Docstrings

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

A complex function

```
def split_and_stack(df, new_names):
  half = int(len(df.columns) / 2)
  left = df.iloc[:, :half]
  right = df.iloc[:, half:]
  return pd.DataFrame(
    data=np.vstack([left.values, right.values]),
    columns=new_names
  )
```

```
def split_and_stack(df, new_names):
  """Split a DataFrame's columns into two halves and then stack
  them vertically, returning a new DataFrame with `new_names` as the
  column names.

  Args:
    df (DataFrame): The DataFrame to split.
    new_names (iterable of str): The column names for the new DataFrame.

  Returns:
    DataFrame
  """
  half = int(len(df.columns) / 2)
  left = df.iloc[:, :half]
  right = df.iloc[:, half:]
  return pd.DataFrame(
    data=np.vstack([left.values, right.values]),
    columns=new_names
  )
```

Anatomy of a docstring

```
def function_name(arguments):
  """
  Description of what the function does.

  Description of the arguments, if any.

  Description of the return value(s), if any.

  Description of errors raised, if any.

  Optional extra notes or examples of usage.
  """
```

Docstring formats

- Google Style
- Numpydoc
- reStructuredText
- EpyText

Google Style - description

```
def function(arg_1, arg_2=42):
  """Description of what the function does.
  """
```

Google style - arguments

```
def function(arg_1, arg_2=42):
  """Description of what the function does.

  Args:
    arg_1 (str): Description of arg_1 that can break onto the next line
      if needed.
    arg_2 (int, optional): Write optional when an argument has a default
      value.
  """
```

Google style - return value(s)

```
def function(arg_1, arg_2=42):
  """Description of what the function does.

  Args:
    arg_1 (str): Description of arg_1 that can break onto the next line
      if needed.
    arg_2 (int, optional): Write optional when an argument has a default
      value.

  Returns:
    bool: Optional description of the return value
    Extra lines are not indented.
  """
```

```
def function(arg_1, arg_2=42):
  """Description of what the function does.

  Args:
    arg_1 (str): Description of arg_1 that can break onto the next line
      if needed.
    arg_2 (int, optional): Write optional when an argument has a default
      value.

  Returns:
    bool: Optional description of the return value
    Extra lines are not indented.

  Raises:
    ValueError: Include any error types that the function intentionally
      raises.

  Notes:
    See https://www.datacamp.com/community/tutorials/docstrings-python
    for more info.
  """
```

Numpydoc

```
def function(arg_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 include a description of the return value.
    Replace "Returns" with "Yields" if this function is a generator.
  """
```

Retrieving docstrings

```
def the_answer():
  """Return the answer to life,
  the universe, and everything.

  Returns:
    int
  """
  return 42
print(the_answer.__doc__)
```

```
Return the answer to life,
  the universe, and everything.

  Returns:
    int
```

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

```
Return the answer to life,
the universe, and everything.

Returns:
  int
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

DRY and "Do One Thing"

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Director of Software Engineering @ American Efficient

Don't repeat yourself (DRY)

```
train = pd.read_csv('train.csv')
train_y = train['labels'].values
train_X = train[col for col in train.columns if col != 'labels'].values
train_pca = PCA(n_components=2).fit_transform(train_X)
plt.scatter(train_pca[:,0], train_pca[:,1])
```

```
val = pd.read_csv('validation.csv')
val_y = val['labels'].values
val_X = val[col for col in val.columns if col != 'labels'].values
val_pca = PCA(n_components=2).fit_transform(val_X)
plt.scatter(val_pca[:,0], val_pca[:,1])
```

```
test = pd.read_csv('test.csv')
test_y = test['labels'].values
test_X = test[col for col in test.columns if col != 'labels'].values
test_pca = PCA(n_components=2).fit_transform(train_X)
```

The problem with repeating yourself

```
train = pd.read_csv('train.csv')
train_y = train['labels'].values
train_X = train[col for col in train.columns if col != 'labels'].values
train_pca = PCA(n_components=2).fit_transform(train_X)
plt.scatter(train_pca[:,0], train_pca[:,1])
```

```
val = pd.read_csv('validation.csv')
val_y = val['labels'].values
val_X = val[col for col in val.columns if col != 'labels'].values
val_pca = PCA(n_components=2).fit_transform(val_X)
plt.scatter(val_pca[:,0], val_pca[:,1])
```

```
test = pd.read_csv('test.csv')
test_y = test['labels'].values
test_X = test[col for col in test.columns if col != 'labels'].values
test_pca = PCA(n_components=2).fit_transform(train_X)  ### yikes! ###
plt.scatter(test_pca[:,0], test_pca[:,1])
```

Another problem with repeating yourself

```
train = pd.read_csv('train.csv')
train_y = train['labels'].values  ### <- there and there --v ###
train_X = train[col for col in train.columns if col != 'labels'].values
train_pca = PCA(n_components=2).fit_transform(train_X)
plt.scatter(train_pca[:,0], train_pca[:,1])
```

```
val = pd.read_csv('validation.csv')
val_y = val['labels'].values  ### <- there and there --v ###
val_X = val[col for col in val.columns if col != 'labels'].values
val_pca = PCA(n_components=2).fit_transform(val_X)
plt.scatter(val_pca[:,0], val_pca[:,1])
```

```
test = pd.read_csv('test.csv')
test_y = test['labels'].values  ### <- there and there --v ###
test_X = test[col for col in test.columns if col != 'labels'].values
test_pca = PCA(n_components=2).fit_transform(test_X)
plt.scatter(test_pca[:,0], test_pca[:,1])
```

Use functions to avoid repetition

```
def load_and_plot(path):
  """Load a data set and plot the first two principal components.

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

  Returns:
    tuple of ndarray: (features, labels)
  """
  data = pd.read_csv(path)
  y = data['label'].values
  X = data[col for col in train.columns if col != 'label'].values
  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])
  return X, y
```

```
train_X, train_y = load_and_plot('train.csv')
val_X, val_y = load_and_plot('validation.csv')
test_X, test_y = load_and_plot('test.csv')
```

```
def load_and_plot(path):
  """Load a data set and plot the first two principal components.

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

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

  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])

  return X, y
```

```
def load_and_plot(path):
  """Load a data set and plot the first two principal components.

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

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

  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])

  return X, y
```

```
def load_and_plot(path):
  """Load a data set and plot the first two principal components.

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

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

  # plot the first two principal components
  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])

  return X, y
```

```
def load_and_plot(path):
  """Load a data set and plot the first two principal components.

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

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

  # plot the first two principle components
  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])

  # return loaded data
  return X, y
```

Do One Thing

```
def load_data(path):
  """Load a data set.

  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
```

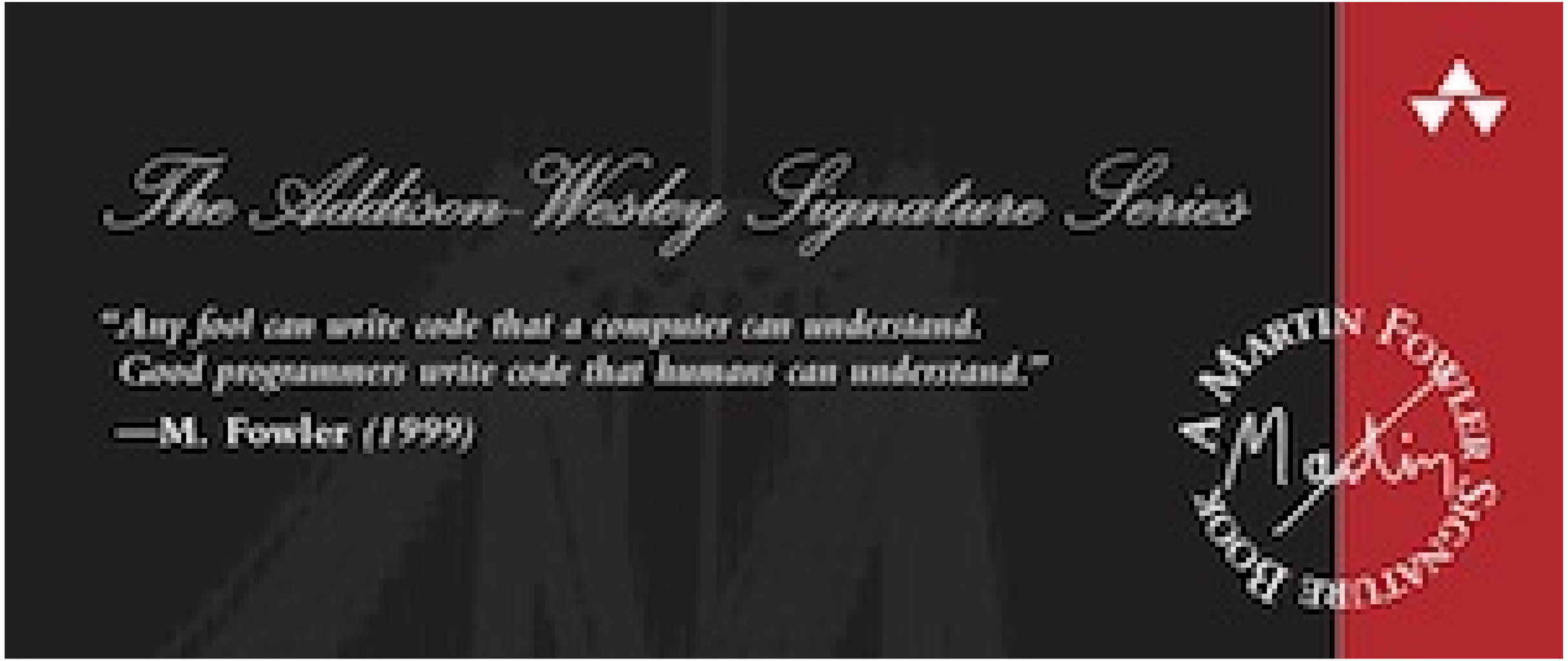
```
def plot_data(X):
  """Plot the first two principal components of a matrix

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

Advantages of doing one thing

The code becomes:

- More flexible
- More easily understood
- Simpler to test
- Simpler to debug
- Easier to change

Code smells and refactoring



Let's practice!

WRITING FUNCTIONS IN PYTHON

Pass by assignment

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

A surprising example

```
def foo(x):
  x[0] = 99

my_list = [1, 2, 3]

foo(my_list)

print(my_list)
```

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

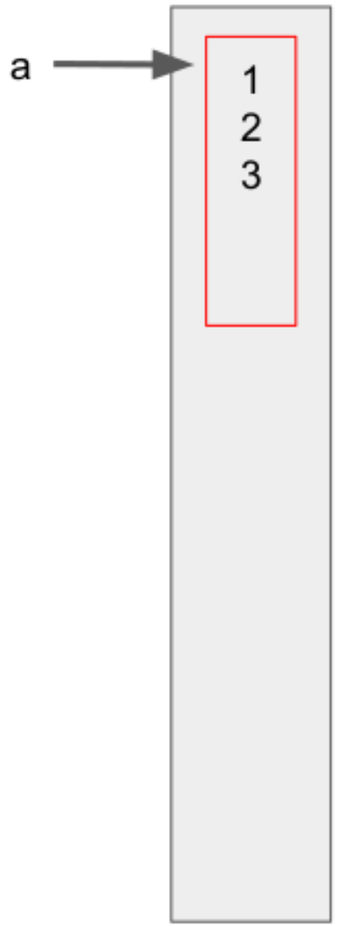
```
def bar(x):
  x = x + 90

my_var = 3

bar(my_var)

print(my_var)
```

```
3
```

Digging deeper

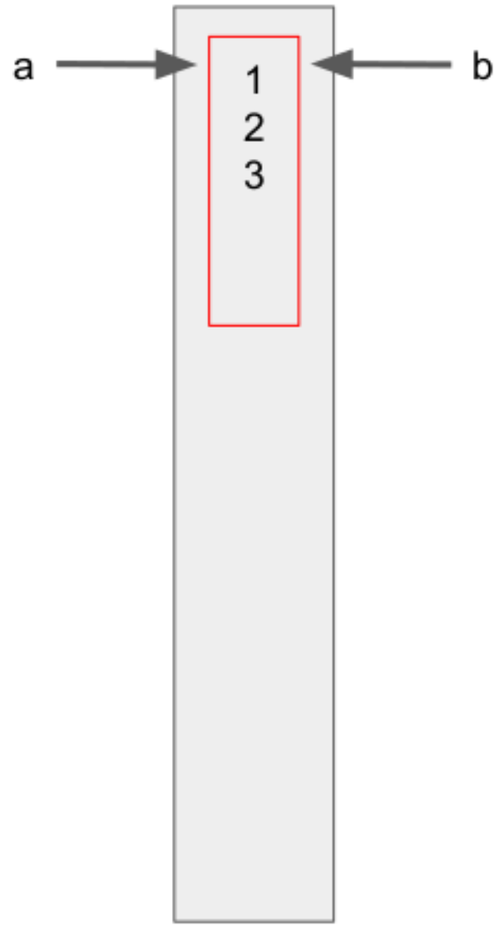
Digging deeper

```
a = [1, 2, 3]
```



Digging deeper

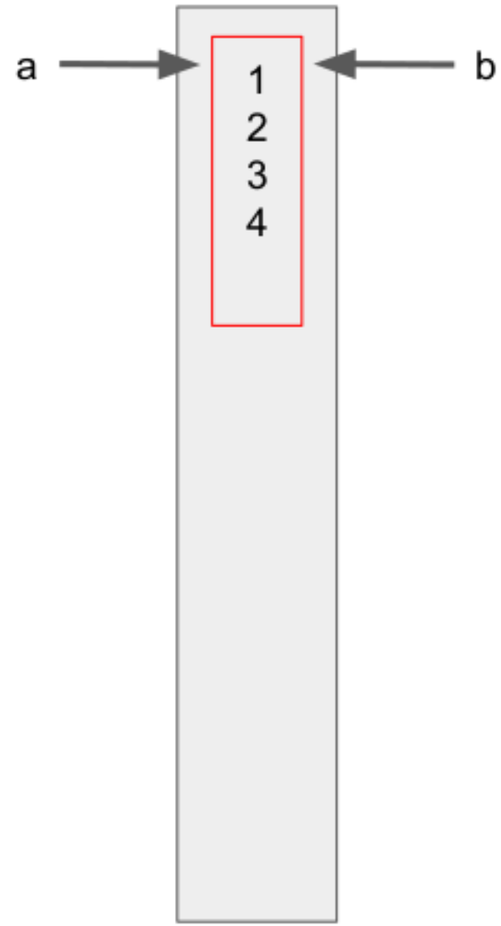
```
a = [1, 2, 3]
b = a
```



Digging deeper

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

```
[1, 2, 3, 4]
```



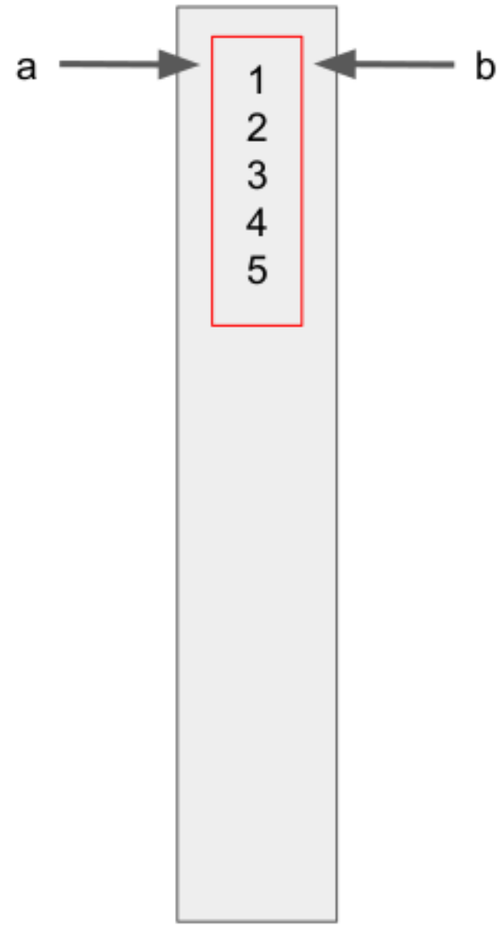
Digging deeper

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

```
[1, 2, 3, 4]
```

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

```
[1, 2, 3, 4, 5]
```



Digging deeper

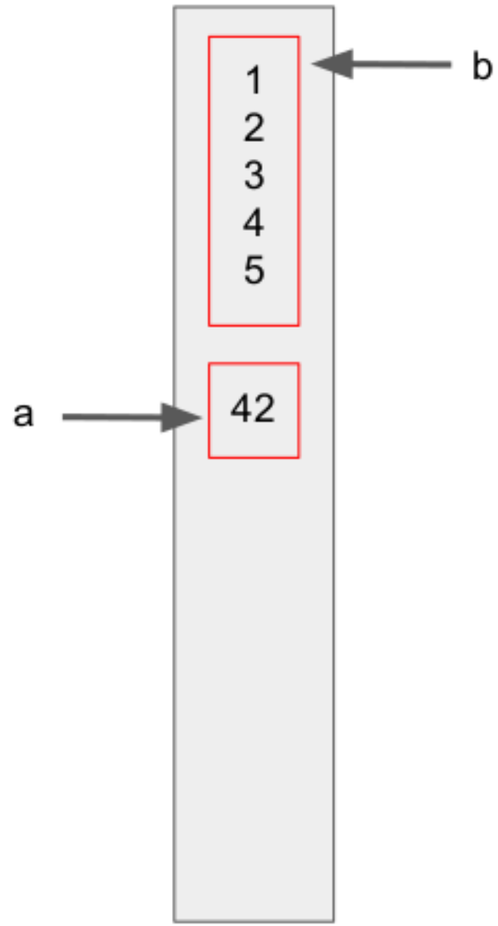
```
a = [1, 2, 3]
b = a
a.append(4)
print(b)
```

```
[1, 2, 3, 4]
```

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

```
[1, 2, 3, 4, 5]
```

```
a = 42
```

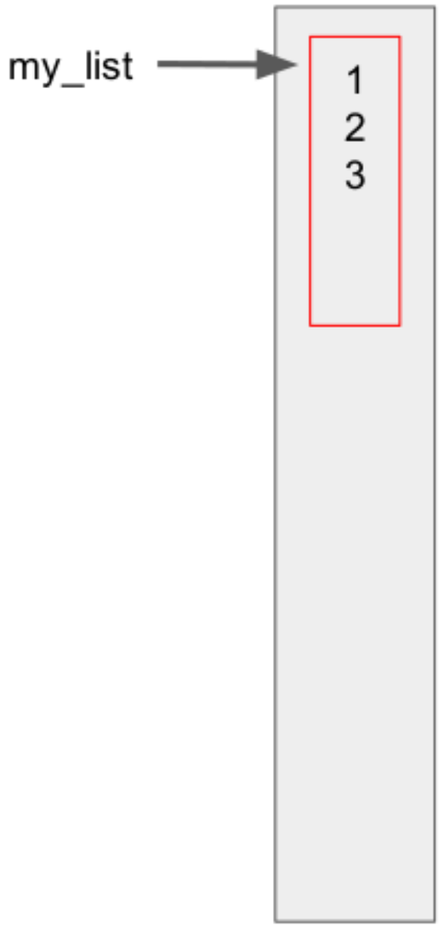


Pass by assignment

```
def foo(x):
  x[0] = 99
```

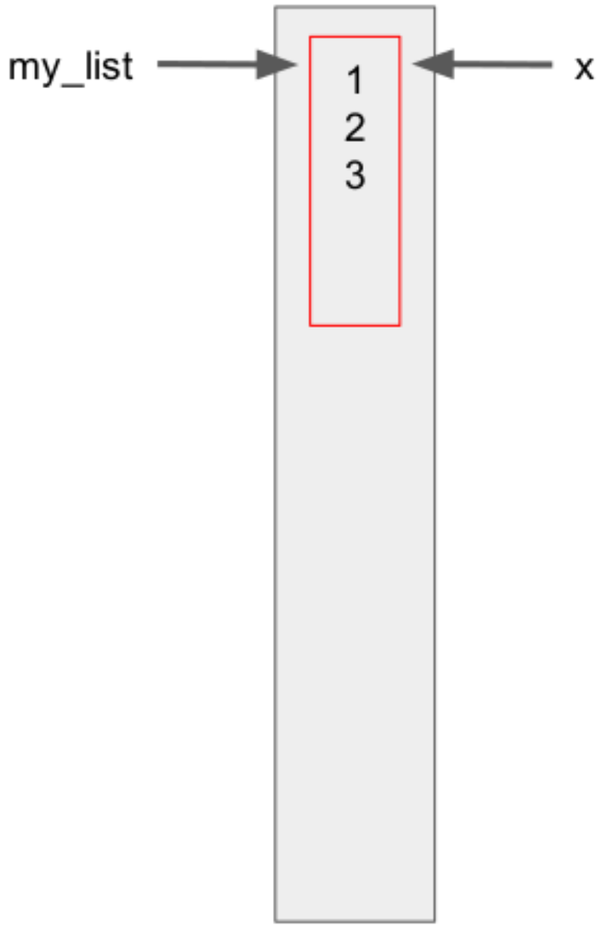
Pass by assignment

```
def foo(x):
  x[0] = 99

my_list = [1, 2, 3]
```



Pass by assignment

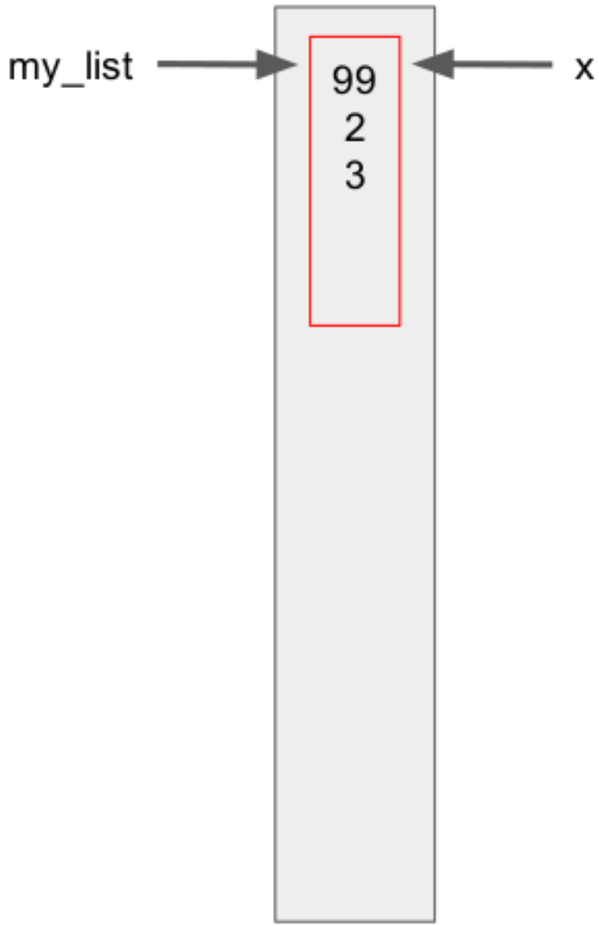
```
def foo(x):
  x[0] = 99

my_list = [1, 2, 3]

foo(my_list)
```



Pass by assignment

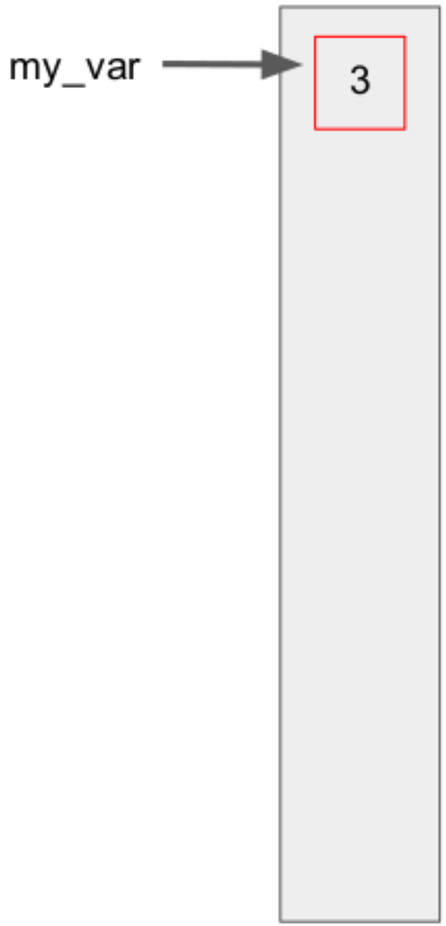
```
def foo(x):
  x[0] = 99

my_list = [1, 2, 3]

foo(my_list)

print(my_list)
```

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



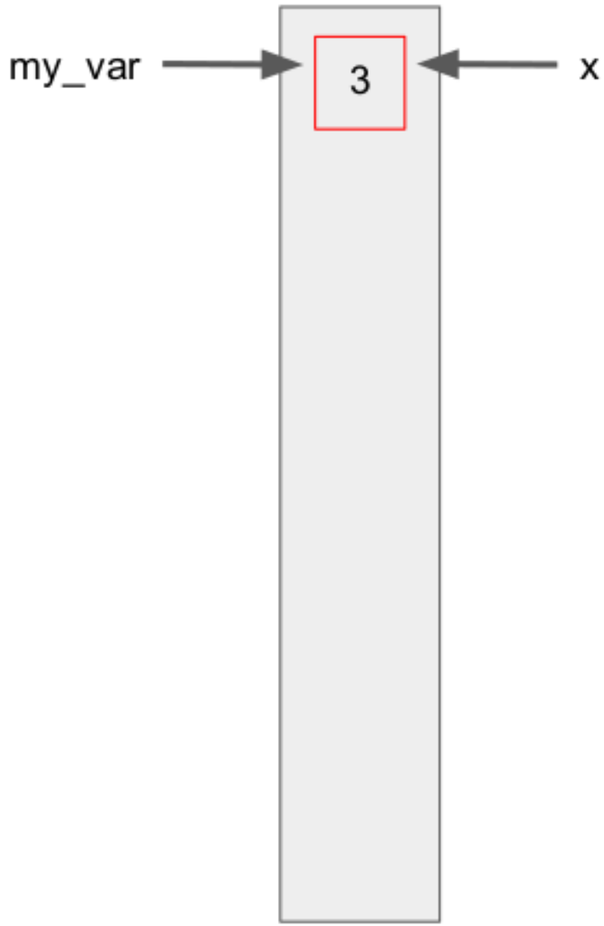
Pass by assignment

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

my_var = 3
```



Pass by assignment

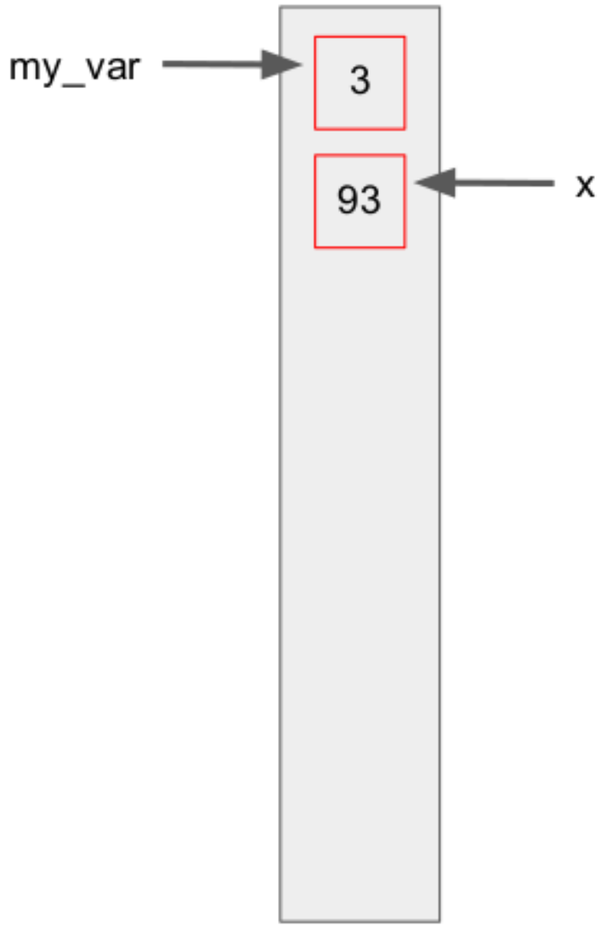
```
def bar(x):
  x = x + 90

my_var = 3

bar(my_var)
```



Pass by assignment

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

my_var = 3

bar(my_var)

my_var
```

```
3
```



Immutable or Mutable?

Immutable

- int
- float
- bool
- string
- bytes
- tuple
- frozenset
- None

Mutable

- list
- dict
- set
- bytearray
- objects
- functions
- almost everything else!

Mutable default arguments are dangerous!

```
def foo(var=[]):
  var.append(1)
  return var
foo()
```

```
[1]
```

```
foo()
```

```
[1, 1]
```

```
def foo(var=None):
  if var is None:
    var = []
  var.append(1)
  return var
foo()
```

```
[1]
```

```
foo()
```

```
[1]
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Using context managers

WRITING FUNCTIONS IN PYTHON

Shayne Miel

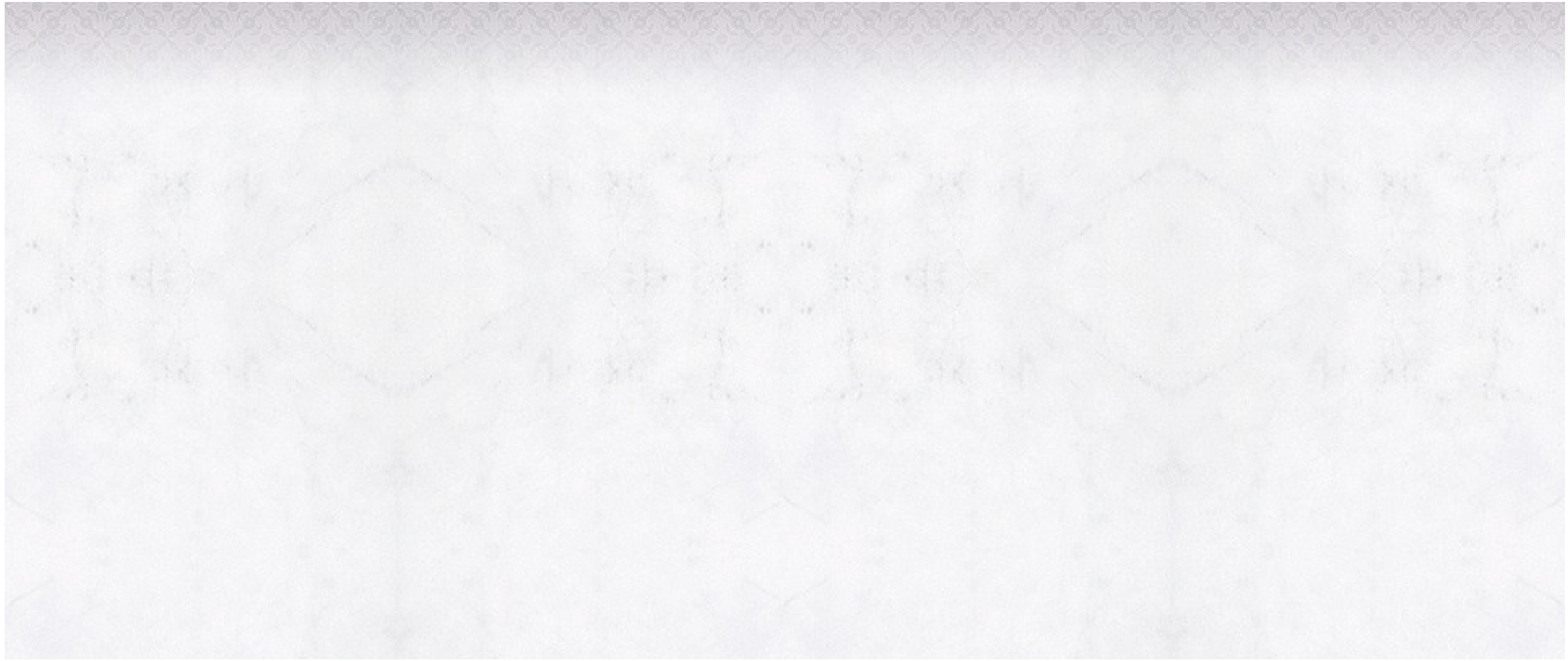
Director of Software Engineering @ American Efficient

What is a context manager?

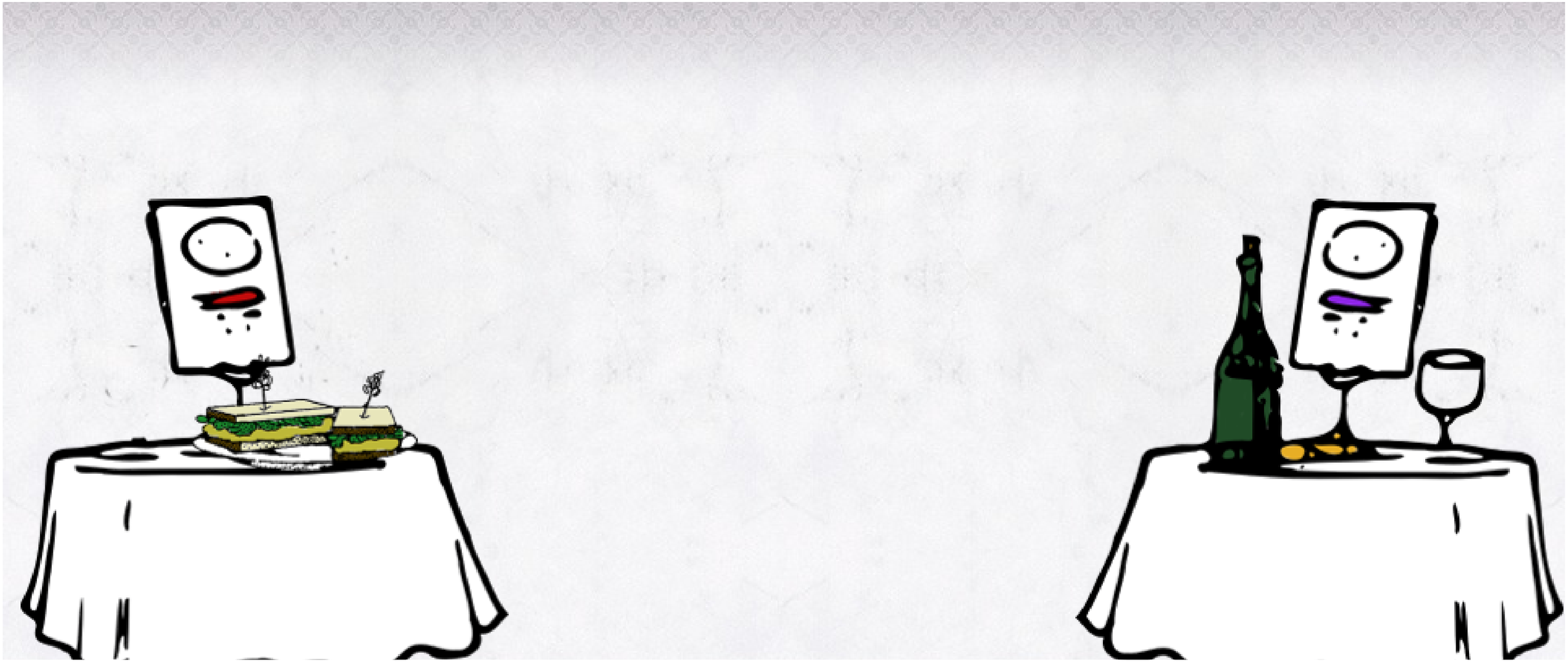
A context manager:

- Sets up a context
- Runs your code
- Removes the context

A catered party

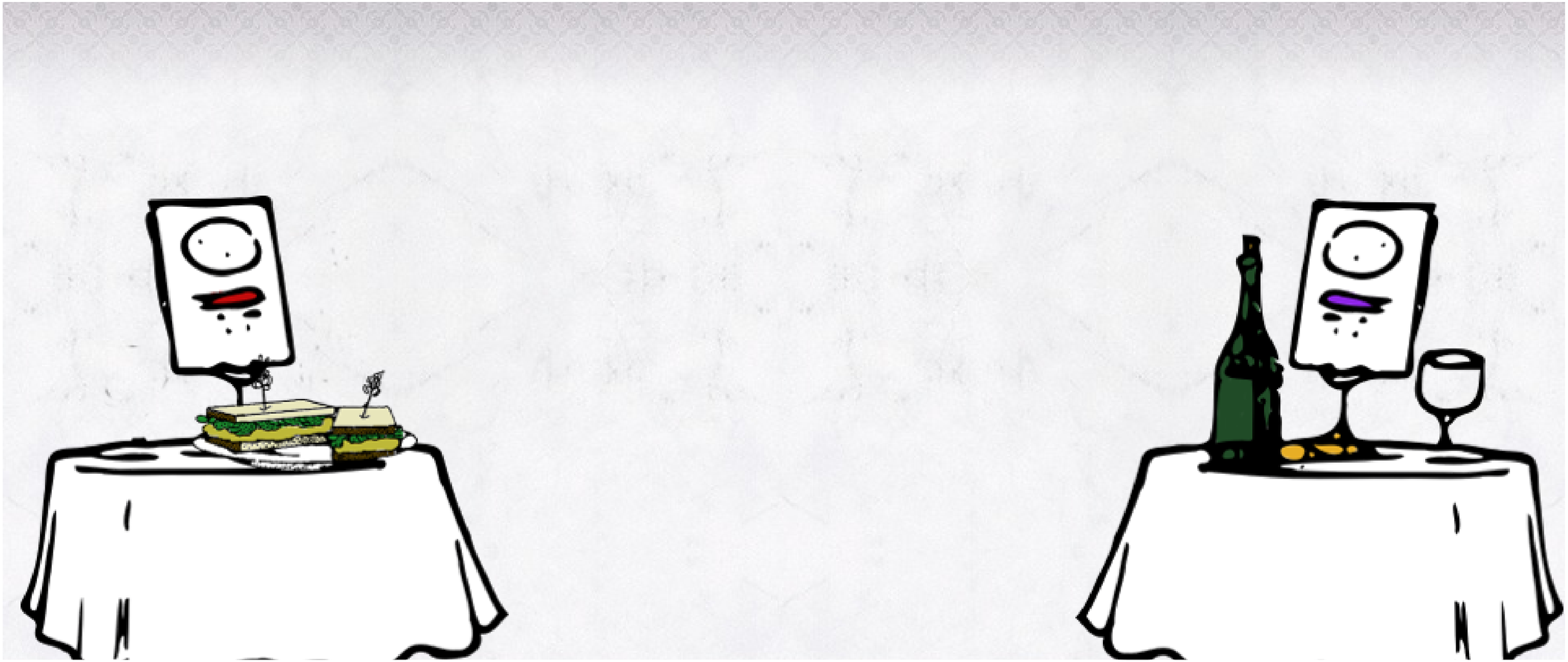


A catered party

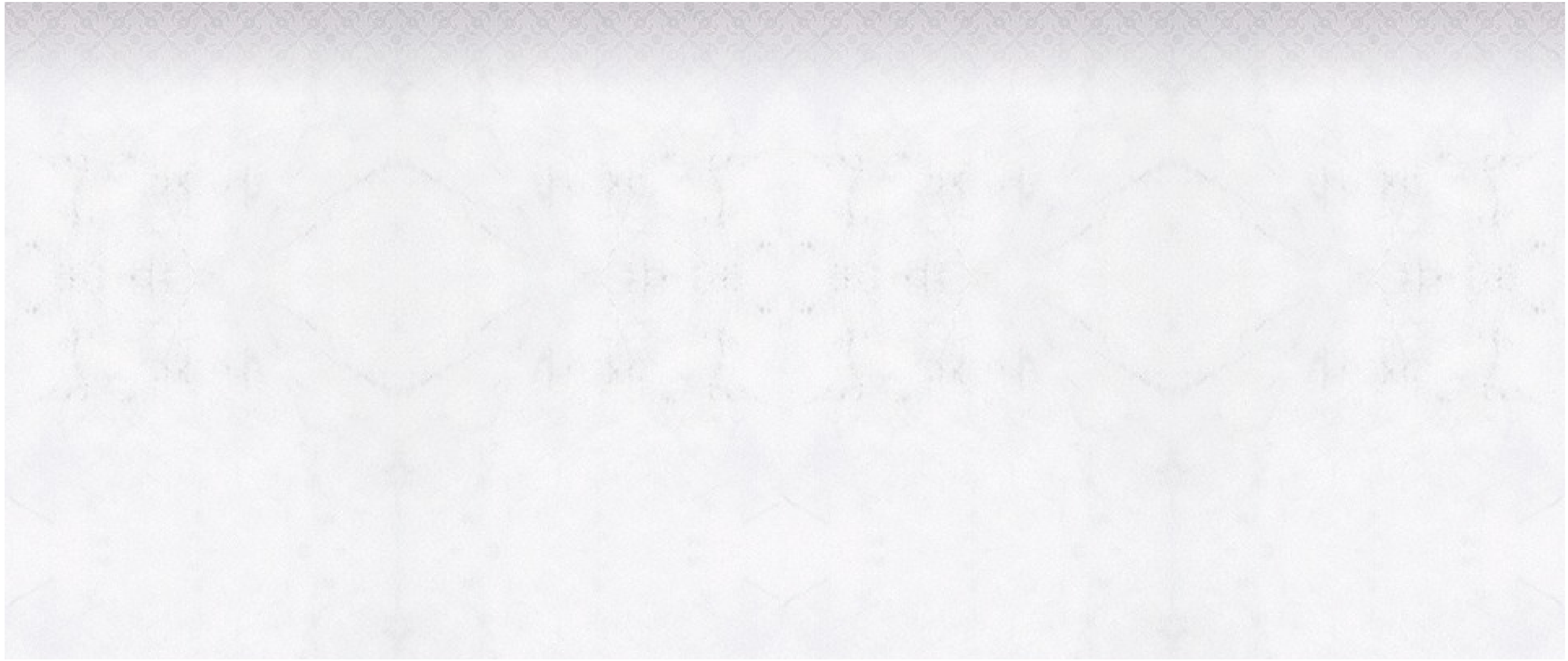


A catered party

A catered party



A catered party



Catered party as context

Context managers:

- Set up a context
- Run your code
- Remove the context

Caterers:

- Set up the tables with food and drink
- Let you and your friends have a party
- Cleaned up and removed the tables

A real-world 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:

- Sets up a context by opening a file
- Lets you run any code you want on that file
- Removes the context by closing the file

Using a context manager

```
with
```

Using a context manager

```
with <context-manager>()
```

Using a context manager

```
with <context-manager>(<args>)
```

Using a context manager

```
with <context-manager>(<args>):
```

Using a context manager

```
with <context-manager>(<args>):
  # Run your code here
  # This code is running "inside the context"
```

Using a context manager

```
with <context-manager>(<args>):
  # Run your code here
  # This code is running "inside the context"


# This code runs after the context is removed
```

Using a context manager

```
with <context-manager>(<args>) as <variable-name>:
  # Run your code here
  # This code is running "inside the context"

# This code runs after the context is removed
```

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

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

Let's practice!

WRITING FUNCTIONS IN PYTHON

Writing context managers

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Two ways to define a context manager

- Class-based
- Function-based

Two ways to define a context manager

- Class-based
- **Function-based ***

How to create a context manager

```
def my_context():
  # Add any set up code you need
  yield
  # Add any teardown code you need
```

1. Define a function.
2. (optional) Add any set up code your context needs.
3. Use the "yield" keyword.
4. (optional) Add any teardown code your context needs.

How to create a context manager

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

1. Define a function.
2. (optional) Add any set up code your context needs.
3. Use the "yield" keyword.
4. (optional) Add any teardown code your context needs.
5. Add the `@contextlib.contextmanager` decorator.

The "yield" keyword

```
@contextlib.contextmanager
def my_context():
  print('hello')
  yield 42
  print('goodbye')
```

```
with my_context() as foo:
  print('foo is {}'.format(foo))
```

```
hello
foo is 42
goodbye
```

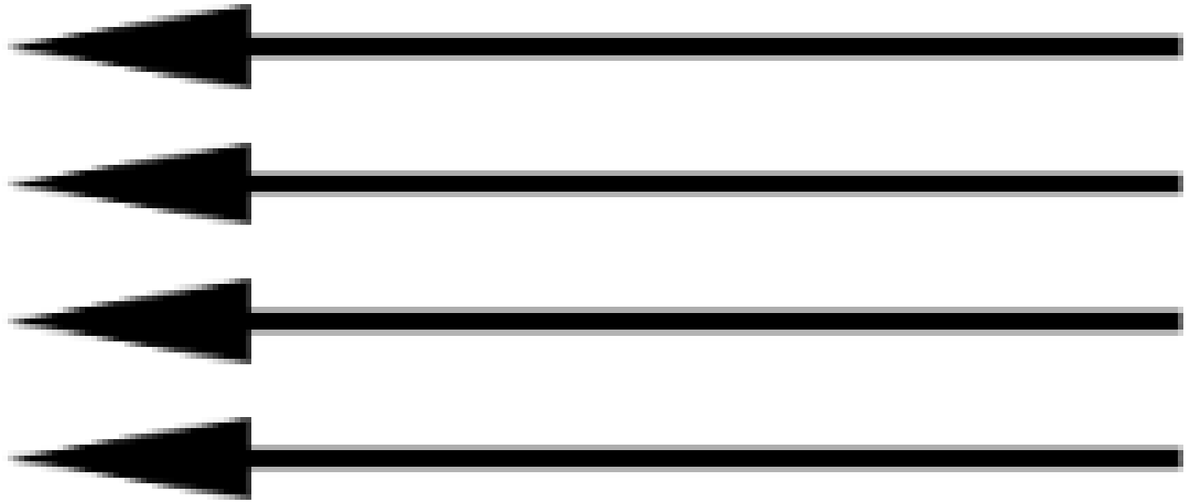
Setup and teardown

```
@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'
  )
```

Setup and teardown

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

  yield db

  # tear down database connection
  db.disconnect()
```

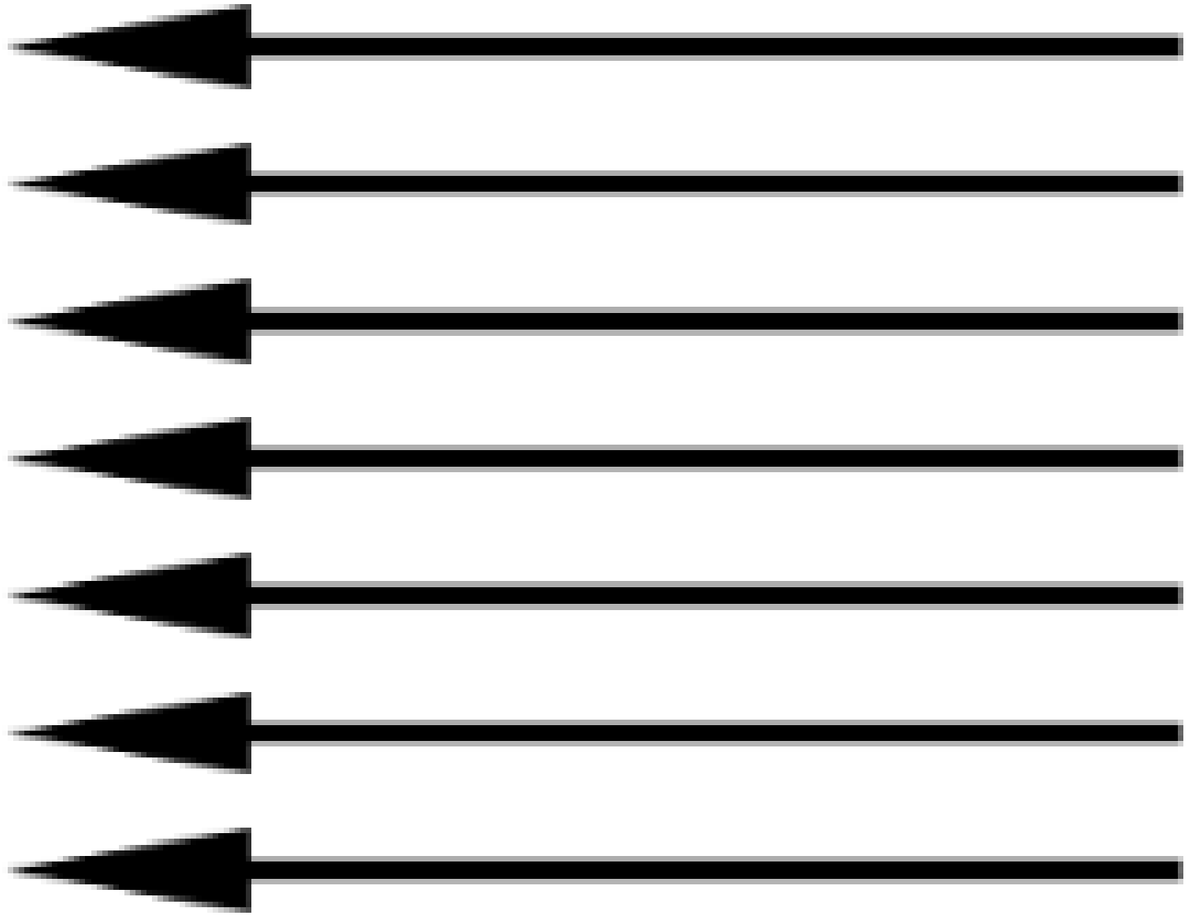


```
url = 'http://datacamp.com/data'
with database(url) as my_db:
  course_list = my_db.execute(
    'SELECT * FROM courses'
  )
```

Setup and teardown

```
@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'
  )
```



Yielding a value or None

```
@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'
  )
```

```
@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)
```

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

Let's practice!

WRITING FUNCTIONS IN PYTHON

Advanced topics

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Nested contexts

```
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 the source file and read in the contents
  with open(src) as f_src:
    contents = f_src.read()

  # Open the destination file and write out the contents
  with open(dst, 'w') as f_dst:
    f_dst.write(contents)
```

Nested contexts

```
with open('my_file.txt') as my_file:
  for line in my_file:
    # do something
```

Nested contexts

```
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

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

  yield

  # This MUST be called or no one else will
  # be able to connect to the printer
  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'])
```

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

Handling errors

```
try:
  # code that might raise an error
except:
  # do something about the error
finally:
  # this code runs no matter what
```

Handling errors

```
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'
```

Context manager patterns

Open	Close
Lock	Release
Change	Reset
Enter	Exit
Start	Stop
Setup	Teardown
Connect	Disconnect

[1] Adapted from Dave Brondsema's talk at PyCon 2012: https://youtu.be/cSbD5SKwak0?t=795

Let's practice!

WRITING FUNCTIONS IN PYTHON

Functions as objects

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Functions are just another type of object

Python objects:

```
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
```

Functions as variables

```
def my_function():
  print('Hello')
x = my_function
type(x)
```

```
<type 'function'>
```

```
x()
```

```
Hello
```

```
PrintyMcPrintface = print
PrintyMcPrintface('Python is awesome!')
```

```
Python is awesome!
```

Lists and dictionaries of functions

```
list_of_functions = [my_function, open, print]
list_of_functions[2]('I am printing with an element of a list!')
```

```
I am printing with an element of a list!
```

```
dict_of_functions = {
  'func1': my_function,
  'func2': open,
  'func3': print
}
dict_of_functions['func3']('I am printing with a value of a dict!')
```

```
I am printing with a value of a dict!
```

Referencing a function

```
def my_function():
  return 42

x = my_function

my_function()
```

```
42
```

```
my_function
```

```
<function my_function at 0x7f475332a730>
```

Functions as arguments

```
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
```

```
def no():
  return 42

def yes():
  """Return the value 42
  """
  return 42
```

```
has_docstring(no)
```

```
False
```

```
has_docstring(yes)
```

```
True
```

Defining a function inside another function

```
def foo():
  x = [3, 6, 9]

  def bar(y):
    print(y)

  for value in x:
    bar(x)
```

Defining a function inside another function

```
def foo(x, y):
  if x > 4 and x < 10 and y > 4 and y < 10:
    print(x * y)
```

```
def foo(x, y):
  def in_range(v):
    return v > 4 and v < 10

  if in_range(x) and in_range(y):
    print(x * y)
```

Functions as return values

```
def get_function():
  def print_me(s):
    print(s)

  return print_me
```

```
new_func = get_function()
new_func('This is a sentence.')
```

```
This is a sentence.
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Scope

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Names

Names

Scope

Scope

Scope

```
x = 7
y = 200
print(x)
```

```
7
```

```
def foo():
    x = 42
    print(x)
    print(y)
```

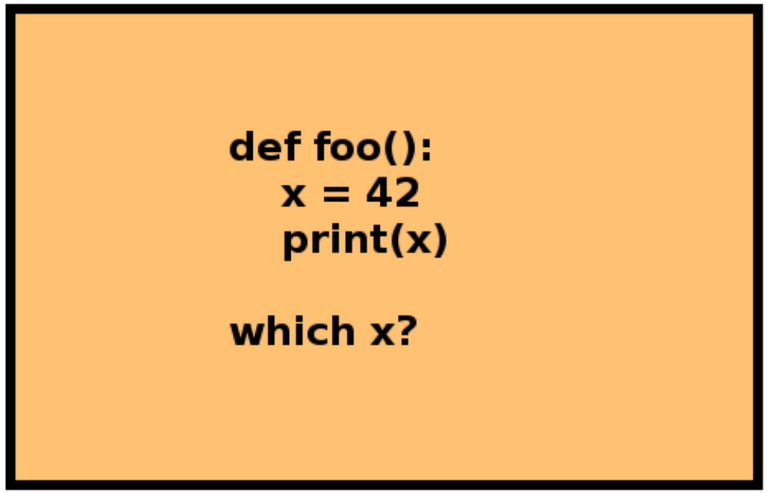
```
foo()
```

```
42
200
```

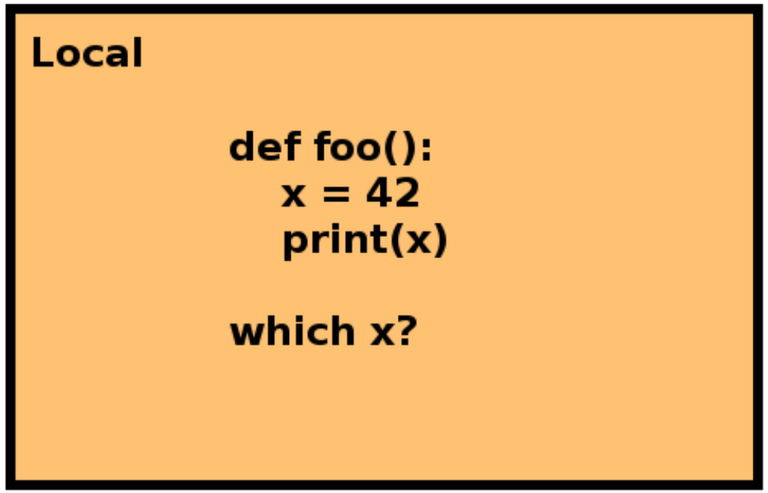
```
print(x)
```

```
7
```

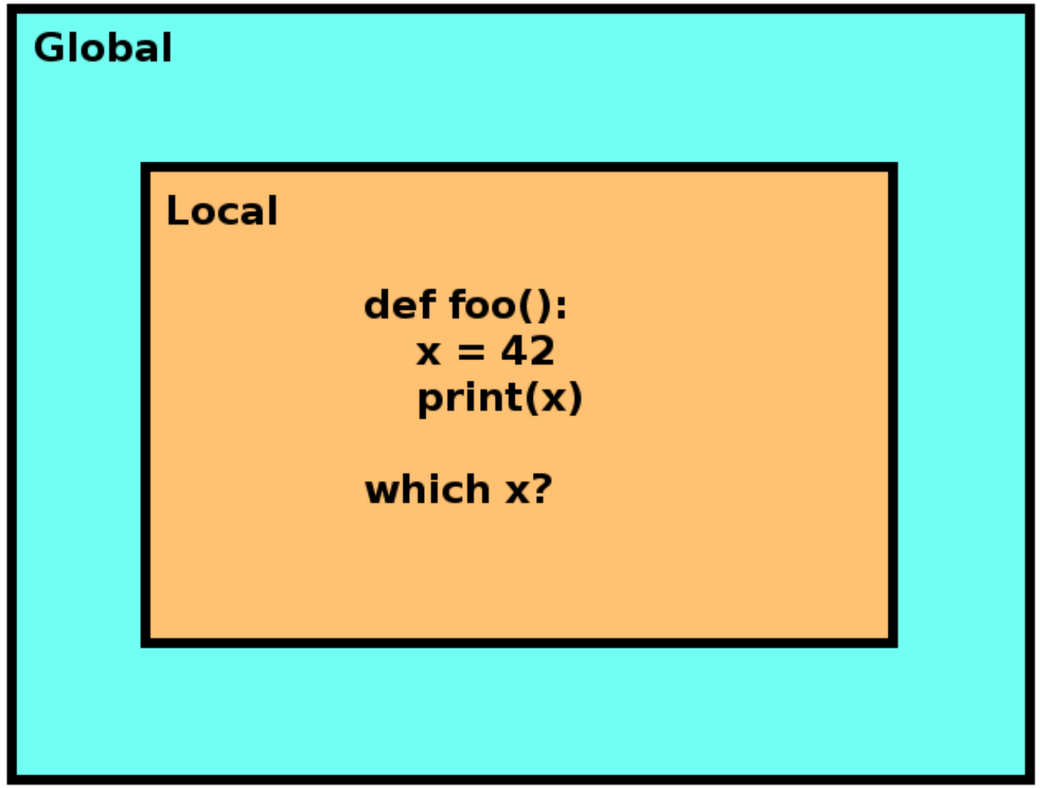
Scope



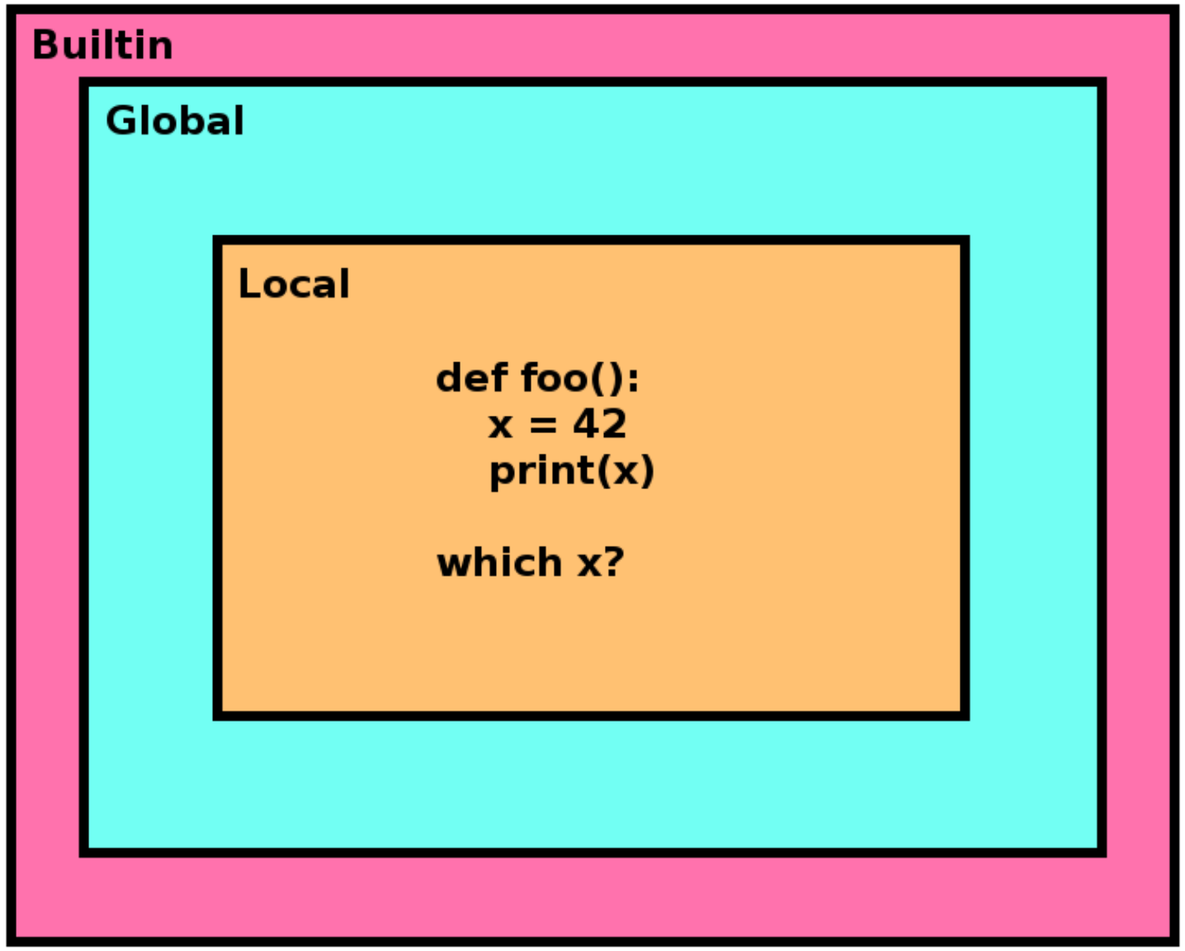
Scope



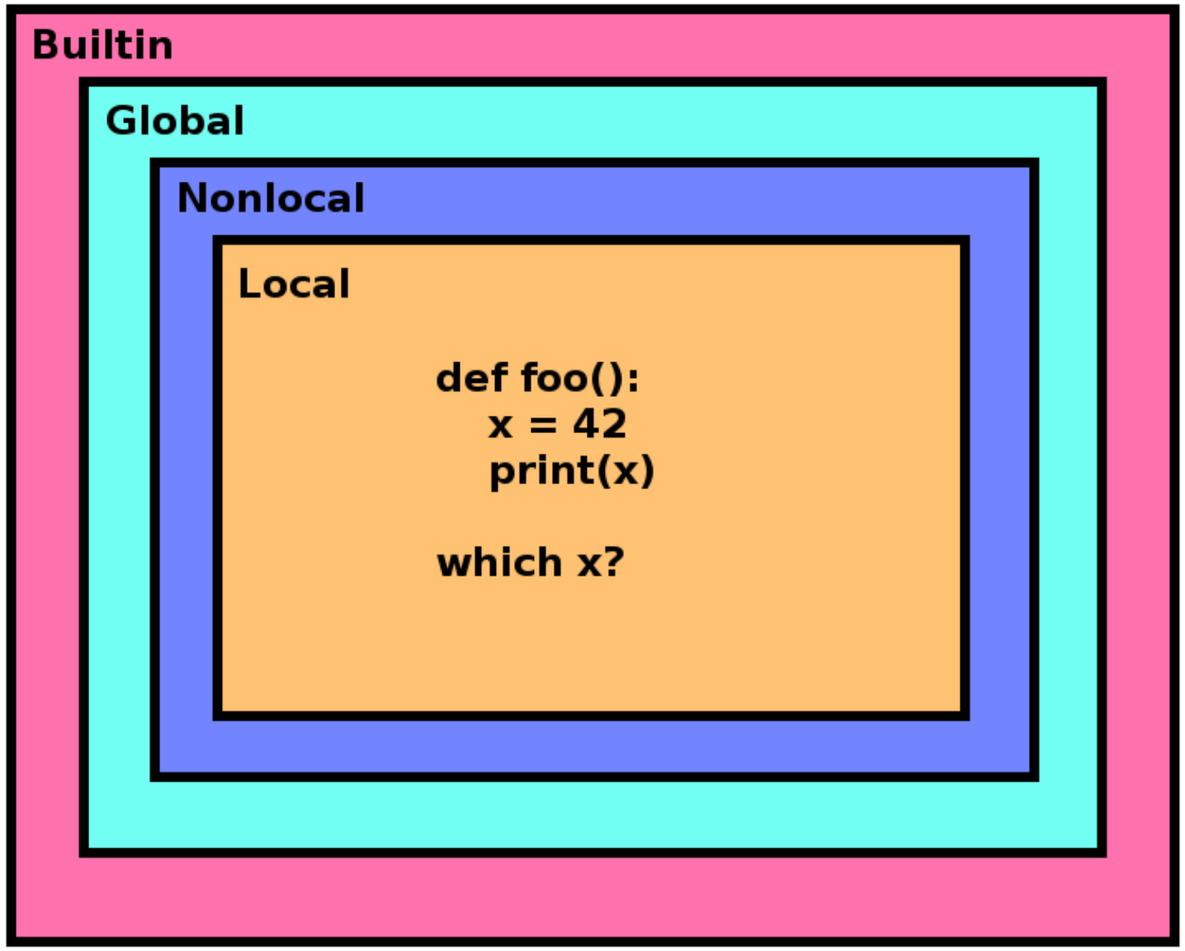
Scope



Scope



Scope



The global keyword

```
x = 7

def foo():
  x = 42
  print(x)

foo()
```

```
42
```

```
print(x)
```

```
7
```

```
x = 7

def foo():
  global x
  x = 42
  print(x)

foo()
```

```
42
```

```
print(x)
```

```
42
```

The nonlocal keyword

```
def foo():
  x = 10

  def bar():
    x = 200
    print(x)

  bar()
  print(x)

foo()
```

```
200
10
```

```
def foo():
  x = 10

  def bar():
    nonlocal x
    x = 200
    print(x)

  bar()
  print(x)

foo()
```

```
200
200
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Closures

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Attaching nonlocal variables to nested functions

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

func = foo()

func()
```

```
5
```

Closures!

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

```
<class 'tuple'>
```

```
len(func.__closure__)
```

```
1
```

```
func.__closure__[0].cell_contents
```

```
5
```

Closures and deletion

```
x = 25

def foo(value):
  def bar():
    print(value)
  return bar

my_func = foo(x)
my_func()
```

```
25
```

```
del(x)
my_func()
```

```
25
```

```
len(my_func.__closure__)
```

```
1
```

```
my_func.__closure__[0].cell_contents
```

```
25
```

Closures and overwriting

```
x = 25

def foo(value):
  def bar():
    print(value)
  return bar

x = foo(x)
x()
```

```
25
```

```
len(x.__closure__)
```

```
1
```

```
x.__closure__[0].cell_contents
```

```
25
```

Definitions - nested function

Nested function: A function defined inside another function.

```
# outer function
def parent():
  # nested function
  def child():
    pass
  return child
```

Definitions - nonlocal variables

Nonlocal variables: Variables defined in the parent function that are used by the child function.

```
def parent(arg_1, arg_2):
  # From child()'s point of view,
  # `value` and `my_dict` are nonlocal variables,
  # as are `arg_1` and `arg_2`.
  value = 22
  my_dict = {'chocolate': 'yummy'}

  def child():
    print(2 * value)
    print(my_dict['chocolate'])
    print(arg_1 + arg_2)

  return child
```

Closure: Nonlocal variables attached to a returned function.

```
def parent(arg_1, arg_2):
  value = 22
  my_dict = {'chocolate': 'yummy'}

  def child():
    print(2 * value)
    print(my_dict['chocolate'])
    print(arg_1 + arg_2)

  return child

new_function = parent(3, 4)

print([cell.cell_contents for cell in new_function.__closure__])
```

```
[3, 4, 22, {'chocolate': 'yummy'}]
```

Why does all of this matter?

Decorators use:

- Functions as objects
- Nested functions
- Nonlocal scope
- Closures

Let's practice!

WRITING FUNCTIONS IN PYTHON

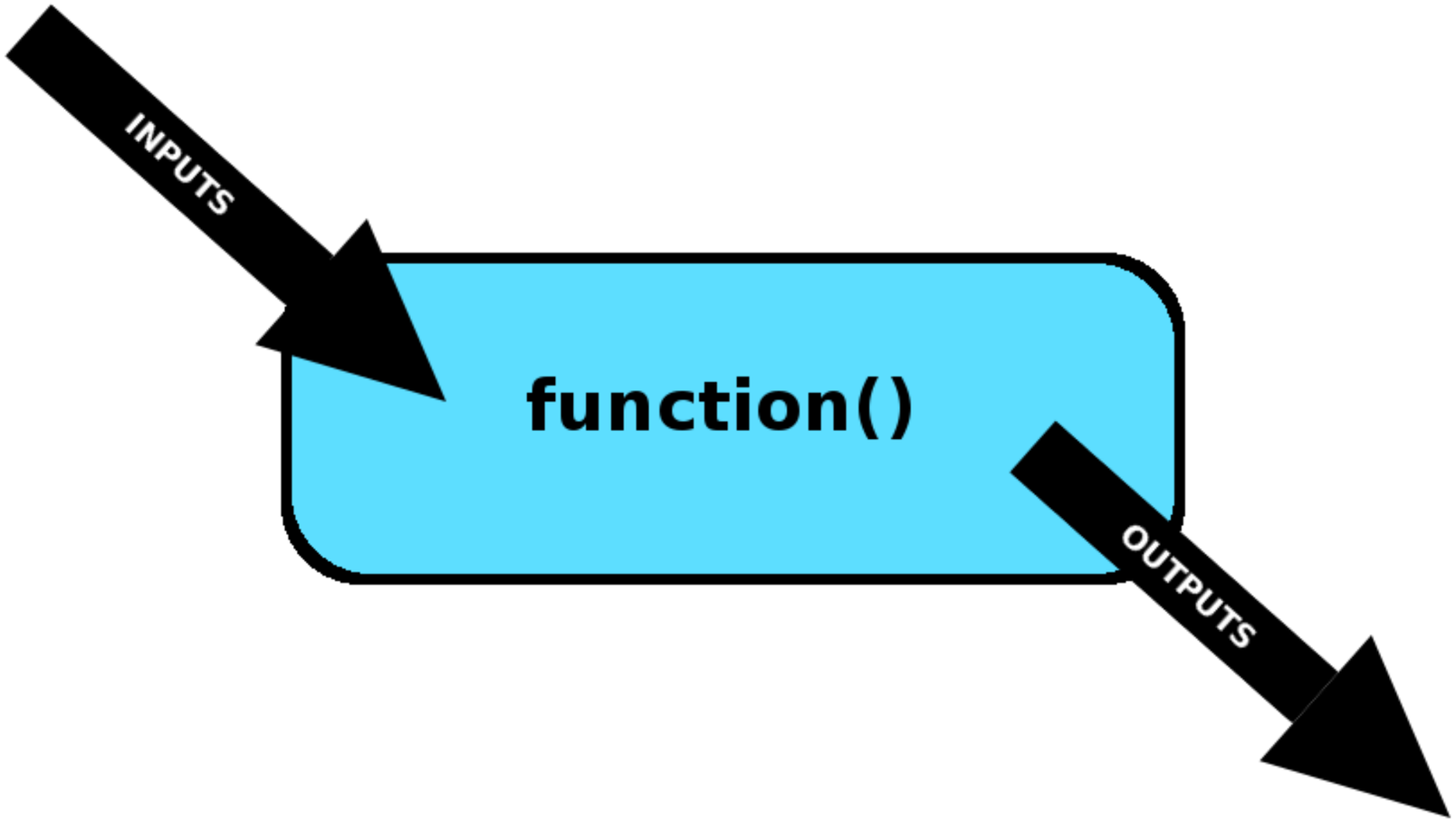
Decorators

WRITING FUNCTIONS IN PYTHON

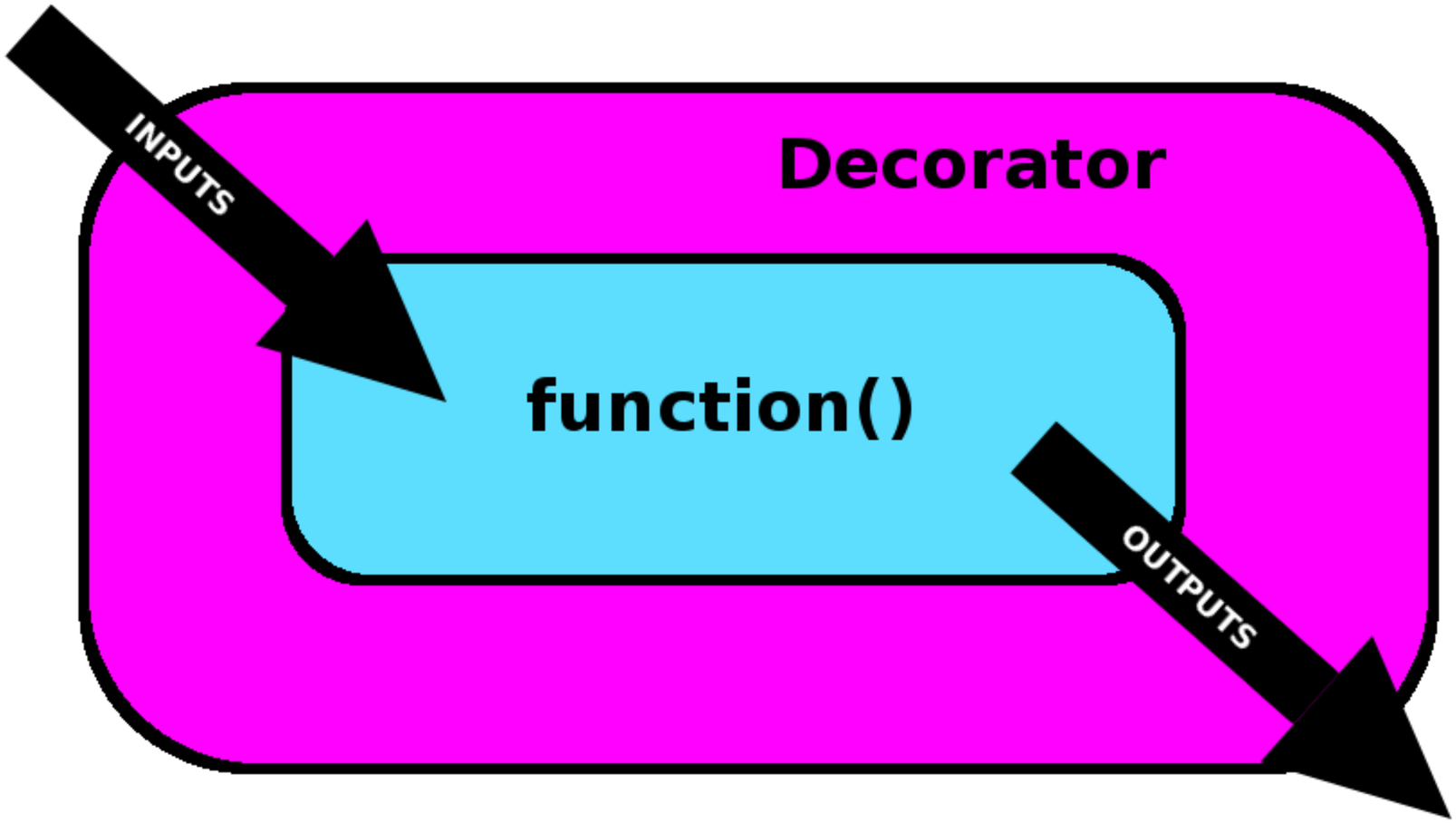
Shayne Miel

Director of Software Engineering @ American Efficient

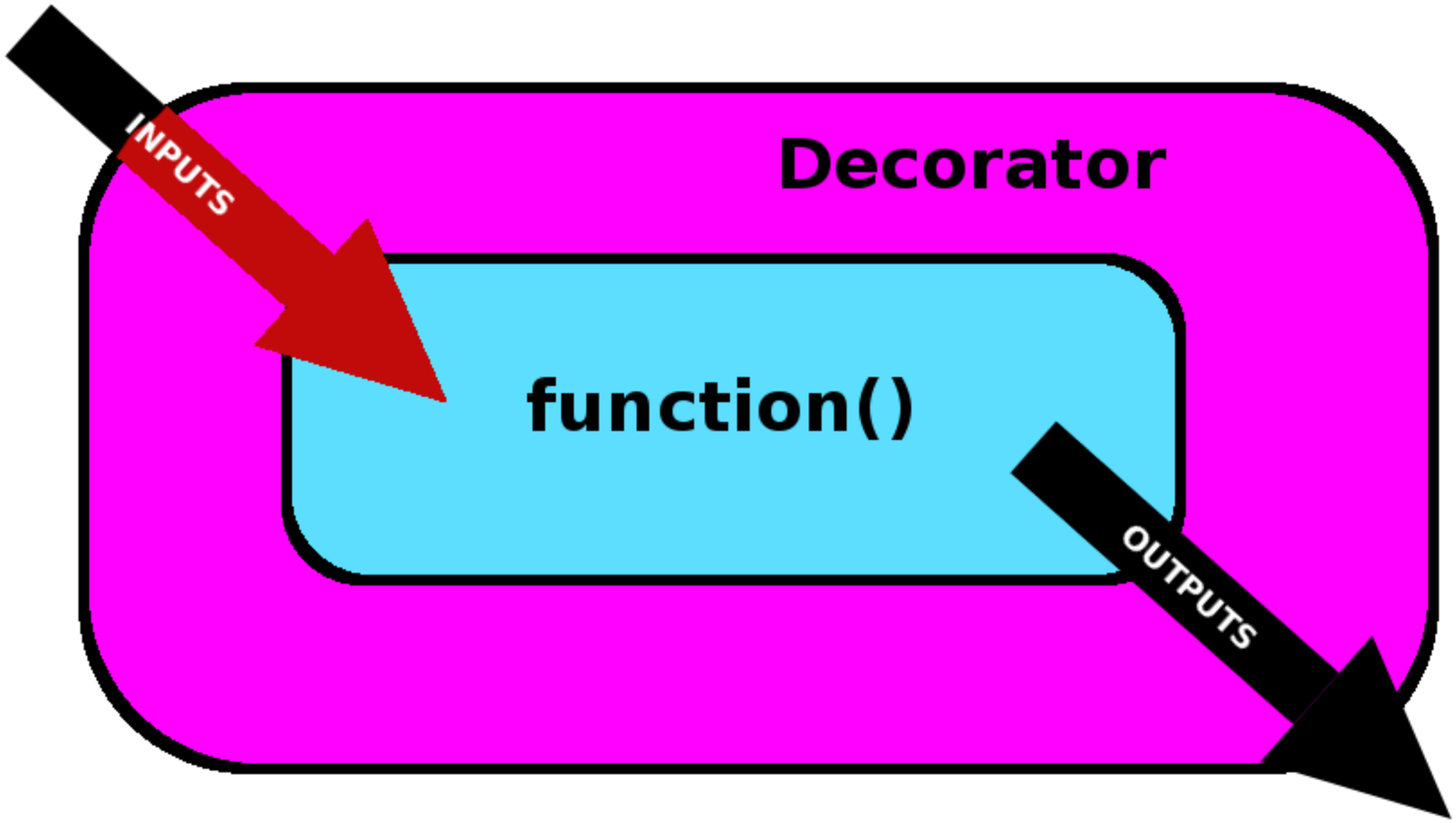
Functions



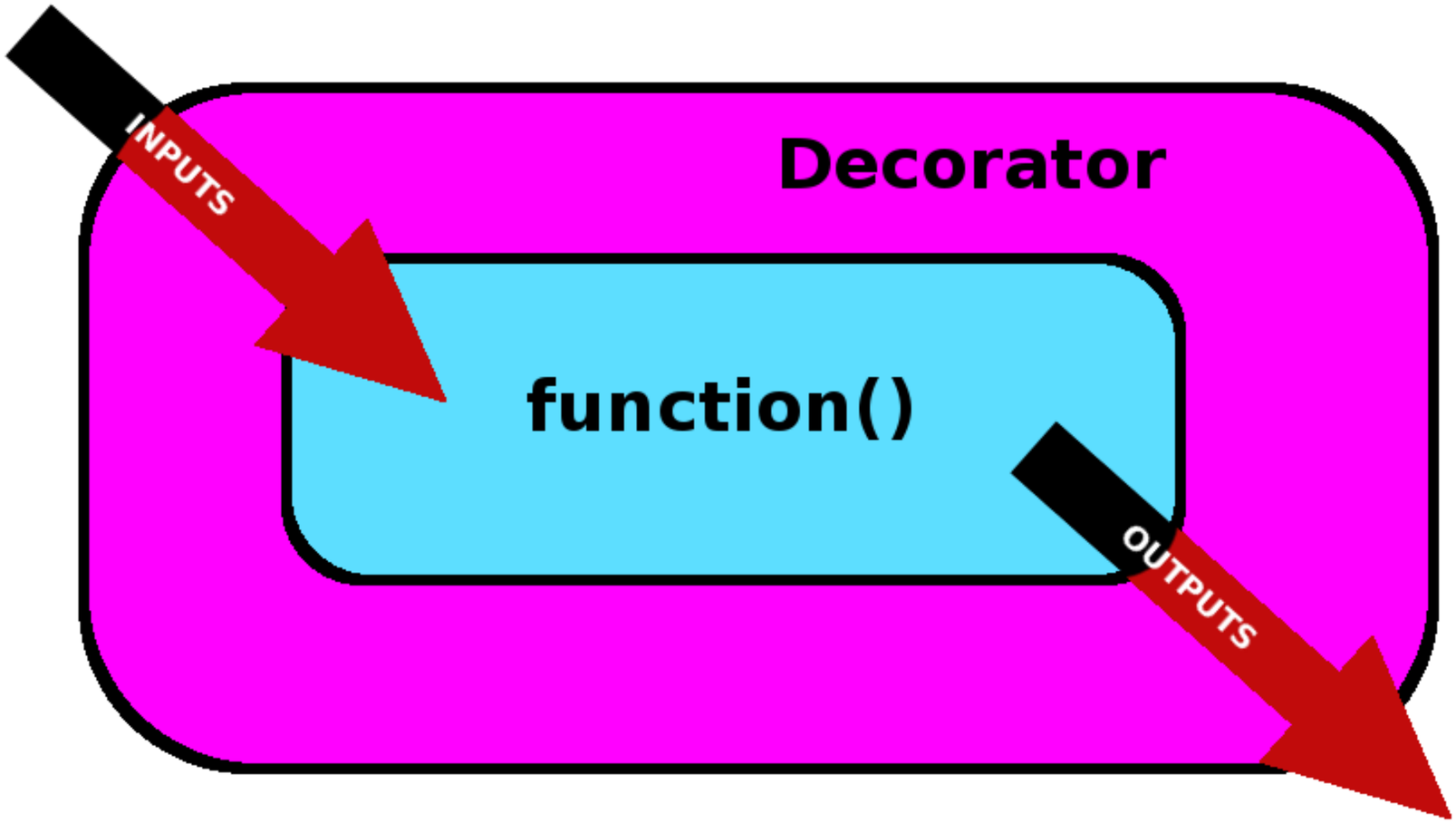
Decorators



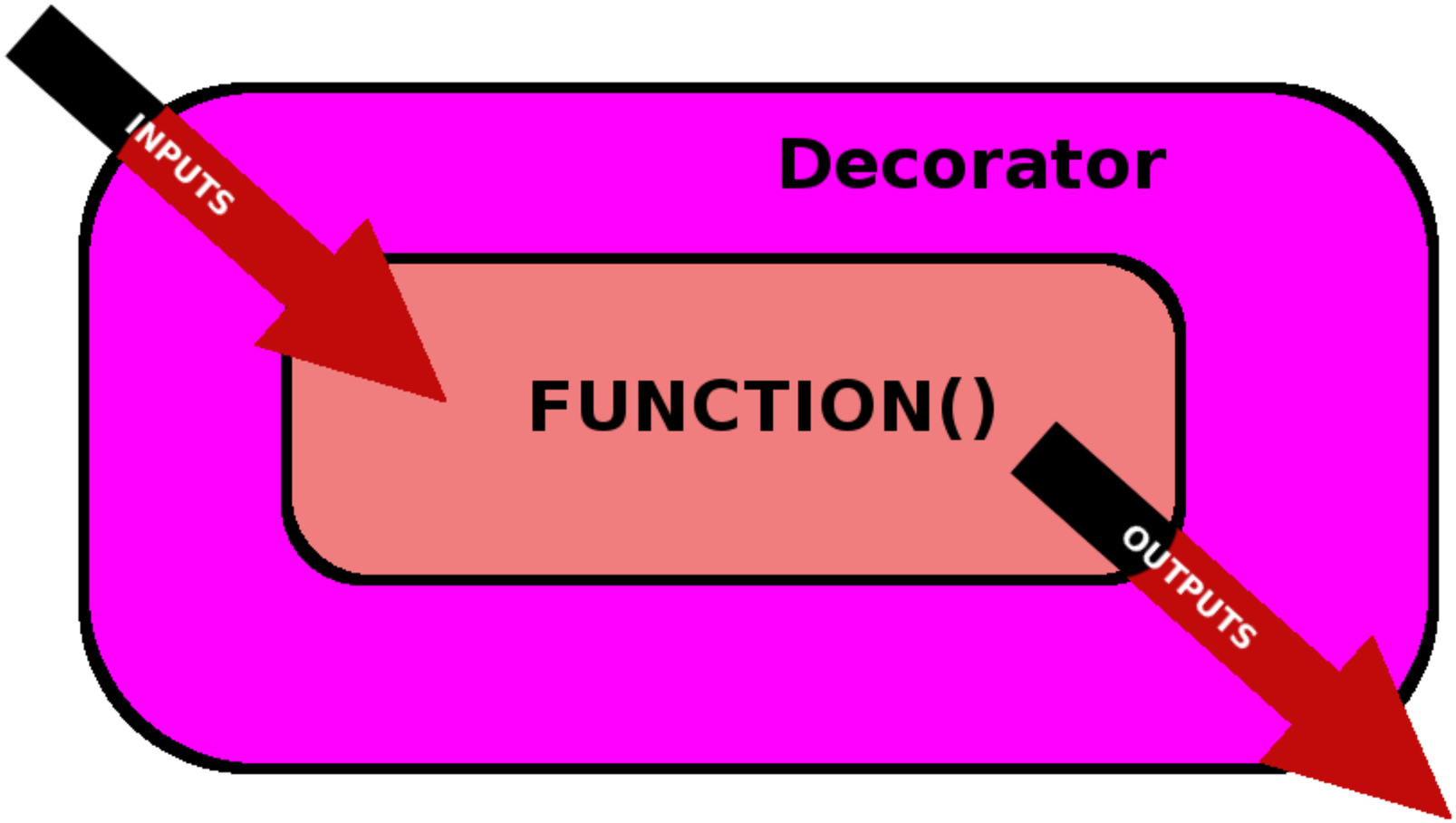
Modify inputs



Modify outputs



Modify function



What does a decorator look like?

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

multiply(1, 5)
```

```
20
```

The double_args decorator

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

def double_args(func):
  return func

new_multiply = double_args(multiply)

new_multiply(1, 5)
```

```
5
```

```
multiply(1, 5)
```

```
5
```

The double_args decorator

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

def double_args(func):
  # Define a new function that we can modify
  def wrapper(a, b):
    # For now, just call the unmodified function
    return func(a, b)
  # Return the new function
  return wrapper

new_multiply = double_args(multiply)

new_multiply(1, 5)
```

```
5
```

The double_args decorator

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

def double_args(func):
  def wrapper(a, b):
    # Call the passed in function, but double each argument
    return func(a * 2, b * 2)
  return wrapper

new_multiply = double_args(multiply)

new_multiply(1, 5)
```

```
20
```

The double_args decorator

```
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
```

```
multiply.__closure__[0].cell_contents
```

```
<function multiply at 0x7f0060c9e620>
```

Decorator syntax

```
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
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Real-world examples

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

Time a function

```
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 5.0050950050354s
```

```
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

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

```
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

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 loooong 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