Mutable or immutable?

The following function adds a mapping between a string and the lowercase␣
 ↪version of that string to a dictionary. What do you expect the values of d␣
 ↪and s to be after the function is called?
def store_lower(_dict, _string):
  """Add a mapping between `_string` and a lowercased version of `_string` to␣
 ↪`_dict`

  Args:
    _dict (dict): The dictionary to update.
    _string (str): The string to add.
  """
  orig_string = _string
  _string = _string.lower()
  _dict[orig_string] = _string

d = {}
s = 'Hello'

store_lower(d, s)
d = {}, s = 'Hello'
This is what you would expect if dictionaries were immutable.
d = {}, s = 'hello'
This is what you would expect if dictionaries were immutable and strings were␣
 ↪mutable.
d = {'Hello': 'hello'}, s = 'Hello'
Correct! Dictionaries are mutable objects in Python, so the function can␣
 ↪directly change it in the _dict[_orig_string] = _string statement. Strings,␣
 ↪on the other hand, are immutable. When the function creates the lowercase␣
 ↪version, it has to assign it to the _string variable. This disconnects what␣
 ↪happens to _string from the external s variable.
d = {'Hello': 'hello'}, s = 'hello'
Yes, dictionaries are mutable, so d gets updated in place. But look at how the␣
 ↪function assigns the lowercase version of _string back to the _string␣
 ↪variable.
d = {'hello': 'hello'}, s = 'hello'
When the function assigns _string.lowercase() back to the _string variable, it␣
 ↪disconnects _string from both s and orig_string.
```

[ ]:
```
Best practice for default arguments
```

```
One of your co-workers (who obviously didn't take this course) has written this␣
↪function for adding a column to a panda's DataFrame. Unfortunately, they␣
↪used a mutable variable as a default argument value! Please show them a␣
↪better way to do this so that they don't get unexpected behavior.

def add_column(values, df=pandas.DataFrame()):
  """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
  """
  df['col_{}'.format(len(df.columns))] = values
  return df
• Change the default value of df to an immutable variable to follow best␣
↪practices.
• Update the code of the function so that a new DataFrame is created if the␣
↪caller didn't pass one.
```

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

Beautiful and best practice! When you need to set a mutable variable as a default argument, always use None and then set the value in the body of the function. This prevents unexpected behavior like adding multiple columns if you call the function more than once.

The number of cats

You are working on a natural language processing project to determine what makes great writers so great. Your current hypothesis is that great writers talk about cats a lot. To prove it, you want to count the number of times the word “cat” appears in “Alice’s Adventures in Wonderland” by Lewis Carroll. You have already downloaded a text file, alice.txt, with the entire contents of this great book.

- Use the open() context manager to open alice.txt and assign the file to the file variable.

[ ]:
```python
# Opening "alice.txt" and assign the file to "file"
with open('alice.txt') as file:
  text = file.read()

n = 0
for word in text.split():
  if word.lower() in ['cat', 'cats']:
    n += 1

print('Lewis Carroll uses the word "cat" {} times'.format(n))
```

Cool cat counting! By opening the file using the with open() statement, you were able to read in the text of the file. More importantly, when you were done reading the text, the context manager closed the file for you.

The speed of cats

You’re working on a new web service that processes Instagram feeds to identify which pictures contain cats (don’t ask why – it’s the internet). The code that processes the data is slower than you would like it to be, so you are working on tuning it up to run faster. Given an image, image, you have two functions that can process it:

- process_with_numpy(image)
- process_with_pytorch(image)

Your colleague wrote a context manager, timer(), that will print out how long the code inside the context block takes to run. She is suggesting you use it to see which of the two options is faster. Time each function to determine which one to use in your web service.

- Use the timer() context manager to time how long process_with_numpy(image) takes to run.
- Use the timer() context manager to time how long process_with_pytorch(image) takes to run.

[ ]:
```python
image = get_image_from_instagram()

# Timing how long process_with_numpy(image) takes to run
with timer():
  print('Numpy version')
  process_with_numpy(image)
```

```
# Timing how long process_with_pytorch(image) takes to run
with timer():
  print('Pytorch version')
  process_with_pytorch(image)
```

Terrific timing! Now that you know the pytorch version is faster, you can use it in your web service to ensure your users get the rapid response time they expect.

You may have noticed there was no as at the end of the with statement in timer() context manager. That is because timer() is a context manager that does not return a value, so the as at the end of the with statement isn't necessary. In the next lesson, you'll learn how to write your own context managers like timer().

The timer() context manager

A colleague of yours is working on a web service that processes Instagram photos. Customers are complaining that the service takes too long to identify whether or not an image has a cat in it, so your colleague has come to you for help. You decide to write a context manager that they can use to time how long their functions take to run.

- Add a decorator from the contextlib module to the timer() function that will make it act like a context manager.
- Send control from the timer() function to the context block

[25]:
```
from contextlib import contextmanager
```

[ ]:
```
# Adding 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)
```

You're managing context like a boss! And your colleague can now use your timer() context manager to figure out which of their functions is running too slow. Notice that the three elements of a context manager are all here: a function definition, a yield statement, and the @contextlib.contextmanager decorator. It's also worth noticing that timer() is a context manager that does not return an explicit value, so yield is written by itself without specifying anything to return.

A read-only open() context manager

You have a bunch of data files for your next deep learning project that took you months to collect and clean. It would be terrible if you accidentally overwrote one of those files when trying to read it in for training, so you decide to create a read-only version of the open() context manager to use in your project.

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
- sends 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

[ ]:
```
@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')
  # Yielding read_only_file so it can be assigned to my_file
  yield read_only_file
  # Closing read_only_file
  read_only_file.close()

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

That is a radical read-only context manager! Now you can relax, knowing that every time you use with open_read_only() your files are safe from being accidentally overwritten. This function is an example of a context manager that does return a value, so we write yield read_only_file instead of just yield. Then the read_only_file object gets assigned to my_file in the with statement so that whoever is using your context can call its .read() method in the context block.

Context manager use cases

Which of the following would NOT be a good opportunity to use a context manager?

- A function that starts a timer that keeps track of how long some block of code takes to run.

Starting and stopping a timer is an example of the START/STOP pattern.

- A function that prints all of the prime numbers between 2 and some value n.

Correct! While you might be able to do this with a context manager, it would make much more sense just to do it with a normal function.

- A function that connects to a smart thermostat so that it can be programmed remotely.

Connecting to a thermostat and disconnecting afterward is an example of the CONNECT/DISCONNECT pattern.

- A function that prevents multiple users from updating an online spreadsheet at the same time by locking access to the spreadsheet before every operation.

Locking access to an online spreadsheet and then releasing the lock is an example of the LOCK/RELEASE pattern.

Scraping the NASDAQ

Training deep neural nets is expensive! You might as well invest in NVIDIA stock since you're spending so much on GPUs. To pick the best time to invest, you are going to collect and analyze some data on how their stock is doing. The context manager stock('NVDA') will connect to the NASDAQ and return an object that you can use to get the latest price by calling its .price() method.

You want to connect to stock('NVDA') and record 10 timesteps of price data by writing it to the file NVDA.txt.

- Use the stock('NVDA') context manager and assign the result to nvda.

- Open a file for writing with open('NVDA.txt', 'w') and assign the file object to f_out so you can record the price over time.

[ ]:
```
# Using the "stock('NVDA')" context manager and assigning 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))
```

Super stock scraping! Now you can monitor the NVIDIA stock price and decide when is the exact right time to buy. Nesting context managers like this allows you to connect to the stock market (the CONNECT/DISCONNECT pattern) and write to a file (the OPEN/CLOSE pattern) at the same time.

Changing the working directory

You are using an open-source library that lets you train deep neural networks on your data. Unfortunately, during training, this library writes out checkpoint models (i.e., models that have been trained on a portion of the data) to the current working directory. You find that behavior frustrating because you don't want to have to launch the script from the directory where the models will be saved.

You decide that one way to fix this is to write a context manager that changes the current working directory, lets you build your models, and then resets the working directory to its original location.

You'll want to be sure that any errors that occur during model training don't prevent you from resetting the working directory to its original location.

- Add a statement that lets you handle any errors that might occur inside the context.
- Add a statement that ensures os.chdir(current_dir) will be called, whether there was an error or not.

[29]:
```
def in_dir(directory):
  """Change current working directory to `directory`,
  allow the user to run some code, and change back.

  Args:
    directory (str): The path to a directory to work in.
  """
  current_dir = os.getcwd()
  os.chdir(directory)

  # Add code that lets you handle errors
  try:
    yield
  # Ensure the directory is reset,
  # whether there was an error or not
  finally:
    os.chdir(current_dir)
```

[ ]:
```
Excellent error handling! Now, even if someone writes buggy code when using␣
↪your context manager, you will be sure to change the current working␣
↪directory back to what it was when they called in_dir(). This is important␣
↪to do because your users might be relying on their working directory being␣
↪what it was when they started the script. in_dir() is a great example of the␣
↪CHANGE/RESET pattern that indicates you should use a context manager.
```

Building a command line data app

You are building a command line tool that lets a user interactively explore a data set. We've defined four functions: mean(), std(), minimum(), and maximum() that users can call to analyze their data. Help finish this section of the code so that your users can call any of these functions by typing the function name at the input prompt.

Note: The function get_user_input() in this exercise is a mock version of asking the user to enter a command. It randomly returns one of the four function names. In real life, you would ask for input and wait until the user entered a value.

[35]:
```
# Adding the missing function references to the function map
function_map = {
  'mean': mean,
  'std': std,
  'minimum': minimum,
  'maximum': maximum
}

data = load_data()
print(data)

func_name = get_user_input()

# Calling the chosen function and pass "data" as an argument
function_map[func_name](data)
```

Phenomenal function referencing! By adding the functions to a dictionary, you can select the function based on the user's input. You could have also used a series of if/else statements, but putting them in a dictionary like this is much easier to read and maintain.

[ ]:
```
# Reviewing your co-worker's code

# Your co-worker is asking you to review some code that they've written and␣
 ↪give them some tips on how to get it ready for production. You know that␣
 ↪having a docstring is considered best practice for maintainable, reusable␣
 ↪functions, so as a sanity check you decide to run this has_docstring()␣
 ↪function on all of their functions.

def has_docstring(func):
  """Check to see if the function
  `func` has a docstring.

  Args:
    func (callable): A function.

  Returns:
    bool
  """
  return func.__doc__ is not None
```

[ ]:
```
# 1. Calling has_docstring() on your co-worker's load_and_plot_data() function.

# Calling has_docstring() on the load_and_plot_data() function
ok = has_docstring(load_and_plot_data)

if not ok:
  print("load_and_plot_data() doesn't have a docstring!")
else:
  print("load_and_plot_data() looks ok")


# 2. Check if the function as_2D() has a docstring.
# Calling has_docstring() on the as_2D() function
ok = has_docstring(as_2D)

if not ok:
  print("as_2D() doesn't have a docstring!")
else:
  print("as_2D() looks ok")


# 3. Checking if the function log_product() has a docstring.

# Calling has_docstring() on the log_product() function
ok = has_docstring(log_product)

if not ok:
  print("log_product() doesn't have a docstring!")
else:
  print("log_product() looks ok")
```

Awesome job writing functions as arguments! You have discovered that your co-worker forgot to write a docstring for log_product(). You have learned enough about best practices to tell them how to fix it.

To pass a function as an argument to another function, you had to determine which one you were calling and which one you were referencing. Keeping those straight will be important as we dig deeper into this chapter. From the function names can you think of any other advice you might give your co-worker about their functions?

[39]:
```
def create_math_function(func_name):
  if func_name == 'add':
    def add(a, b):
      return a + b
    return add
  elif func_name == 'subtract':
    # Define the subtract() function
    def subtract(a, b):
      return a - b
    return subtract
  else:
    print("I don't know that one")

add = create_math_function('add')
print('5 + 2 = {}'.format(add(5, 2)))

subtract = create_math_function('subtract')
print('5 - 2 = {}'.format(subtract(5, 2)))
```

```
5 + 2 = 7
```

```
5 - 2 = 3
```

Nice nested function! Now that you've implemented the subtract() function, you can keep going to include multiply() and divide(). I predict this game is going to be even bigger than Fortnite!

Notice how we assign the return value from create_math_function() to the add and subtract variables in the script. Since create_math_function() returns a function, we can then call those variables as functions.

Scope:

Scope determines which variables can be accessed at different points in your code.

Below is an example.

Understanding scope

What four values does this script print?

[40]:
```python
x = 50

def one():
  x = 10

def two():
  global x
  x = 30

def three():
  x = 100
  print(x)

for func in [one, two, three]:
  func()
  print(x)
```

```
50
30
100
30
```

One() doesn't change the global x, so the first print() statement prints 50.

two() does change the global x so the second print() statement prints 30.

The print() statement inside the function three() is referencing the x value that is local to three(), so it prints 100.

But three() does not change the global x value so the last print() statement prints 30 again.

Modifying variables outside local scope

Sometimes your functions will need to modify a variable that is outside of the local scope of that function. While it's generally not best practice to do so, it's still good to know-how in case you need to do it. Update these functions so they can modify variables that would usually be outside of their scope.

[57]:
```
# Adding a keyword that lets us update call_count from inside the function.
call_count = 0

def my_function():
  # Use a keyword that lets us update call_count
  global call_count
  call_count += 1

  print("You've called my_function() {} times!".format(
    call_count
  ))

for _ in range(20):
  my_function()
```

```
You've called my_function() 1 times!
You've called my_function() 2 times!
You've called my_function() 3 times!
You've called my_function() 4 times!
You've called my_function() 5 times!
You've called my_function() 6 times!
You've called my_function() 7 times!
You've called my_function() 8 times!
You've called my_function() 9 times!
You've called my_function() 10 times!
You've called my_function() 11 times!
You've called my_function() 12 times!
You've called my_function() 13 times!
You've called my_function() 14 times!
You've called my_function() 15 times!
You've called my_function() 16 times!
You've called my_function() 17 times!
You've called my_function() 18 times!
You've called my_function() 19 times!
You've called my_function() 20 times!
```

[ ]:
```
# Adding a keyword that lets us modify file_contents from inside␣
 ↪save_contents().
def read_files():
  file_contents = None

  def save_contents(filename):
    # Add a keyword that lets us modify file_contents
    nonlocal file_contents
    if file_contents is None:
```

```
    file_contents = []
    with open(filename) as fin:
      file_contents.append(fin.read())

  for filename in ['1984.txt', 'MobyDick.txt', 'CatsEye.txt']:
    save_contents(filename)

  return file_contents

print('\n'.join(read_files()))
```

[61]:
```
import random
# Adding a keyword to done in check_is_done() so that wait_until_done()␣
 ↪eventually stops looping.
def wait_until_done():
  def check_is_done():
    # Add a keyword so that wait_until_done()
    # doesn't run forever
    global done
    if random.random() < 0.1:
      done = True

  while not done:
    check_is_done()

done = False
wait_until_done()

print('Work done? {}'.format(done))
```

```
Work done? True
```

Stellar scoping! By adding global done in check_is_done(), you ensure that the done being referenced is the one that was set to False before wait_until_done() was called. Without this keyword, wait_until_done() would loop forever because the done = True in check_is_done() would only be changing a variable that is local to check_is_done(). Understanding what scope your variables are in will help you debug tricky situations like this one.

Checking for closure

You're teaching your niece how to program in Python, and she is working on returning nested functions. She thinks she has written the code correctly, but she is worried that the returned function won't have the necessary information when called. Show her that all of the nonlocal variables she needs are in the new function's closure.

[62]:
```
# Using an attribute of the my_func() function to show that it has a closure␣
 ↪that is not None.

def return_a_func(arg1, arg2):
  def new_func():
    print('arg1 was {}'.format(arg1))
    print('arg2 was {}'.format(arg2))
  return new_func

my_func = return_a_func(2, 17)

# Showing that my_func()'s closure is not None
print(my_func.__closure__ is not None)
```

```
True
```

[63]:
```
# Showing that there are two variables in the closure.

def return_a_func(arg1, arg2):
  def new_func():
    print('arg1 was {}'.format(arg1))
    print('arg2 was {}'.format(arg2))
  return new_func

my_func = return_a_func(2, 17)

print(my_func.__closure__ is not None)

# Showing that there are two variables in the closure
print(len(my_func.__closure__) == 2)
```

```
True
True
```

Get the values of the variables in the closure so you can show that they are equal to [2, 17], the arguments passed to return_a_func().

Each item in the closure is called a "cell".

Each cell has "contents" that stores the value of one variable.

[64]:
```
def return_a_func(arg1, arg2):
  def new_func():
    print('arg1 was {}'.format(arg1))
    print('arg2 was {}'.format(arg2))
  return new_func

my_func = return_a_func(2, 17)

print(my_func.__closure__ is not None)
print(len(my_func.__closure__) == 2)

# Getting the values of the variables in the closure
```

```
closure_values = [
  my_func.__closure__[i].cell_contents for i in range(2)
]
print(closure_values == [2, 17])
```

```
True
True
True
```

Case closed! Your niece is relieved to see that the values she passed to return_a_func() are still accessible to the new function she returned, even after the program has left the scope of return_a_func().

Values get added to a function's closure in the order they are defined in the enclosing function (in this case, arg1 and then arg2), but only if they are used in the nested function. That is, if return_a_func() took a third argument (e.g., arg3) that wasn't used by new_func(), then it would not be captured in new_func()'s closure.

Closures keep your values safe

You are still helping your niece understand closures. You have written the function get_new_func() that returns a nested function. The nested function call_func() calls whatever function was passed to get_new_func(). You've also written my_special_function() which simply prints a message that states that you are executing my_special_function().

You want to show your niece that no matter what you do to my_special_function() after passing it to get_new_func(), the new function still mimics the behavior of the original my_special_function() because it is in the new function's closure.

[65]:
```
# Showing that you still get the original message even if you redefine␣
↪my_special_function() to only print "hello".
def my_special_function():
  print('You are running my_special_function()')

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

new_func = get_new_func(my_special_function)

# Redefining my_special_function() to just print "hello"
def my_special_function():
  print('hello')

new_func()
```

```
You are running my_special_function()
```

[66]:
```python
# Showing that even if you delete my_special_function(), you can still call␣
↪new_func() without any problems.

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

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

new_func = get_new_func(my_special_function)

# Deleting my_special_function()
del(my_special_function)

new_func()
```

```
You are running my_special_function()
```

[67]:
```python
# Showing that you still get the original message even if you overwrite␣
↪my_special_function() with the new function.
def my_special_function():
  print('You are running my_special_function()')

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

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

my_special_function()
```

```
You are running my_special_function()
```

Well done! Your niece feels like she understands closures now. She has seen that you can modify, delete, or overwrite the values needed by the nested function, but the nested function can still access those values because they are stored safely in the function's closure. She even realized that you could run into memory issues if you wound up adding a very large array or object to the closure, and has resolved to keep her eye out for that sort of problem.

Using decorator syntax

You have written a decorator called print_args that prints out all of the arguments and their values any time a function that it is decorating gets called.

```
[ ]: # Decorating my_function() with the print_args() decorator by redefining␣
     ↪my_function().

     print_args (a + b + c)
     def my_function(a, b, c):
       print(a + b + c)

     # Decorating my_function() with the print_args() decorator
     my_function = print_args(my_function)

     my_function(1, 2, 3)

     # Decorating my_function() with the print_args() decorator using decorator␣
     ↪syntax.
     # Decorating my_function() with the print_args() decorator
     @print_args
     def my_function(a, b, c):
       print(a + b + c)

     my_function(1, 2, 3)
```

What a delightful decorator! Note that @print_args before the definition of my_function is exactly equivalent to my_function = print_args(my_function). Remember, even though decorators are functions themselves, when you use decorator syntax with the @ symbol you do not include the parentheses after the decorator name.

Defining a decorator

Your buddy has been working on a decorator that prints a "before" message before the decorated function is called and prints an "after" message after the decorated function is called. They are having trouble remembering how wrapping the decorated function is supposed to work. Help them out by finishing their print_before_and_after() decorator.

- Call the function being decorated and pass it the positional arguments *args.
- Return the new decorated function.

```
[75]: def print_before_and_after(func):
        def wrapper(*args):
          print('Before {}'.format(func.__name__))
          # Call the function being decorated with *args
          func(*args)
          print('After {}'.format(func.__name__))
        # Return the nested function
        return wrapper

      @print_before_and_after
      def multiply(a, b):
        print(a * b)
```

```
multiply(5, 10)
```

```
Before multiply
50
After multiply
```

What a darling decorator! The decorator print_before_and_after() defines a nested function wrapper() that calls whatever function gets passed to print_before_and_after(). wrapper() adds a little something else to the function call by printing one message before the decorated function is called and another right afterwards. Since print_before_and_after() returns the new wrapper() function, we can use it as a decorator to decorate the multiply() function.

Printing the return type

You are debugging a package that you've been working on with your friends. Something weird is happening with the data being returned from one of your functions, but you're not even sure which function is causing the trouble. You know that sometimes bugs can sneak into your code when you are expecting a function to return one thing, and it returns something different. For instance, if you expect a function to return a numpy array, but it returns a list, you can get unexpected behavior. To ensure this is not what is causing the trouble, you decide to write a decorator, print_return_type(), that will print out the type of the variable that gets returned from every call of any function it is decorating.

[76]:
```
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}))
```

```
foo() returned type <class 'int'>
42
foo() returned type <class 'list'>
[1, 2, 3]
foo() returned type <class 'dict'>
```

```
{'a': 42}
```

Righteous return types! Your new decorator helps you examine the results of your functions at runtime. Now you can apply this decorator to every function in the package you are developing and run your scripts. Being able to examine the types of your return values will help you understand what is happening and will hopefully help you find the bug.

Counter

You're working on a new web app, and you are curious about how many times each of the functions in it gets called. So you decide to write a decorator that adds a counter to each function that you decorate. You could use this information in the future to determine whether there are sections of code that you could remove because they are no longer being used by the app.

```
[77]: 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))
```

```
calling foo()
calling foo()
foo() was called 2 times.
```

Cool counting! Now you can go decorate a bunch of functions with the counter() decorator, let your program run for a while, and then print out how many times each function was called.

It seems a little magical that you can reference the wrapper() function from inside the definition of wrapper() as we do here on line 3. That's just one of the many neat things about functions in Python – any function, not just decorators.

Preserving docstrings when decorating functions

Your friend has come to you with a problem. They've written some nifty decorators and added them to the functions in the open-source library they've been working on. However, they were running some tests and discovered that all of the docstrings have mysteriously disappeared from their decorated functions. Show your friend how to preserve docstrings and other metadata when writing decorators.

[78]:
```python
def add_hello(func):
  def wrapper(*args, **kwargs):
    print('Hello')
    return func(*args, **kwargs)
  return wrapper

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

print_sum(10, 20)
print(print_sum.__doc__)
```

```
Hello
30
None
```

- To show your friend that they are printing the wrapper() function's docstring, not the print_sum() docstring, add the following docstring to wrapper():

"""“Print ‘hello’ and then call the decorated function .”""

[79]:
```python
def add_hello(func):
  # Add a docstring to wrapper
  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(print_sum.__doc__)
```

```
Hello
30
Print 'hello' and then call the decorated function.
```

- Importing a function that will allow you to add the metadata from print_sum() to the decorated version of print_sum().

[80]:
```python
# Importing the function you need to fix the problem
from functools import wraps
```

```python
def add_hello(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(print_sum.__doc__)
```

```
Hello
30
Print 'hello' and then call the decorated function.
```

- Finally, decorate wrapper() so that the metadata from func() is preserved in the new decorated function.

[81]:
```python
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(print_sum.__doc__)
```

```
Hello
30
Adds two numbers and prints the sum
```

That's a wrap! Your friend was concerned that they couldn't print the docstrings of their functions. They now realize that the strange behavior they were seeing was caused by the fact that they were accidentally printing the wrapper() docstring instead of the docstring of the original function. After adding @wraps(func) to all of their decorators, they see that the docstrings are back where they expect them to be.

Measuring decorator overhead

Your boss wrote a decorator called check_everything() that they think is amazing, and they are insisting you use it on your function. However, you've noticed that when you use it to decorate your functions, it makes them run much slower. You need to convince your boss that the decorator is adding too much processing time to your function. To do this, you are going to measure how long the decorated function takes to run and compare it to how long the undecorated function would have taken to run. This is the decorator in question:

[93]:
```
def check_everything(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    check_inputs(*args, **kwargs)
    result = func(*args, **kwargs)
    check_outputs(result)
    return result
  return wrapper
```

[94]:
```
from time import sleep
import random
import time
```

[ ]:
```
@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))
```

Your function ran approximately 10,000 times faster without your boss's decorator. At least they were smart enough to add @wraps(func) to the nested wrapper() function so that you were able to access the original function. You should show them the results of this test. Be sure to ask for a raise while you're at it!

Run_n_times()

Below is a code for a decorator. Practice different ways of applying the decorator to the function print_sum(). Then I'll show you a funny prank you can play on your co-workers.

[97]:
```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
```

Adding the run_n_times() decorator to print_sum() using decorator syntax so that print_sum() runs 10 times.

Using run_n_times() to create a decorator run_five_times() that will run any function five times.

Here's the prank: use run_n_times() to modify the built-in print() function so that it always prints 20 times!

[98]:
```python
# Making print_sum() run 10 times with the run_n_times() decorator
@run_n_times(10)
def print_sum(a, b):
  print(a + b)

print_sum(15, 20)
```

```
35
35
35
35
35
35
35
35
35
35
```

[99]:
```python
# Using run_n_times() to create the run_five_times() decorator
run_five_times = run_n_times(5)

@run_five_times
def print_sum(a, b):
  print(a + b)

print_sum(4, 100)
```

```
104
```

```
104
104
104
104
```

[ ]:
```
# Modifying the print() function to always run 20 times

print = run_n_times(20)(print)

print('What is happening?!?!')
```

You've become an expert at using decorators. Notice how when you use decorator syntax for a decorator that takes arguments, you need to call the decorator by adding parentheses, but you don't add parenthesis for decorators that don't take arguments.

Warning: overwriting commonly used functions is probably not a great idea, so think twice before using these powers for evil.

HTML Generator

You are writing a script that generates HTML for a webpage on the fly. So far, you have written two decorators that will add bold or italics tags to any function that returns a string. You notice, however, that these two decorators look very similar. Instead of writing a bunch of other similar looking decorators, you want to create one decorator, html(), that can take any pair of opening and closing tags.

[101]:
```
def bold(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    msg = func(*args, **kwargs)
    return '<b>{}</b>'.format(msg)
  return wrapper
def italics(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    msg = func(*args, **kwargs)
    return '<i>{}</i>'.format(msg)
  return wrapper
```

• Returning the decorator and the decorated function from the correct places in the new html() decorator.

[102]:
```
def html(open_tag, close_tag):
  def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
      msg = func(*args, **kwargs)
      return '{}{}{}'.format(open_tag, msg, close_tag)
    # Return the decorated function
    return wrapper
  # Return the decorator
  return decorator
```

Using the html() decorator to wrap the return value of hello() in and (the HTML tags that mean "bold").

[103]:
```
# Makeing hello() return bolded text
@html('<b>', '</b>')
def hello(name):
  return 'Hello {}!'.format(name)

print(hello('Alice'))
```

```
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
<b>Hello Alice!</b>
```

Using html() to wrap the return value of goodbye() in and (the HTML tags that mean "italics").

[104]:
```
# Making goodbye() return italicized text
@html('<i>', '</i>')
def goodbye(name):
  return 'Goodbye {}.'.format(name)

print(goodbye('Alice'))
```

```
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
```

```
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
<i>Goodbye Alice.</i>
```

[105]:
```
# Wrapping the result of hello_goodbye() in <div> and </div>
@html('<div>', '</div>')
def hello_goodbye(name):
  return '\n{}\n{}\n'.format(hello(name), goodbye(name))

print(hello_goodbye('Alice'))
```

```
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
```

```
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
```

```
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
<div>
<b>Hello Alice!</b>
<i>Goodbye Alice.</i>
</div>
```

That's some HTML hotness! With the new html() decorator you can focus on writing simple functions that return the information you want to display on the webpage and let the decorator take care of wrapping them in the appropriate HTML tags.

Tag your functions

Tagging something means that you have given that thing one or more strings that act as labels. For instance, we often tag emails or photos so that we can search for them later. You've decided to write a decorator that will let you tag your functions with an arbitrary list of tags. You could use these tags for many things:

Adding information about who has worked on the function, so a user can look up who to ask if they run into trouble using it. Labeling functions as "experimental" so that users know that the inputs and outputs might change in the future. Marking any functions that you plan to remove in a future version of the code. Etc.

[106]:
```python
def tag(*tags):
  # Defining a new decorator, named "decorator", to return
  def decorator(func):
    # Ensure the decorated function keeps its metadata
    @wraps(func)
    def wrapper(*args, **kwargs):
      # Calling the function being decorated and return the result
      return func(*args, **kwargs)
    wrapper.tags = tags
    return wrapper
  # Returning the new decorator
  return decorator

@tag('test', 'this is a tag')
def foo():
  pass

print(foo.tags)
```

```
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
```

```
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
('test', 'this is a tag')
```

Terrific tagging! With this new decorator, you can do some really interesting things. For instance, you could tag a bunch of image transforming functions, and then write code that searches for all of the functions that transform images and apply them, one after the other, on a given input image. What other neat uses can you come up with for this decorator?

Check the return type

Python's flexibility around data types is usually cited as one of the benefits of the language. It can occasionally cause problems though if incorrect data types go unnoticed. You've decided that in order to make sure your code is doing exactly what you want it to do, you will explicitly check the return types of all of your functions and make sure they are what you expect them to be. To do that, you are going to create a decorator that checks that the return type of the decorated function is correct.

Note: assert(condition) is a function that you can use to test whether something is true. If condition is True, this function doesn't do anything. If condition is False, this function raises an error. The type of error that it raises is called an AssertionError.

[107]:
```
def returns_dict(func):
  # Complete the returns_dict() decorator
  def wrapper(func):
    result = wrapper
    assert(type(result) == dict)
    return result
  return wrapper

@returns_dict
def foo(value):
  return value

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

```
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
```

Now complete the returns() decorator, which takes the expected return type as an argument.

[108]:
```
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!')
```

```
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
foo() did not return a dict!
```

You did it! We took the training wheels off on this exercise, and you still did a great job. You know how to write your own decorators now, but even more importantly, you know why they work the way they do.

Thanks a lot for your attention.

```
[ ]:
```