Docstrings

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Software Architect @ Duo Security

A complex function

```python
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
  )
```

```python
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

```python
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

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

Google style - arguments

```python
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)

```python
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.
    """
```

```python
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

```python
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

```python
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
```

```python
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
Software Architect @ Duo Security

Don't repeat yourself (DRY)

```python
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])
```

```python
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])
```

```python
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)
plt.scatter(test_pca[:,0], test_pca[:,1])
```

The problem with repeating yourself

```python
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])
```

```python
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])
```

```python
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

```python
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])
```

```python
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])
```

```python
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

```python
def load_and_plot(path):
  """Load a dataset 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 data.columns if col != 'label'].values
  pca = PCA(n_components=2).fit_transform(X)
  plt.scatter(pca[:,0], pca[:,1])
  return X, y
```

```python
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')
```

```python
def load_and_plot(path):
  """Load a dataset 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 data.columns if col != 'label'].values

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

  return X, y
```

```python
def load_and_plot(path):
  """Load a dataset 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 data.columns if col != 'label'].values

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

  return X, y
```

```python
def load_and_plot(path):
  """Load a dataset 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 data.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
```

```python
def load_and_plot(path):
    """Load a dataset 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 data.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

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

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

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

```python
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

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler (1999)

Let's practice!

WRITING FUNCTIONS IN PYTHON

Pass by assignment

WRITING FUNCTIONS IN PYTHON
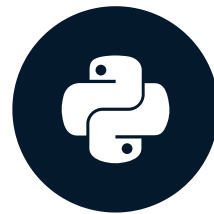
Shayne Miel
Software Architect @ Duo Security

A surprising example

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

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

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

```
3
```

Digging deeper

Digging deeper

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

Digging deeper

```python
a = [1, 2, 3]
b = a
```
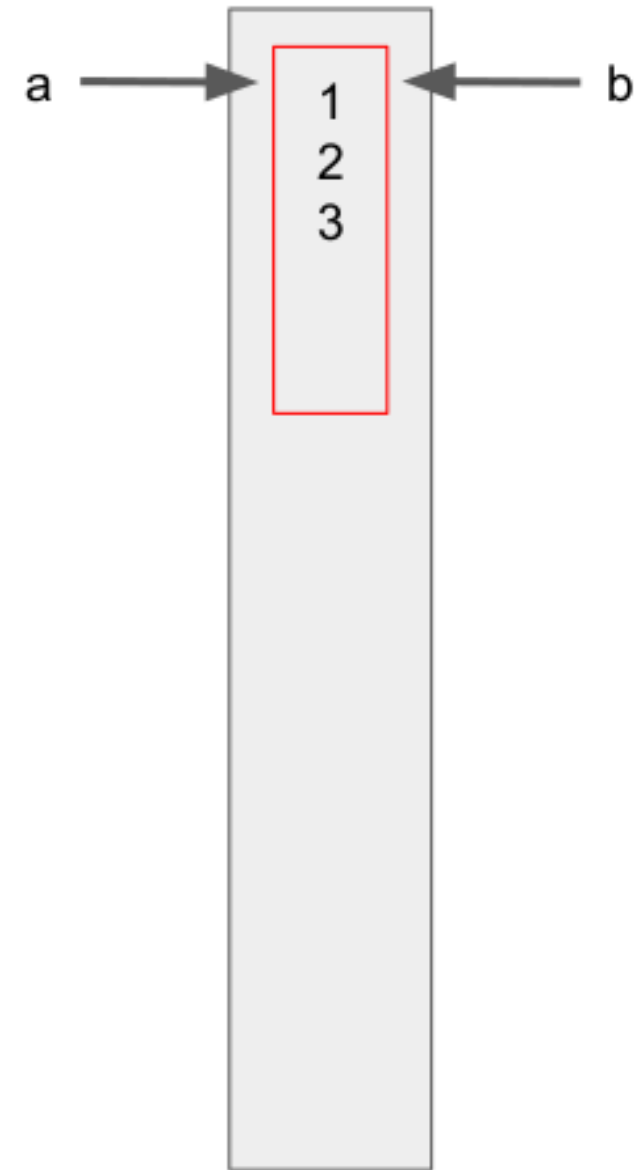
Digging deeper

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

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

Digging deeper

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

```
[1, 2, 3, 4]
```
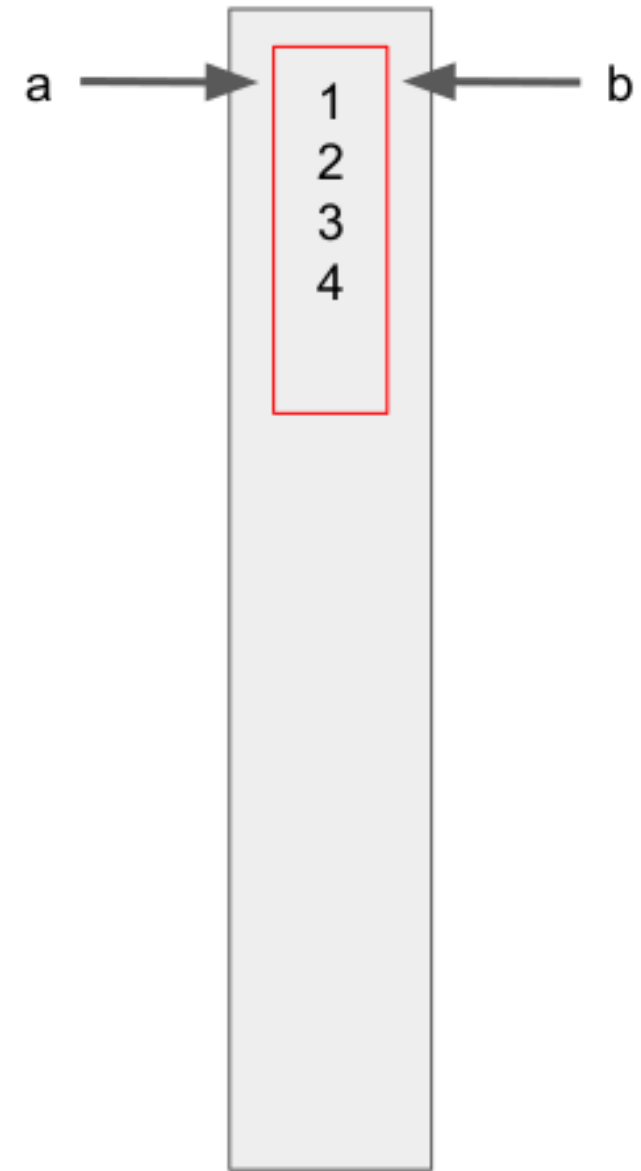
```python
b.append(5)
print(a)
```
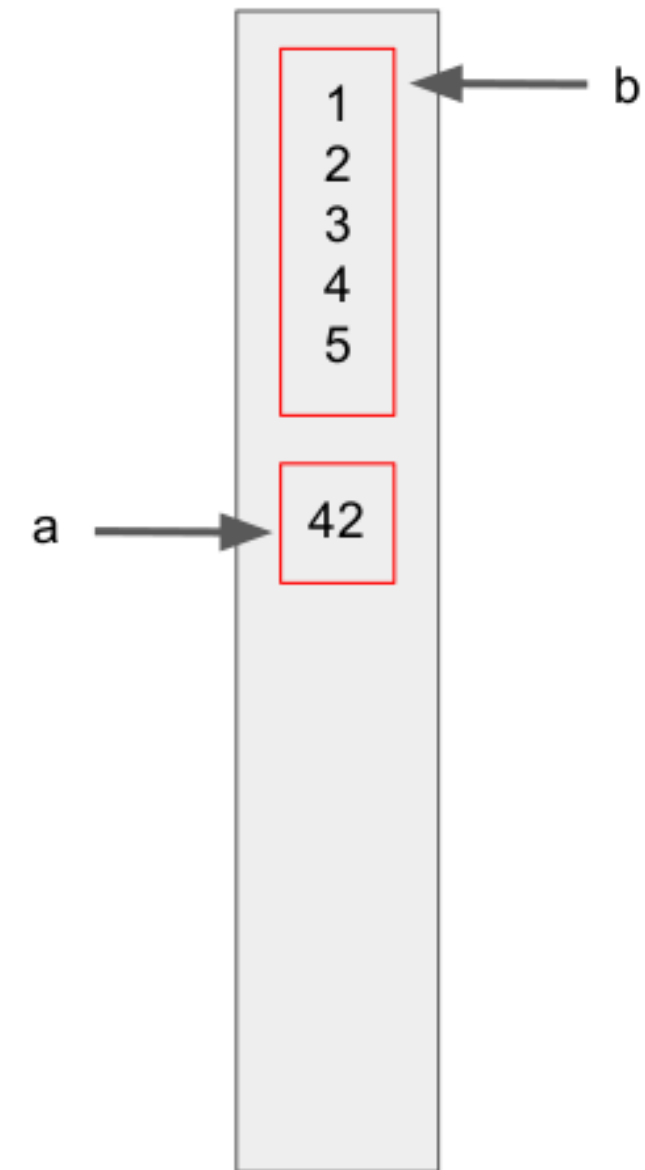
```
[1, 2, 3, 4, 5]
```

Digging deeper

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

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

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

```
[1, 2, 3, 4, 5]
```
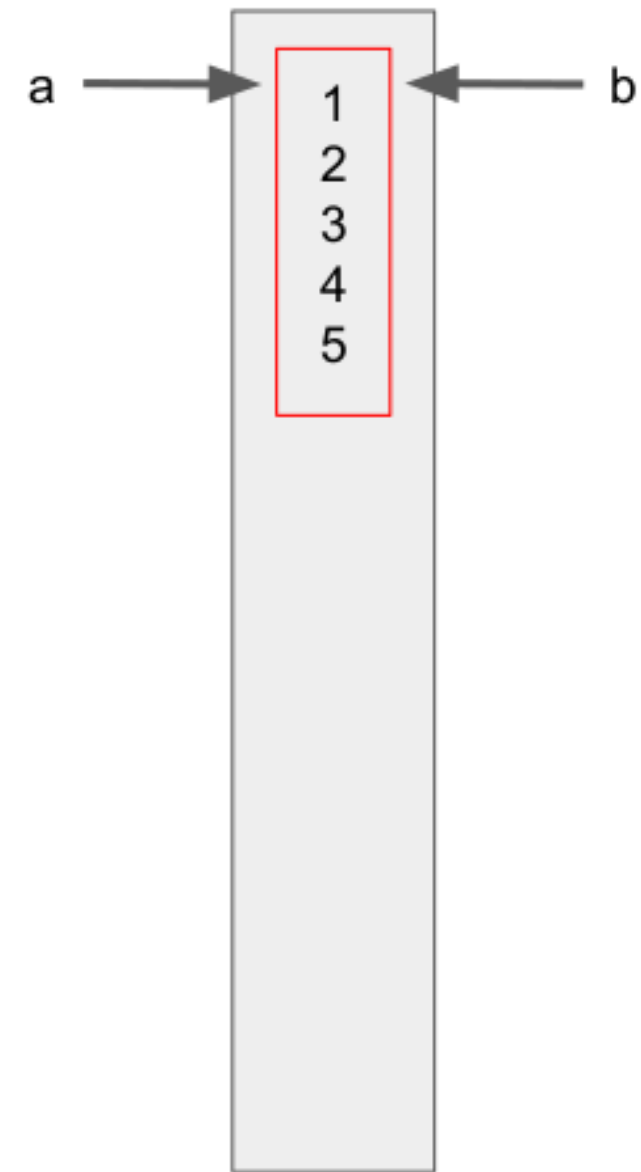
```python
a = 42
```
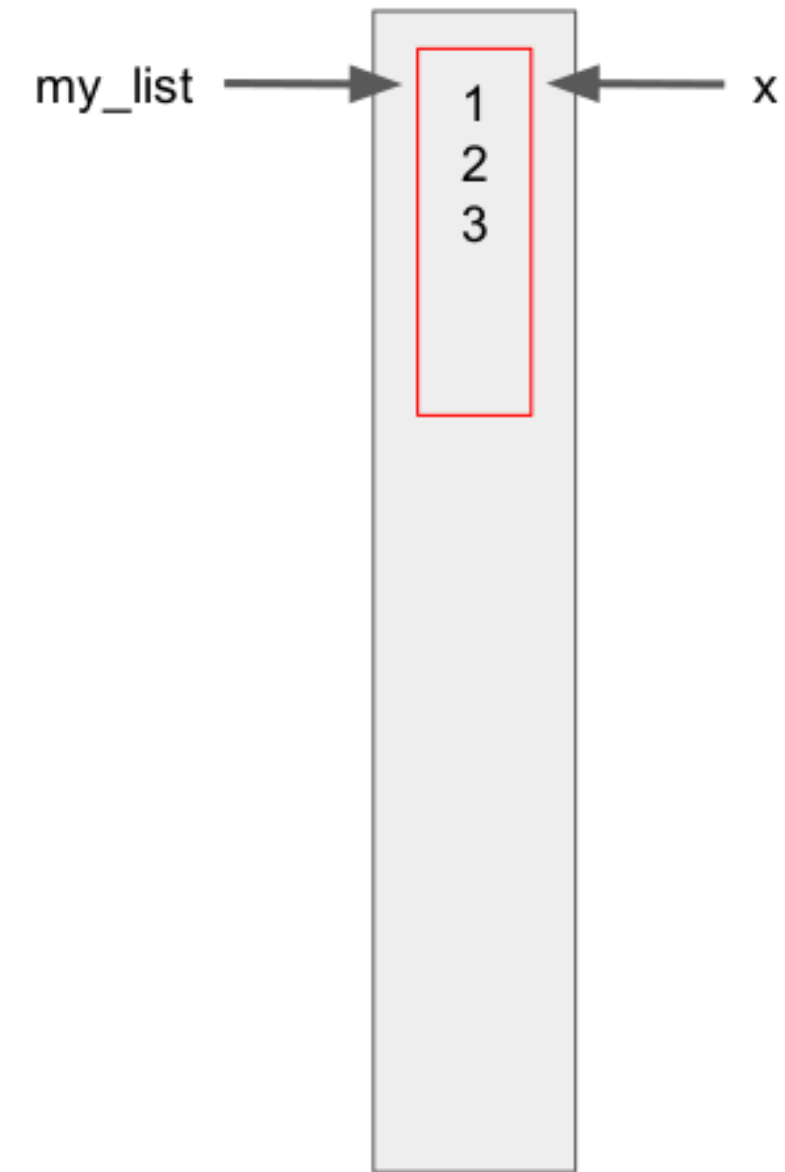
Pass by assignment

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

Pass by assignment

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

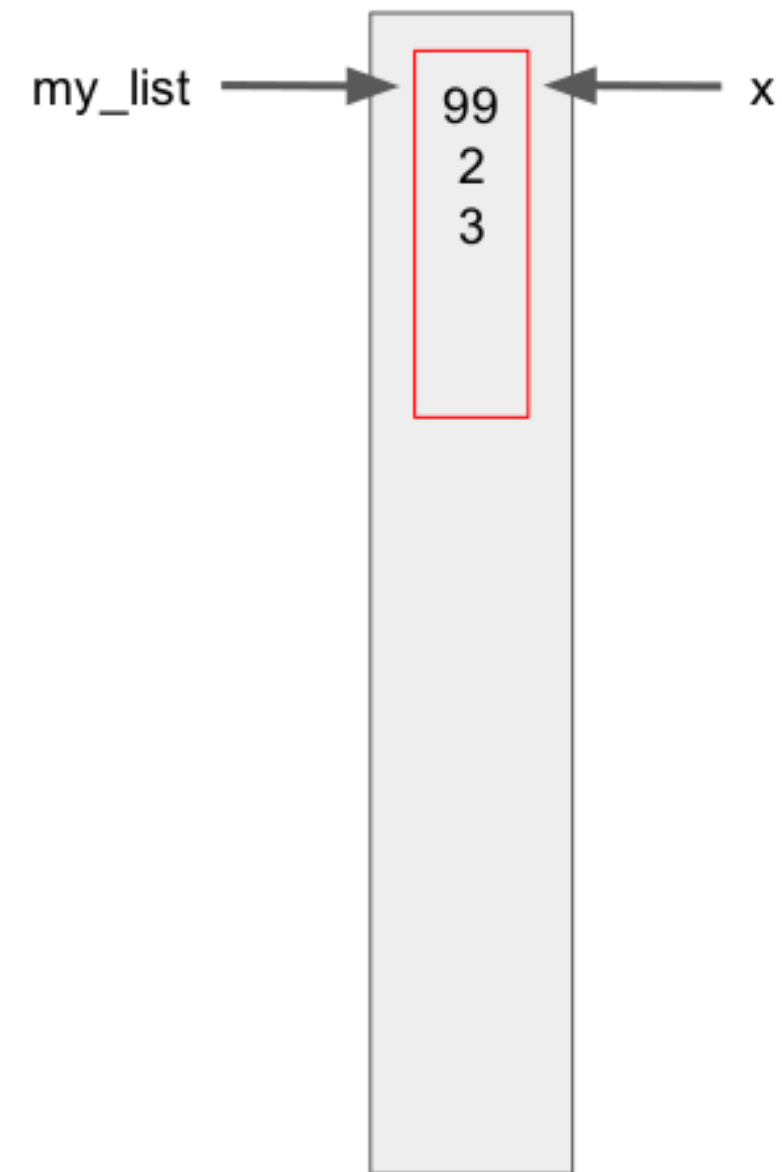my_list →

1
2
3

Pass by assignment

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

Pass by assignment

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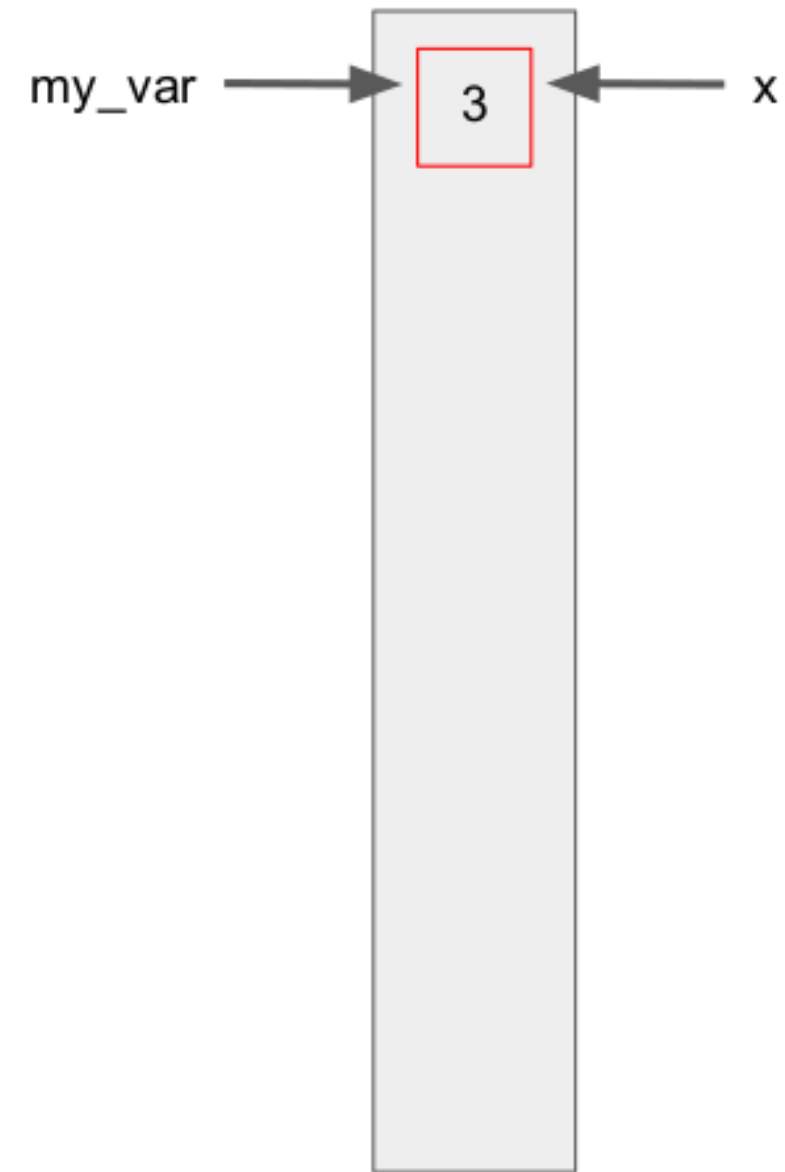
```
[99, 2, 3]
```

Pass by assignment

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

my_var = 3
```
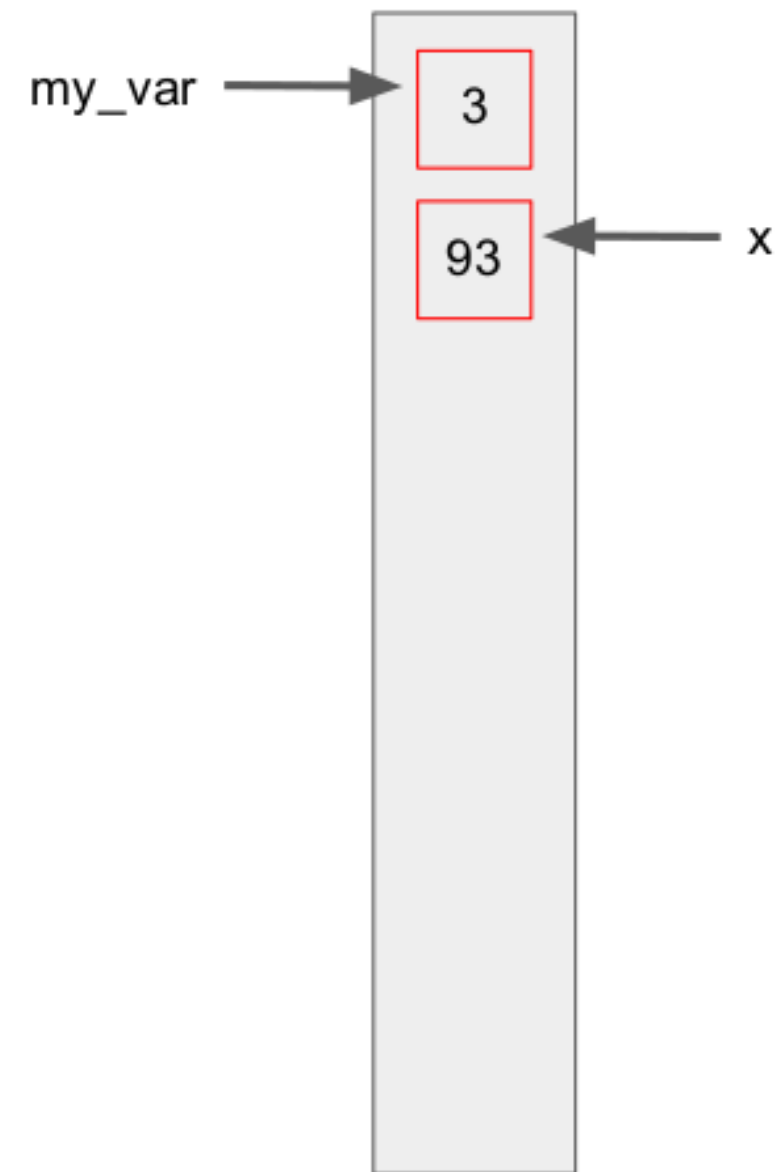
Pass by assignment

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

my_var = 3
bar(my_var)
```

Pass by assignment

```python
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!

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

```
[1]
```

```python
foo()
```

```
[1, 1]
```

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

```
[1]
```

```python
foo()
```

```
[1]
```

Let's practice!

WRITING FUNCTIONS IN PYTHON