# Introduction

These case studies are designed to apply what has been learned to real-life scenarios. Although it is broken into chapters so that you can build the case studies bit by bit as you learn, it is likely that you will not have enough time during the course to complete most of the tasks. For this reason, the exercises are written in a way that will enable you to go through them independently. This way you can do them during class to reinforce what you have learned or do them later at a more casual pace to refresh your memory and extend what you have learned.

Some tasks do require knowledge beyond what will be taught in the course. I have tried to keep these to a minimum so we can focus on the basics, but if I have also marked them as ‘challenge’ questions so that you can skip them if you want to.

Alongside this guide, I have also provided the solutions in two formats. The first is a copy of the guide with the solutions (all code and the answers to questions). If you are new to Python, then you might want to work with this version so that you can see the code and try it for yourself. In this way you can focus on the meaning of the code without worrying about the (sometimes fiddly) details. If you are reasonably confident, however, then feel free to work out the code for yourself.

The solutions have also been provided for each chapter in .py files with names based on the question number. This is so that you can run them for yourself to prove that they work, and to see the expected behaviour for yourself. It also means that if you have not completed an exercise you can copy the solutions and move on to the next chapter so that you don’t get left behind.

Our case study will be a sales application involving price lists. These lists can be filtered, modified printed (to file or on screen). Price lists contain products of various types which have a variety of fields and different ways of calculating tax, etc. Price lists must be set up with dates, and context (such as company name, refund policy, etc).

You will find the final version of the case study in the ‘Final Version’ folder. Simply run pricelist.py.

Enjoy your exploration of Python.

# Chapter 1

A package for products, a package for price lists, a package to manage price lists

## Question 1

The first time you run a module Python might not be able to find it. This question will deal with this issue by using sys.path.

1. In REPL try to import the manage\_price\_lists module in the ‘Price List Application’ folder.

#### Results:

>>> import manage\_price\_lists

Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module>

import manage\_price\_lists

ModuleNotFoundError: No module named 'manage\_price\_lists'

1. In file explorer search for the manage\_price\_lists module. Make a note of it’s location (you can copy and paste it if you want). You will need to set up the path for Python to find this module.

#### Results:

It will depend on where you downloaded the course files. My file was located here:

'C:\Users\mtthw\Desktop\Business\Courses\DWT\Python\Python Intermedidate\Case Study\Price List Application’

1. Import sys and look at sys.path to see that your location is not currently in the various locations that Python recognises.

#### Results:

>>> import sys

>>> sys.path

['', 'C:\\Windows\\system32', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0\\python38.zip', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0\\DLLs', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0\\lib', 'C:\\Users\\mtthw\\AppData\\Local\\Microsoft\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_qbz5n2kfra8p0', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0', 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.8\_3.8.1776.0\_x64\_\_qbz5n2kfra8p0\\lib\\site-packages']

1. Use sys.path to append the location of this file to the path. (Hint: you can paste the location from File Explorer as long as you add r in front of the string to indicate it is a raw string rather than one containing \\)

#### Results:

>>> sys.path.append(r'C:\Users\mtthw\Desktop\Business\Courses\DWT\Python\Python Intermedidate\Case Study\Price List Application')

1. Check the last path in your sys.paths. It should be the path you just added.

#### Results:

>>> sys.path[-1]

'C:\\Users\\mtthw\\Desktop\\Business\\Courses\\DWT\\Python\\Python Intermedidate\\Case Study\\Price List Application'

1. Import manage\_price\_lists. It should now work.

#### Results:

>>> import manage\_price\_lists

managing price lists

## Question 2

Check out the directory structure of our application. Write some code in application that shows all the products we have. You will need to do an import.

1. In REPL, try to import products. Notice that it doesn’t recognise the products folder as a module.

#### Results:

>>> import products

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

import products

ModuleNotFoundError: No module named 'products'

1. Create a file called \_\_init\_\_.py that contains prints the message ‘Products available for this price list’ and save it in the products module.

#### Solution:

#Save this to a file called \_\_init\_\_.py in the products directory

Print(‘Products available for this price list’)

1. Now try to import the products package. (a package is simply a directory that contains an \_\_init\_\_.py file, and acts like a module, except that it can contain other modules).

#### Results:

>>> import products

Products available for this price list

Note: If this fails, make sure you have named the files correctly and placed them in the right directory. The name and location of files in a package is very significant.

# Chapter 2

## Question 1

We are building a retail application and would like to create a class that will be a simply shopping cart that will allow us to keep track of the objects we want to buy.

1. Create a class called Shopping cart. To start with just create an empty SET of products.

#### Results:

#### >>> mycart=ShoppingCart()

#### >>> mycart

#### <\_\_main\_\_.ShoppingCart object at 0x000001E313A6A730>

#### >>> mycart.\_products

#### set()

1. Alter the code so that the mycart instance is callable. Each time you call it a product should be added to the cart. Test it by adding “Peas” and “Corn” to your shopping list.

#### Results:

#### >>> mycart=ShoppingCart()

#### >>> mycart("Peas")

#### >>> mycart.\_products

#### {'Peas'}

#### >>> mycart("Corn")

#### >>> mycart.\_products

#### {'Peas', 'Corn'}

1. In the previous example both the class and the instance were callable functions to confirm this.

#### Results:

>>> callable(ShoppingCart)

True

>>> callable(mycart)

True

1. When we tested our shopping cart before we called the class. Where did we do this?
2. We also called the instance. What code did we use to do this?
3. When we defined our code, which line of code defined what to do when we called the class?
4. Which lines defined what to do when we called the instance?
5. To finish defining our cart, create two functions, one that will test if a product is already in our cart and another that will clear it. (set it to the empty set)

#### Results:

#### >>> mycart=ShoppingCart()

#### >>> mycart("Peas")

#### >>> mycart("Corn")

#### >>> mycart.contains("Peas")

#### True

#### >>> mycart.clear()

#### >>> mycart.contains("Peas")

#### False

#### >>> mycart.\_products

#### set()

## Question 2

The products we put in our shopping cart were just names. Real products are much more than just a name. Let’s create a product class that will include all kinds of information we might want to know about products. At a minimum, our product should include a name and a price. A class has already been created for you that does this. We want to expand this class to include more specific information about our products such as the weight, dimensions, colour, etc.

1. Test the products.py module by creating “corn” that costs 3.14 and then printing this product

#### Results:

#### >>> from products import Product

#### >>> corn=Product("corn",3.14)

#### >>> corn.printProduct()

#### Corn $3.14

1. Modify the constructor so that you can also specify a tuple called dimensions to include the weight, height, length, and width of the product (in that order). You can do this using variable arguments. Modify the printProduct function so that it will print these dimensions (as a tuple). Test it with the corn from the previous example as well as 500g of peas that cost $2.17 and a book that costs $10 and weighs 200g and is 12.5cm by 15cm by 0.5cm.

#### Results:

#### >>> corn=Product("corn",3.14)

#### >>> corn.printProduct()

#### Corn $3.14

#### Weight (g), height (cm), length(cm), width(cm): ()

#### >>> peas=Product("peas",2.17,500)

#### >>> peas.printProduct()

#### Peas $2.17

#### Weight (g), height (cm), length(cm), width(cm): (500,)

#### >>> book=Product("book",10,200,12.5,15,0.5)

#### >>> book.printProduct()

#### Book $10.00

#### Weight (g), height (cm), length(cm), width(cm): (200, 12.5, 15, 0.5)

1. We also need to allow for other properties of a product. For example, capsicums have colour. Chips have a flavour. Modify the constructor to also allow for these variable keyword arguments. In the printProduct function loop through each of these keyword argument pairs printing out the key and the value. Test it on the corn, but also on green and red capsicums which cost 0.50 and 0.75 each respectively (no dimensions), and 500g chicken flavoured chips that cost $3

#### Results:

#### >>> corn=Product("corn",3.14)

#### >>> corn.printProduct()

#### Corn $3.14

#### Weight (g), height (cm), length(cm), width(cm): ()

#### >>> redCapsicum=Product("capsicum",0.75,colour="red")

#### >>> redCapsicum.printProduct()

#### Capsicum $0.75

#### Weight (g), height (cm), length(cm), width(cm): ()

#### colour : red

#### >>> greenCapsicum=Product("capsicum",0.5,colour="green")

#### >>> greenCapsicum.printProduct()

#### Capsicum $0.50

#### Weight (g), height (cm), length(cm), width(cm): ()

#### colour : green

#### >>> chickenChips=Product("chips",3,500, flavour="chicken")

#### >>> chickenChips.printProduct()

#### Chips $3.00

#### Weight (g), height (cm), length(cm), width(cm): (500,)

#### flavour : chicken

1. As we will be using this often during this course, create a file called test\_products.py that creates all the products above, and prints out each product, with a blank line between them to make it more readable.

#### Results:

Corn $3.14

Weight (g), height (cm), length(cm), width(cm): ()

Peas $2.17

Weight (g), height (cm), length(cm), width(cm): (500,)

Book $10.00

Weight (g), height (cm), length(cm), width(cm): (200, 12.5, 15, 0.5)

Capsicum $0.75

Weight (g), height (cm), length(cm), width(cm): ()

colour : red

Capsicum $0.50

Weight (g), height (cm), length(cm), width(cm): ()

colour : green

Chips $3.00

Weight (g), height (cm), length(cm), width(cm): (500,)

flavour : chicken

## Question 3

We will do a quick investigation of lambda expressions by creating one that we can use to get a products name and then, when we add products to our shopping cart we can easily see the names of all the products we have added.

1. Create a lambda expression that simply fetches the name from a product.
2. Test the lambda expression by creating a product (e.g. “peas” from previous example), and then use the lambda expression to display its name.

#### Results:

>>> peas=Product("peas",2.17,500)

>>> name(peas)

'peas'

1. Import the shopping cart and add each of the items we created in the previous exercise to our cart.

#### Results:

>>> from ShoppingBasket import ShoppingCart

>>> cart=ShoppingCart()

>>> cart(Product("corn",3.14))

>>> cart(Product("peas",2.17,"500g"))

>>> cart(Product("book",10,200,12.5,15,0.5))

>>> cart(Product("capsicum",0.75,colour="red"))

>>> cart(Product("capsicum",0.5,colour="green"))

>>> cart(Product("chips",3,500,flavour="chicken"))

1. Loop through the products in our cart using the lambda expression to print out the names of these products.

#### Results:

chips

capsicum

corn

peas

capsicum

book

## Question 4

For debugging purposes, we want to create a function that we will use to trace other functions, even if we have no idea what they do.

1. Look at the trace.py file (as shown in the book), and make sure you understand what it is doing, especially the variable arguments.
2. Test it against our own Product function using chips.

#### Results:

#### >>> from products import Product

#### >>> trace(Product,"chips",3,500,flavour="chicken")

#### args = ('chips', 3, 500)

#### kwargs = {'flavour': 'chicken'}

#### result = <products.Product object at 0x0000022AD6AEE3A0>

#### <products.Product object at 0x0000022AD6AEE3A0>

# Chapter 3

## Question 1

We want to create a utility function to format our products data. Firstly we will create two functions. The first one to format a number, and the second one to format currency

1. In a file called formatting.py create a function called number\_format that formats a value a currency with two decimal places. Use 1.23 to test it.

#### Results:

#### >>> number\_format(1.23)

#### '$1.23'

1. Add another function called character\_format that formats a value by capitalizing it. and 'hEllO WOrld' to test it.

#### Results:

#### >>> character\_format('hEllO WOrld')

#### 'Hello world'

## Question 2

Now we will convert this code into a function factory. The idea is that our function factory will return a function called format based on which type of data is requested.

1. Add a new function at the start of formatting.py called format that accepts a parameter called type. To make sure it works just return the value of type.

#### Results:

#### >>> format('character')

#### 'character'

#### >>> format('currency')

#### 'currency’

1. Rename the number\_format and character\_format functions so they are both called format. Modify the format(type) function to return the appropriate function based on the type specified.
2. In the REPL create the number\_format function using the format(type) function and test this function using 1.23

#### Results:

>>> number\_format=format('currency')

>>> number\_format(1.23)

'$1.23'

1. In the same way create the character\_format function and test it using 'hEllO WOrld'

#### Results:

>>> character\_format=format('character')

>>> character\_format('hEllO WOrld')

'Hello world'

## Question 3

Now we are going to use this format factory as a decorator. First, we will create some getters and setters for price and name in our Products class.

1. Add getPrice and getName that will return the name and price of our product (unformatted). Test this by creating and printing our chips product

#### Results:

#### >>> chips=Product("chips",3,500,flavour="chicken")

#### >>> chips.getName()

#### 'chips'

#### >>> chips.getPrice()

#### 3

1. Modify the printProduct function to use these functions instead of fetching the values directly. Again use chips to test this.

#### Results:

#### >>> chips.printProduct()

#### chips 3

#### Weight (g), height (cm), length(cm), width(cm): (500,)

#### flavour : chicken

## Question 4

Note that in the previous question the price and product name are now both unformatted. What we want to do is create a decorator which we can apply to any function that will automatically format the data the function returns.

1. In our formatting.py module create a function called format\_as\_character that accepts a function as a parameter. It should use the factory to create a function that formats the character type. It should then return a function that wraps the formatting function around the function that was passed in as a parameter.
2. In products.py, import all the functions from formatting. Use the format\_as\_character function as a decorator for getName().
3. In REPL test the getName function to make sure it is now formatted.

#### Results:

>>> chips=Product("chips",3,500,flavour="chicken")

>>> chips.getName()

'Chips'

1. Similarly, create a format\_as\_currency decorator
2. Use the decorator on getPrice() and test it in REPL

#### Results:

#### >>> chips=Product("chips",3,500,flavour="chicken")

#### >>> chips.getPrice()

#### '$3.00'

1. Now make sure that the result of printing a product is also formatted.

#### Results:

#### >>> chips.printProduct()

#### Chips $3.00

#### Weight (g), height (cm), length(cm), width(cm): (500,)

#### flavour : chicken

## Question 5

Nesting your functions inside decorators can lead to difficulties when dealing with metadata. We need to fix this if people are likely to use facilities such as ‘help’ to work out what your module does.

1. Use help to find out about your getName() and getPrice() functions

#### Results:

#### >>> help(Product.getName)

#### Help on function format\_result in module formatting:

#### format\_result(\*args, \*\*kwargs)

#### >>> help(Product.getPrice)

#### Help on function format\_result in module formatting:

#### format\_result(\*args, \*\*kwargs)

1. Manually define the name and doc of the format\_result function to be the name and doc of the getName function

#### Results:

#### >>> help(Product.getName)

#### Help on function getName in module formatting:

#### getName(\*args, \*\*kwargs)

#### >>> help(Product.getPrice)

#### Help on function getPrice in module formatting:

#### getPrice(\*args, \*\*kwargs)

1. You might also want to try using the functools.wraps decorator to achieve the same result.

## Question 6

This is a challenge questions for those who have some spare time and want to go a bit further. Your challenge is to turn the two decorators into a single decorator with a parameter to determine what type to use (as shown in the results below). Test this with printProduct() as before to make sure it still works. The only clue I will give is that it is similar to the technique using in validating arguments. You will need to wrap the decorator that formats the results inside a decorator factory that handles the type of formatting you want.

#### Results:

#### @format\_function('character')

#### def getName(self):

#### return self.\_name

#### @format\_function('currency')

#### def getPrice(self):

#### return self.\_price

# Chapter 4

## Question 1

We would like each product instance to have a serial\_number which will increment each time we create a new product. To do this we will need to keep track of the last\_serial\_number in the class

1. Create a class attribute called last\_serial\_number which we set to 0 initially. Modify the initialisation method to increment the last\_serial\_number before assigning it to the seral\_number instance attribute. (Don’t forget to use the class name as a prefix to reference the class attribute).
2. Create two products and check their serial numbers.

#### Results:

>>> corn=Product("corn",3.14)

>>> peas=Product("peas",2.17,500)

>>> corn.\_serial\_number

1

>>> peas.\_serial\_number

2

>>> Product.last\_serial\_number

2

1. Modify the printProduct to print out a products serial number before any other information. Run the test\_products.py script from Chapter 2 to confirm that each product has an appropriate serial number.

#### Results:

#### Product number: 1

#### Corn $3.14

#### Weight (g), height (cm), length(cm), width(cm): ()

#### Product number: 2

#### Peas $2.17

#### Weight (g), height (cm), length(cm), width(cm): (500,)

#### …

## Question 2

We would like a method that returns the last serial number that was assigned to a product

1. Why is this different from a normal instance method?
2. Which option is do you think will be better to achieve the result we are after, a static method or a class method?
3. Create the appropriate method and run the test\_product script. Once you have done this, use the new method to display the last serial number.

#### Results:

>>> Product.\_get\_last\_serial\_number()

6

## Question 3

We would like to be able to create products using methods that cater for the different kinds of information those products might have.

1. Create a class method called basic\_product that requires just a name and price.
2. Modify the test\_products script to use this method to create the corn product
3. Create another named constructor called product\_with\_dimensions to take care of products that have dimensions.
4. Modify the test\_products script to use this method to create the peas and the book

## Question 4

At the moment we can do almost anything to the attributes of a product. To help manage products we should not allow people to change the name of a product. We also need restrictions on the price of products. With this in mind, we will make both the name and the price of a product a property.

1. Assuming we have just run the test\_products script above, In REPL set the price of corn to -$5 and the name to “nothing”

#### Results:

>>> corn.\_price=-5

>>> corn.\_name="nothing"

>>> corn.printProduct()

Product number: 1

Nothing $-5.00

Weight (g), height (cm), length(cm), width(cm): ()

1. Change the getName and getPrice functions to name and price and prefix them with the @property descriptor. Modify the print product function to use the properties we have just defined. Make sure that the test\_products script still works
2. In REPL try to change the name and price ***properties*** (not the \_name and \_price attributes).

#### Results:

>>> corn.price=-5

Traceback (most recent call last):

File "<pyshell#7>", line 1, in <module>

corn.price=-5

AttributeError: can't set attribute

>>> corn.name="nothing"

Traceback (most recent call last):

File "<pyshell#8>", line 1, in <module>

corn.name="nothing"

AttributeError: can't set attribute

1. Define a setter for price. Make sure it raises a value error if the price is less than 0.
2. Run the test\_product script. In REPL change to price of corn to 3 and confirm that it worked, then try to change it to -5

#### Results:

>>> corn.price=3

>>> corn.printProduct()

Product number: 1

Corn $3.00

Weight (g), height (cm), length(cm), width(cm): ()

>>> corn.price=-5

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

corn.price=-5

File "D:/Case Study/Solutions for each chapter/Chapter 4/4d\products.py", line 28, in price

raise ValueError("Negative values not permitted")

ValueError: Negative values not permitted

>>>

## Question 5

We have applied validation to make sure that we can’t change the values to something inappropriate, but what about when we construct the original product?

1. Create the corn product with a negative value using the constructor. Why has this worked?

#### Results:

>>> corn=Product.basic\_product("corn",-5)

>>> corn.printProduct()

Product number: 7

Corn $-5.00

Weight (g), height (cm), length(cm), width(cm): ()

>>>

1. Modify the constructor to use the price properties (rather than the \_price attribute). Confirm that you can no longer create a product with a negative value.

#### Results:

>>> corn=Product.basic\_product("corn",-5)

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

corn=Product.basic\_product("corn",-5)

File "D:/Case Study/Solutions for each chapter/Chapter 4/5\products.py", line 48, in basic\_product

return cls(name,price)

File "D:/Case Study/Solutions for each chapter/Chapter 4/5\products.py", line 8, in \_\_init\_\_

self.price=price

File "D:/Case Study/Solutions for each chapter/Chapter 4/5\products.py", line 28, in price

raise ValueError("Negative values not permitted")

ValueError: Negative values not permitted

>>>

## Question 6

Properties often behave differently when we have subclasses. The easiest way to handle this is to make sure that our properties don’t do the work, they just call a method which does the work. This is what is meant by the template pattern. Doing it this means that our subclasses can override these methods to change the behaviour of the properties rather than the properties themselves.

1. A subclass called cheap\_products has been created for you. It’s only purpose is to make sure than a product can not cost more than $10 (and no be negative either). We have attempted to do this by overriding the definition of our price property as follows:

class MyProduct(Product):

@property

@format\_as\_currency

def price(self):

return self.\_price

@price.setter

def price(self,value):

if 0<=value<=10:

raise ValueError("Negative values not permitted")

self.\_price=value

Run this module and then In REPL create an instance of this this subclass called corn with a price of $15 and again with a price of -$5 to confirm that this technique doesn’t pick check the prices properly.

#### Results:

>>> corn=MyProduct("corn",-5)

>>> corn=MyProduct("corn",15)

>>>

1. In the products module, modify the setter so that validation is done in a separate function called validate\_price. Now modify cheap\_products so that it overrides validate\_price rather than the price property itself. Test it in REPL to confirm it is now working.

# Chapter 5

## Question 1

For debugging purposes, when we print out a representation of a product, we want to see all the possible properties of a product and their values even if they are empty, along with any custom properties.

1. Run the test\_product module from the previous chapter.
2. Using corn as an example print the product compare this with what happens when you call the in-built repr() and str() functions

#### Results:

>>> print(corn)

<products.Product object at 0x0000027522DAF0C8>

>>> str(corn)

'<products.Product object at 0x000001BF2F93F288>'

>>> repr(corn)

'<products.Product object at 0x000001BF2F93F288>'

>>>

1. Override the \_\_repr\_\_() function using PrintProduct as the basis for our code. The function returns a string. Instead of printing the values we need to concatenate them to our products description (with a space in between). Any numerical data can be converted to a string with the str() function. You should add new lines using “\n” and for neatness spread you commands over more than one line using the \ continuation character. Don’t forget to return the description at the end of the function.
2. Modify our test\_products module to print out the representation.
3. Run the test\_products to makes sure it is still printing correctly.
4. Again using corn as an example print the product compare this with what happens when you call the in-built repr() and str() functions

#### Results:

>>> repr(corn)

Product number: 1

Corn $3.14

Weight (g), height (cm), length(cm), width(cm): ()

>>> str(corn)

Product number: 1

Corn $3.14

Weight (g), height (cm), length(cm), width(cm): ()

>>>

1. Override the \_\_str\_\_() function so that it only prints out the name and price of the product when you call print it.

#### Results:

#### >>> str(corn)

#### 'Corn $3.14'

## Question 2

In chapter 2 we created a shopping basket. Let’s have a look at how collections and their contents are printed, and then use the Table class discussed in pages 145-146 to print out the contents of our shopping cart in a tabular format.

1. In test\_products, import ShoppingCart from the ShoppingBasket module. Create a ShoppingCart called test\_cart and add corn and peas to it. Remember that the class is callable so all you need to do is call test\_cart in order to add a product to it.
2. In REPL, pint out the test\_cart, then print out the test\_cart contents.

#### Results:

>>> print(test\_cart)

<ShoppingBasket.ShoppingCart object at 0x000001DFDEDD6EC8>

>>> print(test\_cart.\_products)

{Product number: 1

Corn $3.14

Weight (g), height (cm), length(cm), width(cm): ()

, Product number: 2

Peas $2.17

Weight (g), height (cm), length(cm), width(cm): (500,)

}

>>>

1. Observe that when you are printing the contents of a collections it uses the products repr() not str().

## Question 3

We want to print out the details of our shopping cart in tabular form. We can use the code in the case study to do this.

1. Open the tabular\_data module. Read the code and then run the module.
2. Test it out in the REPL by using the code in the book to create a table of first and last names

#### Results:

>>> t = Table(['First name', 'Last name'],

['Fred', 'George', 'Scooby'],

['Flintstone', 'Jetson', 'Doo'])

>>> print(str(t))

First name Last name

========== ==========

Fred Flintstone

George Jetson

Scooby Doo

>>> print(repr(t))

Table(header=['First name', 'Last name'])

>>>

1. In REPL import test\_products. Create a comprehension that generates a list of product names from our test cart.

#### Results:

>>> [p.name for p in test\_cart.\_products]

['Corn', 'Peas']

1. Create a comprehension that generates a list of product names from our test cart.

#### Results:

>>> [p.price for p in test\_cart.\_products]

['$3.14', '$2.10']

1. Create a table based on these comprehensions and print it.

#### Results:

>>> from tabular\_data import Table

>>> t = Table(['Product Name', 'Product Price'],

[p.name for p in test\_cart.\_products],

[p.price for p in test\_cart.\_products])

>>> print(t)

Product Name Product Price

============ =============

Corn $3.14

Peas $2.17

>>>

1. Now that we have tested it, create a str() function in our ShoppingBasket that creates a table based on the products names and price and returns it as a string. When you run the test\_products module and print out our test\_cart it should display the names and prices of the products in tabular format.

#### Results:

>>> print(test\_cart)

Product Name Product Price

============ =============

Peas $2.17

Corn $3.14

>>>

# Chapter 6

## Question 1

When we print out the contents of our shopping cart we would also like to show the total cost at the end.

1. Run test\_product and in REPL calculate the total of the test\_cart. (Hint: Use the \_price attribute not the price property as this is a formatted string).
2. Check the accuracy of the calculation
3. We need to make price a decimal. Fortunately this is quite simple. Now that price is a property all we need to do is make sure that we convert the value in the setter to a string and then use the constructor to create a Decimal based on this number. Test it by rerunning the test\_products module and calculating the total.

#### Results:

>>> sum([p.\_price for p in test\_cart.\_products])

Decimal('5.31')

>>>

1. Now that our sum calculation is working add it (along with a suitable title) to the end of the string that str() returns in the Shopping. Rerun the test\_product module and print the test\_cart.

#### Results:

>>> print(test\_cart)

Product Name Product Price

============ =============

Peas $2.17

Corn $3.14

Total: 5.31

>>>

## Question 2

We are going to use dates in our application, but before we do let’s get familiar with how they work.

#### In REPL import the datetime module and put today’s date in a variable called today

#### Results:

#### >>> today

#### datetime.datetime(2021, 10, 7, 0, 45, 20, 963151)

#### Print out today with str, and in iso and calendar format. Also print it out in full. (You may need to check p179 of the manual or go to the documentation online).

#### Results:

#### '2021-10-07 00:45:20.963151'

#### '2021-10-07T00:45:20.963151'

#### (2021, 40, 4)

#### 'Thursday 07 October 2021'

#### Create a variable called delivery\_date for 7 days time, at 6pm. Explicitly name the parameters to make it clear which numbers are for the year, month, etc. Display the results as a string.

#### Results:

#### '2021-10-14 06:00:00'

#### Create a variable called duration that stores the time between today’s date and the delivery date. Display it as a string and then show the number of days in this duration.

#### Results:

#### '7 days, 5:14:39.036849'

#### 7

#### Extract the time component from the delivery date and display it as a string. Now display it in 12 hours format with am/pm indicator. (You may need to search the documentation to find out which symbols to use in the format string).

#### Results:

18:00:00

06:00 PM

## Question 3

Now that we are familiar with dates, we want to be able to checkout our shopping cart. That is, we want to create an order based on the products in our shopping cart. Our order will also include the date the order was placed and a pick up (or delivery) date.

1. Create a checkout module that contains a class called order. The constructor should accept two parameters, a shopping cart and the date and time the order will be picked up or delivered. It should keep a local copy of the product from the shopping cart, record the delivery date, and automatically populate as attribute called order\_date with today’s date.
2. In order to make the printout neat, use the same str() function as you did for your shopping cart. Run the checkout module and in REPL import test\_products so that we can create an order based on test\_cart. Also import datetime so that we can specify a pick up date of 6pm in 7 days. Print out the order and delivery date explicitly to make sure they are correct.

#### Results:

#### >>> from test\_products import test\_cart

#### >>> import datetime

#### >>> test\_order=order(test\_cart,datetime.datetime(year=2021,month=10,day=14,hour=18))

#### >>> print(test\_order)

#### Product Name Product Price

#### ============ =============

#### Corn $3.14

#### Peas $2.17

#### Total: 5.31

#### >>> test\_order.\_order\_date

#### datetime.datetime(2021, 10, 7, 2, 9, 54, 869383)

#### >>> test\_order.\_delivery\_date

#### datetime.datetime(2021, 10, 14, 18, 0)

1. Once you have testing creating an order in the REPL, move the code that creates the test\_order into test\_products. Don’t forget to do the necessary imports. Run test\_products and print out the test\_order it to make sure it works.
2. Modify the str() function to display the order and delivery dates (in full format without the time) at the beginning. After displaying the total, include how many days till delivery / pickup and the time pickup or delivery should happen. Make sure it displays it in 12 hour time.

#### Results:

Order Date: Thursday 07 October 2021

Product Name Product Price

============ =============

Peas $2.17

Corn $3.14

Total: 5.31

Your delivery is due in 7 days and should be picked up (or delivered) at 06:00 PM

## Question 4 (Optional)

If you would like to see a physical representation of the inaccuracies of floats work your way through the case study from page 188. The purpose is to display a whole lot of points which will be white if they are above a given line, black if they are below the line and grey if they are on a line.

1. Run the collinear\_using\_float module. You should see a whole lot of numbers. These are the points we will be mapping on our diagram as being above (+1), below(+1), or on the line (0). With a completely accurate results the only pair that should give a 0 is (0.5,0.5)

#### Results:

orientation((0.5, 0.49999999999999 ) q, r) -> -1

orientation((0.5, 0.49999999999999006) q, r) -> -1

orientation((0.5, 0.4999999999999901 ) q, r) -> -1

…

orientation((0.5, 0.4999999999999999 ) q, r) -> 0

orientation((0.5, 0.49999999999999994) q, r) -> 0

orientation((0.5, 0.5 ) q, r) -> 0

orientation((0.5, 0.5000000000000001 ) q, r) -> 0

orientation((0.5, 0.5000000000000002 ) q, r) -> 0

…

orientation((0.5, 0.5000000000000098 ) q, r) -> 1

orientation((0.5, 0.5000000000000099 ) q, r) -> 1

orientation((0.5, 0.50000000000001 ) q, r) -> 1

1. To see this graphically run collinear\_graphically\_using\_float and open the file above\_below\_float.bmp that it generates.
2. Now try running the version that uses Fractions, collinear\_using\_fractions.

#### Results:

orientation((0.5, 0.49999999999999 ) q, r) -> -1

orientation((0.5, 0.49999999999999006) q, r) -> -1

orientation((0.5, 0.4999999999999901 ) q, r) -> -1

…

orientation((0.5, 0.4999999999999999 ) q, r) -> 0

orientation((0.5, 0.49999999999999994) q, r) -> 0

orientation((0.5, 0.5 ) q, r) -> 0

orientation((0.5, 0.5000000000000001 ) q, r) -> 0

orientation((0.5, 0.5000000000000002 ) q, r) -> 0

…

orientation((0.5, 0.5000000000000098 ) q, r) -> 1

orientation((0.5, 0.5000000000000099 ) q, r) -> 1

orientation((0.5, 0.50000000000001 ) q, r) -> 1

1. Yes there are clearly still inaccuracies, but check out what happens if you generate the file above\_below\_fractions.bmp using collinear\_graphical\_using\_fractions. It clearly illustrates how much more accurate fractions are than floating points in python.

# Chapter 7

## Question 1

This question is based on the case study on pages 245 to 247 of the manual. Solutions can be found there. The idea is to build a class that will iterate over a (potentially infinite) set of events and return information about that event. For example we might want to keep an eye on the vital statistics of our computer. In our case we are simply going to iterate over randomly generated number.

1. Our iteration will be driven by the current date and time. Create a new module and import datetime. At this stage simply have the module create a variable called timestamps and set it today’s date and then print it out.

#### Results:

#### 2021-10-07 05:53:32.072688

#### >>>

1. To be an iterator, we must generate many values. Use the extended iter() function on datetime.datetime.today to do this for ever (ie the second parameter should be None). Now that we have more than one value write a loop to print the timestamp and take a 1 second break (using time.sleep) before moving on to the next one. Run the module and when you are satisfied that it is working hit Ctrl+C to kill it.

#### Results:

#### 2021-10-07 06:12:58.519691

#### 2021-10-07 06:12:59.576990

#### 2021-10-07 06:13:00.665225

#### 2021-10-07 06:13:01.800023

#### 2021-10-07 06:13:02.820222

#### Traceback (most recent call last):

#### File "D:/Case Study/Solutions for each chapter/Chapter 7/1b/sensor.py", line 8, in <module>

#### time.sleep(1)

#### KeyboardInterrupt>>>

1. iter(datetime.datetime.today,None) is the iterator. To see this simply show it’s directory and observe that it has both \_\_iter\_\_ and \_\_next\_\_ functions.

#### Results:

>>> dir(iter(datetime.datetime.today,None))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_']

>>>

1. We need another iterator that generates our sensor data (which is just a random number in our case, but it could be any data we want to track). Create a class called Sensor which has the two functions essential to an iterator. The \_\_iter\_\_ function should just return itself, and the \_\_next\_\_ function should just generate a random number using the built-in random module. Use zip to combine the timestamps and the sensor data and print out both the stamp and value each time.

#### Results:

2021-10-07 06:38:38.642464 0.15337551717476516

2021-10-07 06:38:39.700536 0.014841885597338567

2021-10-07 06:38:40.738043 0.11801862783160522

2021-10-07 06:38:41.826875 0.572378694206436

2021-10-07 06:38:42.861766 0.601055752750746

2021-10-07 06:38:43.888005 0.13342380336226312

2021-10-07 06:38:44.992839 0.27999248521595366

2021-10-07 06:38:46.097497 0.5507154135684557

Traceback (most recent call last):

File "D:/Case Study/Solutions for each chapter/Chapter 7/1b/sensor.py", line 17, in <module>

time.sleep(1)

KeyboardInterrupt

>>>

1. The final example in the book also uses the itertools.isslice() function to just fetch the first 10 rows (that than having to kill it manually) but since the purpose is to monitor sensor data ad infinitum this kind of defeat the purpose.

# Chapter 8

## Question 1

We have named constructors to take care of the different types of products. It might be more helpful if we set these up as subtypes

1. In the products module, remove the named constructors and change the name of the Products class to basic\_product instead. You will also need to change the prefix in front of our last\_serial\_number class variable. Run this module and in REPL create corn as a basic product to test that it works.

#### Results:

#### >>> corn=basic\_product("corn",3.14)

#### >>> print(repr(corn))

#### Product number: 1

#### Corn $3.14

#### Weight (g), height (cm), length(cm), width(cm): ()

1. Create two subtypes called product\_with\_dimensions and product\_with\_keyword, copying and pasting the init and repr methods to both subtypes. Be careful with indentation, we want to make sure that each subtype is defined in the module itself not inside the basic product.
2. Now we can modify each subclass to be specific. Do each one and test it before moving on to the next. For basic\_product remove anything in init or repr that has to do with the dimensions and keyword arguments. Test this on corn.

#### Results:

#### >>> corn=basic\_product("corn",3.14)

#### >>> print(repr(corn))

#### Product number: 1

#### Corn $3.14

1. Now modify products with dimensions by removing anything that has to do with the keyword arguments. Test this on peas.

#### Results:

#### >>> peas=product\_with\_dimensions("peas",2.17,500)

#### >>> print(repr(peas))

#### Product number: 1

#### Peas $2.17

#### Weight (g), height (cm), length(cm), width(cm): (500,)

1. Finally modify our test\_products module by changing any reference to Product into a reference to the appropriate type or subtype. Don’t forget to also change the import statements to import all of our classes.

#### Results:

>>>

= RESTART: D:\Case Study\Solutions for each chapter\Chapter 8\1\test\_products.py

Product number: 1

Corn $3.14

Product number: 2

Peas $2.17

Weight (g), height (cm), length(cm), width(cm): (500,)

Product number: 3

Book $10.00

Weight (g), height (cm), length(cm), width(cm): (200, 12.5, 15, 0.5)

Product number: 4

Capsicum $0.75

Weight (g), height (cm), length(cm), width(cm): ()

colour: red

…

## Question 2

You will notice that each of the init and repr methods repeat a lot of code that the basic subtype already has. We can utilise this by calling super().

1. Modify the constructor for product\_with\_dimensions by calling the init methods for super, passing in the name and price. Remove (or comment out) the lines that we no longer need. Run test\_products to make sure that test\_products still works.
2. Do the same modification to product\_with\_keywords and check to make sure test\_products still works.
3. In a similar way modify the repr method in both subclasses to make use of the repr method in the parent class. Yet again check that test\_products still works.

## Question 3

We can refine our code even further. The subclass product\_with\_keywords deals with dimensions as well. Maybe we can refine this subtype so that it only deals with keywords, then if a product has both dimensions and keywords we can make a third subclass that is simply a subclass of both product\_with\_dimensions and product\_with\_keywords.

1. In the product\_with\_keywords class, remove all the code involved with dimensions. Run the test\_products module and make sure that the only product that fails is the chickenChips Ithe only one with both dimensions and keywords).

#### Results:

#### Product number: 1

#### Corn $3.14

#### …

#### Product number: 5

#### Capsicum $0.50

#### colour: green

#### Traceback (most recent call last):

#### File "D:\Case Study\Solutions for each chapter\Chapter 8\3\test\_products.py", line 26, in <module>

#### chickenChips=products.product\_with\_keywords("chips",3,500, flavour="chicken")

#### TypeError: \_\_init\_\_() takes 3 positional arguments but 4 were given

1. Define a new subtype called product\_full that inherits both product\_with\_dimensions and product\_with\_keyword. Create a constructor that accepts name, price, dimensions and keywords). For now, just assign the values directly (don’t use super() yet).

Change chickenChips in the test\_products module to use products\_full and test it to make sure that it works.

#### Results:

…

Product number: 6

Chips $3.00

flavour: chicken

Weight (g), height (cm), length(cm), width(cm): (500,)

…

>>>

1. What do you notice about what was printed out? We will investigate how this worked in the next question

## Question 4

There are several things we need to understand when it comes to inheritance (particularly multiple inheritance). In the previous question we saw that repr seemed to somehow magically know that it needed to print out all the attributes even though we didn’t tell it to, and none of our subtypes did either. We will investigate this, and use what we have learned to good effect in simplifying our constructor.

1. Assuming you have just run the results of the previous question, explicitly run the repr(chickenChips) in REPL. Observe that it has created a string with all the information from basic product, product\_with\_keywords and product\_with\_dimensions in that order. Now check out the mro (method resolution order) for this class. Try to consolidate this with the results you see on the screen.

#### Results:

>>> import products

>>> products.product\_full

<class 'products.product\_full'>

>>> products.product\_full.\_\_mro\_\_

(<class 'products.product\_full'>, <class 'products.product\_with\_dimensions'>, <class 'products.product\_with\_keywords'>, <class 'products.basic\_product'>, <class 'object'>)

1. To help make it clear what is going on here I have created another version of products where the repr for product\_full has been declared explicity to call super() (we don’t need to do this, but for the purposes of illustration it should make things clear). I then added trace comments in each of the repr definitions to show you if and when each version of repr is called.
   * 1. Open the file and have a quick look
     2. Run this module.
     3. Create the chicken chips using this modules product\_full type
     4. Print out the repr for this product.

#### Results:

>>> chickenChips=product\_full("chips",3,500, flavour="chicken")

>>> print(repr(chickenChips))

start product full.

start product with dimension.

start product with keywords.

start product.

end product:

Product number: 1

Chips $3.00

end product with keywords:

Product number: 1

Chips $3.00

flavour: chicken

end product with dimension:

Product number: 1

Chips $3.00

flavour: chicken

Weight (g), height (cm), length(cm), width(cm): (500,)

end product full:

Product number: 1

Chips $3.00

flavour: chicken

Weight (g), height (cm), length(cm), width(cm): (500,)

Product number: 1

Chips $3.00

flavour: chicken

Weight (g), height (cm), length(cm), width(cm): (500,)

>>>

1. You can think of super().\_\_repr\_\_() as calling the method from the next class in the mro. If you want to call a specific version of the repr then super can take two optional parameters, the first specifies which type is providing the mro and the second is the instance you want to call. For example try calling super(basic\_product,chickenChips).\_\_repr\_\_()
2. Now lets see what happens when I use super() in my constructor.

# Chapter 9

## Question 1

In order to work with our shopping cart we need to think of it as a collection of products. This suggests many methods we need to provide in our class. Since there is a lot of work to be done lets take it a step at a time. We will start with the basics. We need to check to see if our shopping cart contains a particular product. This is known as the container protocol. Not that our collecton is based on a set (not a list) so the sequence protocol does not apply.

1. We already have a contains method, but this is misleading as it is not using the name of the product to do the comparison. Run test\_product and confirm that it is matching using the name of the variable we assigned it to in test\_products not the name of the product. In addition to this try using the in operator to ask the same questoin.

#### Results:

#### >>> test\_cart.contains('Peas')

#### False

#### >>> test\_cart.contains(peas)

#### True

>>> 'Peas' in test\_cart

Traceback (most recent call last):

File "<pyshell#115>", line 1, in <module>

'Peas' in test\_cart

TypeError: argument of type 'ShoppingCart' is not iterable

1. Define a new method called \_\_contains\_\_ passing in the name of the product we are looking for and checking to see if this is a name of one of the products in our shopping cart. Remove the other method as it is not useful.

#### Results:

#### >>> test\_cart.\_\_contains\_\_('Peas')

#### True

#### >>> test\_cart.\_\_contains\_\_(peas)

#### False

#### >>>

1. One reason we are using the ‘dunder’ notation is that this is a special internal method that is used for in.

#### Results:

>>> 'Peas' in test\_cart

True

>>>

## Question 2

The pattern we have seen in the first question is the same as we will see throughout the exercises. The protocol will suggest some functionality and describe and internal ‘dunder’ method to write. You write this method and it will override the default functionality, and in most cases will simply call the same functionality from the underlying self.\_products collection. However writing the functions ourselves give us complete control over all aspects of how our collection functions if we need to customise it.

1. Write the code for \_\_len\_\_ that will return how long a collection is (in our case, how many items are in our cart).

#### Results:

>>> len(test\_cart)

2

>>>

## Question 3

We will certainly want to loop through our shopping cart instead of looping through the products in our shopping cart. We have already covered iterations and iterable objects in a previous chapter so let’s use that knowledge here.

1. Write an \_\_iter\_\_ method that will enable us to loop through our shopping cart. Test it in a for loop that prints out the details of each product and also in a comprehension that just shows us the names.

#### Results:

>>> for p in test\_cart:

print(p)

Corn $3.14

Peas $2.17

>>> [p.name for p in test\_cart]

['Corn', 'Peas']

>>>

## Question 4

The final thing we want to consider is what is meant when we say two shopping carts are equal. This could be important if we are worried about a customer losing connection and want to make sure when they come back their own shopping cart matches the one we last had for them. The basic rule is that two shopping carts are equal if they contain the same products.

1. We will be using the underlying == operator, so we should play with it a bit first to understand it. In REPL create a new cart and add peas and corn to the cart. In theory these two carts should be the same, so lets test them with ==

#### Results:

>>> new\_cart=ShoppingCart()

>>> new\_cart(peas)

>>> new\_cart(corn)

>>> new\_cart==test\_cart

False

1. Now try to compere the content of their products.

#### Results:

new\_cart.\_products==test\_cart.\_products

True

>>>

1. This should be how we compare two carts, by matching their contents. Let’’s define a new method called \_\_eq\_\_ that does this comparison. Test it by comparing the two carts directly with each other.

#### Results:

>>> new\_cart=ShoppingCart()

>>> new\_cart(peas)

>>> new\_cart(corn)

>>> test\_cart==new\_cart

True

>>>

# Chapter 10

We have provided a update version of our checkout script which contains a payment class that collect information about credit card so that it can process payments. There are many things that could go wrong when users are trying to provide information. Whatever the issues are, we would like to handle them in a user friendly and consistent manner.

## Question 1

To get an idea of the code we will be using for the exercises it is worth running through it first to see what it does. The checkout module contains two classes. The one we are focused on is the payment class

1. Construct a payment by providing a name, the card (we only accept visa and master card at his time), the number, an expiry date (in the format ’04/23’) and a four digit pin number. Once you have created the payment, print it out to confirm the details are correct. (Please don’t use actual card numbers, etc, just make it up)

#### Results:

>>> p=payment('Matthew Gregory','Visa',123456789,'04/23',1234)

>>> print(p)

Name: Matthew Gregory

Type of card: Visa

Card number: 123456789

Expiry: 04/23

Pin: 1234

1. Interestingly there is also an interactive version that prompt you for the details. Run this by calling the class method interactive\_payment() on the payment class.

#### Results:

>>> p=payment.interative\_payment()

Please enter the name on your credit card: Matthew Gregory

Please enter the type of credit card: Visa

Please enter the credit card number: 123456789

Please enter the expiry date of credit card (mm/yy): 04/23

Please enter the pin: 1234

>>> print(p)

Name: Matthew Gregory

Type of card: Visa

Card number: 123456789

Expiry: 04/23

Pin: 1234

>>>

1. Try breaking it by entering a blank string for the name and an invalid names for the type of card (e.g. American Express) and a pin number that is more than 4 characters long. These are all validation done by our code.

#### Results:

>>> p=payment('','American Express',123456789,'04/23',12345)

You must enter a value for this field

Sorry we only accept Visa or Mastercard payments

Input must not be more than 4 characters

1. Try breaking it by entering characters instead of numbers for the pin, or a wrong format for the expiry date or None for our credit card number. These error are all picked up by python not by our code.

#### Results:

>>> p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

Traceback (most recent call last):

File "<pyshell#64>", line 1, in <module>

p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 31, in \_\_init\_\_

self.\_pin=self.validate(pin,'int',size=4)

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 83, in validate

return int(input)

ValueError: invalid literal for int() with base 10: 'abcd'

>>> p=payment('Matthew Gregory','Visa',123456789,'30/04/2023',1234)

Traceback (most recent call last):

File "<pyshell#65>", line 1, in <module>

p=payment('Matthew Gregory','Visa',123456789,'30/04/2023',1234)

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 30, in \_\_init\_\_

self.\_card\_expiry=self.validate(card\_expiry,'date',date\_format='%m/%y')

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 85, in validate

return datetime.datetime.strptime(input,kwargs.get('date\_format'))

File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.7\_3.7.2544.0\_x64\_\_qbz5n2kfra8p0\lib\\_strptime.py", line 577, in \_strptime\_datetime

tt, fraction, gmtoff\_fraction = \_strptime(data\_string, format)

File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.7\_3.7.2544.0\_x64\_\_qbz5n2kfra8p0\lib\\_strptime.py", line 359, in \_strptime

(data\_string, format))

ValueError: time data '30/04/2023' does not match format '%m/%y'

>>> p=payment('Matthew Gregory','Visa',None,'04/23',1234)

Traceback (most recent call last):

File "<pyshell#66>", line 1, in <module>

p=payment('Matthew Gregory','Visa',None,'04/23',1234)

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 29, in \_\_init\_\_

self.\_card\_number=self.validate(card\_number,'int',size=9)

File "D:/Case Study/Solutions for each chapter/Chapter 10/4/checkout\_start.py", line 83, in validate

return int(input)

TypeError: int() argument must be a string, a bytes-like object or a number, not 'NoneType'

>>> p=payment('Matthew Gregory','Visa',123456789,04/23,1234)

SyntaxError: invalid token

>>>

## Question 2

In the previous question we had a look at some of the problems you could have with our payments system. Now we want to look at dealing with all these situations in a consistent manner by user exceptions.

1. Firstly some of the issue already have exceptions, but the messages are not necessarily meaningful to the person using our application. In the payment constructor add a try/except clause around the testing of pin number to handle the ValueError by raising another ValueError with a more appropriate message.

#### Results:

>>> p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

Traceback (most recent call last):

File "D:/Case Study/Solutions for each chapter/Chapter 10/2/checkout.py", line 32, in \_\_init\_\_

self.\_pin=self.validate(pin,'int',size=4)

File "D:/Case Study/Solutions for each chapter/Chapter 10/2/checkout.py", line 86, in validate

return int(input)

ValueError: invalid literal for int() with base 10: 'abcd'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

File "D:/Case Study/Solutions for each chapter/Chapter 10/2/checkout.py", line 34, in \_\_init\_\_

raise ValueError('The pin number should be 4 digits')

ValueError: The pin number should be 4 digits

1. You may have noticed that it has shown both errors, the one we raised and the original. To suppress the original add the phrase ‘from None’ to the raise command. This tells python to raise this exception as if there was not parent.

#### Results:

>>> p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

p=payment('Matthew Gregory','Visa',123456789,'04/23','abcd')

File "D:/Case Study/Solutions for each chapter/Chapter 10/2/checkout.py", line 34, in \_\_init\_\_

raise ValueError('The pin number should be 4 digits') from None

ValueError: The pin number should be 4 digits

## Question 3

It may well be worth recognising which conditions are specific to credit cards, such as the length of the fields, and the fact that the pin number need to be four digits, and those that are not. For this reason we will define our own exception specifically for credit card checking.

1. In the checkout module define a new exception called CardException as a subtype of ValueError.
2. Refactor all the messages in our validation to raise CardException with the message as a payload instead of just printing it. Check it by rerunning the code from Queston 1 c and making sure that it raises an exception with the right message.

#### Results:

#### >>> p=payment('','American Express',123456789,'04/23',12345)

#### Traceback (most recent call last):

#### File "<pyshell#7>", line 1, in <module>

#### p=payment('','American Express',123456789,'04/23',12345)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 30, in \_\_init\_\_

#### self.\_customer\_name=self.validate(customer\_name,'str',size=30)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 86, in validate

#### raise CreditException('You must enter a value for this field')

#### CreditException: You must enter a value for this field

#### >>> p=payment('Matthew Gregory','American Express',123456789,'04/23',12345)

#### Traceback (most recent call last):

#### File "<pyshell#9>", line 1, in <module>

#### p=payment('Matthew Gregory','American Express',123456789,'04/23',12345)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 31, in \_\_init\_\_

#### self.\_card\_type=self.validate(card\_type,'card',size=20)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 95, in validate

#### raise CreditException('Sorry we only accept Visa or Mastercard payments')

#### CreditException: Sorry we only accept Visa or Mastercard payments

#### >>> p=payment('Matthew Gregory','Visa',123456789,'04/23',12345)

#### Traceback (most recent call last):

#### File "<pyshell#10>", line 1, in <module>

#### p=payment('Matthew Gregory','Visa',123456789,'04/23',12345)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 37, in \_\_init\_\_

#### raise ValueError('The pin number should be 4 digits') from None

#### ValueError: The pin number should be 4 digits

1. Refactor Question 2 so that it also raises the CardException rather than the ValueError.
2. Modify the validate code so that we can also make the formatting of the expiry date a CreditException, this time however use the same message that was in the ValueError.

#### Results:

#### >>> p=payment('Matthew Gregory','Visa',123456789,'30/04/2023',1234)

#### Traceback (most recent call last):

#### File "<pyshell#14>", line 1, in <module>

#### p=payment('Matthew Gregory','Visa',123456789,'30/04/2023',1234)

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 33, in \_\_init\_\_

#### self.\_card\_expiry=self.validate(card\_expiry,'date',date\_format='%m/%y')

#### File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 94, in validate

#### raise CreditException(str(ve.args)) from None

#### CreditException: ("time data '30/04/2023' does not match format '%m/%y'",)

## Question 4

The validation method we have just been working on goes through all the different type of validation, and by the end assumes whatever is left is a string. But how can we be sure?

1. Check out the code in the constructor to see examples of how our validate method is being called. In REPL use this method to validate 1234 as an int of max size 4

#### Results:

#### >>> p.validate(1234,'int',size=4)

#### 1234

1. Rerun this with ‘integer’ instead of ‘int’. It still works but now it didn’t run the validation required of an int.
2. Include an assertion at the end of the validate functions (just before the last ‘return’) to check that at the end of our code the type reallly is a string. Rerun the validation in REPL and observe the results.

#### Results:

>>> p.validate(1234,'integer',size=4)

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

p.validate(1234,'integer',size=4)

File "D:\Case Study\Solutions for each chapter\Chapter 10\3\checkout.py", line 101, in validate

assert isinstance(input,str)

AssertionError

>>>

# Chapter 11

The final step in our project is to develop our checkout process. This involves stetting up a delivery date, taking the customers shopping cart, displaying it, also showing things they have previously ordered (along with id) and asking if they would like to include them also, then collecting customer information and credit card information, validating the credit card and then final confirmation before placing the order. Once the order has been successfully placed, we want to add it to the already purchased list, save the customer details, give a confirmation message and display the order to the customer, on failure we want to remove explain why the process failed and ask if they would like to try again? If not print a message then exit, if yes then start the process over again.

## Question 1

Firstly we are going to write the class to checkout without worrying about context managers.

1. Run checkout, notice how we can change the date and time, type in ‘Change Date’ and a different date and time. Make sure it worked, when it asks you again if you want to proceed type in Cancel. It will raise and exception.

#### Results:

>>> from test\_products import \*

>>> co=checkout(test\_cart)

>>> co.place\_order()

Please enter your pickup/delviery date (DD/MM/YY): 23/10/21

Please enter your pickup/devlivery time (HH:MM AM/PM): 8:45 AM

Please confirm you wish to proceed? Yes / Change Date/ Cancel: Change Date

Please enter your pickup/delviery date (DD/MM/YY): 23/10/23

Please enter your pickup/devlivery time (HH:MM AM/PM): 10:40 PM

Your new delivery date and time is: 23/10/23 10:40 PM

Please confirm you wish to proceed? Yes / Change Date/ Cancel: Cancel

Traceback (most recent call last):

File "<pyshell#148>", line 1, in <module>

co.place\_order()

File "D:\Case Study\Solutions for each chapter\Chapter 11\1\checkout.py", line 136, in place\_order

raise CheckoutException("Order cancelled by operator")

CheckoutException: Order cancelled by operator

1. Try again and this time type in ‘Yes’ to proceed. In the next step we are required to enter an address (make up an address), and then credit card details. For the credit card enter some valid data according to the rules we saw in the last chapter. You should see a confirmation that the order was placed.

#### Results:

>>> co.place\_order()

Please enter your pickup/delviery date (DD/MM/YY): 23/10/23

Please enter your pickup/devlivery time (HH:MM AM/PM): 10:40 PM

Please confirm you wish to proceed? Yes / Change Date/ Cancel: Yes

Please enter your address: 123 any street, somewhere

Please enter the name on your credit card: Matthew Gregory

Please enter the type of credit card: Visa

Please enter the credit card number: 123456789

Please enter the expiry date of credit card (mm/yy): 4/23

Please enter the pin: 1234

Order Confirmed.

Thank you for shopping with us.

We look forward to seeing you again soon.

Order Date: Sunday 10 October 2021

Product Name Product Price

============ =============

Corn $3.14

Peas $2.17

Total: 5.31

Your delivery is due in 742 days and should be picked up (or delivered) at 10:40 PM

>>>

1. Run it once more but this time give details for a credit card that are not valid (e.g. use “American Express” for the card). Notice there is no chance to recover and change the details of a credit card. Once it fails that is it!

#### Results:

>>> co.place\_order()

Please enter your pickup/delviery date (DD/MM/YY): 23/10/23

Please enter your pickup/devlivery time (HH:MM AM/PM): 10:40 PM

Please confirm you wish to proceed? Yes / Change Date/ Cancel: Yes

Please enter your address: 123 any street, somewhere

Please enter the name on your credit card: Matthew Gregory

Please enter the type of credit card: 123456789

Please enter the credit card number: 123456789

Please enter the expiry date of credit card (mm/yy): 4/23

Please enter the pin: 1234

Traceback (most recent call last):

File "<pyshell#149>", line 1, in <module>

co.place\_order()

File "D:\Case Study\Solutions for each chapter\Chapter 11\1\checkout.py", line 142, in place\_order

current\_payment=payment.interative\_payment()

File "D:\Case Study\Solutions for each chapter\Chapter 11\1\checkout.py", line 49, in interative\_payment

cls.get\_pin())

File "D:\Case Study\Solutions for each chapter\Chapter 11\1\checkout.py", line 34, in \_\_init\_\_

self.\_card\_type=self.validate(card\_type,'card',size=20)

File "D:\Case Study\Solutions for each chapter\Chapter 11\1\checkout.py", line 100, in validate

raise CheckoutException('Sorry we only accept Visa or Mastercard payments')

CheckoutException: Sorry we only accept Visa or Mastercard payments

## Question 2

Turn the code that repeats into a context manager

1. In checkout create a new class called Response with \_\_enter\_\_, \_\_exit\_\_ methods. The enter method should set an attribute called response to an initial values of ‘none’ and return itself to the user so they can interact with it. The exit should prompt for a yes/no/change response and print out and message thanking them for their response and display it.
2. To test in REPL use this Response class in a with clause and make sure you print out the response.

#### Results:

none

Please confirm you wish to proceed? yes/no/change: yes

Thank you for your response of: yes

>>>

## Question 3

1. Inside the Response class create an init that accepts a function and if it is callable saves it as an attribute other wise it should raise a TypeError with the message 'Objects passed to Response must be callable'.

Also create a function called run() that can be used to run the function if the user wants to call it in the main block.

1. Test this by creating a function called getaddress() which prompts for an address. Use this when creating the Response in a with clause. In the body of the code simply call the run() method from our response to populate the address then print it out. It should also ask for a confirmation that we are happy to continue. Reply ‘yes’

#### Results:

>>> def getaddress():

return input('Please enter your address: ')

>>> with Response(getaddress) as r:

address=r.run()

print(address)

Please enter your address: 123 any street, somewhere

123 any street, somewhere

Please confirm you wish to proceed? yes/no/change: yes

Thank you for your response of: yes

>>>

1. Modify your exit so that when someone responds ‘change’ it recursively calls the function using the with clause and the Response object. When someone responds ‘cancel’ raise a CheckoutException with the message "Cancelled by operator". When someone responds with yes give them an thank you message.

#### Results:

>>> def getaddress():

return input('Please enter your address: ')

>>> with Response(getaddress) as r:

address=r.run()

print(address)

Please enter your address: 123 any street, somewhere

123 any street, somewhere

Please confirm you wish to proceed? yes/no/change: change

Please enter your address: another address

Please confirm you wish to proceed? yes/no/change: no

Traceback (most recent call last):

File "<pyshell#81>", line 3, in <module>

print(address)

File "D:/Case Study/Solutions for each chapter/Chapter 11/3/checkout.py", line 170, in \_\_exit\_\_

return self.\_function()

File "D:/Case Study/Solutions for each chapter/Chapter 11/3/checkout.py", line 172, in \_\_exit\_\_

raise CheckoutException("Cancelled by operator")

CheckoutException: Cancelled by operator

>>> with Response(getaddress) as r:

address=r.run()

print(address)

Please enter your address: 123 any street, somewhere

123 any street, somewhere

Please confirm you wish to proceed? yes/no/change: yes

Thank you for your response

>>>

## Question 4

Although it would be quite a time consuming task, you may wish to go back and refactor your code to use this new capacity to simplify prompting for all kind of information, both in the credit card and checkout processes. It will also come in useful any time someone want to get a input from the use and would like it confirmed before they move on. The code was written using a class based context manager, but you might also like to try writing it as a function based context manager using @contextlib.

# Chapter 12

Build the introspection tool as described on page 398. The instructions and explanations are all in the book. It is a longish exercise, but the skills involved are a good summary of what we have learned in this course and the tool that you will build could end up being useful for debugging or understanding code.