Docstrings

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

datacamp

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
  )
```

3. A complex function with a docstring
With a docstring though, it is much easier to tell what the expected inputs and outputs should be, as well as what the function does. This makes it easier for you and other engineers to use your code in the future.

```
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
  )
```

4. Anatomy of a docstring

A docstring is a string written as the first line of a function. Because docstrings usually span multiple lines, they are enclosed in triple quotes, Python's way of writing multi-line strings. Every docstring has some (although usually not all) of these five key pieces of information: what the function does, what the arguments are, what the return value or values should be, info about any errors raised, and anything else you'd like to say about the function.

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

6. Google Style - description
In Google style, the docstring starts with a concise description of what the function does. This should be in imperative language. For instance: "Split the data frame and stack the columns" instead of "This function will split the data frame and stack the columns".

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.
  """
```

7. Google style - arguments

Next comes the "Args" section where you list each argument name, followed by its expected type in parentheses, and then what its role is in the function. If you need extra space, you can break to the next line and indent as I've done here. If an argument has a default value, mark it as "optional" when describing the type. If the function does not take any parameters, feel free to leave this section out.

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.
  """
```

8. Google style - return value(s)

The next section is the "Returns" section, where you list the expected type or types of what gets returned. You can also provide some comment about what gets returned, but often the name of the function and the description will make this clear. Additional lines should not be indented.

9. Google-style - errors raised and extra notes

Finally, if your function intentionally raises any errors, you should add a "Raises" section. You can also include any additional notes or examples of usage in free form text at the end.

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.
  """
```

10. Numpydoc
The Numpydoc format is very similar and is the most common format in the scientific Python community. Personally, I think it looks better than the Google style. It takes up more vertical space though, so this course will either use Google-style or leave out the docstrings entirely to keep the examples compact and legible.

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
```

11. Retrieving docstrings
Every function in Python comes with a __doc__ attribute that holds this information. Notice that the __doc__ attribute contains the raw docstring, including any tabs or spaces that were added to make the words line up visually. To get a cleaner version, with those leading spaces removed, you can use the getdoc() function from the inspect module. The inspect module contains a lot of useful methods for gathering information about functions.

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

2. Don't repeat yourself (DRY)
When you are writing code to look for answers to a research question, it is totally normal to copy and paste a bit of code, tweak it slightly, and re-run it. However, this kind of repeated code can lead to real problems. In this code snippet, I load my train, validation, and test data, and plot the first two principal components of each data set. I wrote the code for the train data set, then copied it and pasted it into the next two blocks, updating the paths and the variable names.

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])
```

3. The problem with repeating yourself
But one of the problems with copying and pasting is that it is easy to accidentally introduce errors that are hard to spot. If you'll notice in the last block, I accidentally took the principal components of the train data instead of the test data. Yikes!

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])
```

4. Another problem with repeating yourself
Another problem with repeated code is that if you want to change something, you often have to do it in multiple places. For instance, if we realized that our CSVs used the column name "label" instead of "labels", we would have to change our code in six places. Repeated code like this is a good sign that you should write a function. So let's do that.

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
```

5. Use functions to avoid repetition

Wrapping the repeated logic in a function and then calling that function several times makes it much easier to avoid the kind of errors introduced by copying and pasting. And if you ever need to change the column "label" back to "labels", or you want to swap out PCA for some other dimensionality reduction technique, you only have to do it in one or two places.

```
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
```

6. Problem: it does multiple things
However, there is still a big problem with this function.

7. Problem: it does multiple things
First, it loads the data.
8. Problem: it does multiple things
Then it plots the data.

9. Problem: it does multiple things
And then it returns the loaded data. **This function violates another software engineering principle: Do One Thing. Every function should have a single responsibility.** Let's look at how we could split this one up.

```
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])
```

10. Do One Thing
Instead of one big function, we could have a more nimble function that just loads the data and a second one for plotting. We get several advantages from splitting the load_and_plot() function into two smaller functions. First of all, our code has become more flexible. Imagine that later on in your script, you just want to load the data and not plot it. That's easy now with the load_data() function. Likewise, if you wanted to do some transformation to the data before plotting, you can do the transformation and then call the plot_data() function. We have decoupled the loading functionality from the plotting functionality.

Advantages of doing one thing

The code becomes:

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

11. Advantages of doing one thing
The code will also be easier for other developers to understand, and it will be more pleasant to test and debug. Finally, if you ever need to update your code, functions that each have a single responsibility make it easier to predict how changes in one place will affect the rest of the code.

Code smells and refactoring

12. Code smells and refactoring
Repeated code and functions that do more than one thing are examples of "code smells", which are indications that you may need to refactor.
Refactoring is the process of improving code by changing it a little bit at a time. This process is well described in Martin Fowler's book, "Refactoring", which is a good read for any aspiring software engineer.

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)
```

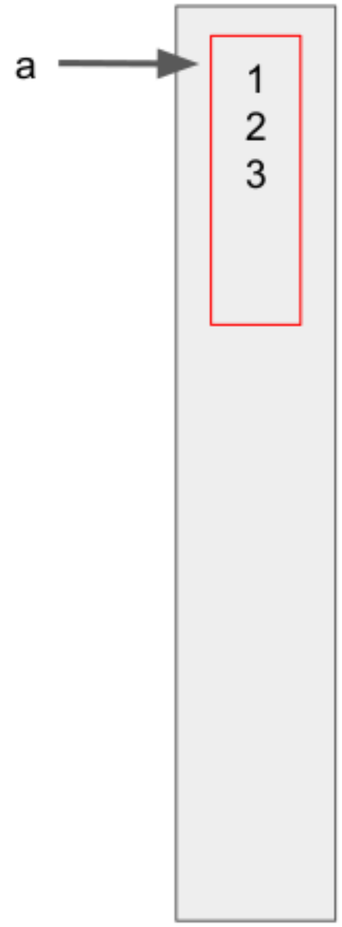
```
3
```

2. A surprising example
Let's say we have a function foo() that takes a list and sets the first value of the list to 99. Then we set "my_list" to the value [1, 2, 3] and pass it to foo(). What do you expect the value of "my_list" to be after calling foo()? If you said "[99, 2, 3]", then you are right. Lists in Python are mutable objects, meaning that they can be changed. Now let's say we have another function bar() that takes an argument and adds ninety to it. Then we assign the value 3 to the variable "my_var" and call bar() with "my_var" as the argument. What do you expect the value of "my_var" to be after we've called bar()? If you said "3", you're right. In Python, integers are immutable, meaning they can't be changed.

Digging deeper

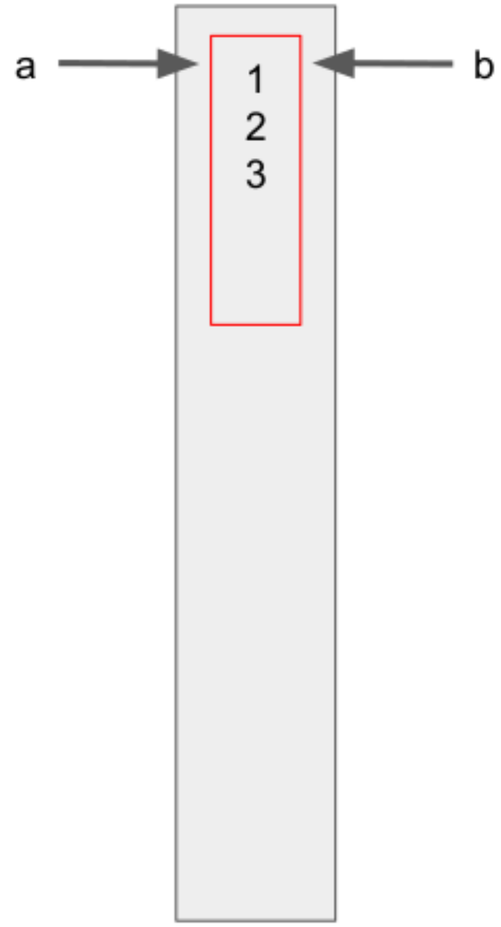
Digging deeper

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



Digging deeper

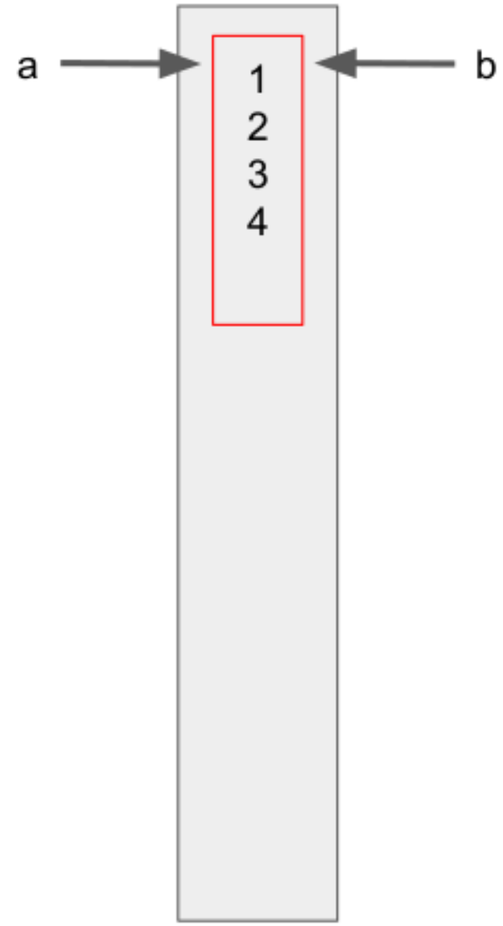
```
a = [1, 2, 3]
b = a
```



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)
```

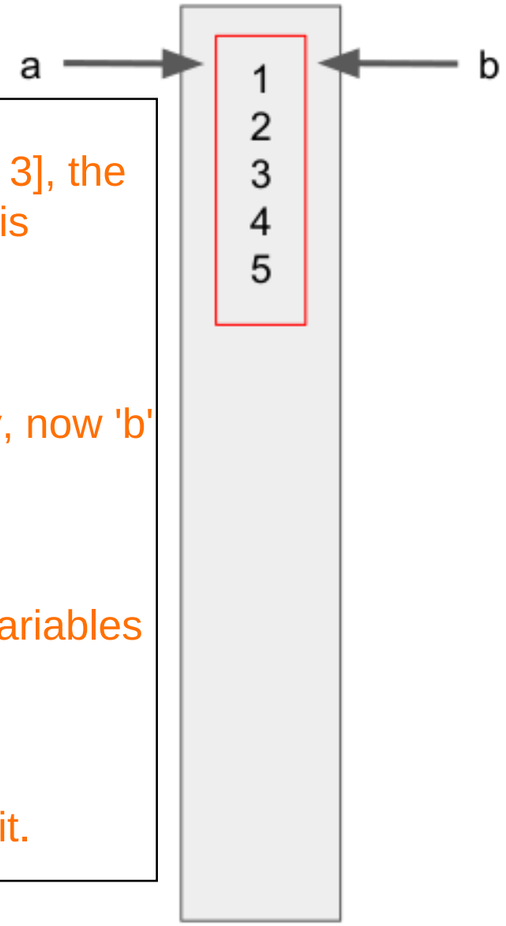
```
[1, 2, 3, 4, 5]
```

4. Digging deeper
When we set the variable "a" equal to the list [1, 2, 3], the Python interpreter says, "Okay, now 'a' points to this location in memory."

5. Digging deeper
Then if we type "b = a", the interpreter says, "Okay, now 'b' points to whatever 'a' is pointing to."

6. Digging deeper
So if we were to append 4 to the end of "a", both variables get it because there is only one list.

7. Digging deeper
Likewise, if we append 5 to "b", both variables get it.



Digging deeper

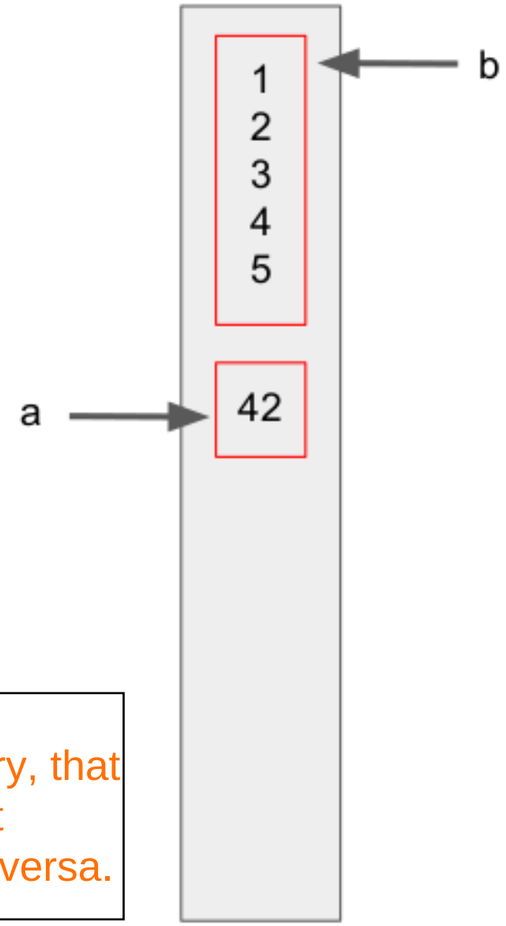
```
a = [1, 2, 3]
b = a
a.append(4)
print(b)
```

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

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

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

```
a = 42
```



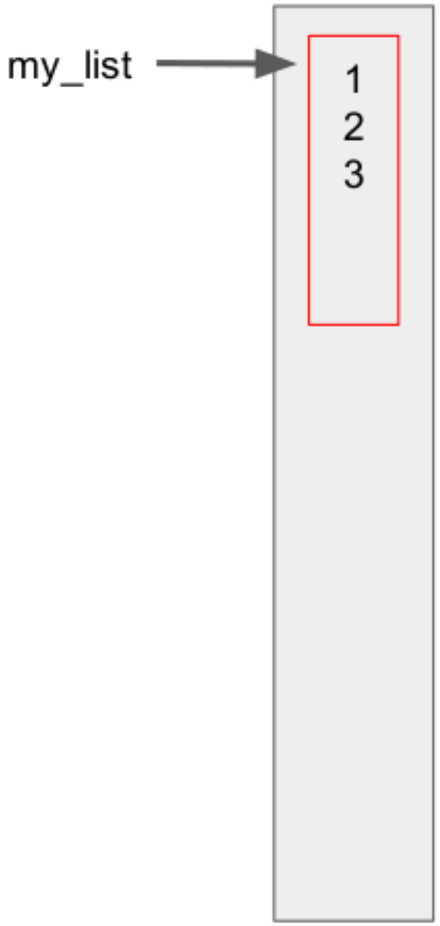
8. Digging deeper
However, if we assign "a" to a different object in memory, that does not change where "b" is pointing. Now, things that happen to "a" are no longer happening to "b", and vice versa.

Pass by assignment

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

Pass by assignment

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



9. Pass by assignment
How does this relate to the example functions we saw earlier?

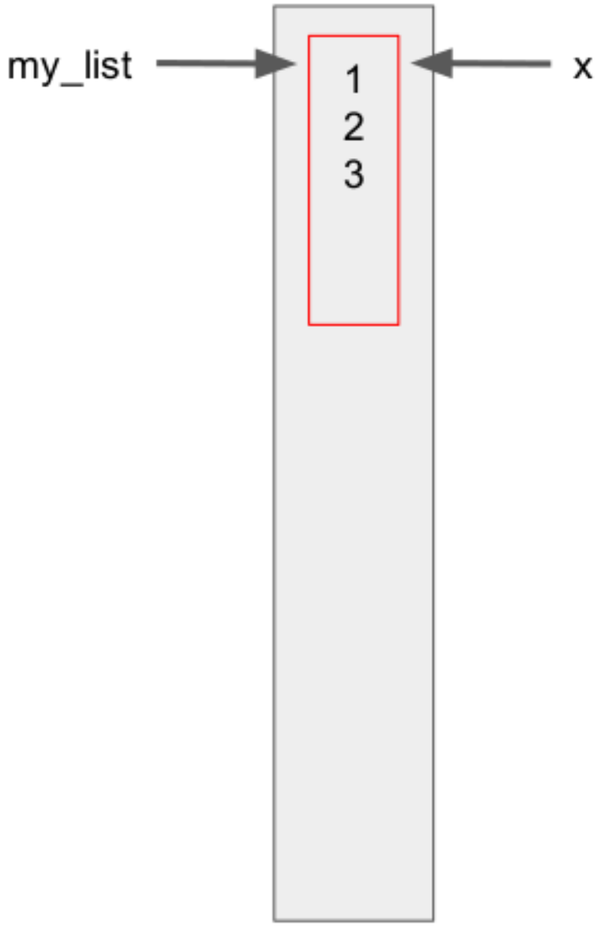
10. Pass by assignment
When we assign a list to the variable "my_list", it sets up a location in memory for it.

Pass by assignment

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

11. Pass by assignment
Then, when we pass "my_list" to the function foo(), the parameter "x" gets assigned to that same location.

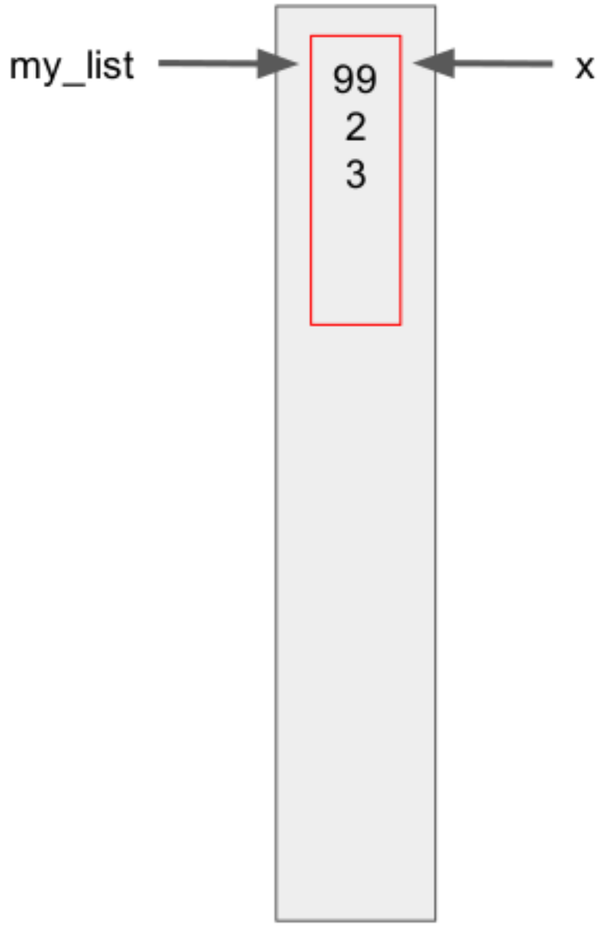
12. Pass by assignment
So when the function modifies the thing that "x" points to, it is also modifying the thing that "my_list" points to.



Pass by assignment

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

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



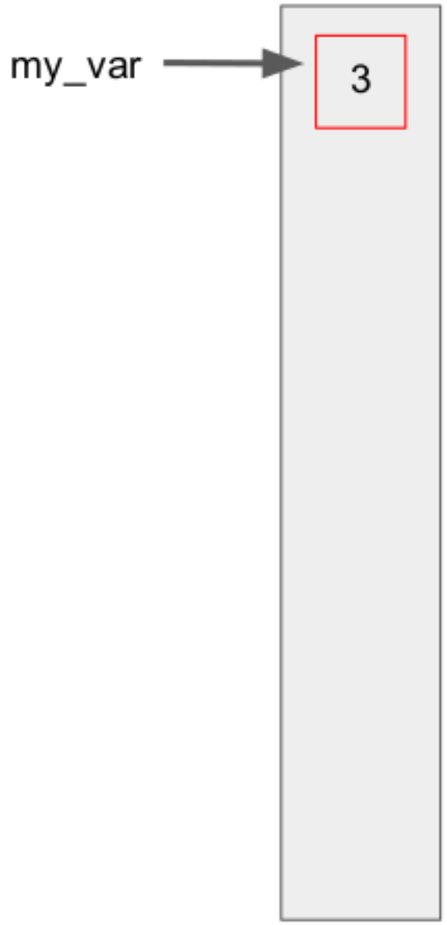
13. Pass by assignment
In the other example, we created a variable "my_var" and assigned it the value 3.

14. Pass by assignment
Then we passed it to the function bar(), which caused the argument "x" to point to the same place "my_var" is pointing.

Pass by assignment

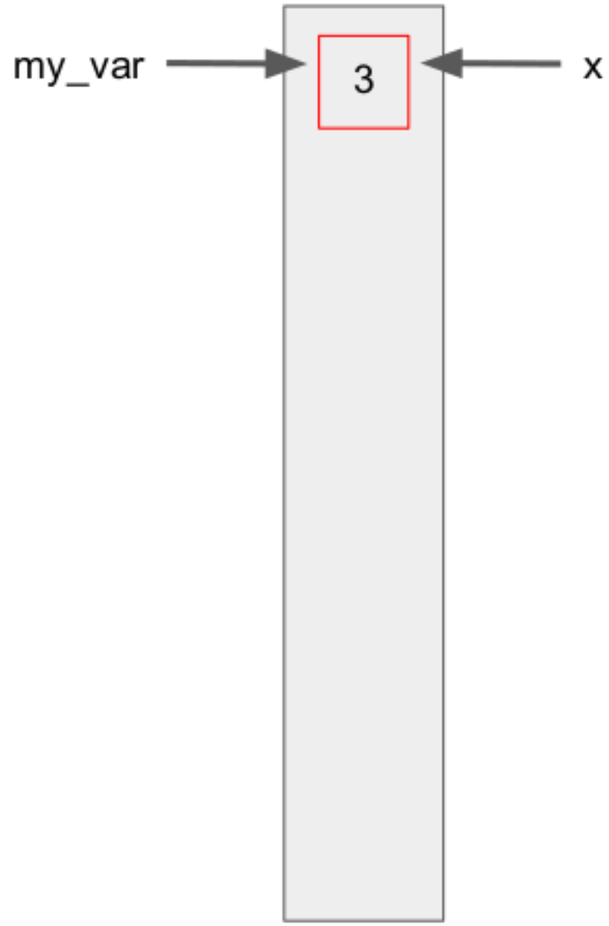
```
def bar(x):
  x = x + 90
my_var = 3
```

15. Pass by assignment
But the bar() function assigns "x" to a new value, so the "my_var" variable isn't touched. In fact, there is no way in Python to have changed "x" or "my_var" directly, because integers are immutable variables.



Pass by assignment

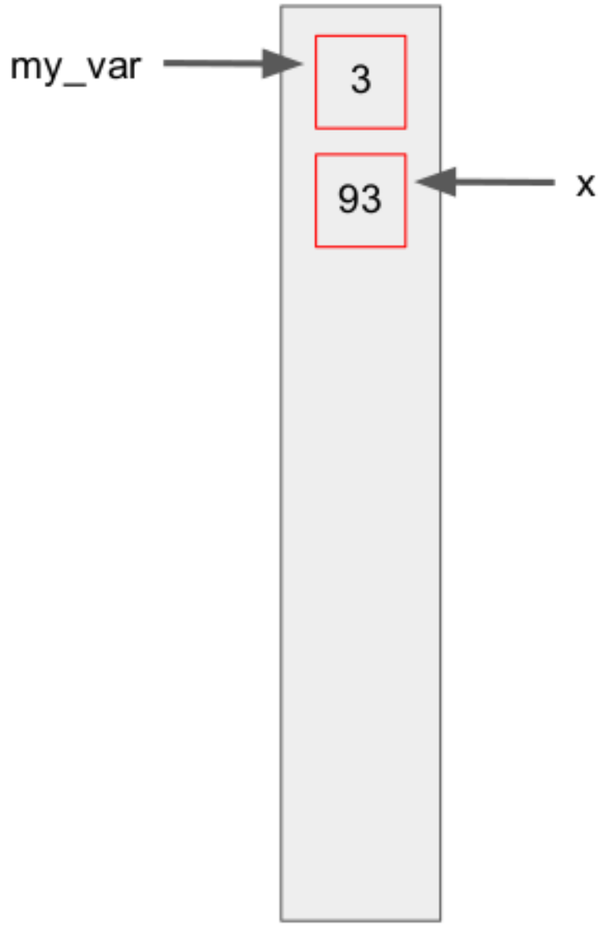
```
def bar(x):
  x = x + 90
my_var = 3
bar(my_var)
```



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

16. Immutable or Mutable?
There are only a few immutable data types in Python because almost everything is represented as an object. The only way to tell if something is mutable is to see if there is a function or method that will change the object without assigning it to a new variable.

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]
```

17. Mutable default arguments are dangerous!
Finally, here is a thing that can get you into trouble. foo() is a function that appends the value 1 to the end of a list. But, whoever wrote this function gave the argument an empty list as a default value. When we call foo() the first time, we get what you would expect, a list with one entry. But, when we call foo() again, the default value has already been modified! If you really want a mutable variable as a default value, consider defaulting to None and setting the argument in the function.

Let's practice!

WRITING FUNCTIONS IN PYTHON