Jack Hughes CO664 Logbook Exercises

September 27, 2023

1 CO664WBL Design Patterns

1.1 CW1 Design Patterns Logbook and Report

Jack Hughes - 22044198

GitHub link - https://github.com/Jack272Hughes/uni-design-patterns

1.2 Logbook Exercise 1

Insert a 'code' cell below. In this do the following:

- line 1 - create a list named "shopping_list" with items: milk, eggs, bread, cheese, tea, coffee, rice, pasta, milk, tea (NOTE: the duplicate items are intentional
- line 2 - print the list along with a message e.g. "This is my shopping list …"
- line 3 - create a tuple named "shopping_tuple" with the same items
- line 4 - print the tuple with similar message e.g. "This is my shopping tuple …"
- line 5 - create a set named "shopping_set" from "shopping_list" by using the set() method
- line 6 - print the set with appropriate message and check duplicate items have been removed
- line 7 - make a dictionary "shopping_dict" - copy and paste the following items and prices: "milk": "£1.20", "eggs": "£0.87", "bread": "£0.64", "cheese": "£1.75", "tea": "£1.06", "coffee": "£2.15", "rice": "£1.60", "pasta": "£1.53".
- line 8 - print the dictionary with an appropriate message

An example of fully described printed output is presented below (some clues here also) Don't worry of your text output is different - it is the contents of the compund variables that matter

```
This is my shopping list ['milk', 'eggs', 'bread', 'cheese', 'tea', 'coffee', 'rice', 'pasta',
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese', 'tea', 'coffee', 'rice', 'pasta'
This is my Shopping_set with duplicates removes {'rice', 'milk', 'pasta', 'cheese', 'eggs', 'te
This is my shopping_dict {'milk': '£1.20', 'eggs': '£0.87', 'bread': '£0.64', 'cheese': '£1.75
```

[ ]:
```
shopping_list = ["milk", "eggs", "bread", "cheese", "tea", "coffee", "rice",␣
 ↪"pasta", "milk", "tea"]
print("This is my shopping list", shopping_list)

shopping_tuple = ("milk", "eggs", "bread", "cheese", "tea", "coffee", "rice",␣
 ↪"pasta", "milk", "tea")
print("This is my shopping tuple", shopping_tuple)
```

```
shopping_set = set(shopping_list)
print("This is my shopping set with duplicates removed", shopping_set)

shopping_dict = { "milk": "£1.20", "eggs": "£0.87", "bread": "£0.64", "cheese":␣
↪"£1.75", "tea": "£1.06", "coffee": "£2.15", "rice": "£1.60", "pasta": "£1.
↪53" }
print("This is my shopping dictionary", shopping_dict)
```

1.3 Logbook Exercise 2

Create a 'code' cell below. In this do the following:

- line 1 - Use a comment to title your exercise - e.g. "Unit 2 Exercise"
- line 2 - create a list ... li = ["USA","Mexico","Canada"]
- line 3 - append "Greenland" to the list
- l4 - print the list to de,monstrate that Greenland is attached
- l5 - remove "Greenland"
- l6 - print the list to de,monstrate that Greenland is removed
- l7 - insert "Greenland" at the beginning of the list
- l8 - print the resul of l7
- l9 - shorthand slice the list to extract the first two items - simultaneausly print the output
- l10 - use a negative index to extract the second to last item - simultaneausly print the output
- l11 - use a splitting sequence to extract the middle two items - simultaneausly print the output

An example of fully described printed output is presented below (some clues here also) Don't worry of your text output is different - it is the contents of the list that matter

```
li.append('Greenland') gives ...  ['USA', 'Mexico', 'Canada', 'Greenland']
li.remove('Greenland') gives ...  ['USA', 'Mexico', 'Canada']
li.insert(0,'Greenland') gives ...  ['Greenland', 'USA', 'Mexico', 'Canada']
li[:2] gives ...  ['Greenland', 'USA']
li[-2] gives ...  Mexico
li[1:3] gives ...  ['USA', 'Mexico']
```

[ ]:
```
# Unit 2 Exercise
countries = ["USA", "Mexico", "Canada"]
countries.append("Greenland")
print("After appending 'Greenland' we have", countries)

countries.remove("Greenland")
print("After removing 'Greenland' we have", countries)

countries.insert(0, "Greenland")
print("After inserting 'Greenland' at index 0 we have", countries)

print("The first two countries are", countries[:2])
print("The second to last country is '{0}'".format(countries[-2]))
print("The two middle countries are", countries[1:3])
```

1.4 Logbook Exercise 3

Create a 'code' cell below. In this do the following: - on the first line create the following set … a=[0,1,2,3,4,5,6,7,8,9,10] - on the second line create the following set … b=[0,5,10,15,20,25] - on the third line create the following dictionary … topscores={"Jo":999, "Sue":987, "Tara":960; "Mike":870} - use a combination of print() and type() methods to produce the following output

```
list a is ...  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list b is ...  [0, 5, 10, 15, 20, 25]
The type of a is now ... <class 'list'>
```

- on the next 2 lines convert list a and b to sets using set()
- on the following lines use a combination of print(), type() and set notaion (e.g. 'a & b', 'a | b', 'b-a') to obtain the following output

```
set a is ...  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
set b is ...  {0, 5, 10, 15, 20, 25}
The type of a is now ... <class 'set'>
Intersect of a and b is [0, 10, 5]
Union of a and b is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25]
Items unique to set b are {25, 20, 15}
```

- on the next 2 lines use print(), '.keys()' and '.values()' methods to obtain the following output

```
topscores dictionary keys are dict_keys(['Jo', 'Sue', 'Tara', 'Mike'])
topscores dictionary values are dict_values([999, 987, 960, 870])
```

[ ]:
```
a = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
b = [ 0, 5, 10, 15, 20, 25 ]
top_scores = { "Jo": 999, "Sue": 987, "Tara": 960, "Mike": 870 }

print("List a is", a)
print("List b is", b)
print("The type of a is currently:", type(a))

a = set(a)
b = set(b)

print("Set a is", a)
print("Set b is", b)
print("The type of a is now:", type(a))
print("The intersect of a and b is", a & b)
print("The union of sets a and b is", a | b)
print("The items unique to set b are", b - a)

print("top_scores dictionary keys are", top_scores.keys())
print("top_scores dictionary values are", top_scores.values())
```

1.4.1 Logbook Exercise 4

Create a 'code' cell below. In this do the following: - Given the following 4 lists of names, house number and street addresses, towns and postcodes …

```
["T Cruise","D Francis","C White"]
["2 West St","65 Deadend Cls","15 Magdalen Rd"]
["Canterbury", "Reading", "Oxford"]
["CT8 23RD", "RG4 1FG", "OX4 3AS"]
```

- write a Custom 'address_machine' function that formats 'name', 'hs_number_street', 'town', 'postcode' with commas and spaces between items
- create a 'newlist' that repeatedly calls 'address_machine' and 'zips' items from the 4 lists
- write a 'for loop' that iterates over 'new list' and prints each name and address on a separate line
- the output should appear as follows

```
T Cruise, 2 West St, Canterbury, CT8 23RD
D Francis, 65 Deadend Cls, Reading, RG4 1FG
C White, 15 Magdalen Rd, Oxford, OX4 3AS
```

- HINT: look at "# CUSTOM FUNCTION WORKED EXAMPLES 3 & 4" above

[ ]:
```
names = ["T Cruise", "D Francis", "C White"]
addresses = ["2 West St", "65 Deadend Cls", "15 Magdalen Rd"]
towns = ["Canterbury", "Reading", "Oxford"]
postcodes = ["CT8 23RD", "RG4 1FG", "OX4 3AS"]

def address_machine(name, hs_number_street, town, postcode):
    return ", ".join([name, hs_number_street, town, postcode])

newlist = [address_machine(name, address, town, postcode) for name, address,␣
 ↪town, postcode in zip(names, addresses, towns, postcodes)]
for item in newlist:
    print(item)
```

1.5 Logbook Exercise 5

Create a 'code' cell below. In this do the following: - Create a super class "Person" that takes three string and one integer parameters for first and second name, UK Postcode and age in years. - Give "Person" a method "greeting" that prints a statement along the lines "Hello, my name is Freddy Jones. I am 22 years old and my postcode is HP6 7AJ" - Create a "Student" class that extends/inherits "Person" and takes additional parameters for degree_subject and student_ID. - give "Student" a "studentGreeting" method that prints a statement along the lines "My student ID is SN123456 and I am reading Computer Science" - Use either Python {} format or C-type %s/%d notation to format output strings - Create 3 student objects and persist these in a list - Iterate over the three objects and call their "greeting" and "studentGreeting" methods - Output should be along the lines of the following

```
Hello, my name is Dick Turpin. I am 32 years old and my postcode is HP11 2JZ
My student ID is DT123456 and I am reading Highway Robbery
```

```
Hello, my name is Dorothy Turpin. I am 32 years old and my postcode is SO14 7AA
My student ID is DT123457 and I am reading Law

Hello, my name is Oliver Cromwell. I am 32 years old and my postcode is OX35 14RE
My student ID is OC123456 and I am reading History
```

[ ]:
```python
class Person:
    def __init__(self, forename, surname, postcode, age):
        self._forename = forename
        self._surname = surname
        self._postcode = postcode
        self._age = age

    def greeting(self):
        print("Hello, my name is %s %s. I am %d years old and my postcode is␣
 ↪%s" % (self._forename, self._surname, self._age, self._postcode))

class Student(Person):
    def __init__(self, forename, surname, postcode, age, degree_subject,␣
 ↪student_ID):
        super().__init__(forename, surname, postcode, age)
        self._degree_subject = degree_subject
        self._student_ID = student_ID

    def studentGreeting(self):
        print("My student ID is {0} and I am reading {1}".format(self.
 ↪_student_ID, self._degree_subject))

students = [
    Student("Freddy", "Jones", "HP6 7AJ", 22, "Computer Science", "SN123456"),
    Student("Amy", "Charles", "SW1A 0AA", 23, "Biology", "SN654321"),
    Student("Charlie", "Able", "SE1 7PB", 21, "Mechanical Engineering",␣
 ↪"SN123654")
]

for student in students:
    student.greeting()
    student.studentGreeting()
```

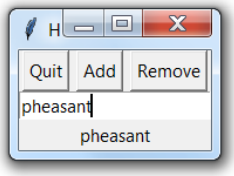
1.6 Logbook Exercise 6

- Examine Steve Lipton's "simplest ever" version for the MVC
- Note how when the MyController object is initialised it:
- passes a reference to itself to the MyModel and MyView objects it creates
- thereby allowing MyModel and MyView to create 'delegate' vc (virtual control) aliases to call back the MyController object
- When you feel you have understood MVC messaging and delegation add code for a new

button that removes items from the list

- The end result should be capable of creating the output below
- Clearly ***comment your code*** to highlight the insertions you have made
- Note: if you don't see the GUI immediately look for the icon Jupyter icon in your task bar (also higlighted below)

```
returned
['duck', 'duck', 'goose', 'penguin']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey', 'pheasant']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey']
```



[ ]:
```python
from tkinter import *

class MyController():
    def __init__(self,parent):
        self.parent = parent
        self.model = MyModel(self)
        self.view = MyView(self)
        self.view.setEntry_text('Add to Label')
        self.view.setLabel_text('Ready')

    def quitButtonPressed(self):
        self.parent.destroy()
    def addButtonPressed(self):
        self.model.addToList(self.view.entry_text.get())
    # CHANGED - Added method to be used by the remove button
    def removeButtonPressed(self):
        # Calls the model to remove an item based on the view's entry text
        self.model.removeFromList(self.view.entry_text.get())

    # CHANGED - Using this method as a "list changed" event handler
    def listChangedDelegate(self):
        # CHANGED - Moved setting label text to when list changes
        self.view.setLabel_text(self.view.entry_text.get())
        # CHANGED - Moved all logging for the list to when list changes
        print("returned\n", self.model.getList())

class MyView(Frame):

    def __init__(self,vc):
        self.frame=Frame()
        self.frame.grid(row = 0,column=0)
        self.vc = vc
        self.entry_text = StringVar()
```

```
        self.entry_text.set('nil')
        self.label_text = StringVar()
        self.label_text.set('nil')
        self.loadView()

    def loadView(self):
        quitButton = Button(self.frame,text = 'Quit', command= self.vc.
↪quitButtonPressed).grid(row = 0,column = 0)
        addButton = Button(self.frame,text = "Add", command = self.vc.
↪addButtonPressed).grid(row = 0, column = 1)
        # CHANGED - Added button for removing items which uses the␣
↪removeButtonPressed function from the controller
        removeButton = Button(self.frame,text = "Remove", command = self.vc.
↪removeButtonPressed).grid(row = 0, column = 2)
        entry = Entry(self.frame,textvariable = self.entry_text).grid(row = 1,␣
↪column = 0, columnspan = 3, sticky = EW)
        label = Label(self.frame,textvariable = self.label_text).grid(row = 2,␣
↪column = 0, columnspan = 3, sticky = EW)

    def getEntry_text(self):
        return self.entry_text.get()
    def setEntry_text(self,text):
        self.entry_text.set(text)
    def getLabel_text(self):
        return self.label_text.get()
    def setLabel_text(self,text):
        self.label_text.set(text)

class MyModel():
    def __init__(self,vc):
        self.vc = vc
        self.myList = ['duck','duck','goose']
        self.count = 0
    def listChanged(self):
        self.vc.listChangedDelegate()
    def getList(self):
        return self.myList
    def initListWithList(self, aList):
        self.myList = aList
    def addToList(self,item):
        # CHANGED - Directly changin the internal list rather than assigning to␣
↪variable first
        self.myList.append(item)
        self.listChanged()
    # CHANGED - Added method for removing items from list
    def removeFromList(self, item):
```

```
        # Removes items directly from the internal list
        self.myList.remove(item)
        self.listChanged()

def main():
    root = Tk()
    frame = Frame(root )
    root.title('Hello TEST')
    app = MyController(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

1.7 Logbook Exercise 7

- Your task is to extend the Observer example below with a pie-chart view of model data and to copy this cell and the solution to your logbook
- The bar chart provides a useful example of structure
- Partial code is provided below for insertion, completion (note '####' requires appropriate replacement) and implementation
- you will also need to create an 'observer' object from the PieView class and attach it to the first 'model'

```
# Pie chart viewer/ConcreteObserver - overrides the update() method
class PieView(####):

    def update(####, ####): #Alert method that is invoked when the notify() method in a concret
        # Pie chart, where the slices will be ordered and plotted counter-clockwise:
        labels = ####
        sizes = ####
        explode = (0.1, 0, 0, 0, 0, 0)  # only "explode" the 1st slice
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startang
        ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
        plt.####()
```

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np


# Nicely abstracted structure by which any model can notify any observer (view)␣
 ↪of changes in the model
class Subject(object): #Represents what is being 'observed'

    def __init__(self):
        self._observers = [] # This where references to all the observers are␣
 ↪being kept
```

```
                            # Note that this is a one-to-many relationship:␣
↪there will be one subject to be observed by multiple _observers

    def attach(self, observer):
        if observer not in self._observers: #If the observer is not already in␣
↪the observers list
            self._observers.append(observer) # append the observer to the list

    def detach(self, observer): #Simply remove the observer
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notify(self, modifier=None):
        for observer in self._observers: # For all the observers in the list
            if modifier != observer: # Don't notify the observer who is␣
↪actually doing the updating
                observer.update(self) # Alert the observers!

# Represents the 'data' for which changes will produce notifications to any␣
↪registered view/observer objects
class Model(Subject): # Extends the Subject class

    def __init__(self, name):
        Subject.__init__(self)
        self._name = name # Set the name of the model
        self._labels = ['Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp']
        self._data = [10,8,6,4,2,1]

    @property #Getter that gets the labels
    def labels(self):
        return self._labels

    @labels.setter #Setter that sets the labels
    def labels(self, labels):
        self._labels = labels
        self.notify() # Notify the observers whenever somebody changes the␣
↪labels

    @property #Getter that gets the data
    def data(self):
        return self._data

    @data.setter #Setter that sets the labels
    def data(self, data):
        self._data = data
```

```
        self.notify() # Notify the observers whenever somebody changes the data


# This is the 'standard' view/observer which also acts as an 'abstract' class␣
 ↪whereby deriving Bar/Chart/Table views override the update() method
# This 'abstracted' layer is always shown in examples but is important to␣
 ↪demonstrate potential polymorphic behaviour of update()
class View():

    def __init__(self, name=""):
         self._name = name #Set the name of the Viewer

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        print("Generalised Viewer '{}' has: \nName = {}; \nLabels = {}; \nData␣
 ↪= {}".format(self._name, subject._name, subject._labels, subject._data))

# Table 'chart' viewer/ConcreteObserver - overrides the update() method
class TableView(View):

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        fig = plt.figure(dpi=80)
        ax = fig.add_subplot(1,1,1)
        table_data = list(map(list,zip(subject._labels, subject._data)))
        table = ax.table(cellText=table_data, loc='center')
        table.set_fontsize(14)
        table.scale(1,4)
        ax.axis('off')
        plt.show()

# Bar chart viewer/ConcreteObserver - overrides the update() method
class BarView(View):

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        objects = subject._labels
        y_pos = np.arange(len(objects))
        performance = subject._data
        plt.bar(y_pos, performance, align='center', alpha=0.5)
        plt.xticks(y_pos, objects)
        plt.ylabel('Usage')
        plt.title('Programming language usage')
        plt.show()

# Pie chart viewer/ConcreteObserver - overrides the update() method
```

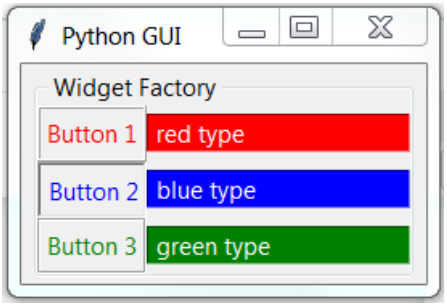
```
class PieView(View):

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        # Pie chart, where the slices will be ordered and plotted␣
 ↪counter-clockwise:
        labels = subject.labels
        sizes = subject.data
        explode = (0.1, 0, 0, 0, 0, 0)  # only "explode" the 1st slice
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',␣
 ↪shadow=True, startangle=90)
        ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a␣
 ↪circle.
        plt.show()

#Let's create our subjects
m1 = Model("Model 1")
m2 = Model("Model 2") # This is never used!

#Let's create our observers
v1 = View("1: standard text viewer")
v2 = TableView("2: table viewer")
v3 = BarView("3: bar chart viewer")
v4 = PieView("4: pie chart viewer")
####

#Let's attach our observers to the first model
m1.attach(v1)
m1.attach(v2)
m1.attach(v3)
m1.attach(v4)
####

# Let's just call the notify() method to see all the charts in their unchanged␣
 ↪state
m1.notify()

# Now Let's change the properties of our first model
# Change 1 triggers all 4 views and updates their labels
m1.labels = ['C#','PHP','JavaScript','ASP','Python','Smalltalk']
# Change 2 triggers all 4 views and updates their data
m1.data = [1,18,8,60,3,1]
```

1.8 Logbook Exercise 8

- Your task is to extend the modified version of Burkhard Meier's button factory (below) to create a text field factory
- In tkinter textfields are 'Entry' widgets
- similary to the button factory structure you will need:
- a concrete Entry widget factory class - name it TextFactory()
- the TextFactory's Factory Method - name it createText(…)
- an abstract product - name it TextBase() and give it default attributes 'textvariable' and 'background'
- a getTextConfig(…) method
- 3x concrete text products - name these Text_1/2/3
- … and assign them textvariable values of " red/blue/green type" respectively
- … and assign them background values of 'red/blue/green' respectively
- to extend the OOP class with a createTextFields() method
- … that creates a factory object
- and the Entry fields … the code for the first Entry Field is as follows

```
# Entry field 1
sv=tk.StringVar()
tx = factory.createText(0).getTextConfig()[0]
sv.set(tx)
bg  = factory.createText(0).getTextConfig()[1]
action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg, foreground="white"
action.grid(column=1, row=1)
```

- the end product should look something like this …



Challenge question

- At present the example has a 2x 'concrete' creators. As a comment just above the line `import tkinter as tk` briefly explain what you would need to do to refactor code so that the concrete creators extended an abstract class/interface called ***Creator*** (i.e. just as it appears in Gamma et al. (1995) and on slide 2)

[ ]:
```
%%python3
'''
Example modified from Meier, B (2017) Python GUI Programming Cookbook, Second␣
↪Edition.  Packt Publishing
```

```
R Mather June 2020

'''
# You would first need to create a base class called 'FactoryBase' with a␣
 ↪variable called 'types' initialised to an empty list
# The 'FactoryBase' class would have a create method which returns the element␣
 ↪in the 'types' variable at the given index
# Any creators extending this class then only need to update the 'types'␣
 ↪variable with a list of all the possible types
# it can return, such as the current 'buttonTypes' variable defined below
import tkinter as tk
from tkinter import ttk
from tkinter import Menu


class ButtonFactory():
    def createButton(self, type_):
        return buttonTypes[type_]()

class ButtonBase():
    relief     ='flat'
    foreground ='white'
    def getButtonConfig(self):
        return self.relief, self.foreground

class ButtonRidge(ButtonBase):
    relief     ='ridge'
    foreground ='red'

class ButtonSunken(ButtonBase):
    relief     ='sunken'
    foreground ='blue'

class ButtonGroove(ButtonBase):
    relief     ='groove'
    foreground ='green'

buttonTypes = [ButtonRidge, ButtonSunken, ButtonGroove]

class TextFactory():
    def createText(self, _type):
        return textTypes[_type]()

class TextBase():
    textvariable = "black"
    background = "white"
```

```
    def getTextConfig(self):
        return self.textvariable, self.background

class Text_1(TextBase):
    textvariable = "red"
    background = "red"

class Text_2(TextBase):
    textvariable = "blue"
    background = "blue"

class Text_3(TextBase):
    textvariable = "green"
    background = "green"

textTypes = [Text_1, Text_2, Text_3]


class OOP():
    def __init__(self):
        self.win = tk.Tk()
        self.win.title("Python GUI")
        self.createWidgets()

    def createWidgets(self):
        self.widgetFactory = ttk.LabelFrame(text=' Button Factory ')
        self.widgetFactory.grid(column=0, row=0, padx=8, pady=4)

        self.createButtons()
        self.createTextFields()

    def createButtons(self):

        factory = ButtonFactory()

        # Button 1
        rel = factory.createButton(0).getButtonConfig()[0]
        fg  = factory.createButton(0).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(0+1),␣
 ↪relief=rel, foreground=fg)
        action.grid(column=0, row=1)

        # Button 2
        rel = factory.createButton(1).getButtonConfig()[0]
        fg  = factory.createButton(1).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(1+1),␣
 ↪relief=rel, foreground=fg)
```

```
        action.grid(column=0, row=2)

        # Button 3
        rel = factory.createButton(2).getButtonConfig()[0]
        fg  = factory.createButton(2).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(2+1),␣
 ↪relief=rel, foreground=fg)
        action.grid(column=0, row=3)

    def createTextFields(self):
        factory = TextFactory()

        # Entry field 1
        sv=tk.StringVar()
        tx = factory.createText(0).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(0).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=1)

        # Entry field 2
        sv=tk.StringVar()
        tx = factory.createText(1).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(1).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=2)

        # Entry field 3
        sv=tk.StringVar()
        tx = factory.createText(2).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(2).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=3)


#=========================
oop = OOP()
oop.win.mainloop()
```

1.9 Logbook Exercise 9

- Extend the Jungwoo Ryoo's Abstract Factory below to mirror the structure used by a statically typed languages by:
- adding a 'CatFactory' and a 'Cat' class with methods that are compatible with 'DogFactory' and 'Dog' respectively
- providing an Abstract Factory class/interface named 'AnimalFactory' and make both the Dog and Cat factories implement this
- providing an AbstractProduct (name this 'Animal') and make both Dog and cat classes implement this
- Use in-code comments (#) to identify the abstract and concrete entities present in Gamma et al. (1995)
- comments should include: "# Abstract Factory #"; "# Concrete Factory #"; "# Abstract Product #"; "# Concrete Product #"; and "# The Client #"
- Implement the CatFactory … the end output should look something like this …

```
Our pet is 'Dog'!
Our pet says hello by 'Woof'!
Its food is 'Dog Food'!

Our pet is 'Cat'!
Our pet says hello by 'Meeoowww'!
Its food is 'Cat Food'!
```

[ ]:
```python
# Abstract Product #
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses should implement the speak␣
 ↪method")

# Concrete Product #
class Cat(Animal):
    def speak(self):
        return "Meeoowww"
    def __str__(self):
        return "Cat"

# Concrete Product #
class Dog(Animal):
    """One of the objects to be returned"""
    def speak(self):
        return "Woof"
    def __str__(self):
        return "Dog"

# Abstract Factory #
class AnimalFactory:
    def get_pet(self):
```

```
        raise NotImplementedError("Subclasses should implement the get_pet␣
 ↪method")
    def get_food(self):
        raise NotImplementedError("Subclasses should implement the get_food␣
 ↪method")

# Concrete Factory #
class CatFactory(AnimalFactory):
    def get_pet(self):
        return Cat()
    def get_food(self):
        return "Cat Food"

# Concrete Factory #
class DogFactory(AnimalFactory):
    """Concrete Factory"""
    def get_pet(self):
        """Returns a Dog object"""
        return Dog()
    def get_food(self):
        """Returns a Dog Food object"""
        return "Dog Food"

# The Client #
class PetStore:
    """ PetStore houses our Abstract Factory """
    def __init__(self, pet_factory=None):
        """ pet_factory is our Abstract Factory """
        self._pet_factory = pet_factory
    def show_pet(self):
        """ Utility method displays details of objects returned by DogFactory␣
 ↪"""
        pet = self._pet_factory.get_pet()
        pet_food = self._pet_factory.get_food()
        print("Our pet is '{}'!".format(pet))
        print("Our pet says hello by '{}'!".format(pet.speak()))
        print("Its food is '{}'!".format(pet_food))


#Create a Concrete Factory
factory = DogFactory()
#Create a pet store housing our Abstract Factory
shop = PetStore(factory)
#Invoke the utility method to show the details of our pet
shop.show_pet()

factory = CatFactory()
```

```
shop = PetStore(factory)
shop.show_pet()
```

1.10 Logbook Exercise 10

- Modify Jungwoo Ryoo's Strategy Pattern to showcase ***OpenCV*** capabilities with different image processing strategies
- We will use the ***OpenCV*** (Open Computer Vision) library which has been reproduced with Python bindings
- OpenCV has many standard computer science image-processing filters and includes powerful AI machine learning algorithms
- The following resources provide more information on OpenCv with Python …
- Beyeler, M. (2015). OpenCV with Python blueprints. Packt Publishing Ltd.
- Joshi, P. (2015). OpenCV with Python by example. Packt Publishing Ltd.
- The cartoon effect is from http://www.askaswiss.com/2016/01/how-to-create-cartoon-effect-opencv-python.html and https://www.tutorialspoint.com/cartooning-an-image-using-opencv-in-python

1.10.1 Development stages

- Install the OpenCV package - we have to do this manually …
- Start Anaconda Navigator
- From Anaconda run the CMD.exe terminal
- At the prompt type … `conda install opencv-python`
- The process may pause with a prompt … `Proceed ([y]/n)?` … just accept this … `y`
- Copy Jungwoo Ryoo's code to a code cell below this one
- As well as the ***types*** module you will need to provide access to OpenCV and numpy as follows

```
# Import OpenCV
import cv2
import numpy as np
```

- Please place a copy of clouds.jpg in the same directory as your Jupyter logbook
- The code for each strategy and some notes on implementing these are below …
- The output should look something like this … but if you wish feel free to experiment with something else … cats etc.!

1.10.2 Implementing image processing strategies

- There will be six strategy objects s0-s5, where s0 is the default strategy of the **Strategy** class
- Instead of assigning a name to each strategy object, you will need to reference the image to be processed - 'clouds.jpg'
- i.e. s0.image = "clouds.jpg"
- The *body* code for each strategy is below, you will need to provide the method signatures and their executions

strategy s0 The default ***execute()*** method that simply displays the image sent to it

```
print("The image {} is used to execute Strategy 0 - Display image".format(self.image))
img_rgb = cv2.imread(self.image)
cv2.imshow('Image', img_rgb)
```

strategy s1 This converts a colour image into a mononochrome one - suggested strategy method name is ***strategy_greyscale***

```
img_rgb = cv2.imread(self.image)
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
cv2.imshow('Greyscale image', img_gray)
```

strategy s2 This applies a blur filter to an image - suggested strategy method name is ***strategy_blur***

```
img_rgb = cv2.imread(self.image)
img_blur = cv2.medianBlur(img_rgb, 7)
cv2.imshow('Blurred image', img_blur)
```

strategy s3 This produces a colour negative image from a colour one - suggested strategy method name is ***strategy_colNegative***

```
img_rgb = cv2.imread(self.image)
  for x in np.nditer(img_rgb, op_flags=['readwrite']):
      x[...] = (255 - x)
  cv2.imshow('Colour negative', img_rgb)
```

strategy s4 This produces a monochrome negative image from a colour one - suggested strategy method name is ***strategy_greyNegative***

```
img_rgb = cv2.imread(self.image)
img_grey = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
for x in np.nditer(img_grey, op_flags=['readwrite']):
    x[...] = (255 - x)
cv2.imshow('Monochrome negative', img_grey)
```

strategy s5 This produces a cartoon-like effect - suggested strategy method name is ***strategy_cartoon***

```
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    img_rgb = cv2.imread(self.image)
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv2.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of applying one large filter
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=9, sigmaSpace=7)
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv2.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale and apply median blur
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect and enhance edges
    img_edge = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BIN
    # Combine color image with edge mask & display picture, convert back to color, bit-AND wit
    img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
    img_cartoon = cv2.bitwise_and(img_color, img_edge)
    # display
    cv2.imshow("Cartoon-ised image", img_cartoon); cv2.waitKey(0); cv2.destroyAllWindows()
```

[ ]:
```
%%python3
# Example from Jungwoo Ryoo (2015)

# The 'types' module is needed to support dynamic creation of new types. In␣
 ↪this case we dynamically create a new method type
import types #Import the types module
import cv2
import numpy as np

# Although 'concrete' this performs the role of the Strategy interface in the␣
 ↪GoF example
class Strategy:
    """The Strategy Pattern class"""

    def __init__(self, function=None):
        self.name = "Default Strategy"
        self.image = "./images/clouds.jpg"

        #If a reference to a function is provided, replace the execute() method␣
 ↪with the given function
        if function:
            self.execute = types.MethodType(function, self)
```

```
    # Although 'concrete' this performs an equivalent role to the GoF␣
↪"AlgorithmInterface()"
    def execute(self): # This gets replaced by another version if another␣
↪strategy is provided.
        print("The image {} is used to execute Strategy 0 - Display image".
↪format(self.image))
        img_rgb = cv2.imread(self.image)
        cv2.imshow('Image', img_rgb)

def strategy_greyscale(self):
    img_rgb = cv2.imread(self.image)
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    cv2.imshow('Greyscale image', img_gray)

def strategy_blur(self):
    img_rgb = cv2.imread(self.image)
    img_blur = cv2.medianBlur(img_rgb, 7)
    cv2.imshow('Blurred image', img_blur)

def strategy_colNegative(self):
    img_rgb = cv2.imread(self.image)
    for x in np.nditer(img_rgb, op_flags=['readwrite']):
        x[...] = (255 - x)
    cv2.imshow('Colour negative', img_rgb)

def strategy_greyNegative(self):
    img_rgb = cv2.imread(self.image)
    img_grey = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    for x in np.nditer(img_grey, op_flags=['readwrite']):
        x[...] = (255 - x)
    cv2.imshow('Monochrome negative', img_grey)

def strategy_cartoon(self):
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    img_rgb = cv2.imread(self.image)
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv2.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of applying one large␣
↪filter
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=9,␣
↪sigmaSpace=7)
```

```
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv2.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale and apply median␣
 ↪blur
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect and enhance edges
    img_edge = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C,␣
 ↪cv2.THRESH_BINARY, blockSize=9, C=2)
    # Combine color image with edge mask & display picture, convert back to␣
 ↪color, bit-AND with color image
    img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
    img_cartoon = cv2.bitwise_and(img_color, img_edge)
    # display
    cv2.imshow("Cartoon-ised image", img_cartoon); cv2.waitKey(0); cv2.
 ↪destroyAllWindows()

# Let's create our default strategy
s0 = Strategy()
# Let's execute our default strategy
s0.execute()

s1 = Strategy(strategy_greyscale)
s1.name = "Strategy Grey Scale"
s1.execute()

s2 = Strategy(strategy_blur)
s2.name = "Strategy Blur"
s2.execute()

s3 = Strategy(strategy_colNegative)
s3.name = "Strategy Colour Negative"
s3.execute()

s4 = Strategy(strategy_greyNegative)
s4.name = "Strategy Grey Negative"
s4.execute()

s5 = Strategy(strategy_cartoon)
s5.name = "Strategy Cartoon"
s5.execute()
```

1.11 Logbook Exercise 11

- Demonstrate the use of `__iter__()`, `__next()__` and `StopIteration` using …
- … the first four items from the top10books list (see above) …

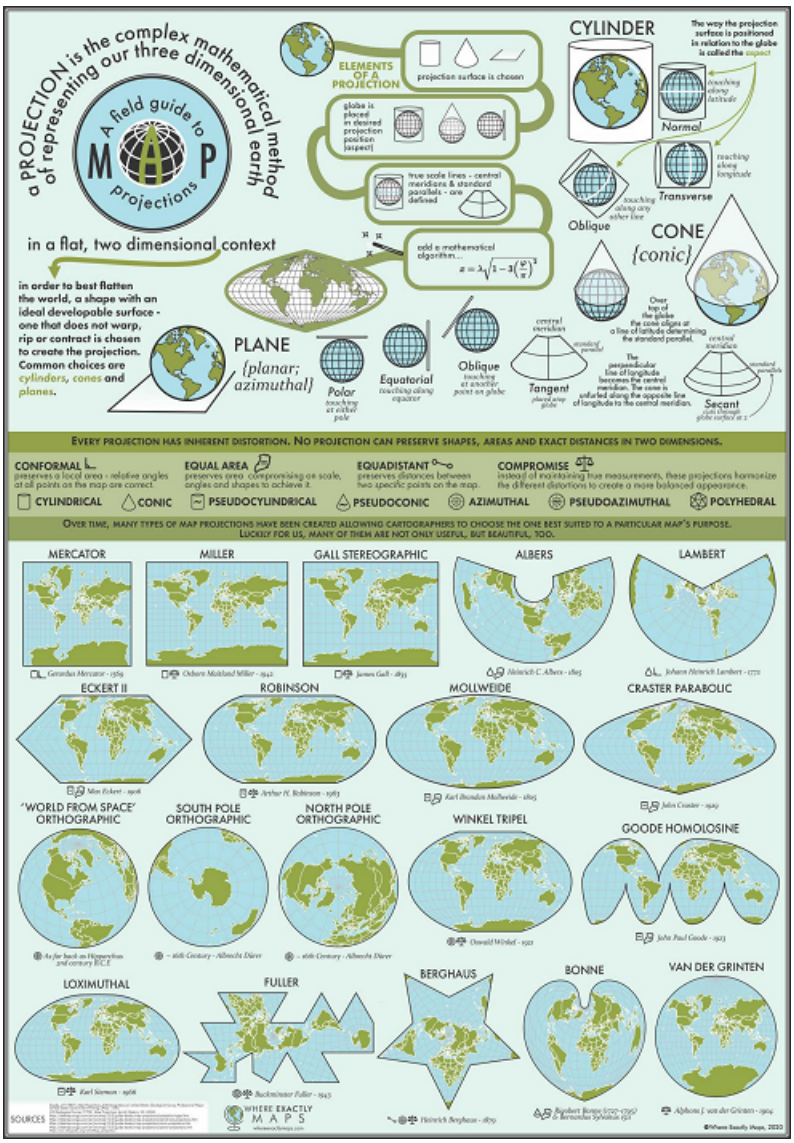
- … and the following structure

```
mylist = ['item1', 'item2', 'item3']

iter_mylist = iter(mylist)

try:
    print( next(iter_mylist))
    print( next(iter_mylist))
    print( next(iter_mylist))
    # Exceeds numbe of items so should raise StopIteration exception
    print( next(iter_mylist))
except Exception as e:
    print(e)
    print(sys.exc_info())
```

[ ]:
```
import sys

top4books = ["Anna Karenina by Leo Tolstoy", "Madame Bovary by Gustave␣
 ↪Flaubert", "War and Peace by Leo Tolstoy", "Lolita by Vladimir Nabokov"]

class BookShelf:
    def __init__(self, books):
        self._books = books

    def __iter__(self):
        self._index = -1
        return self

    def __next__(self):
        self._index += 1
        if len(self._books) <= self._index:
            raise StopIteration()
        return self._books[self._index]

bookshelf = BookShelf(top4books)
iter_mylist = iter(bookshelf)

try:
    print(next(iter_mylist))
    print(next(iter_mylist))
    print(next(iter_mylist))
    print(next(iter_mylist))
    # Exceeds number of items so should raise StopIteration exception
    print( next(iter_mylist))
except Exception as e:
    print(e)
    print(sys.exc_info())
```

1.12 Logbook Exercise 12 - The Adapter DP

- Modify Jungwoo Ryoo's Adapter Pattern example (the one with 'country' classes that 'speak' greetings) to showcase:
- the ***polymorphic*** capability of the Adpater DP
- the geo-data capabilities of ***matplotlib geographical projections*** ...
- in combination with ***Cartopy geospatial data processing*** package to ***produce maps and other geospatial data analyses***.
- A frequent problem in handling geospatial data is that the user often needs to convert it from one form of map projection (essentially a formula to convert the globe into a plane for map-representation) to another map projection



- Fortunately other clever people have written the algorithms we need
- Less fortunately, the interfaces of all the classes that return projections are different

1.13 Development Stages

- We need …
- first, to install the Python cartographic ***Cartopy*** package. In Anaconda launch a CMD.exe terminal and enter the following …

```
conda install -c conda-forge cartopy
```

- to insert a code cell below this one … and copy the extended example of Ryoo's Adapter above (with 'speak' methods in Korean, British and German) in this …
- an ***Adapter*** - Ryoo's adapter is already a well-engineered solution that requires no modification
- then to import some essential packages

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
```

- then add the 'adaptee' classes - here represented by the plot axes and their map projections

```
class PlateCarree:
    def __init__(self):
        self.name = "PlateCarree"
    def project_PlateCarree(self):
        ax = plt.axes(projection=ccrs.PlateCarree())
        return ax

class InterruptedGoodeHomolosine:
    def __init__(self):
        self.name = "InterruptedGoodeHomolosine"
    def project_InterruptedGoodeHomolosine(self):
        ax = plt.axes(projection=ccrs.InterruptedGoodeHomolosine())
        return ax

class AlbersEqualArea:
    def __init__(self):
        self.name = "AlbersEqualArea"
    def project_AlbersEqualArea(self):
        ax = plt.axes(projection=ccrs.AlbersEqualArea())
        return ax

class Mollweide:
    def __init__(self):
        self.name = "Mollweide"
    def project_Mollweide(self):
        ax = plt.axes(projection=ccrs.Mollweide())
        return ax
```

- similarly to Ryoo's example you will need a collection to store projection objects
- again, simlarly to Ryoo, to create all the projection objects (e.g. `plateCarree =PlateCarree()`)
- again, similarly to Ryoo, to append to the collection key-value pairs for each projection and

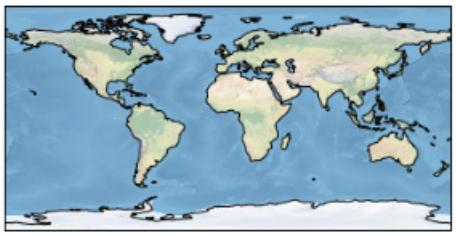
its projection method

- finally to traverse the list of objects to:

```
# Create an axes with the specified projection
ax = obj.project()
# Attach Cartopy's default geospatially registered map/image of the world
ax.stock_img()
# Add the coastlines - highlight these with a black vector
ax.coastlines()
# Print the name of the object/projection
print(obj.name)
# Plot the axes, projection and render the map-image to the projection
plt.show()
```

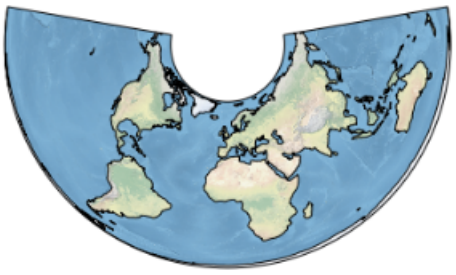
- The output should look something like this ...

PlateCarree

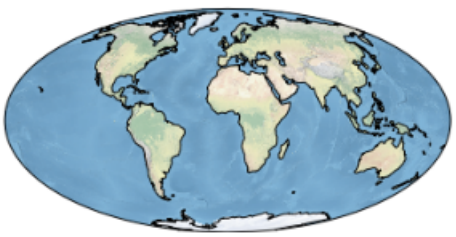


InterruptedGoodeHomolosine

AlbersEqualArea



Mollweide



[ ]:
```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

class PlateCarree:
       def __init__(self):
              self.name = "PlateCarree"
       def project_PlateCarree(self):
              ax = plt.axes(projection=ccrs.PlateCarree())
              return ax

class InterruptedGoodeHomolosine:
       def __init__(self):
              self.name = "InterruptedGoodeHomolosine"
       def project_InterruptedGoodeHomolosine(self):
              ax = plt.axes(projection=ccrs.InterruptedGoodeHomolosine())
              return ax

class AlbersEqualArea:
       def __init__(self):
              self.name = "AlbersEqualArea"
       def project_AlbersEqualArea(self):
              ax = plt.axes(projection=ccrs.AlbersEqualArea())
              return ax

class Mollweide:
       def __init__(self):
              self.name = "Mollweide"
       def project_Mollweide(self):
              ax = plt.axes(projection=ccrs.Mollweide())
              return ax

class Adapter:
       # adapted_methods argument excepts any number of key-value pairs where␣
 ↪the key is the method name and the value is the method to run
       def __init__(self, object, **adapted_methods):
              self._object = object
              # Add all key-value mappings to adapter's own dictionary
              self.__dict__.update(adapted_methods)

       def __getattr__(self, attr):
              # Get the attribute from the stored object if it is not found on␣
 ↪the adapter
              return getattr(self._object, attr)

plateCarree = PlateCarree()
interruptedGoodeHomolosine = InterruptedGoodeHomolosine()
albersEqualArea = AlbersEqualArea()
```

```
mollweide = Mollweide()

adapters = [
       Adapter(plateCarree, project=plateCarree.project_PlateCarree),
       Adapter(interruptedGoodeHomolosine, project=interruptedGoodeHomolosine.
 ↪project_InterruptedGoodeHomolosine),
       Adapter(albersEqualArea, project=albersEqualArea.
 ↪project_AlbersEqualArea),
       Adapter(mollweide, project=mollweide.project_Mollweide)
]

for adapter in adapters:
       ax = adapter.project()
       # Attach Cartopy's default geospatially registered map/image of the␣
 ↪world
       ax.stock_img()
       # Add the coastlines - highlight these with a black vector
       ax.coastlines()
       # Print the name of the object/projection
       print(adapter.name)
       # Plot the axes, projection and render the map-image to the projection
       plt.show()
```

1.14 Logbook Exercise 13 - The Decorator DP

- Repair the code below so that the decorator reveals the name and docstring of aTestMethod()
- Note … the @wrap decorator is NOT needed here

```
<<< Name of the 'decorated' function ...  aTestMethod  >>>
<<< Docstring for the 'decorated' function is ...  This is a method to test the docStringDe
What is your name? ... Buggy Code
Hello ... Buggy Code
```

[ ]:
```
def docStringDecorator(function):
    def printFunctionInfo():
        '''Decorator that automatically reports name and docstring for a␣
 ↪decorated function'''
        print("<<< Name of the 'decorated' function ... ", function.__name__, "␣
 ↪>>> ")
        print("<<< Docstring for the 'decorated' function is ... ", function.
 ↪__doc__, " >>>")
        return function()

    return printFunctionInfo

@docStringDecorator
def aTestMethod():
```

```
    '''This is a method to test the docStringDecorator'''
    nm = input("What is your name? ... ")
    msg = "Hello ... " + nm
    return msg

print(aTestMethod())
```

1.15 Logbook Exercise 14 - The 'conventional' Singleton DP

- Insert a code cell below here
- Copy the code from Dusty Philips' singleton
- Create two objects
- Test output using `print( ... )` and `repr( ... )` as well as the `==` operator to determine whether or not the two objects are the same and occupy the same memory addresses
- Make a note below of your findings

1.15.1 My observations having tested the two objects are ...

Once the first object is instantiated from the class, all subsequent 'new' objects will use the exact same instance. Therefore the memory address will always be the same and making changes to one will affect the others.

[ ]:
```
## SINGLETON - Extended from Dusty Philips (2015) ##

class OneOnly:
    _singleton =  None
    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
            cls._singleton = super(OneOnly, cls).__new__(cls, *args, **kwargs)
        return cls._singleton

obj1 = OneOnly()
obj2 = OneOnly()

print(repr(obj1))
print(repr(obj2))

print("obj1 == obj2 returns " + str(obj1 == obj2))
```

1.16 Logbook Exercise 15 - The 'Borg' Singleton DP

- Repeat the exercise above ...
- Insert a code cell below here
- Copy the code from Alex Martelli's 'Borg' singleton
- Create THREE objects ... ***NOTE***: pass a name for the object when you call the constructor
- Test output using `print( ... )` and `repr( ... )` as well as the `==` operator to determine whether or not the objects are the same and occupy the same memory addresses

- ***Also*** can you use `print( ... )` to test the assertion in the notes above that … "-`_shared_state` is effectively static and is only created once, when the first singleton is instantiated"
- Make a note below of your findings

1.16.1 My observations having tested the three objects are …

For each object that is instantiated they are unique in the sense that they have different memory locations. However they do still share the same internal state (___dict___), so changing an attribute on one will update it on another. The memory location of those instantiated objects' internal dictonary will also always stay the same.

[ ]:
```
# Singleton/BorgSingleton.py
# Alex Martelli's 'Borg'

class Borg:
    _shared_state = {}

    def __init__(self):
        self.__dict__ = self._shared_state
        print("Value of self._shared_state is ..." + str(self._shared_state))

class Singleton(Borg):
    def __init__(self, arg):
        # Here the 'static' Borg class is updated with the state of the new␣
 ↪singleton object
        Borg.__init__(self)
        self.val = arg
    def __str__(self):
        return self.val

print(id(Borg._shared_state))

obj1 = Singleton("s1")
print(id(obj1.__dict__))

obj2 = Singleton("s2")
print(id(obj2.__dict__))

obj3 = Singleton("s3")
print(id(obj3.__dict__))

print(repr(obj1))
print(repr(obj2))
print(repr(obj3))

print("obj1 == obj2 returns " + str(obj1 == obj2))
print("obj1 == obj3 returns " + str(obj1 == obj3))
```

```
print("obj2 == obj3 returns " + str(obj2 == obj3))
```

2 Evaluation and Refactor

This section is for me to extend, modify or reinterpret the implementations above as well as critically evaluate the effectiveness, use cases or alternatives of the design patterns used in that exercise.

2.1 Logbook Exercise 6

This implementation does not use the traditional approach to MVC which normally would allow the view and model to commicate with each other. In instead has all events, UI handling methods and variable values go through the controller which will then pass it to the correct component. I believe this to be a more reusable approach since by removing the coupling between the model and view, it allows you to swap their implementations to adhere to different operating systems or different UI layouts.

The advantage of using the MVC design pattern is that it allows you to separate the logic of your UI from the data. This separation of concern makes it more testable, reusable, and readable. As mentioned above it also allows you to add modifications or completely new implementation (potentially with other libraries/frameworks) without affecting the entire application.

This does come with some disadvantages though. Firstly, in large projects this separation can increase the overall size of the project as you create more files to adhere to the three-part structure, although not as noticable in a small product, this can slow down any attempt to find specific functionality in the code or being able to understand the flow of data when debugging. Furthermore, even though the separation allows you swap out implementations, these new files would usually still need to comply with a specific contract. For example, even though they're decoupled, if you have a really complicated model that the view uses to render a page, you can't just replace this model with one that contains less information, otherwise errors will be thrown by the controller when the view requests data that the model no longer has.

Overall, I think the MVC design pattern can be very beneficial to a project since it gives programmers more freedom on the technologies they can use and, with a good project structure, you can avoid it getting too complicated. Moreover, it is also possible to combine two of the three parts to this design pattern, so if you have a simple component with not very much logic, you could consider combining the controller and model into one.

[ ]:
```
from tkinter import *

class MyController():
    def __init__(self, root):
        # CHANGED - Renamed from parent to root
        self.root = root
        self.view = MyView(self)
        self.model = MyModel(self)
        # CHANGED - Call internal method to update list (remove duplicate code)
        self.listChangedEvent()

    def quitButtonPressed(self):
```

```
        self.root.destroy()

    def addButtonPressed(self):
        # CHANGED - Moved updating of view to when list is changed
        self.model.addToList(self.view.entry_text)

    # CHANGED - Added method to be used by the remove button
    def removeButtonPressed(self):
        # Calls the model to remove an item based on the view's entry text
        self.model.removeFromList(self.view.entry_text)

    # CHANGED - Renamed and used as an event handler for when the list changes
    def listChangedEvent(self):
        # CHANGED - Calls the view to update the displayed items using the␣
↪model's list
        # The label text is set using the model's setter
        self.view.label_text = self.model.list

class MyView(Frame):

    def __init__(self, vc):
        self.vc = vc
        self.frame=Frame(vc.root)
        self.frame.grid(row = 0,column=0)
        # CHANGED - Make these StringVar's private and intialise default value␣
↪in constructor
        self._entry_text = StringVar(vc.root, 'Add to Label')
        self._label_text = StringVar(vc.root)
        self.loadView()

    def loadView(self):
        quitButton = Button(self.frame,text = 'Quit', command= self.vc.
↪quitButtonPressed).grid(row = 0,column = 0)
        addButton = Button(self.frame,text = "Add", command = self.vc.
↪addButtonPressed).grid(row = 0, column = 1)
        # CHANGED - Added button for removing items which uses the␣
↪removeButtonPressed function from the controller
        removeButton = Button(self.frame,text = "Remove", command = self.vc.
↪removeButtonPressed).grid(row = 0, column = 2)
        entry = Entry(self.frame,textvariable = self._entry_text).grid(row = 1,␣
↪column = 0, columnspan = 3, sticky = EW)
        label = Label(self.frame,textvariable = self._label_text).grid(row = 2,␣
↪column = 0, columnspan = 3, sticky = EW)

    # CHANGED - Converted entry and label text methods into actual python␣
↪getters and setters
```

```
    @property
    def entry_text(self):
        return self._entry_text.get()

    @entry_text.setter
    def entry_text(self, text):
        self._entry_text.set(text)

    @property
    def label_text(self):
        return self._label_text.get()

    @label_text.setter
    def label_text(self, items):
        text = "\n".join(items)
        self._label_text.set(text)

class MyModel():
    def __init__(self, vc):
        self.vc = vc
        # CHANGED - Renamed to list and made private, also removed unnecessary␣
↪'count' variable
        self._list = ['duck', 'duck', 'goose']

    # CHANGED - Convert to actual python getter
    @property
    def list(self):
        return self._list

    # CHANGED - Removed unused 'initListWithList' method

    def addToList(self, item):
        # CHANGED - Refactored to directly update list and log on new item
        self._list.append(item)
        # CHANGED - Logs when a new item is added since the list is displayed␣
↪in the view
        print("Added item: " + item)
        # CHANGED - Removed listChanged method and just call the event method␣
↪on controller
        self.vc.listChangedEvent()

    # CHANGED - Added method for removing an item from the list similar to␣
↪adding
    def removeFromList(self, item):
        self._list.remove(item)
        # CHANGED - Logs when an item is removed since the list is displayed in␣
↪the view
```

```
        print("Removed item: " + item)
        self.vc.listChangedEvent()

def main():
    root = Tk()
    # CHANGED - Updated with a more production ready title and removed unused␣
 ↪frame variable
    root.title('List Manager')
    app = MyController(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

2.2 Logbook Exercise 7

The observer pattern is mainly used in situations where a particular event should lead to some other functions being triggered. The advantage of it is that you don't end up with a god class that "knows" which functions to call on which event cause this will everntually become very difficult to add additional functionality to. Instead the function should explicity subscribe to the events itself. This also fixes the alternative of having the subscribing functions constantly check if there are changes or needing to filter the events themselves. Lastly, this approach also allows for a more reactive or asynchronous approach that can be scaled to run the subscriber functions on multiple threads.

There are also some minor disadvantages though with the most prevalent being memory management in programming languages that don't utilise garbage collection. It can be hard to know what object owns the lifecycle of a subscriber and if they are unsubscribed, and then not used again, they could be lost and start causing memory leaks. Additionally, programmers should take care when implementing this design pattern as to not mutate information or introduce dependencies between subscribers as there is no guarenteed order and mutating data could lead to some hard-to-debug side effects.

Overall, if implemented responsibly this design pattern can be super effective as it decouples the source of data from how it is used, in an easy to understand way, and it allows you to keep adding subscribers without affecting the existing functionality.

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np


class Subject(object):

    def __init__(self):
        self._observers = []

    def attach(self, observer):
        if observer not in self._observers:
```

```
            self._observers.append(observer)

    def detach(self, observer):
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notify(self, modifier=None):
        for observer in self._observers:
            if modifier != observer:
                observer.update(self)

class Model(Subject):

    def __init__(self, name):
        Subject.__init__(self)
        self._name = name
        self._labels = ['Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp']
        self._data = [10,8,6,4,2,1]

    @property
    def labels(self):
        return self._labels

    @labels.setter
    def labels(self, labels):
        self._labels = labels
        self.notify()

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, data):
        self._data = data
        self.notify()

class View():

    def __init__(self, name=""):
         self._name = name

    def update(self, subject):
        print("Generalised Viewer '{}' has: \nName = {}; \nLabels = {}; \nData␣
 ↪= {}".format(self._name, subject._name, subject._labels, subject._data))
```

```
class TableView(View):

    def update(self, subject):
        fig = plt.figure(dpi=80)
        ax = fig.add_subplot(1,1,1)
        table_data = list(map(list,zip(subject._labels, subject._data)))
        table = ax.table(cellText=table_data, loc='center')
        table.set_fontsize(14)
        table.scale(1,4)
        ax.axis('off')
        plt.show()

class BarView(View):

    def update(self, subject):
        objects = subject._labels
        y_pos = np.arange(len(objects))
        performance = subject._data
        plt.bar(y_pos, performance, align='center', alpha=0.5)
        plt.xticks(y_pos, objects)
        plt.ylabel('Usage')
        plt.title('Programming language usage')
        plt.show()

# Code added as part of main exercise
class PieView(View):

    def update(self, subject):
        labels = subject.labels
        sizes = subject.data
        explode = (0.1, 0, 0, 0, 0, 0)
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',␣
 ↪shadow=True, startangle=90)
        ax1.axis('equal')
        plt.show()

# CHANGED - Removed unused model
m1 = Model("Model 1")

# CHANGED - Converted from inline variable
m1.attach(View("1: standard text viewer"))
m1.attach(TableView("2: table viewer"))
m1.attach(BarView("3: bar chart viewer"))
m1.attach(PieView("4: pie chart viewer"))
```

```
m1.notify()

m1.labels = ['C#','PHP','JavaScript','ASP','Python','Smalltalk']
m1.data = [1,18,8,60,3,1]
```

2.3 Logbook Exercise 8

The factory pattern is used when you want to leave the creation of an object up to class implementing the interface. This has the benefit of removing the coupling between the product and the creator, since it no longer needs to know which type of object to create. The alternative is to have a God class, instead of a creator class, that contains methods for all the different implementations for creating objects. By extracting these functions out into subclasses, which implement a common interface, you have introduced separation of concerns, that in turn allows you to add new implementations without affecting the code of the creator class.

There are only a few minor downsides to this pattern, firstly that each implementation has to comply with a specific interface which should be easy to do with proper user of dependency injection. Secondly, for large projects, this could end up generating a lot of subclasses making it more difficult to traverse or understand but I believe that this is very minor as long as the file structure is appropriate and the interface has easy to follow method names.

[ ]:
```
%%python3
import tkinter as tk
from tkinter import ttk
from tkinter import Menu

# CHANGED - Moved to using getters for the properties and marked variables as␣
 ↪private
class ButtonBase():
    _relief     = 'flat'
    _foreground = 'white'

    @property
    def relief(self):
        return self._relief

    @property
    def foreground(self):
        return self._foreground

class ButtonRidge(ButtonBase):
    _relief     = 'ridge'
    _foreground = 'red'

class ButtonSunken(ButtonBase):
    _relief     = 'sunken'
    _foreground = 'blue'
```

```
class ButtonGroove(ButtonBase):
    _relief     = 'groove'
    _foreground = 'green'

# CHANGED - Moved to using getters for the properties, marked variables as␣
 ↪private and renamed textvariable to text
class TextBase():
    _text       = "black"
    _background = "white"

    @property
    def text(self):
        return self._text

    @property
    def background(self):
        return self._background

# CHANGED - Renamed classes to more readable names
class TextRed(TextBase):
    _text       = "red"
    _background = "red"

class TextBlue(TextBase):
    _text       = "blue"
    _background = "blue"

class TextGreen(TextBase):
    _text       = "green"
    _background = "green"

# CHANGED - Created Creator class to be extended by other creators
#           Moved from using a list to a dictionary for better readability and␣
 ↪usability
class FactoryBase():
    types = {}

    def create(self, _type):
        # CHANGED - Checks type exists before returning, to allow for a more␣
 ↪meaningful error message should it not exist
        if _type not in self.types:
            raise Exception(f"Type '{_type}' is not supported")

        return self.types[_type]()

# CHANGED - Updated to use creator base class and types dictionary
class ButtonFactory(FactoryBase):
```

```
    types = {
        "ridge": ButtonRidge,
        "sunken": ButtonSunken,
        "groove": ButtonGroove
    }

# CHANGED - Updated to use creator base class and types dictionary
class TextFactory(FactoryBase):
    types = {
        "red": TextRed,
        "blue": TextBlue,
        "green": TextGreen
    }

class OOP():
    def __init__(self):
        self.win = tk.Tk()
        self.win.title("Python GUI")
        self.createWidgets()

    def createWidgets(self):
        self.widgetFactory = ttk.LabelFrame(text=' Button Factory ')
        self.widgetFactory.grid(column=0, row=0, padx=8, pady=4)

        self._createButtons()
        self._createTextFields()

    # CHANGED - Marked function as private
    def _createButtons(self):
        factory = ButtonFactory()
        typesToDisplay = ["ridge", "sunken", "groove"]

        for position, buttonType in enumerate(typesToDisplay, 1):
            # The create method only needs to be called once per button
            self._createButton(position, factory.create(buttonType))

    # CHANGED - Moved common code for creating a button into this private␣
 ↪function
    def _createButton(self, position, button):
        # CHANGED - Text for button now uses an f string and other values are␣
 ↪retrieved from button object
        action = tk.Button(self.widgetFactory, text=f"Button {position}",␣
 ↪relief=button.relief, foreground=button.foreground)
        action.grid(column=0, row=position)

    # CHANGED - Marked function as private
    def _createTextFields(self):
```

```
        factory = TextFactory()
        typesToDisplay = ["red", "blue", "green"]

        for position, textType in enumerate(typesToDisplay, 1):
            # The create method only needs to be called once per button
            self._createTextField(position, factory.create(textType))

    # CHANGED - Moved common code for creating a text field into this private␣
 ↪function
    def _createTextField(self, position, textField):
        # CHANGED - Values are now retrieved from text object
        stringVar = tk.StringVar()
        stringVar.set(textField.text)
        action = tk.Entry(self.widgetFactory, textvariable=stringVar,␣
 ↪background=textField.background, foreground="white")
        action.grid(column=1, row=position)

#=========================
oop = OOP()
oop.win.mainloop()
```

2.4 Logbook Exercise 9

The abstract factory pattern is similar to the factory pattern except it allows for a collection of related objects as well. This also shares the benefits of the factory method in that it simplifies your creator class since it only needs to interact with one interface and adding new implementations of this interface doesn't affect that. Additionally, it also ensures that objects created from the same factory are related and compatible with each other.

However, the amount of extra classes you would be generating is worse compared to the factory method. If you had an interface with four methods that had been implemented by five other classes then adding a fifth method would require you to make changes to those five classes which in turn would require you to introduce five more object. Over time this number could get larger and larger leading to a very complicated file structure.

In conclusion this design pattern can be very effective as long as your whole program is based around this pattern, since adding more implementations could end up becoming very cumbersome and time consuming.

[ ]:
```
from abc import ABC, abstractmethod

# CHANGED - Implemented abstract class using Python's abstract class module␣
 ↪(abc)
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
```

```
class Cat(Animal):
    def speak(self):
        return "Meeoowww"
    def __str__(self):
        return "Cat"

class Dog(Animal):
    def speak(self):
        return "Woof"
    def __str__(self):
        return "Dog"

# CHANGED - Implemented abstract class using Python's abstract class module␣
 ↪(abc)
class AnimalFactory(ABC):
    # CHANGED - Added Animal type hint for the return value
    @abstractmethod
    def get_pet(self) -> Animal:
        pass
    # CHANGED - Added str type hint for the return value
    @abstractmethod
    def get_food(self) -> str:
        pass

class CatFactory(AnimalFactory):
    def get_pet(self):
        return Cat()
    def get_food(self):
        return "Cat Food"

class DogFactory(AnimalFactory):
    def get_pet(self):
        return Dog()
    def get_food(self):
        return "Dog Food"

class PetStore:
    # CHANGED - Added a AnimalFactory type hint
    def __init__(self, pet_factory: AnimalFactory=None):
        self._pet_factory = pet_factory

    # CHANGED - Added a setter with a type hint, so a new instance of pet store␣
 ↪doesn't need to be created per factory
    def set_pet_factory(self, pet_factory: AnimalFactory):
        self._pet_factory = pet_factory

    def show_pet(self):
```

```
        # CHANGED - Added check for if a factory has not been provided
        if not self._pet_factory:
            print("We currently have no pets")
            return

        pet = self._pet_factory.get_pet()
        pet_food = self._pet_factory.get_food()
        print("Our pet is '{}'!".format(pet))
        print("Our pet says hello by '{}'!".format(pet.speak()))
        print("Its food is '{}'!".format(pet_food))


dog_factory = DogFactory()
cat_factory = CatFactory()
shop = PetStore()

shop.set_pet_factory(dog_factory)
shop.show_pet()

shop.set_pet_factory(cat_factory)
shop.show_pet()
```

2.5 Logbook Exercise 10

The strategy pattern is helpful when you have a family of implementations that produce the same type of outcome but with different inputs and you need to separate these out. Similarly to the factory patterns above, the alternative is to have a god class that just contains methods with each algorithm and then when it is called it will check the inputs (most likely with a switch or if..else statement) and delegate it to the correct method. By having all these algorithms implement a common interface you can easily switch out these objects and call the same method name with different outcomes.

The downside again being that this introduces many subclasses, plus users of this need to make sure they understand which implementation they are using, since from their point of view it is just an interface with specific methods and you don't want the situation where you called the wrong function without realising.

In conclusion I do think this is an effective approach for a large code base but programmers should be aware that if it is a small set of implementations that don't change very often, then it may be best not to overcomplicate the project and just keep them together, or better yet, make use of more modern programming features that allow for anonymous, in-line functions.

[ ]:
```
%%python3
import types
import cv2
import numpy as np

class Strategy:
```

```
    # CHANGED - The strategy function no longer replaces the execute method but␣
 ↪is what is called to process the image
    # The Gamma et al says to have a class which implements a common method e.g.
 ↪ "process_image()" but that has been skipped for simplicity's sakes
    # The name has also been added to the constructor so that is clearer to the␣
 ↪programmer that it needs to be changed
    def __init__(self, name: str="Default", process_image_strategy=None):
        self.image = "./images/clouds.jpg"
        # CHANGED - Name made private and should be changed via argument
        self._name = name
        self._process_image = process_image_strategy or self._strategy_default

    def _strategy_default(self, image):
        return image

    # CHANGED - This is no longer overriden and instead calls the method which␣
 ↪is passed in, with the read image as an argument
    # It expects to get a processed image back which it can display. This is to␣
 ↪avoid the duplicated code of reading then displaying the image.
    def execute(self):
        image = cv2.imread(self.image)
        processed_image = self._process_image(image)
        cv2.imshow(self._name + ' Image', processed_image)

# CHANGED - Updated to use passed in image and return processed one
def strategy_greyscale(image):
    return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# CHANGED - Updated to use passed in image and return processed one
def strategy_blur(image):
    return cv2.medianBlur(image, 7)

# CHANGED - Updated to use passed in image and return processed one. The image␣
 ↪is copied as to not mutate the original image instance.
def strategy_colNegative(image):
    img_negative = image.copy()
    for x in np.nditer(img_negative, op_flags=['readwrite']):
        x[...] = (255 - x)
    return img_negative

# CHANGED - Updated to use passed in image and return processed one
def strategy_greyNegative(image):
    img_grey = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    for x in np.nditer(img_grey, op_flags=['readwrite']):
        x[...] = (255 - x)
    return img_grey
```

```
# CHANGED - Updated to use passed in image and return processed one, also␣
 ↪copies the image to prevent mutation
def strategy_cartoon(image):
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    # downsample image using Gaussian pyramid
    img_color = image.copy()
    for _ in range(num_down):
        img_color = cv2.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of applying one large␣
 ↪filter
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=9,␣
 ↪sigmaSpace=7)
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv2.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale and apply median␣
 ↪blur
    img_gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect and enhance edges
    img_edge = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C,␣
 ↪cv2.THRESH_BINARY, blockSize=9, C=2)
    # Combine color image with edge mask & display picture, convert back to␣
 ↪color, bit-AND with color image
    img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
    return cv2.bitwise_and(img_color, img_edge)

s0 = Strategy()
s0.execute()

# CHANGED - All strategies updated to match new constructor args
s1 = Strategy("Grey Scale", strategy_greyscale)
s1.execute()

s2 = Strategy("Blur", strategy_blur)
s2.execute()

s3 = Strategy("Colour Negative", strategy_colNegative)
s3.execute()

s4 = Strategy("Grey Negative", strategy_greyNegative)
s4.execute()
```

```
s5 = Strategy("Cartoon", strategy_cartoon)
s5.execute()

# CHANGED - Moved these options out of the cartoon strategy since they are␣
 ↪unrelated to that image processing method
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2.6 Logbook Exercise 11

The iterator pattern is used when you want to traverse a collection but provide a facade that abstracts the underlying structure from the user.

The advantages of such are that you can abstract any complicated traversal logic behind predetermined methods such as `next()` and `hasNext()`. This further decouples your implementation from the collection type you use allowing you to potentially swap this out in the future if, for example, you find a more optimised collection. Moreover, the iterator pattern allows you to have multiple iterators for the same object since they are independent of each other (as long as you're not mutating the underlying colletion) and you can pause processing and continue at any time.

It is worth noting however, that this pattern can very easily be misused since most the time, for simple collections like arrays, it is cleaner to just loop over it like you would traditionally or for overly complex collections, it may be more recommended to just use its own methods rather than wrapping it in less efficient iterator.

[ ]:
```
import sys

top4books = ["Anna Karenina by Leo Tolstoy", "Madame Bovary by Gustave␣
 ↪Flaubert", "War and Peace by Leo Tolstoy", "Lolita by Vladimir Nabokov"]

class BookShelf:
    def __init__(self, books):
        self._books = books

    def __iter__(self):
        self._index = -1
        return self

    def __next__(self):
        # CHANGED - Added a check in case the iter dunder method has not been␣
 ↪run
        if self._index is None:
            raise Exception("Iterator has not been initialised")

        self._index += 1
        if len(self._books) <= self._index:
            raise StopIteration()
```

```
        return self._books[self._index]

bookshelf = BookShelf(top4books)
# CHANGED - Renamed to be clearer
iter_bookshelf = iter(bookshelf)

try:
    # CHANGED - Updated to use a while loop so the length of the list does not␣
 ↪matter
    while item := next(iter_bookshelf):
        print(item)
# CHANGED - Be more specific with the catch cause in case something unrelated␣
 ↪to StopIteration is thrown and then accidentally ignored
# This is inline with the "Zen of python" that states "Errors should never pass␣
 ↪silently. Unless explicitly silenced"
except StopIteration as e:
    print(e)
    print(sys.exc_info())
```

2.7 Logbook Exercise 12

The adapter pattern is extrememly useful when you have a bunch of classes that all do similar things in their own implementation, but they all have different method names that need to be converted into more of a standard using an interface. This also has the added bonus that you can very easily add additional adapters, without needing to change or break other implementations.

However, it is best to only use this design pattern if you are trying to integrate with a library or code that you are unable to change. There is no benefit to introducing complex interfaces and multiple adapter classes into your codebase if you are able to just change the source code instead.

[ ]:
```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

class PlateCarree:
       def __init__(self):
              self.name = "PlateCarree"
       def project_PlateCarree(self):
              ax = plt.axes(projection=ccrs.PlateCarree())
              return ax

class InterruptedGoodeHomolosine:
       def __init__(self):
              self.name = "InterruptedGoodeHomolosine"
       def project_InterruptedGoodeHomolosine(self):
              ax = plt.axes(projection=ccrs.InterruptedGoodeHomolosine())
              return ax

class AlbersEqualArea:
```

```
        def __init__(self):
                self.name = "AlbersEqualArea"
        def project_AlbersEqualArea(self):
                ax = plt.axes(projection=ccrs.AlbersEqualArea())
                return ax

class Mollweide:
        def __init__(self):
                self.name = "Mollweide"
        def project_Mollweide(self):
                ax = plt.axes(projection=ccrs.Mollweide())
                return ax

# CHANGED - Updated name to be more specific
class ProjectionAdapter:
        # CHANGED - The supported adapted classes are now stored as a class␣
 ↪instance
        # It is a dictionary of the supported classes as the keys and the method␣
 ↪to use as the value
        # If more complicated logic is needed the value could be changed to a␣
 ↪lambda or method reference
        supportedTypes = {
                PlateCarree: "project_PlateCarree",
                InterruptedGoodeHomolosine: "project_InterruptedGoodeHomolosine",
                AlbersEqualArea: "project_AlbersEqualArea",
                Mollweide: "project_Mollweide"
        }

        # CHANGED - The user no longer needs to map the methods themselves
        def __init__(self, object):
                self._object = object

                # Will throw an exception if the type of the object is not␣
 ↪supported
                methodName = self.supportedTypes.get(type(object))
                if not methodName:
                        raise LookupError("Unsupported projection class '{0}'".
 ↪format(type(object).__name__))

                # Set the adapters method to be that of the object to adapt, but␣
 ↪under a normalised name
                self.project = getattr(object, methodName)

        def __getattr__(self, attr):
                # Get the attribute from the stored object if it is not found on␣
 ↪the adapter
```

```
            return getattr(self._object, attr)

adapters = [
        ProjectionAdapter(PlateCarree()),
        ProjectionAdapter(InterruptedGoodeHomolosine()),
        ProjectionAdapter(AlbersEqualArea()),
        ProjectionAdapter(Mollweide())
]

for adapter in adapters:
        ax = adapter.project()
        # Attach Cartopy's default geospatially registered map/image of the␣
 ↪world
        ax.stock_img()
        # Add the coastlines - highlight these with a black vector
        ax.coastlines()
        # Print the name of the object/projection
        print(adapter.name)
        # Plot the axes, projection and render the map-image to the projection
        plt.show()
```

2.8 Logbook Exercise 13

In the original implementation, the decorator function did not take advantage of Python's `@wraps` annotation. Although this does not effect the execution of this method, it does overwrite any useful information that could be useful to other programmers such as the docstrings or the underlying `__name__` of the function.

The wrapper pattern is useful when you have a "chain" of operations that you need to do but want to avoid having a God class which has these implementations in it. The advantages of this pattern are that you can change which operations are executed at runtime without needing to make large code changes and if you needed to add more operations then it is as simple as making another wrapper class.

This can come with some downsides including the fact that these wrappers can sometimes depend on others which can cause unexected bugs if users don't order them correctly. This can also make it difficult to debug the order in which these wrappers are executed or if they are run at all. Lastly, if not properly decoupled from each other, you lose the advantage of being able to add or remove operations at runtime.

[ ]:
```
from functools import wraps

def docStringDecorator(function):
    @wraps(function)
    def printFunctionInfo():
        '''Decorator that automatically reports name and docstring for a␣
 ↪decorated function'''
```

```
        print("<<< Name of the 'decorated' function ... ", function.__name__, "␣
↪>>> ")
        print("<<< Docstring for the 'decorated' function is ... ", function.
↪__doc__, " >>>")
        return function()

    return printFunctionInfo

@docStringDecorator
def aTestMethod():
    '''This is a method to test the docStringDecorator'''
    nm = input("What is your name? ... ")
    msg = "Hello ... " + nm
    return msg

print(aTestMethod())
assert aTestMethod.__name__ == "aTestMethod"
assert aTestMethod.__doc__ == "This is a method to test the docStringDecorator"
```

2.9 Logbook Exercise 14 & 15

The singleton pattern is one of the most well known and highly used design patterns. With this design pattern you can ensure that an object is only instantiated once which can save on resources or doing expensive operations in the constructor that only need to be done once, such as connecting to a database.

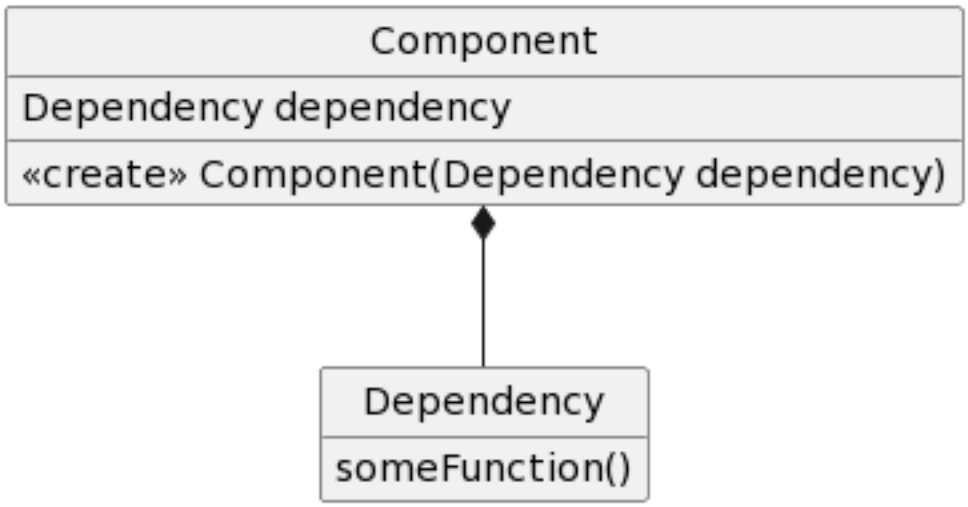
Although it is highly used, it can also be commonly misused. Firstly, in an application that is multithreaded, programmers need to be aware of what could happen if a singleton's methods are used concurrently or that the object is not accidentally getting instantiated multiple times due to multiple sources trying to access it simultaneously. This pattern could also potentially be hiding code smells that are "fixed" with the use of a global instance, such as other objects "knowing" too much about the singleton. Finally it could also impact how you test your classes since it can be hard to mock global instances, this however could be fixed with the dependency injection design pattern.

3 Two Additional Design Patterns

This section details two additional design patterns, that have not been covered by the lecture materials, where I will provide a description on the intent, motivation, structure, and evaluate their effectiveness with an example implemented in python.

3.1 Logbook Exercise 16

3.2 Dependency Injection Design Pattern



Dependency injection is a creational design pattern with the intent to decouple the creation of dependencies from the parent, to help conform to programming principles such as SOLID's "Single responsibility". The main motivation behind this is that the alternatives to this lead to less cohesive code and make testing impossible or complicated, due to the need to potentially mock objects created within the class. An example alternative is using a factory that has a get method for generating objects, although this removes the responsibility of creating the object from the class, it still needs to know what to call on the factory, and mocking the response of this factory (so that we can control the object that is returned) could prove difficult.

3.2.1 Advantages

As previously stated a few of the advantages are that it reduces coupling, increases cohesion and makes testing easier since mocked objects can passed directly into the constructor. Additionally, it can reduce the overall code since these components can be more easily reused and singletons don't need to be given special code to prevent multiple instantiations since instead the single instance is just passed around everywhere. One final advantage is that it can also have the added benefit of providing documentation on the dependencies for a component so that it is clear to the user what it is using.

3.2.2 Disadvantages

Similar to all patterns it does come with some potential disadvantages if not used responsibly. This design pattern is an implementation of the Inversion of Control pattern, which has the intent of redirecting the control of something to an external component or process. With dependency injection you are redirecting control of dependency creation, to the component calling the constructor, however, you do not want to have object creation scattered throughout your project and would need to implement a mechanism for and storing all object instances in one location. Fortunately, there are libraries out there to assist with this, an example for Java would be Spring. Another point is that your constructors could become very cumbersome if a component has a lot of dependencies,

and it can then be hard to make these optional or default them. For external libraries this can lead to a much worse user experience as the user needs to understand every dependency the component uses and what to use as a default, even if they're only utilising a single part of the functionality. Luckily, this can also be solved with a different approach to dependency injection through the users of setters. Although this does introduce mutability into your component, the constructor would then only need to have the mandatory dependencies and can instatiate its own defaults, that would get overriden should the user need to use the set methods to change them. On the other hand, you could also overload the constructor to take different combinations of dependencies but programmers should be careful not to create too many as not to overcomplicate the code.

3.2.3 Implementation

Below is an example implementation of a service which uses a database to get a list of users. There is also a handler which then uses this service to get information about a particular user. Both the service and handler both have a required dependency which is passed in using the two methods detailed above: constructor and setters.

[ ]:
```python
from abc import ABC, abstractmethod

class Database:
    def execute(self, sqlQuery: str):
        # Pretending that the query is executed and returns a list of users
        print("Executing request: " + sqlQuery)
        return [{ "id": 1, "name": "Eve" }, { "id": 2, "name": "Jack" }, { "id": 3, "name": "Charlie" }]

# Service interface
class UserService(ABC):
    @abstractmethod
    def get_users(self) -> list:
        pass

class DatabaseUserService(UserService):
    # The other way to do dependency injection is through setters
    # This is more useful if you have a lot of dependencies or to make managing␣
    ↪defaults easier
    def set_database(self, database: Database):
        self.database = database

    def get_users(self):
        if not self.database: raise Exception("Database has not been set")
        return self.database.execute("SELECT * FROM users;")

class UserHandler:
    # Does not need to find a global instance or instantiate a class cause it␣
    ↪is passed in
    # Also uses the interface as the type so service implementation can be␣
    ↪easily swapped out
```

```
    def __init__(self, service: UserService):
        self.service = service

    def get_user_info(self, id: int):
        users = self.service.get_users()
        return next(user for user in users if user["id"] == id)

# Could use another component which manages all object instances within the␣
 ↪program
database = Database()
service = DatabaseUserService()
service.set_database(database)
handler = UserHandler(service)

print(handler.get_user_info(2))
```

3.3 Logbook Exercise 17

3.4 Reactor Design Pattern

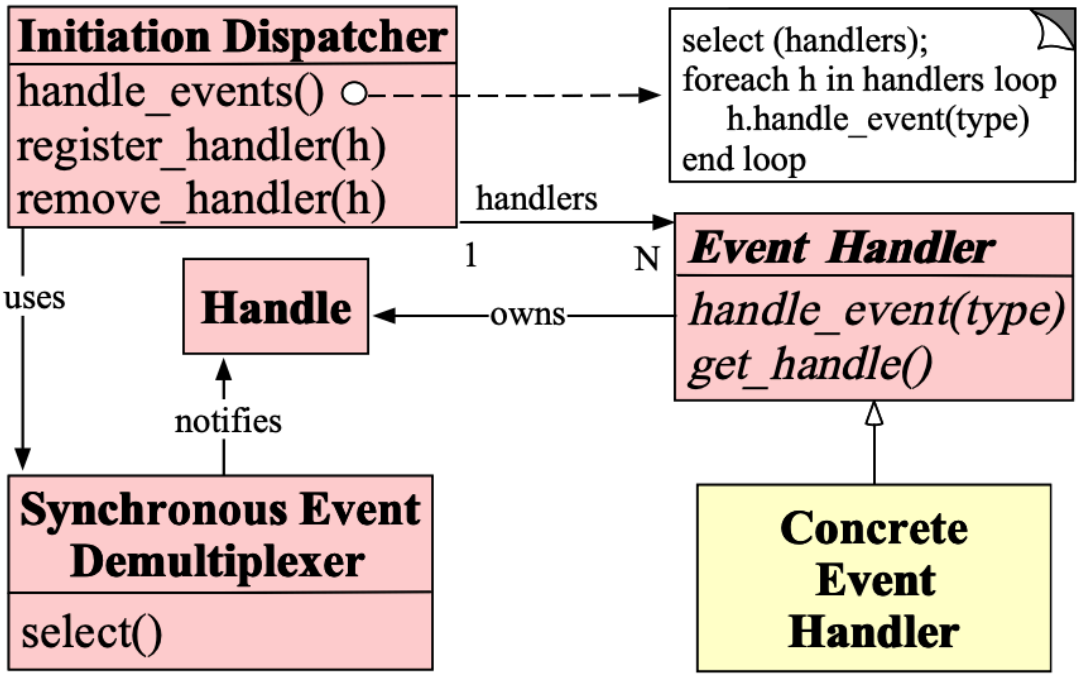


Diagram from Schmidt and Coplien, 1995 (page 3)

The Reactor design pattern is a concurrency design pattern with the intent of handling concurrent requests from one or more clients. All functionality should be given an event handler which implements a common interface such that when an event comes in, it can be dispatched to its corresponding event handler. The motivation for this design pattern is to avoid needing to man-

age multiple threads which can cause concurrency issues, usually fixed with locks or synchronising threads for specific functions. This design pattern instead has a singular event loop which takes in an event and execute it on an event handler. This event handler runs in the same thread but should not block it, so for example, if it sends a HTTP request it should free up that thread and once the response comes back, a new event is generated.

3.4.1 Advantages

The best way to see the advantages of this pattern are to compare it to alternative solutions. Consider an API that takes in GET, ADD and DELETE requests which are forwarded to a database: - The non-reactor solution is to generate a new thread for each request that comes in, it needs to determine what type of request they have sent, then call the database and wait for its response, then return the database response to the user. Throughout this journey the application had to do a lot of work to determine how to handle the request, waste time by waiting for the databases response, and it involves the overhead of creating a new thread. - The reactor solution on the otherhand would receive the request and dispatch an event to the handler which is responsible for it. The handler would then process the request and send a call to the database where it would then release the thread. Once the response for the database comes back, it will go back through the event handler which would then dispatch a different event, this time to the handler which handles returning the response to the original caller.

As shown in the example above, the reactor pattern makes better use of resources by not managing multiple threads and not executing non-blocking calls but instead delegating that to a callback approach. Furthermore, this also introduces a separation of concern since each handler should only be executing non-blocking code, if you need to wait for a response, it should instead be handled by a second handler under a different event. This also improves upon the scalability of the application, since eight instances of a reactor application would be using the same amount of resources as a non-reactor application with eight threads, while being able to handle more requests.

Lastly, in low level languages the managing of threads is in control of the programmer which leads to complicated, custom multithread implementations that can be hard to maintain.

3.4.2 Disadvantages

Understandably, there are some potential problems that can arise from this pattern. Firstly, it should be noted that each even handler can only handle quick, non-blocking requests, so if you are developing an image server for storing images that users upload, the handler for handling the saving of these images, will probably take some considerable time depending on the size which would ultimately cause events to back up.

Moreover, the implementation of the handler is very important as developers need to make sure they are not accidentally introducing blocking code, and any exceptions thrown by the handlers need to be correctly handled otherwise it will bring down the entire application.

Finally, from a debugging point of view it is very difficult to follow the flow of data due to how events are processed in order of arrival and how instead of blocking it uses callbacks. If trying to "step through" the process, developers will find it difficult since it is not a simple case as going line by line but instead following the program through its complex infrastructure of reading the event and dispatching it to the correct handler.

3.4.3 Implementation

Below is an example of a simplified Reactor implementation. Normally, there are a lot more components which make up a reactor application as it improves the readability and adheres to SOLID principles (like single responsibility) better.

[ ]:
```
# Example modified from jpanganiban's, "Simple Reactor Pattern Implementation"␣
 ↪- https://gist.github.com/jpanganiban/4221200

from typing import Callable
import threading
import traceback

class EventLoop:
    def __init__(self):
        self._handlers = {}

    def on(self, event: str, handler: Callable[[any], None]):
        handlers = self._handlers.get(event, [])
        if not handler in handlers:
            handlers.append(handler)
            self._handlers[event] = handlers

    def emit(self, event: str, value):
        handlers = self._handlers.get(event, [])
        # If there are any handlers loop through and execute them
        # Should an exception occur it will just print it to the console but␣
 ↪error handlers or a default error handler could be implemented instead
        for handler in handlers:
            try:
                handler(value)
            except Exception:
                print(traceback.format_exc())

eventLoop = EventLoop()

# Simulating an external logging service that needs to be called
# Once the call has been made it can process other events (if any) while it␣
 ↪waits for a response
# The response will come back in the form of another event hence why it doesn't␣
 ↪need to wait
def callLoggingService(val):
    threading.Timer(0.5, lambda: eventLoop.emit("log", val)).start()

def logMultiply(value):
    print("Sending multiply result to logger...")
    callLoggingService(value * 2)
```

```
def logDivide(value):
    print("Sending divide result to logger...")
    callLoggingService(value / 2)

def logValue(value):
    print(f"Log: {value}")

def raiseException(value):
    raise Exception(f"Throwing with value {value}")

eventLoop.on("multiply", logMultiply)
eventLoop.on("divide", logDivide)
eventLoop.on("log", logValue)
eventLoop.on("throw", raiseException)

# Simulating asyncronous calls coming into the event loop
threading.Timer(2, lambda: eventLoop.emit("multiply", 5)).start() # Will␣
 ↪multiply by 2 and print the result
threading.Timer(3, lambda: eventLoop.emit("multiply", 15)).start() # Will␣
 ↪multiply by 2 and print the result
threading.Timer(1, lambda: eventLoop.emit("divide", 10)).start() # Will divide␣
 ↪by 2 and print the result
eventLoop.emit("throw", 1) # Throws exception before other events are emitted␣
 ↪which is cleanly handled by event loop
```

4 References

- Bhat, S. and Hegde, V. (2021). Overview of Dependency Injection Design Pattern. International Journal of Scientific Research and Engineering Development, [online] 4(3), pp.1546–1549. Available at: http://www.ijsred.com/volume4/issue3/IJSRED-V4I3P213.pdf [Accessed 22 Sep. 2023].
- developer.apple.com. (n.d.). Model-View-Controller. [online] Available at: https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html [Accessed 1 Sep. 2023].
- java-design-patterns.com. (n.d.). Dependency Injection. [online] Available at: https://java-design-patterns.com/patterns/dependency-injection/ [Accessed 22 Sep. 2023].
- kamituel (2014). What are the differences between event-driven and thread-based server system? [online] Stack Overflow. Available at: https://stackoverflow.com/a/25280366 [Accessed 22 Sep. 2023].
- LinkedIn (2023a). What are the benefits and drawbacks of using observer pattern in event-driven systems? [online] www.linkedin.com. Available at: https://www.linkedin.com/advice/0/what-benefits-drawbacks-using-observer-pattern [Accessed 1 Sep. 2023].
- LinkedIn (2023b). What are the benefits and drawbacks of using the Model-View-Controller (MVC) pattern? [online] www.linkedin.com. Available at: https://www.linkedin.com/advice/3/what-benefits-drawbacks-using-model-view-controller [Accessed 1 Sep. 2023].

- Professionalqa.com. (2017). Dependency Injection |Professionalqa.com. [online] Available at: https://professionalqa.com/dependency-injection [Accessed 22 Sep. 2023].
- Refactoring Guru (2014). The Catalog of Design Patterns. [online] refactoring.guru. Available at: https://refactoring.guru/design-patterns/catalog [Accessed 15 Sep. 2023].
- Schmidt, D. and Coplien, J. (1995). Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events. [online] Available at: http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf [Accessed 22 Sep. 2023].