Annotated follow-along guide_Data structures in Python

January 7, 2024

1 Annotated follow-along guide: Data structures in Python

This notebook contains the code used in the instructional videos from Module 4: Data structures in Python.

1.1 Introduction

This follow-along guide is an annotated Jupyter Notebook organized to match the content from each module. It contains the same code shown in the videos for the module. In addition to content that is identical to what is covered in the videos, you'll find additional information throughout the guide to explain the purpose of each concept covered, why the code is written in a certain way, and tips for running the code.

As you watch each of the following videos, an in-video message will appear to advise you that the video you are viewing contains coding instruction and examples. The in-video message will direct you to the relevant section in the notebook for the specific video you are viewing. Follow along in the notebook as the instructor discusses the code.

To skip directly to the code for a particular video, use the following links:

1. Introduction to lists

[1]:
```
# Assign a list using brackets, with elements separated by commas.
x = ["Now", "we", "are", "cooking", "with", 7, "ingredients"]
```

```
# Print element at index 3.
print(x[3])
```

```
cooking
```

[2]:
```
# Trying to access an index not in list will result in IndexError.
print(x[7])
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-2-7595372b0a8f> in <module>
  1 # Trying to access an index not in list will result in IndexError.
----> 2 print(x[7])


IndexError: list index out of range
```

[ ]:
```
# Access part of a list by slicing.
x[1:3]
```

[ ]:
```
# Omitting the first value of the slice implies a value of 0.
x[:2]
```

[ ]:
```
# Omitting the last value of the slice implies a value of len(list).
x[2:]
```

[ ]:
```
# Check the data type of an object using type() function.
type(x)
```

[ ]:
```
# The `in` keyword lets you check if a value is contained in the list.
x = ["Now", "we", "are", "cooking", "with", 7, "ingredients"]
"This" in x
```

2. Modify the contents of a list

[ ]:
```
# The append() method adds an element to the end of a list.
fruits = ['Pineapple', 'Banana', 'Apple', 'Melon']
fruits.append('Kiwi')
print(fruits)
```

[ ]:
```
# The insert() method adds an element to a list at the specified index.
fruits.insert(1, 'Orange')
print(fruits)
```

[ ]:
```
# The insert() method adds an element to a list at the specified index.
fruits.insert(0, 'Mango')
print(fruits)
```

[ ]:
```
# The remove() method deletes the first occurrence of an element in a list.
fruits.remove('Banana')
print(fruits)
```

[ ]:
```
# Trying to remove an element that doesn't exist results in an error.
fruits.remove('Strawberry')
print(fruits)
```

[ ]:
```
# The pop() method removes the element at a given index and returns it.
# If no index is given, it removes and returns the last element.
fruits.pop(2)
print(fruits)
```

[ ]:
```
# Reassign the element at a given index with a new value.
fruits[1] = 'Mango'
```

[ ]:
```
print(fruits)
```

[3]:
```
# Strings are immutable because you need to reassign them to modify them.
power = '1.21'
power = power + ' gigawatts'
print(power)
```

```
1.21 gigawatts
```

[4]:
```
# You cannot reassign a specific character within a string.
power[0] = '2'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-4-7834e7b6b0fa> in <module>
  1 # You cannot reassign a specific character within a string.
----> 2 power[0] = '2'
```

```
TypeError: 'str' object does not support item assignment
```

[ ]:
```
# Lists are mutable because you can overwrite their elements
power = [1.21, 'gigawatts']
power[0] = 2.21
print(power)
```

3. Introduction to tuples

[ ]:
```
# Tuples are instantiated with parentheses.
fullname = ('Masha', 'Z', 'Hopper')

# Tuples are immutable, so their elements cannot be overwritten.
fullname[2] = 'Copper'
print(fullname)
```

[ ]:
```
# You can combine tuples using addition.
fullname = fullname + ('Jr',)
print(fullname)
```

[ ]:
```
# The tuple() function converts an object's data type to tuple.
fullname = ['Masha', 'Z', 'Hopper']
fullname = tuple(fullname)
print(fullname)
```

[ ]:
```
# Functions that return multiple values return them in a tuple.
def to_dollars_cents(price):
    '''
    Split price (float) into dollars and cents.
    '''
    dollars = int(price // 1)
    cents = round(price % 1 * 100)

    return dollars, cents
```

[ ]:
```
# Functions that return multiple values return them in a tuple.
to_dollars_cents(6.55)
```

[ ]:
```
# "Unpacking" a tuple allows a tuple's elements to be assigned to variables.
dollars, cents = to_dollars_cents(6.55)
print(dollars + 1)
print(cents + 1)
```

[ ]:
```
# The data type of an element of an unpacked tuple is not necessarily a tuple.
type(dollars)
```

[ ]:
```
# Create a list of tuples, each representing the name, age, and position of a
# player on a basketball team.
team = [('Marta', 20, 'center'),
        ('Ana', 22, 'point guard'),
        ('Gabi', 22, 'shooting guard'),
        ('Luz', 21, 'power forward'),
        ('Lorena', 19, 'small forward'),
        ]
```

[ ]:
```
# Use a for loop to loop over the list, unpack the tuple at each iteration, and
# print one of the values.
for name, age, position in team:
    print(name)
```

[ ]:
```
# This code produces the same result as the code in the cell above.
for player in team:
    print(player[0])
```

4. More with loops, lists, and tuples

[ ]:
```
# Create a list of tuples, each representing the name, age, and position of a
# player on a basketball team.
team = [
    ('Marta', 20, 'center'),
    ('Ana', 22, 'point guard'),
    ('Gabi', 22, 'shooting guard'),
    ('Luz', 21, 'power forward'),
    ('Lorena', 19, 'small forward'),
    ]
```

[ ]:
```
# Create a function to extract and names and positions from the team list and
# format them to be printed. Returns a list.
def player_position(players):
    result = []
    for name, age, position in players:
        result.append('Name: {:>19} \nPosition: {:>15}\n'.format(name,␣
 ↪position))

    return result
```

[ ]:
```
# Loop over the list of formatted names and positions produced by
# player_position() function and print them.
for player in player_position(team):
    print(player)
```

[ ]:
```
# Nested loops can produce the different combinations of pips (dots) in
# a set of dominoes.
```

```
for left in range(7):
    for right in range(left, 7):
        print(f"[{left}|{right}]", end=" ")
    print('\n')
```

[ ]:
```
# Create a list of dominoes, with each domino reprented as a tuple.
dominoes = []
for left in range(7):
    for right in range(left, 7):
        dominoes.append((left, right))
dominoes
```

[ ]:
```
# Select index 1 of the tuple at index 4 in the list of dominoes.
dominoes[4][1]
```

In the following code cells are two ways to add the total number of pips on each individual domino to a list, as indicated in this diagram:

The first way uses a for loop. The second way uses a list comprehension.

[ ]:
```
# You can use a for loop to sum the pips on each domino and append
# the sum to a new list.
pips_from_loop = []
for domino in dominoes:
    pips_from_loop.append(domino[0] + domino[1])
print(pips_from_loop)
```

[ ]:
```
# A list comprehension produces the same result with less code.
pips_from_list_comp = [domino[0] + domino[1] for domino in dominoes]
pips_from_loop == pips_from_list_comp
```

5. Introduction to dictionaries

[ ]:
```
# Create a dictionary with pens as keys and the animals they contain as values.
# Dictionaries can be instantiated using braces.
zoo = {
    'pen_1': 'penguins',
    'pen_2': 'zebras',
    'pen_3': 'lions',
    }

# Selecting the `pen_2` key returns `zebras`, the value stored at that key.
zoo['pen_2']
```

[ ]:
```
# You cannot access a dictionary's values by name using bracket indexing
# because the computer interprets this as a key, not a value.
zoo['zebras']
```

[ ]:
```
# Dictionaries can also be instantiated using the dict() function.
zoo = dict(
    pen_1='monkeys',
    pen_2='zebras',
    pen_3='lions',
    )

zoo['pen_2']
```

[ ]:
```
# Another way to create a dictionary using the dict() function.
zoo = dict(
    [
     ['pen_1', 'monkeys'],
     ['pen_2', 'zebras'],
     ['pen_3', 'lions'],
    ]
)

zoo['pen_2']
```

[ ]:
```
# Assign a new key: value pair to an existing dictionary.
zoo['pen_4'] = 'crocodiles'
zoo
```

[ ]:
```
# Dictionaries are unordered and do not support numerical indexing.
zoo[2]
```

[ ]:
```
# Use the `in` keyword to produce a Boolean of whether a given key exists in a␣
 ↪dictionary.
print('pen_1' in zoo)
print('pen_7' in zoo)
```

6. Dictionary methods

[ ]:
```
# Create a list of tuples, each representing the name, age, and position of a
# player on a basketball team.
team = [
    ('Marta', 20, 'center'),
    ('Ana', 22, 'point guard'),
    ('Gabi', 22, 'shooting guard'),
    ('Luz', 21, 'power forward'),
    ('Lorena', 19, 'small forward'),
    ]
```

[ ]:
```
# Add new players to the list.
team = [
    ('Marta', 20, 'center'),
    ('Ana', 22, 'point guard'),
    ('Gabi', 22, 'shooting guard'),
    ('Luz', 21, 'power forward'),
    ('Lorena', 19, 'small forward'),
    ('Sandra', 19, 'center'),
    ('Mari', 18, 'point guard'),
    ('Esme', 18, 'shooting guard'),
    ('Lin', 18, 'power forward'),
    ('Sol', 19, 'small forward'),
    ]
```

[ ]:
```
# Instantiate an empty dictionary.
new_team = {}

# Loop over the tuples in the list of players and unpack their values.
for name, age, position in team:
    if position in new_team:                    # If position already a key in␣
↪new_team,
        new_team[position].append((name, age))  # append (name, age) tup to␣
↪list at that value.
    else:
        new_team[position] = [(name, age)]      # If position not a key in␣
↪new_team,
                                                # create a new key whose value␣
↪is a list
                                                # containing (name, age) tup.
new_team
```

[ ]:
```
# Examine the value at the 'point guard' key.
new_team['point guard']
```

[ ]:
```
# You can access the a dictionary's keys by looping over them.
for x in new_team:
    print(x)
```

[ ]:
```
# The keys() method returns the keys of a dictionary.
new_team.keys()
```

[ ]:
```
# The values() method returns all the values in a dictionary.
new_team.values()
```

[ ]:
```
# The items() method returns both the keys and the values.
for a, b in new_team.items():
    print(a, b)
```

7. Introduction to sets

[ ]:
```
# The set() function converts a list to a set.
x = set(['foo', 'bar', 'baz', 'foo'])
print(x)
```

[ ]:
```
# The set() function converts a tuple to a set.
x = set(('foo','bar','baz', 'foo'))
print(x)
```

[ ]:
```
# The set() function converts a string to a set.
x = set('foo')
print(x)
```

[ ]:
```
# You can use braces to instantiate a set
x = {'foo'}
print(type(x))

# But empty braces are reserved for dictionaries.
y = {}
print(type(y))
```

[ ]:
```
# Instantiating a set with braces treats the contents as literals.
x = {'foo'}
print(x)
```

[ ]:
```
# The intersection() method (&) returns common elements between two sets.
set1 = {1, 2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8, 9}
print(set1.intersection(set2))
print(set1 & set2)
```

[ ]:
```
# The union() method (|) returns all the elements from two sets, each␣
 ↪represented once.
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}
print(x1.union(x2))
print(x1 | x2)
```

[ ]:
```
# The difference() method (-) returns the elements in set1 that aren't in set2
set1 = {1, 2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8, 9}
print(set1.difference(set2))
print(set1 - set2)
```

[ ]:
```
# ... and the elements in set2 that aren't in set1.
print(set2.difference(set1))
```

```
print(set2 - set1)
```

[ ]:
```
# The symmetric_difference() method (^) returns all the values from each set␣
↪that
# are not in both sets.
set1 = {1, 2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8, 9}
set2.symmetric_difference(set1)
set2 ^ set1
```

8. Introduction to NumPy

[ ]:
```
# Lists cannot be multiplied together.
list_a = [1, 2, 3]
list_b = [2, 4, 6]

list_a * list_b
```

[ ]:
```
# To perform element-wise multiplication between two lists, you could
# use a for loop.
list_c = []
for i in range(len(list_a)):
    list_c.append(list_a[i] * list_b[i])

list_c
```

[ ]:
```
# NumPy arrays let you perform array operations.

# Import numpy, aliased as np.
import numpy as np

# Convert lists to arrays.
array_a = np.array(list_a)
array_b = np.array(list_b)

# Perform element-wise multiplication between the arrays.
array_a * array_b
```

9. Basic array operations

[ ]:
```
import numpy as np

# The np.array() function converts an object to an ndarray
x = np.array([1, 2, 3, 4])
x
```

[ ]:
```
# Arrays can be indexed.
x[-1] = 5
x
```

[ ]:
```
# Trying to access an index that doesn't exist will throw an error.
x[4] = 10
```

[ ]:
```
# Arrays cast every element they contain as the same data type.
arr = np.array([1, 2, 'coconut'])
arr
```

[ ]:
```
# NumPy arrays are a class called `ndarray`.
print(type(arr))
```

[ ]:
```
# The dtype attribute returns the data type of an array's contents.
arr = np.array([1, 2, 3])
arr.dtype
```

[ ]:
```
# The shape attribute returns the number of elements in each dimension
# of an array.
arr.shape
```

[ ]:
```
# The ndim attribute returns the number of dimensions in an array.
arr.ndim
```

[ ]:
```
# Create a 2D array by passing a list of lists to np.array() function.
arr_2d = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
print(arr_2d.shape)
print(arr_2d.ndim)
arr_2d
```

[ ]:
```
# Create a 3D array by passing a list of two lists of lists to np.array()␣
↪function.
arr_3d = np.array([[[1, 2, 3],
                    [3, 4, 5]],

                   [[5, 6, 7],
                    [7, 8, 9]]]
)

print(arr_3d.shape)
print(arr_3d.ndim)
arr_3d
```

[ ]:
```
# The reshape() method changes the shape of an array.
arr_2d = arr_2d.reshape(2, 4)
arr_2d
```

[ ]:
```
# Create new array
arr = np.array([1, 2, 3, 4, 5])

# The mean() method returns the mean of the elements in an array.
np.mean(arr)
```

[ ]:
```
# The log() method returns the natural logarithm of the elements in an array.
np.log(arr)
```

[ ]:
```
# The floor() method returns the value of a number rounded down
# to the nearest integer.
np.floor(5.7)
```

[ ]:
```
# The ceil() method returns the value of a number rounded up
# to the nearest integer.
np.ceil(5.3)
```

10. Introduction to pandas

[ ]:
```
# NumPy and pandas are typically imported together.
# np and pd are conventional aliases.
import numpy as np
import pandas as pd
```

[ ]:
```
# Read in data from a .csv file.
dataframe = pd.read_csv('train.csv')

# Print the first 25 rows.
dataframe.head(25)
```

[ ]:
```
# Calculate the mean of the Age column.
dataframe['Age'].mean()
```

[ ]:
```
# Calculate the maximum value contained in the Age column.
dataframe['Age'].max()
```

[ ]:
```
# Calculate the minimum value contained in the Age column.
dataframe['Age'].min()
```

[ ]:
```
# Calculate the standard deviation of the values in the Age column.
dataframe['Age'].std()
```

[ ]:
```
# Return the number of rows that share the same value in the Pclass column.
dataframe['Pclass'].value_counts()
```

[ ]:
```
# The describe() method returns summary statistics of the dataframe.
dataframe.describe()
```

[ ]:
```
# Filter the data to return only rows where value in Age column is greater than␣
↪60
# and value in Pclass column equals 3.
dataframe[(dataframe['Age'] > 60) & (dataframe['Pclass'] == 3)]
```

[ ]:
```
# Create a new column called 2023_Fare that contains the inflation-adjusted
# fare of each ticket in 2023 pounds.
dataframe['2023_Fare'] = dataframe['Fare'] * 146.14
dataframe
```

[ ]:
```
# Use iloc to access data using index numbers.
# Select row 1, column 3.
dataframe.iloc[1][3]
```

[ ]:
```
# Group customers by Sex and Pclass and calculate the total paid for each group
# and the mean price paid for each group.
fare = dataframe.groupby(['Sex', 'Pclass']).agg({'Fare': ['count', 'sum']})
fare['fare_avg'] = fare['Fare']['sum'] / fare['Fare']['count']
fare
```

11. pandas basics

[ ]:
```
import pandas as pd

# Use pd.DataFrame() function to create a dataframe from a dictionary.
data = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data=data)
df
```

[ ]:
```
# Use pd.DataFrame() function to create a dataframe from a NumPy array.
df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
                   columns=['a', 'b', 'c'], index=['x', 'y', 'z'])
df2
```

[ ]:
```
# Use pd.read_csv() function to create a dataframe from a .csv file
# from a URL or filepath.
df3 = pd.read_csv('train.csv')
df3.head()
```

[ ]:
```
# Print class of first row
print(type(df3.iloc[0]))

# Print class of "Name" column
print(type(df3['Name']))
```

[ ]:
```
# Create a copy of df3 named 'titanic'.
titanic = df3
```

```
# The head() method outputs the first 5 rows of dataframe.
titanic.head()
```

[ ]:
```
# The columns attribute returns an Index object containing the dataframe's␣
↪columns.
titanic.columns
```

[ ]:
```
# The shape attribute returns the shape of the dataframe (rows, columns).
titanic.shape
```

[ ]:
```
# The info() method returns summary information about the dataframe.
titanic.info()
```

[ ]:
```
# You can select a column by name using brackets.
titanic['Age']
```

[ ]:
```
# You can select a column by name using dot notation
# only when its name contains no spaces or special characters.
titanic.Age
```

[5]:
```
# You can create a DataFrame object of specific columns using a list
# of column names inside brackets.
titanic[['Name', 'Age']]
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-c861d7b1377b> in <module>
  1 # You can create a DataFrame object of specific columns using a list
  2 # of column names inside brackets.
----> 3 titanic[['Name', 'Age']]


NameError: name 'titanic' is not defined
```

[ ]:
```
# Use iloc to return a Series object of the data in row 0.
titanic.iloc[0]
```

[ ]:
```
# Use iloc to return a DataFrame view of the data in row 0.
titanic.iloc[[0]]
```

[ ]:
```
# Use iloc to return a DataFrame view of the data in rows 0, 1, 2.
titanic.iloc[0:3]
```

[ ]:
```
# Use iloc to return a DataFrame view of rows 0-2 at columns 3 and 4.
titanic.iloc[0:3, [3, 4]]
```

[ ]:
```
# Use iloc to return a DataFrame view of all rows at column 3.
titanic.iloc[:, [3]]
```

[ ]:
```
# Use iloc to access value in row 0, column 3.
titanic.iloc[0, 3]
```

[ ]:
```
# Use loc to access values in rows 0-3 at just the Name column.
titanic.loc[0:3, ['Name']]
```

[ ]:
```
# Create a new column in the dataframe containing the value in the Age column +␣
↪100.
titanic['Age_plus_100'] = titanic['Age'] + 100
titanic.head()
```

12. Boolean masking

[ ]:
```
# Instantiate a dictionary of planetary data.
data = {'planet': ['Mercury', 'Venus', 'Earth', 'Mars',
                   'Jupiter', 'Saturn', 'Uranus', 'Neptune'],
        'radius_km': [2440, 6052, 6371, 3390, 69911, 58232,
                      25362, 24622],
        'moons': [0, 0, 1, 2, 80, 83, 27, 14]
         }
# Use pd.DataFrame() function to convert dictionary to dataframe.
planets = pd.DataFrame(data)
planets
```

[ ]:
```
# Create a Boolean mask of planets with fewer than 20 moons.
mask = planets['moons'] < 20
mask
```

[ ]:
```
# Apply the Boolean mask to the dataframe to filter it so it contains
# only the planets with fewer than 20 moons.
planets[mask]
```

[ ]:
```
# Define the Boolean mask and apply it in a single line.
planets[planets['moons'] < 20]
```

[ ]:
```
# Boolean masks don't change the data. They're just views.
planets
```

[ ]:
```
# You can assign a dataframe view to a named variable.
moons_under_20 = planets[mask]
moons_under_20
```

[ ]:
```
# Create a Boolean mask of planets with fewer than 10 moons OR more than 50␣
↪moons.
mask = (planets['moons'] < 10) | (planets['moons'] > 50)
mask
```

[ ]:
```
# Apply the Boolean mask to filter the data.
planets[mask]
```

[ ]:
```
# Create a Boolean mask of planets with more than 20 moons, excluding them if␣
↪they
# have 80 moons or if their radius is less than 50,000 km.
mask = (planets['moons'] > 20) & ~(planets['moons'] == 80) &␣
↪~(planets['radius_km'] < 50000)

# Apply the mask
planets[mask]
```

13. Grouping and aggregation

[ ]:
```
import numpy as np
import pandas as pd

# Instantiate a dictionary of planetary data.
data = {'planet': ['Mercury', 'Venus', 'Earth', 'Mars',
                   'Jupiter', 'Saturn', 'Uranus', 'Neptune'],
        'radius_km': [2440, 6052, 6371, 3390, 69911, 58232,
                      25362, 24622],
        'moons': [0, 0, 1, 2, 80, 83, 27, 14],
        'type': ['terrestrial', 'terrestrial', 'terrestrial', 'terrestrial',
                 'gas giant', 'gas giant', 'ice giant', 'ice giant'],
        'rings': ['no', 'no', 'no', 'no', 'yes', 'yes', 'yes','yes'],
        'mean_temp_c': [167, 464, 15, -65, -110, -140, -195, -200],
        'magnetic_field': ['yes', 'no', 'yes', 'no', 'yes', 'yes', 'yes', 'yes']
        }

# Use pd.DataFrame() function to convert dictionary to dataframe.
planets = pd.DataFrame(data)
planets
```

[ ]:
```
# The groupby() function returns a groupby object.
planets.groupby(['type'])
```

[ ]:
```
# Apply the sum() function to the groupby object to get the sum
# of the values in each numerical column for each group.
planets.groupby(['type']).sum()
```

[ ]:
```
# Apply the sum function to the groupby object and select
# only the 'moons' column.
planets.groupby(['type']).sum()[['moons']]
```

[ ]:
```
# Group by type and magnetic_field and get the mean of the values
# in the numeric columns for each group.
planets.groupby(['type', 'magnetic_field']).mean()
```

[ ]:
```
# Group by type, then use the agg() function to get the mean and median
# of the values in the numeric columns for each group.
planets.groupby(['type']).agg(['mean', 'median'])
```

[ ]:
```
# Group by type and magnetic_field, then use the agg() function to get the
# mean and max of the values in the numeric columns for each group.
planets.groupby(['type', 'magnetic_field']).agg(['mean', 'max'])
```

[ ]:
```
# Define a function that returns the 90 percentile of an array.
def percentile_90(x):
    return x.quantile(0.9)
```

[ ]:
```
# Group by type and magnetic_field, then use the agg() function to apply the
# mean and the custom-defined `percentile_90()` function to the numeric
# columns for each group.
planets.groupby(['type', 'magnetic_field']).agg(['mean', percentile_90])
```

14. Merging and joining data

[ ]:
```
import numpy as np
import pandas as pd

# Instantiate a dictionary of planetary data.
data = {'planet': ['Mercury', 'Venus', 'Earth', 'Mars'],
        'radius_km': [2440, 6052, 6371, 3390],
        'moons': [0, 0, 1, 2],
        }
# Use pd.DataFrame() function to convert dictionary to dataframe.
df1 = pd.DataFrame(data)
df1
```

[ ]:
```
# Instantiate a dictionary of planetary data.
data = {'planet': ['Jupiter', 'Saturn', 'Uranus', 'Neptune'],
        'radius_km': [69911, 58232, 25362, 24622],
        'moons': [80, 83, 27, 14],
```

```
        }
# Use pd.DataFrame() function to convert dictionary to dataframe.
df2 = pd.DataFrame(data)
df2
```

[ ]:
```
# The pd.concat() function can combine the two dataframes along axis 0,
# with the second dataframe being added as new rows to the first dataframe.
df3 = pd.concat([df1, df2], axis=0)
df3
```

[ ]:
```
# Reset the row indices.
df3 = df3.reset_index(drop=True)
df3
```

[ ]:
```
# NOTE: THIS CELL WAS NOT SHOWN IN THE INSTRUCTIONAL VIDEO BUT WAS RUN AS A
#       SETUP CELL.
data = {'planet': ['Earth', 'Mars','Jupiter', 'Saturn', 'Uranus',
                   'Neptune', 'Janssen', 'Tadmor'],
        'type': ['terrestrial', 'terrestrial','gas giant', 'gas giant',
                 'ice giant', 'ice giant', 'super earth','gas giant'],
        'rings': ['no', 'no', 'yes', 'yes', 'yes','yes', 'no', None],
        'mean_temp_c': [15, -65, -110, -140, -195, -200, None, None],
        'magnetic_field': ['yes', 'no', 'yes', 'yes', 'yes', 'yes', None, None],
        'life': [1, 0, 0, 0, 0, 0, 1, 1]
        }
df4 = pd.DataFrame(data)
```

[ ]:
```
df4
```

[ ]:
```
# Use pd.merge() to combine dataframes.
# Inner merge retains only keys that appear in both dataframes.
inner = pd.merge(df3, df4, on='planet', how='inner')
inner
```

[ ]:
```
# Use pd.merge() to combine dataframes.
# Outer merge retains all keys from both dataframes.
outer = pd.merge(df3, df4, on='planet', how='outer')
outer
```

[ ]:
```
# Use pd.merge() to combine dataframes.
# Left merge retains only keys that appear in the left dataframe.
left = pd.merge(df3, df4, on='planet', how='left')
left
```

[ ]:
```
# Use pd.merge() to combine dataframes.
# Right merge retains only keys that appear in right dataframe.
right = pd.merge(df3, df4, on='planet', how='right')
```

```
right
```

Congratulations! You've completed this lab. However, you may not notice a green check mark next to this item on Coursera's platform. Please continue your progress regardless of the check mark. Just click on the "save" icon at the top of this notebook to ensure your work has been logged.