Q :

Understanding Object Dependencies

Which of the following is the basic building block for any of Python's scientific computing, data science, and general programming libraries we use today?

A :

Python object.

press

Correct! Everything in Python starts as an object.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ref :

Creating functions

In this exercise, we will review functions, as they are key building blocks of object-oriented programs.

For this, we will create a simple function average\_numbers() which averages a list of numbers. Remember that lists are a basic data type in Python that we can build using the [] bracket notation.

Here is an example of a function that returns the square of an integer:

def square\_function(x):

x\_squared = x\*\*2

return x\_squared

Q :

Create a function average\_numbers(), which takes a list num\_list as input and then returns avg as output.

Inside the function, create a variable, avg, that takes the average of all the numbers in the list.

Call the average\_numbers function on the list [1, 2, 3, 4, 5, 6] and assign the output to the variable my\_avg.

Print out my\_avg.

# Create function that returns the average of an integer list

def average\_numbers(num\_list):

avg = sum(num\_list)/float(len(num\_list)) # divide by length of list

return avg

# Take the average of a list: my\_avg

my\_avg = average\_numbers([1, 2, 3, 4, 5, 6])

# Print out my\_avg

print(my\_avg)

<script.py> output:

3.5

Good job! Now that you have created a function, try playing around by adding your own list.

Ref :

Creating a complex data type

In this exercise, we'll take a closer look at the flexibility of the list data type, by creating a list of lists.

In Python, lists usually look like our list example below, and can be made up of either simple strings, integers, or a combination of both.

list = [1,2]

In creating a list of lists, we're building up to the concept of a NumPy array.

Create a variable called matrix, and assign it the value of a list.

Within the matrix list, include two additional lists: [1,2,3,4] and [5,6,7,8].

Print the matrix list.

# Create a list that contains two lists: matrix

matrix = [[1,2,3,4] , [5,6,7,8]]

# Print the matrix list

print(matrix)

<script.py> output:

[[1, 2, 3, 4], [5, 6, 7, 8]]

Good job! What happens when you nest two lists in a list? How can you access those elements? Try creating a tripply-nested list and finding out.

Ref :

**What are NumPy Arrays most similar to?**

What other Python data structure can one-dimensional NumPy arrays be thought of as similar, but not exactly identical to?

A : Lists.

Press

Correct! One-dimensional NumPy arrays look a lot like lists, and building them is very similar.

Ref :

# Create a function that returns a NumPy array

In this exercise, we'll continue working with the numpy package and our previous structures.

We'll create a NumPy array of the float (numerical) data type so that we can work with a multi-dimensional data objects, much like columns and rows in a spreadsheet.

Q :

* Import numpy as np.
* Declare variable my\_matrix and set it to [[1,2,3,4], [5,6,7,8]].
* Declare a function called return\_array(), which takes a list matrix as input, and returns an array object as output. In the body, declare a variable array set it to np.array(matrix, dtype = float).
* Call return\_array() on the my\_matrix list, and print out the output.

# Import numpy as np

import numpy as np

# List input: my\_matrix

my\_matrix = [[1,2,3,4], [5,6,7,8]]

# Function that converts lists to arrays: return\_array

def return\_array(matrix):

array = np.array(matrix, dtype = float)

return array

# Call return\_array on my\_matrix, and print the output

print(return\_array(my\_matrix))

<script.py> output:

[[1. 2. 3. 4.]

[5. 6. 7. 8.]]

Good job! Now that you have created a NumPy array, take a look at the NumPy docs (by googling) and investigate other data types arrays can take (as arguments).

Ref :

# Creating a class

We're going to be working on building a class, which is a way to organize functions and variables in Python. To start with, let's look at the simplest possible way to create a class.

Q :

* Declare a class called DataShell.
* Our class will not do much: simply include the passstatement in the body of the DataShell class.

# Create a class: DataShell

class DataShell:

pass

Good job! Try creating different classes with different names. See what kind of naming convention Python will accept.

Q :

# Difference between a class and an object

Fill in the blanks: \_\_\_\_ are instances of \_\_\_\_ and can have both variables and functions.

A :

Objects, classes.

Press

# Understanding what we're building

What high-level Python object will our DataShell be most like once we're done finishing building it?

##### Answer the question

**50 XP**

##### Possible Answers



A Pandas dataframe.

press1



A Python method.

press2



A scikit-learn model.

press3



A NumPy Array.

press

That's right! We're trying to build out a Pandas object.

Ref :

# Object: Instance of a Class

As we learned earlier, a class is like a blueprint: we can make many copies of our class.

When we do this, we say that we are instantiating our class. These instances are called objects.

Here is an example of class instantiation:

object\_name = ClassName()

Q :

* Create an empty class called DataShell. Only include the pass statement inside of the class definition.
* Instantiate the DataShell class and assign the newly created object to the my\_data\_shell variable.
* Print my\_data\_shell and explore its contents.

# Create empty class: DataShell

class DataShell:

# Pass statement

pass

# Instantiate DataShell: my\_data\_shell

my\_data\_shell = DataShell()

# Print my\_data\_shell

print(my\_data\_shell)

<script.py> output:

<\_\_main\_\_.DataShell object at 0x7f413b4d30f0>

Good job! Now you have created an instance of the DataShell class. Try creating additional instances with different names!

# Ref :

# The Init Method

Now it's time to explore the special \_\_init\_\_ method!

\_\_init\_\_ is an initialization method used to construct class instances in custom ways. In this exercise we will simply introduce the utilization of the method, and in subsequent ones we will do fancier things.

**Q :**

* Create a class called DataShell.
* Include the \_\_init\_\_() method, and pass it the self argument.
* In the body of the \_\_init\_\_() method, include the pass statement.
* Instantiate the DataShell class and assign the newly created object to the my\_data\_shell variable. Then print my\_data\_shell and explore its contents.

# Create class: DataShell

class DataShell:

# Initialize class with self argument

def \_\_init\_\_(self):

# Pass statement

pass

# Instantiate DataShell: my\_data\_shell

my\_data\_shell = DataShell()

# Print my\_data\_shell

print(my\_data\_shell)

<script.py> output:

<\_\_main\_\_.DataShell object at 0x7fd7a71e15f8>

Now you know about the initialization method (\_\_init\_\_())! Notice that this method takes in as input argument the self keyword. Could we input other arguments?

# Ref :

# Instance Variables

Class instances are useful in that we can store values in them at the time of instantiation. We store these values in instance variables. This means that we can have many instances of the same class whose instance variables hold different values!

##### **Q :**

* Create a class called DataShell.
* In the class definition, include the \_\_init\_\_() method, and pass it the self and integerInput arguments. In the body of the \_\_init\_\_() method, set the data as the instance variable.
* Create an instance of DataShell called my\_data\_shell. Pass x as an argument to the constructor function.
* Print the my\_data\_shell.data and explore its contents.

# Create class: DataShell

class DataShell:

# Initialize class with self and integerInput arguments

def \_\_init\_\_(self, integerInput):

# Set data as instance variable, and assign the value of integerInput

self.data = integerInput

# Declare variable x with value of 10

x = 10

# Instantiate DataShell passing x as argument: my\_data\_shell

my\_data\_shell = DataShell(x)

# Print my\_data\_shell

print(my\_data\_shell.data)

<script.py> output:

10

Great job declaring instance variables! Notice that instance variables live in the body of the initialization method, as they are initialized when the object is instantiated. Also important to notice that they are preceded by self., as this is referring to the instance itself.

# Ref :

# Multiple Instance Variables

We are not limited to declaring only one instance variable; in fact, we can declare many!

In this exercise we will declare two instance variables: identifier and data. Their values will be specified by the values passed to the initialization method, as before.

##### **Q :**

* Create a class called DataShell.
* Initialize the class with the self, identifier, and data arguments. Set identifier and data to be instance variables.
* Create an instance of DataShell called my\_data\_shell passing x and y to the constructor function.
* Print the my\_data\_shell.identifier and my\_data\_shell.data and explore their contents.

# Create class: DataShell

class DataShell:

# Initialize class with self, identifier and data arguments

def \_\_init\_\_(self, identifier , data):

# Set identifier and data as instance variables, assigning value of input arguments

self.identifier = identifier

self.data = data

# Declare variable x with value of 100, and y with list of integers from 1 to 5

x = 100

y = [1, 2, 3, 4, 5]

# Instantiate DataShell passing x and y as arguments: my\_data\_shell

my\_data\_shell = DataShell(x , y)

# Print my\_data\_shell.identifier

print(my\_data\_shell.identifier)

# Print my\_data\_shell.data

print(my\_data\_shell.data)

<script.py> output:

100

[1, 2, 3, 4, 5]

Excellent! As you saw in this exercise, you can declare more than one instance variable! What kind of instance variables can you think of that might be useful to have in our DataShell class?

# Ref :

# Class Variables

We saw that we can specify different instance variables.

But, what if we want any instance of a class to hold the same value for a specific variable? Enter class variables.

Class variables must not be specified at the time of instantiation and instead, are declared/specified at the class definition phase.

**Q :**

* Create a class called DataShell.
* Declare a class variable called family and assign it the value of "DataShell".
* Create an instance of DataShell called my\_data\_shell, passing x to the initializer method.
* Print the my\_data\_shell.family to explore its contents.

# Create class: DataShell

class DataShell:

# Declare a class variable family, and assign value of "DataShell"

family = 'DataShell'

# Initialize class with self, identifier arguments

def \_\_init\_\_(self, identifier):

# Set identifier as instance variable of input argument

self.identifier = identifier

# Declare variable x with value of 100

x = 100

# Instantiate DataShell passing x as argument: my\_data\_shell

my\_data\_shell = DataShell(x)

# Print my\_data\_shell class variable family

print(my\_data\_shell.family)

<script.py> output:

DataShell

Awesome! Class variables are different from instance variables (which we saw earlier). Even though class variables may be overridden, they are generally set even before object instanciation; therefore, class variable values across instances of the same class tend to be the same.

# Ref :

# Overriding Class Variables

Sometimes our object instances have class variables whose values are not correct, and hence, not useful. For this reason it makes sense to modify our object's class variables.

In this exercise, we will do just that: override class variables with values of our own!

**Q :**

* Create a class called DataShell.
* Declare a class variable called family and assign it the value of "DataShell".
* Create an instance of DataShell called my\_data\_shell passing x to the initializer method, then print the my\_data\_shell.family to explore its contents.
* Override the class variable my\_data\_shell.family by assigning it the value "NotDataShell" to explore its contents.

# Create class: DataShell

class DataShell:

# Declare a class variable family, and assign value of "DataShell"

family = 'DataShell'

# Initialize class with self, identifier arguments

def \_\_init\_\_(self, identifier):

# Set identifier as instance variables, assigning value of input arguments

self.identifier = identifier

# Declare variable x with value of 100

x = 100

# Instantiate DataShell passing x and y as arguments: my\_data\_shell

my\_data\_shell = DataShell(x)

# Print my\_data\_shell class variable family

print(my\_data\_shell.family)

# Override the my\_data\_shell.family value with "NotDataShell"

my\_data\_shell.family = 'NotDataShell'

# Print my\_data\_shell class variable family once again

print(my\_data\_shell.family)

<script.py> output:

DataShell

NotDataShell

Great! Now you have learned how to override class variables. Does this look too different from changing the value of instance variables?

# Ref :

# Methods I

Not only are we able to declare both instance variables and class variables in our objects, we can also cook functions right into our objects as well. These object-contained functions are called methods.

**Q :**

* Create a class called DataShell with its initialization method.
* Define a method called print\_static that only takes the argument self. Inside of this method's body print the string "You just executed a class method!".
* Create an instance of DataShell called my\_data\_shell passing no arguments to the constructor.
* Call the print\_static method and explore its output!

# Create class: DataShell

class DataShell:

# Initialize class with self argument

def \_\_init\_\_(self):

pass

# Define class method which takes self argument: print\_static

def print\_static(self):

# Print string

print("You just executed a class method!")

# Instantiate DataShell taking no arguments: my\_data\_shell

my\_data\_shell = DataShell()

# Call the print\_static method of your newly created object

my\_data\_shell.print\_static()

<script.py> output:

You just executed a class method!

Super! You are on your way to becoming a wizard at writing class methods! Can you think of more interesting functionality than simply printing a static string?

# Ref :

# Methods II

In the previous exercise our print\_static() method was kind of boring.

We can do more interesting things with our objects' methods. For example, we can interact with our objects' data. In this exercise we will declare a method that prints the value of one of our instance variables.

##### 

**Q :**

* Create a class called DataShell with its initialization method, taking self and dataList as arguments. Declare data as an instance variable and assign it the value of dataList.
* Define show() as a class method, taking self as an argument. Inside of the method print the instance variable data.
* Declare a list called integer\_list with integers 1 to 10. Then create an instance of DataShell called my\_data\_shell with integer\_list as an argument to the constructor.
* Call your object's show() method and explore its output.

# Create class: DataShell

class DataShell:

# Initialize class with self and dataList as arguments

def \_\_init\_\_(self, dataList):

# Set data as instance variable, and assign it the value of dataList

self.data = dataList

# Define class method which takes self argument: show

def show(self):

# Print the instance variable data

print(self.data)

# Declare variable with list of integers from 1 to 10: integer\_list

integer\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Instantiate DataShell taking integer\_list as argument: my\_data\_shell

my\_data\_shell = DataShell(integer\_list)

# Call the show method of your newly created object

my\_data\_shell.show()

<script.py> output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Outstanding! You accessed an instance variable from within your method, and were able to wrangle its data to output useful information! What other magic can be done with class and object internals?

# Ref :

# Methods III

In the last exercise our method simply printed out the value of instance variables.

In this one, we'll do something more interesting. We will add another method, avg(), which takes a list of integers, calculates the average value, and prints it out. To make things even more interesting, the list of integers for which avg() does this operations, is one of our object's instance variables.

This means that our object can not only store data, but also can store procedures it can execute on its own data. Awesome.

Note that the variable integer\_list has already been loaded for you.

**Q :**

* Create a class called DataShell with its initialization method, taking self and dataList as arguments. Declare data as an instance variable and assign it the value of the input argument dataList.
* Define show() as a class method, taking self as an argument. Inside of the method's body, print the instance variable data.
* Define avg() as a class method, taking self as an argument. Inside of the method's body, declare the variable avg and assign it the value of the average of the instance variable data. Then print it out.
* Instantiate DataShell as my\_data\_shell passing integer\_list as an argument to the constructor. Then call your object's show() and avg() methods and explore their output.

# Create class: DataShell

class DataShell:

# Initialize class with self and dataList as arguments

def \_\_init\_\_(self, dataList):

# Set data as instance variable, and assign it the value of dataList

self.data = dataList

# Define method that prints data: show

def show(self):

print(self.data)

# Define method that prints average of data: avg

def avg(self):

# Declare avg and assign it the average of data

avg = sum(self.data)/float(len(self.data))

# Print avg

print(avg)

# Instantiate DataShell taking integer\_list as argument: my\_data\_shell

my\_data\_shell = DataShell(integer\_list)

# Call the show and avg methods of your newly created object

my\_data\_shell.show()

my\_data\_shell.avg()

<script.py> output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

5.5

Marvelous! It even feels that instances of the DataShell class are self-aware. Can you think of a way of writing methods that use class variables instead of instance variables?

# Ref :

# Return Statement I

Let's now drill into the return statement.

class DataShell:

def \_\_init\_\_(self, x):

return x

In the code chunk above, you may have expected to see the print() function instead of the return statement. The difference between the two is that print() outputs a string to the console, while the the return statement exits the current function (or method) and hands the returned value back to its caller. In this case, the caller could have another function, among other things. If this sounds confusing have not fear, we will further practice this!

Q :

In the console, enter this code in order to answer the question below:

x = my\_data\_shell.get\_data()

print(x)

**What value does the my\_data\_shell.get\_data() method return?**

##### Possible Answers

* 

[1, 2, 3]

* 

[1, 2, 3, 4, 5]

* 

[10, 20, 30, 40, 50]

* 

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Yes! Notice that in this case, entering into the console data\_shell.get\_data() outputs to the console just as print(my\_data\_shell.get\_data()) does! In what situations might this not be the case?

# Ref :

# Return Statement II: The Return of the DataShell

Let's now go back to the class DataShell that we were working with earlier, and refactor it such that it uses the return statement instead of the print() function.

Notice that since we are now using the return statement, we need to include our calls to object methods within the print()function.

Q :

* Create a class called DataShell with its initialization method, taking self and dataList as arguments. Declare data as an instance variable and assign it the value of the input argument dataList.
* Define show() as a class method, taking self as an argument. Inside of the method's body, **return** the instance variable data.
* Define avg() as a class method, taking self as an argument. Inside of the method's body, declare the variable avg and assign it the value of the average of the instance variable data. Then **return** avg.
* Instantiate DataShell as my\_data\_shell passing integer\_list as an argument to the constructor. Then print the output of your object's show() and avg() methods.

# Create class: DataShell

class DataShell:

# Initialize class with self and dataList as arguments

def \_\_init\_\_(self, dataList):

# Set data as instance variable, and assign it the value of dataList

self.data = dataList

# Define method that returns data: show

def show(self):

return self.data

# Define method that prints average of data: avg

def avg(self):

# Declare avg and assign it the average of data

avg = sum(self.data)/float(len(self.data))

# Return avg

return avg

# Instantiate DataShell taking integer\_list as argument: my\_data\_shell

my\_data\_shell = DataShell(integer\_list)

# Print output of your object's show method

print(my\_data\_shell.show())

# Print output of your object's avg method

print(my\_data\_shell.avg())

<script.py> output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

5.5

Awesome! We saw the return statement earlier in our functions exercise and similarly, the return statement is very useful when writing methods for our classes and methods. Think of it as a way to enable objects to talk to other objects, as they are able to return useful information when called.

# Ref :

# Return Statement III: A More Powerful DataShell

In this exercise our DataShell class will evolve from simply consuming lists to consuming CSV files so that we can do more sophisticated things.

For this, we will employ the return statement once again. Additionally, we will leverage some neat functionality from both the numpy and pandas packages.

Q :

* Import the numpy and pandas packages as np and pd, respectively.
* Create a class called DataShell with its initialization method, taking self and inputFile as arguments. Declare file as an instance variable and assign to it the value of the input argument inputFile.
* Define a method called generate\_csv(), taking self as argument. In the body of this method, declare an instance variable called data\_as\_csv and assign it to the value of pd.read\_csv(self.file). Finally, have generate\_csv()return the value of data\_as\_csv.
* Instantiate DataShell as data\_shell passing us\_life\_expectancy as an argument to the constructor. Declare a variable called df and assign to it the value of data\_shell.generate\_csv(). Finally, print the value of df.

# Load numpy as np and pandas as pd

import numpy as np

import pandas as pd

# Create class: DataShell

class DataShell:

# Initialize class with self and inputFile

def \_\_init\_\_(self, inputFile):

self.file = inputFile

# Define generate\_csv method, with self argument

def generate\_csv(self):

self.data\_as\_csv = pd.read\_csv(self.file)

return self.data\_as\_csv

# Instantiate DataShell with us\_life\_expectancy as input argument

data\_shell = DataShell(us\_life\_expectancy)

# Call data\_shell's generate\_csv method, assign it to df

df = data\_shell.generate\_csv()

# Print df

print(df)

<script.py> output:

country code year life\_expectancy

0 United States USA 1880 39.410000

1 United States USA 1890 45.209999

2 United States USA 1901 49.299999

3 United States USA 1902 50.500000

4 United States USA 1903 50.599998

5 United States USA 1904 49.599998

6 United States USA 1905 50.299999

7 United States USA 1906 50.099998

8 United States USA 1907 50.200001

9 United States USA 1908 51.900002

10 United States USA 1909 52.799999

11 United States USA 1910 51.799999

12 United States USA 1911 53.400002

13 United States USA 1912 54.099998

14 United States USA 1913 53.500000

15 United States USA 1914 54.599998

16 United States USA 1915 55.099998

17 United States USA 1916 54.200001

18 United States USA 1917 54.000000

19 United States USA 1918 47.200001

20 United States USA 1919 55.299999

21 United States USA 1920 55.400002

22 United States USA 1921 58.200001

23 United States USA 1922 58.099998

24 United States USA 1923 57.500000

25 United States USA 1924 58.500000

26 United States USA 1925 58.500000

27 United States USA 1926 57.900002

28 United States USA 1927 59.400002

29 United States USA 1928 58.299999

.. ... ... ... ...

87 United States USA 1986 74.772003

88 United States USA 1987 74.876999

89 United States USA 1988 74.988998

90 United States USA 1989 75.112999

91 United States USA 1990 75.251999

92 United States USA 1991 75.405998

93 United States USA 1992 75.568001

94 United States USA 1993 75.734001

95 United States USA 1994 75.903000

96 United States USA 1995 76.070000

97 United States USA 1996 76.230003

98 United States USA 1997 76.384003

99 United States USA 1998 76.531998

100 United States USA 1999 76.674004

101 United States USA 2000 76.815002

102 United States USA 2001 76.962997

103 United States USA 2002 77.121002

104 United States USA 2003 77.292000

105 United States USA 2004 77.474998

106 United States USA 2005 77.667000

107 United States USA 2006 77.859001

108 United States USA 2007 78.045998

109 United States USA 2008 78.223000

110 United States USA 2009 78.387001

111 United States USA 2010 78.539001

112 United States USA 2011 78.681999

113 United States USA 2012 78.820999

114 United States USA 2013 78.959999

115 United States USA 2014 79.099998

116 United States USA 2015 79.244003

[117 rows x 4 columns]

Great! Now you are starting to build classes with more sophisticated functionality!

# Ref :

# Data as Attributes

In the previous coding exercise you wrote a method within your DataShell class that returns a Pandas Dataframe.

In this one, we will cook the data into our class, as an instance variable. This is so that we can do fancy things later, such as renaming columns, as well as getting descriptive statistics.

The object us\_life\_expectancy is loaded and available in your workspace.

Q :

* Import the numpy and pandas packages as np and pd.
* Create a DataShell class with its initialization method, taking self and filepath as arguments.
* Set filepath as an instance variable, and assign it the value of the filepath parameter. Also set data\_as\_csv as an instance variable, assign it the value of pd.read\_csv(filepath).
* Instantiate DataShell as us\_data\_shell, passing the us\_life\_expectancy as input argument for the filepathparameter. Then print your objects' data\_as\_csv output.

# Import numpy as np, pandas as pd

import numpy as np

import pandas as pd

# Create class: DataShell

class DataShell:

# Define initialization method

def \_\_init\_\_(self, filepath):

# Set filepath as instance variable

self.filepath = filepath

# Set data\_as\_csv as instance variable

self.data\_as\_csv = pd.read\_csv(filepath)

# Instantiate DataShell as us\_data\_shell

us\_data\_shell = DataShell(us\_life\_expectancy)

# Print your object's data\_as\_csv attribute

print(us\_data\_shell.data\_as\_csv)

<script.py> output:

country code year life\_expectancy

0 United States USA 1880 39.410000

1 United States USA 1890 45.209999

2 United States USA 1901 49.299999

3 United States USA 1902 50.500000

4 United States USA 1903 50.599998

5 United States USA 1904 49.599998

6 United States USA 1905 50.299999

7 United States USA 1906 50.099998

8 United States USA 1907 50.200001

9 United States USA 1908 51.900002

10 United States USA 1909 52.799999

11 United States USA 1910 51.799999

12 United States USA 1911 53.400002

13 United States USA 1912 54.099998

14 United States USA 1913 53.500000

15 United States USA 1914 54.599998

16 United States USA 1915 55.099998

17 United States USA 1916 54.200001

18 United States USA 1917 54.000000

19 United States USA 1918 47.200001

20 United States USA 1919 55.299999

21 United States USA 1920 55.400002

22 United States USA 1921 58.200001

23 United States USA 1922 58.099998

24 United States USA 1923 57.500000

25 United States USA 1924 58.500000

26 United States USA 1925 58.500000

27 United States USA 1926 57.900002

28 United States USA 1927 59.400002

29 United States USA 1928 58.299999

.. ... ... ... ...

87 United States USA 1986 74.772003

88 United States USA 1987 74.876999

89 United States USA 1988 74.988998

90 United States USA 1989 75.112999

91 United States USA 1990 75.251999

92 United States USA 1991 75.405998

93 United States USA 1992 75.568001

94 United States USA 1993 75.734001

95 United States USA 1994 75.903000

96 United States USA 1995 76.070000

97 United States USA 1996 76.230003

98 United States USA 1997 76.384003

99 United States USA 1998 76.531998

100 United States USA 1999 76.674004

101 United States USA 2000 76.815002

102 United States USA 2001 76.962997

103 United States USA 2002 77.121002

104 United States USA 2003 77.292000

105 United States USA 2004 77.474998

106 United States USA 2005 77.667000

107 United States USA 2006 77.859001

108 United States USA 2007 78.045998

109 United States USA 2008 78.223000

110 United States USA 2009 78.387001

111 United States USA 2010 78.539001

112 United States USA 2011 78.681999

113 United States USA 2012 78.820999

114 United States USA 2013 78.959999

115 United States USA 2014 79.099998

116 United States USA 2015 79.244003

[117 rows x 4 columns]

Now your classes have the ability of storing data as instance variables, which means you can exercute methods on them!

# Ref :

# Renaming Columns

Methods can be especially useful to manipulate their object's data. In this exercise, we will create a method inside of our DataShell class, so that we can rename our data columns.

numpy and pandas are already imported in your workspace as np and pd, respectively.

Q :

* Create class DataShell, with initialization method, taking self and filepath as arguments. Set filepath and data\_as\_csv as instance variables.
* Define method rename\_column, with arguments self, column\_name, and new\_column\_name. Replace the column name of column\_name with that of new\_column\_name, this is already done for you.
* Instantiate DataShell as us\_data\_shell, taking us\_life\_expectancy as input to the constructor. Print your print the data type of your object's dataascsv attribute. You can do this by passing us\_data\_shell.data\_as\_csv.dtypesinto the print() function.
* Using your new method, rename your object's column 'code' to 'country\_code'. Once again, print the data type of your object's data\_as\_csv attribute.

# Create class DataShell

class DataShell:

# Define initialization method

def \_\_init\_\_(self, filepath):

self.filepath = filepath

self.data\_as\_csv = pd.read\_csv(filepath)

# Define method rename\_column, with arguments self, column\_name, and new\_column\_name

def rename\_column(self, column\_name, new\_column\_name):

self.data\_as\_csv.columns = self.data\_as\_csv.columns.str.replace(column\_name, new\_column\_name)

# Instantiate DataShell as us\_data\_shell with argument us\_life\_expectancy

us\_data\_shell = DataShell(us\_life\_expectancy)

# Print the datatype of your object's data\_as\_csv attribute

print(us\_data\_shell.data\_as\_csv.dtypes)

# Rename your objects column 'code' to 'country\_code'

us\_data\_shell.rename\_column('code', 'country\_code')

# Again, print the datatype of your object's data\_as\_csv attribute

print(us\_data\_shell.data\_as\_csv.dtypes)

<script.py> output:

country object

code object

year int64

life\_expectancy float64

dtype: object

country object

country\_code object

year int64

life\_expectancy float64

dtype: object

Amazing! Now you are able to employ your object's methods to modify its data. In what other ways might you be able to modify your object's data?

# Ref :

# Self-Describing DataShells

In this exercise you will add functionality to your DataShellclass such that it returns information about itself.

numpy and pandas are already imported in your workspace as np and pd, respectively.

Q :

* Create class DataShell, with initialization method, taking self and filepath as arguments. Set filepath and data\_as\_csv as instance variables.
* Create a method rename\_column (this is already done for you). Create a second method, get\_stats, which takes self as argument, and returns self.data\_as\_csv.describe().
* Instantiate DataShell as us\_data\_shell, taking us\_life\_expectancy as input to the constructor.
* Print the output of your object's get\_stats() method.

# Create class DataShell

class DataShell:

# Define initialization method

def \_\_init\_\_(self, filepath):

self.filepath = filepath

self.data\_as\_csv = pd.read\_csv(filepath)

# Define method rename\_column, with arguments self, column\_name, and new\_column\_name

def rename\_column(self, column\_name, new\_column\_name):

self.data\_as\_csv.columns = self.data\_as\_csv.columns.str.replace(column\_name, new\_column\_name)

# Define get\_stats method, with argument self

def get\_stats(self):

# Return a description data\_as\_csv

return self.data\_as\_csv.describe()

# Instantiate DataShell as us\_data\_shell

us\_data\_shell = DataShell(us\_life\_expectancy)

# Print the output of your objects get\_stats method

print(us\_data\_shell.get\_stats())

<script.py> output:

year life\_expectancy

count 117.000000 117.000000

mean 1956.752137 66.556684

std 34.398252 9.551079

min 1880.000000 39.410000

25% 1928.000000 58.500000

50% 1957.000000 69.599998

75% 1986.000000 74.772003

max 2015.000000 79.244003

Great! Not only does your DataShell class store data, it can also modify it, and output information about it! What other kinds of data could you store? And what other procedures could you exercute on such data?

# Naming classes

What's the best practice, according to PEP-8, for naming a class?

##### Answer the question

**50 XP**

##### Possible Answers

* 

lowercase

press1

* 

CamelCase

press2

* 

UPPERCASE

press3

* 

under\_score

press

That's right! CamelCase is a best practice for easy-to-read classes.

# Got Characters?

According to PEP-8, what is the maximum amount of characters per line of code?

##### Answer the question

**50 XP**

##### Possible Answers

* 

69.

press1

* 

70.

press2

* 

79.

press3

* 

80.

press

That's right! PEP-8 recommends a maximum of 79 characters per line of code!

# Which class has appropriate docstrings?

**Option A:**

class DataShell:

A simple class that brings a csv object in-memory as a

numpy matrix so you can perform operations on it.

def \_\_init\_\_(self, filename):

self.filename = filename

**Option B:**

class DataShell:

"""

A simple class that brings a csv object in-memory as a

numpy matrix so you can perform operations on it.

"""

def \_\_init\_\_(self, filename):

self.filename = filename

**Option C:**

class DataShell:

# A simple class that brings a csv object in-memory as a numpy matrix so you can perform operations on it.

def \_\_init\_\_(self, filename):

self.filename = filename

##### Answer the question

**50 XP**

##### Possible Answers

* 

Option A

press1

* 

Option B

press2

* 

Option C

Press

Yes Option B

# Why Inheritance?

Why is it useful to declare a class that inherits from an abstract class?

##### Answer the question

**50 XP**

##### Possible Answers

* 

You want to overwrite your existing base class.

press1

* 

You want to extend the functionality of your current class without overwriting it.

press2

* 

You want to make a copy of the class somewhere else.

Press

That's right! Inheritance allows us to pass features down into a child class.

# Is-A or Has-A

Which of the following represents a valid inheritance relationship?

##### Answer the question

**50 XP**

##### Possible Answers

* 

Cookie inherits from Cake.

press1

* 

Cake inherits from Cookie.

press2

* 

Dessert inherits from Cake.

press3

* 

Cake inherits from Dessert.

Press

That's right! Cake is a type of dessert.

# Ref :

# Animal Inheritance

In this exercise we will code a simple example of an abstract class, and two other classes that inherit from it.

To focus on the concept of inheritance, we will introduce another set of classes: Animal, Mammal, and Reptile.

More specifically, Animal will be our abstract class, and both Mammal and Reptile will inherit from it.

Q :

* Create a class called Animal with its initialization method, taking self and name as arguments. Declare name as an instance variable and assign it the value of the input argument name.
* Create two classes Mammal and Reptile, which inherit from Animal. For both Mammal and Reptile, include the initialization method, taking self, name, and animal\_type as arguments; then, animal\_type as an instance variable and assign it the value of the input argument animal\_type.
* Instantiate Mammal as daisy, passing 'Daisy' as the first argument 'dog' and as the second argument. Then instantiate Reptile as stella, passing 'Stella' as the first argument 'alligator' and as the second argument.
* Print both daisy and stella to explore their contents.

# Create a class Animal

class Animal:

def \_\_init\_\_(self, name):

self.name = name

# Create a class Mammal, which inherits from Animal

class Mammal(Animal):

def \_\_init\_\_(self, name , animal\_type):

self.animal\_type = animal\_type

# Create a class Reptile, which also inherits from Animal

class Reptile(Animal):

def \_\_init\_\_(self, name , animal\_type):

self.animal\_type = animal\_type

# Instantiate a mammal with name 'Daisy' and animal\_type 'dog': daisy

daisy = Mammal('Daisy', 'dog')

# Instantiate a reptile with name 'Stella' and animal\_type 'alligator': stella

stella = Reptile('Stella', 'alligator')

# Print both objects

print(daisy)

print(stella)

<script.py> output:

<\_\_main\_\_.Mammal object at 0x7f71341387b8>

<\_\_main\_\_.Reptile object at 0x7f7134138550>

Wow. You have now written an abstract class, and successfully employed inheritance, a powerful feature of object-oriented programming that helps you simplify code through re-usability!

# Ref :

# Vertebrate Inheritance

In the previous exercise, it seemed almost unnecessary to have an abstract class, as it did not do anything particularly interesting (other than begin to learn inheritance).

In this exercise, we will refine our abstract class and include some class variables in our abstract class so that they can be passed down to our other classes.

Additionally from inheritance, in this exercise we are seeing another powerful object-oriented programming concept: polymorphism. As you explore your code while writing the Mammal and Reptile classes, notice their differences. Because they both inherit from the Vertebrate class, and because they are different, we say that they are polymorphic. How cool!

Q :

* Create a class called Vertebrate with its initialization method, taking self and name as arguments. Declare spinal\_cord as a class variable, and set it to True. Declare name as an instance variable and assign it the value of the input argument name.
* Create two classes Mammal and Reptile, which inherit from Vertebrate. For both Mammal and Reptile, include the initialization method, taking self, name, and animal\_type as arguments; then, animal\_type as an instance variable and assign it the value of the input argument animal\_type. Additionally, include the instance variable temperature\_regulation, set it to True for class Mammal, and set it to False for class Reptile.
* Instantiate Mammal as daisy, passing 'Daisy' as the first argument 'dog' and as the second argument. Then instantiate Reptile as stella, passing 'Stella' as the first argument 'alligator' and as the second argument.
* Print both daisy and stella to explore their contents. Specifically, print their spinal\_cord and temperature\_regulation attributes.

# Create a class Vertebrate

class Vertebrate:

spinal\_cord = True

def \_\_init\_\_(self, name):

self.name = name

# Create a class Mammal, which inherits from Vertebrate

class Mammal(Vertebrate):

def \_\_init\_\_(self, name, animal\_type):

self.animal\_type = animal\_type

self.temperature\_regulation = True

# Create a class Reptile, which also inherits from Vertebrate

class Reptile(Vertebrate):

def \_\_init\_\_(self, name, animal\_type):

self.animal\_type = animal\_type

self.temperature\_regulation = False

# Instantiate a mammal with name 'Daisy' and animal\_type 'dog': daisy

daisy = Mammal('Daisy', 'dog')

# Instantiate a reptile with name 'Stella' and animal\_type 'alligator': stella

stella = Reptile('Stella', 'alligator')

# Print stella's attributes spinal\_cord and temperature\_regulation

print("Stella Spinal cord: " + str(stella.spinal\_cord))

print("Stella temperature regulation: " + str(stella.temperature\_regulation))

# Print daisy's attributes spinal\_cord and temperature\_regulation

print("Daisy Spinal cord: " + str(daisy.spinal\_cord))

print("Daisy temperature regulation: " + str(daisy.temperature\_regulation))

<script.py> output:

Stella Spinal cord: True

Stella temperature regulation: False

Daisy Spinal cord: True

Daisy temperature regulation: True

Awesome. Not only are you learning inheritance, but also polymorphism! Try extending the polymorphism of your (inheriting) classes by adding different class and instance variables.

# Ref :

# Abstract Class DataShell I

We will now switch back to working on our DataShell class. Specifically, we will create an abstract class, such that we can create other classes that then inherit from it!

For this reason, our abstract DataShell class will not do much, resembling some of the earlier exercises in this course.

Q :

* Import numpy as np and pandas as np.
* Create class DataShell with initialization method and input arguments self and inputFile. In the method body, define the instance variable file, and set it to the value of inputFile.
* Instantiate DataShell as my\_data\_shell, passing us\_life\_expectancy as input to the constructor.
* Print my\_data\_shell to explore its contents.

# Load numpy as np and pandas as pd

import numpy as np

import pandas as pd

# Create class: DataShell

class DataShell:

def \_\_init\_\_(self , inputFile):

self.file = inputFile

# Instantiate DataShell as my\_data\_shell

my\_data\_shell = DataShell(us\_life\_expectancy)

# Print my\_data\_shell

print(my\_data\_shell)

<script.py> output:

<\_\_main\_\_.DataShell object at 0x7fa868212400>

Great job! This exercise should feel a bit easier from the previous few. Get ready as we will build upon this abstract class to add inheritance, and begin to explore a new concept!

# Ref :

# Abstract Class DataShell II

Now that we have our abstract class DataShell, we can now create a second class that inherits from it.

Specifically, we will define a class called CsvDataShell. This class will have the ability to import a CSV file. In the following exercises we will add a bit more functionality to make our classes more sophisticated!

Q :

* Import numpy as np and pandas as np.
* Create class DataShell with initialization method and input arguments self and inputFile. In the method body, define the instance variable file, and set it to the value of inputFile. **All this is already done for you**.
* Create a second class, CsvDataShell, which inherits from DataShell. Add an initialization method, which takes as input arguments both self and inputFile. Declare an instance variable data and set it to the value of pd.read\_csv(inputFile).
* Instantiate CsvDataShell as us\_data\_shell, passing us\_life\_expectancy as input to the constructor, and print us\_data\_shell.data to explore its contents.

# Load numpy as np and pandas as pd

import numpy as np

import pandas as pd

# Create class: DataShell

class DataShell:

def \_\_init\_\_(self, inputFile):

self.file = inputFile

# Create class CsvDataShell, which inherits from DataShell

class CsvDataShell(DataShell):

# Initialization method with arguments self, inputFile

def \_\_init\_\_(self, inputFile):

# Instance variable data

self.data = pd.read\_csv(inputFile)

# Instantiate CsvDataShell as us\_data\_shell, passing us\_life\_expectancy as argument

us\_data\_shell = CsvDataShell(us\_life\_expectancy)

# Print us\_data\_shell.data

print(us\_data\_shell.data)

<script.py> output:

country code year life\_expectancy

0 United States USA 1880 39.410000

1 United States USA 1890 45.209999

2 United States USA 1901 49.299999

3 United States USA 1902 50.500000

4 United States USA 1903 50.599998

5 United States USA 1904 49.599998

6 United States USA 1905 50.299999

7 United States USA 1906 50.099998

8 United States USA 1907 50.200001

9 United States USA 1908 51.900002

10 United States USA 1909 52.799999

11 United States USA 1910 51.799999

12 United States USA 1911 53.400002

13 United States USA 1912 54.099998

14 United States USA 1913 53.500000

15 United States USA 1914 54.599998

16 United States USA 1915 55.099998

17 United States USA 1916 54.200001

18 United States USA 1917 54.000000

19 United States USA 1918 47.200001

20 United States USA 1919 55.299999

21 United States USA 1920 55.400002

22 United States USA 1921 58.200001

23 United States USA 1922 58.099998

24 United States USA 1923 57.500000

25 United States USA 1924 58.500000

26 United States USA 1925 58.500000

27 United States USA 1926 57.900002

28 United States USA 1927 59.400002

29 United States USA 1928 58.299999

.. ... ... ... ...

87 United States USA 1986 74.772003

88 United States USA 1987 74.876999

89 United States USA 1988 74.988998

90 United States USA 1989 75.112999

91 United States USA 1990 75.251999

92 United States USA 1991 75.405998

93 United States USA 1992 75.568001

94 United States USA 1993 75.734001

95 United States USA 1994 75.903000

96 United States USA 1995 76.070000

97 United States USA 1996 76.230003

98 United States USA 1997 76.384003

99 United States USA 1998 76.531998

100 United States USA 1999 76.674004

101 United States USA 2000 76.815002

102 United States USA 2001 76.962997

103 United States USA 2002 77.121002

104 United States USA 2003 77.292000

105 United States USA 2004 77.474998

106 United States USA 2005 77.667000

107 United States USA 2006 77.859001

108 United States USA 2007 78.045998

109 United States USA 2008 78.223000

110 United States USA 2009 78.387001

111 United States USA 2010 78.539001

112 United States USA 2011 78.681999

113 United States USA 2012 78.820999

114 United States USA 2013 78.959999

115 United States USA 2014 79.099998

116 United States USA 2015 79.244003

[117 rows x 4 columns]

Awesome. We are now seeing inheritance in action in the context of our DataShell class. What other classes might you create, that inherit from DataShell? Maybe ones that import TSV files? Maybe JSON?

# Ref :

# Composition and Inheritance I

As you may have noticed, we have already been using composition in our classes, we just have not been explicit about it. More specifically, we have been relying on functionality from the pandas package.

In this exercise, we will combine inheritance and composition as we define a class that 1) inherits from another class, and 2) uses functionality from other classes.

Q :

* Create abstract class DataShell, with class variable family set to 'DataShell', and with initialization method and instance variables; all this is already done for you.
* Create a second class, CsvDataShell, which inherits from DataShell.
* Define initialization method, with self, name, and filepath as input arguments. In the method body, declare data as instance variable and set it to pd.read\_csv(filepath). Declare a second instance variable stats and set it to self.data.describe().
* Instantiate CsvDataShell as us\_data\_shell, passing "US" and us\_life\_expectancy as inputs. Finally, print us\_data\_shell.stats to explore its contents.

# Define abstract class DataShell

class DataShell:

# Class variable family

family = 'DataShell'

# Initialization method with arguments, and instance variables

def \_\_init\_\_(self, name, filepath):

self.name = name

self.filepath = filepath

# Define class CsvDataShell

class CsvDataShell(DataShell):

# Initialization method with arguments self, name, filepath

def \_\_init\_\_(self, name, filepath):

# Instance variable data

self.data = pd.read\_csv(filepath)

# Instance variable stats

self.stats = self.data.describe()

# Instantiate CsvDataShell as us\_data\_shell

us\_data\_shell = CsvDataShell("US", us\_life\_expectancy)

# Print us\_data\_shell.stats

print(us\_data\_shell.stats)

<script.py> output:

year life\_expectancy

count 117.000000 117.000000

mean 1956.752137 66.556684

std 34.398252 9.551079

min 1880.000000 39.410000

25% 1928.000000 58.500000

50% 1957.000000 69.599998

75% 1986.000000 74.772003

max 2015.000000 79.244003

Amazing! Now you now how to recycle code from other classes through both inheritance and composition! What other classes might you want to inherit from? What other code might you want to use composition with?

# Ref :

# Composition and Inheritance II

In this exercise, we will create another class TsvDataShellthat inherits from our abstract class DataShell, which also uses composition in recycling functionality from pandasobjects.

Specifically, our new class will be able to read in TSV files, and also give us a description of the data it stores.

Q :

* Create abstract class DataShell, with class variable family set to 'DataShell', and with initialization method and instance variables. Create a second class, CsvDataShell, which inherits from DataShell, also with initialization method and instance variables. **All this is already done for you**.
* Create a third class, TsvDataShell, which inherits from DataShell. Define its initialization method, with self, name, and filepath as input arguments. In the method body, declare data as instance variable and set it to pd.read\_table(filepath). Declare a second instance variable stats and set it to self.data.describe().
* Instantiate CsvDataShell as us\_data\_shell, passing"US" and us\_life\_expectancy as input. Finally, print us\_data\_shell.stats to explore its contents. This is already done for you.
* Finally, instantiate TsvDataShell as france\_data\_shell, passing "France" and france\_life\_expectancy as inputs. Finally, print france\_data\_shell.stats to explore its contents.

# Define abstract class DataShell

class DataShell:

family = 'DataShell'

def \_\_init\_\_(self, name, filepath):

self.name = name

self.filepath = filepath

# Define class CsvDataShell

class CsvDataShell(DataShell):

def \_\_init\_\_(self, name, filepath):

self.data = pd.read\_csv(filepath)

self.stats = self.data.describe()

# Define class TsvDataShell

class TsvDataShell(DataShell):

# Initialization method with arguments self, name, filepath

def \_\_init\_\_(self, name, filepath):

# Instance variable data

self.data = pd.read\_table(filepath)

# Instance variable stats

self.stats = self.data.describe()

# Instantiate CsvDataShell as us\_data\_shell, print us\_data\_shell.stats

us\_data\_shell = CsvDataShell("US", us\_life\_expectancy)

print(us\_data\_shell.stats)

# Instantiate TsvDataShell as france\_data\_shell, print france\_data\_shell.stats

france\_data\_shell = TsvDataShell('France', france\_life\_expectancy)

print(france\_data\_shell.stats)

<script.py> output:

year life\_expectancy

count 117.000000 117.000000

mean 1956.752137 66.556684

std 34.398252 9.551079

min 1880.000000 39.410000

25% 1928.000000 58.500000

50% 1957.000000 69.599998

75% 1986.000000 74.772003

max 2015.000000 79.244003

country,code,year,life\_expectancy

count 200

unique 200

top France,FRA,1979,73.822998

freq 1

Congratulations on getting to the last interactive exercise! Before heading to the last video, compare the differences in printed output for us\_data\_shell and france\_data\_shell. Why might they be different?