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?