Design Patterns Logbook

Stefan Allen

ID: 22135474

26/01/2024

GITHUB LINK: https://github.com/Stefan-Allen/DesignPatterns

Logbook Exercise 1 Task

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', 'milk', 'tea']
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese', 'tea',
'coffee', 'rice', 'pasta', 'milk', 'tea')
This is my Shopping_set with duplicates removes {'rice', 'milk',
```

```
'pasta', 'cheese', 'eggs', 'tea', 'bread', 'coffee'}
This is my 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'}
```

Unit 1 Exercise

```
#The set should be {} not () as stated in the exercise

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 = {'milk', 'eggs', 'bread', 'cheese', 'tea', 'coffee',
'rice', 'pasta', 'milk', 'tea'}
shopping_list = list(set(shopping_set))
print("This is my Shopping_set with duplicates removes",
shopping_list)

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_dict", shopping_dict)
```

```
This is my shopping list ['milk', 'eggs', 'bread', 'cheese', 'tea',
'coffee', 'rice', 'pasta', 'milk', 'tea']
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese', 'tea',
'coffee', 'rice', 'pasta', 'milk', 'tea')
This is my Shopping_set with duplicates removes ['bread', 'tea',
'coffee', 'milk', 'rice', 'pasta', 'eggs', 'cheese']
This is my 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'}
```

Logbook Exercise 2 Task

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

```
#Unit 2 Exercise
li = ["USA","Mexico","Canada"]

print("li.append ")
li.append("Greenland")
print(li)

print("li.remove ")
li.remove("Greenland")
print(li)

print("li.insert ")
li.insert(0, "Greenland")
print(li)

print("li[:2] ")
print(li[:2])

print("li[-2] ")
print(li[-2])

print("li[1:3] ")
print(li[1:3])
```

```
li.append
['USA', 'Mexico', 'Canada', 'Greenland']
li.remove
['USA', 'Mexico', 'Canada']
li.insert
```

```
['Greenland', 'USA', 'Mexico', 'Canada']
li[:2]
['Greenland', 'USA']
li[-2]
Mexico
li[1:3]
['USA', 'Mexico']
```

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])
```

Unit 3 Exercise

```
#The set should be {} not () as stated in the exercise

a = [0,1,2,3,4,5,6,7,8,9,10]
b = [0,5,10,15,20,25]
topscores = {"Jo":999, "Sue":987, "Tara":960, "Mike":870}
print("List a is ...", a)
print("List b is ...", b)
print("The type of a is now ...", 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("Intersect of a and b is", a & b)
print("Union of a and b is", a | b)
print("Items unique to set b are", b - a)

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

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

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 ...

```
"2 West St","65 Deadend Cls","15 Magdalen Rd" ["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

Unit 4 Exercise

```
def address_machine(names, street, town, postcode):
    formatted_address = names + ", " + street + ", " + town + ", " +
postcode
    return formatted_address

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

new_list = [address_machine(names, street, town, postcode) for names,
street, town, postcode in zip(name, street, town, postcode)]
for address in new_list:
    print(address)
```

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

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
```

Unit 5 Exercise

```
class Person:
    def __init__(self, first_name, last_name, postcode, age):
        self.first_name = first_name
        self.last_name = last_name
        self.postcode = postcode
        self.age = age

    def greeting(self):
        print("Hello, my name is {}. I am {} years old and my postcode
is {}".format(self.first_name + " " + self.last_name, self.age,
self.postcode))

class Student(Person):
    def __init__(self, first_name, last_name, postcode, age,
degree_subject, student_ID):
        super().__init__(first_name, last_name, postcode, age)
        self.degree_subject = degree_subject
        self.student_ID = student_ID

    def studentGreeting(self):
        print("My student ID is {} and I am reading
{}".format(self.student_ID, self.degree_subject))

students = [
    Student("Dick", "Turpin", "HP11 2JZ", 32, "Highway Robbery",
"DT123456"),
    Student("Dorothy", "Turpin", "SO14 7AA", 32, "Law", "DT123457"),
    Student("Oliver", "Cromwell", "OX35 14RE", 32, "History",
"OC123456")
]

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

```
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
```

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)

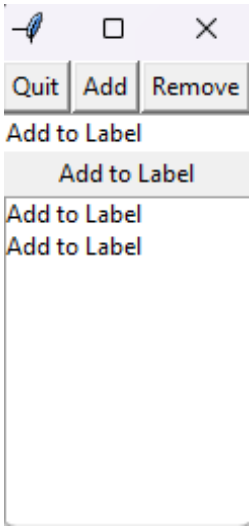
Unit 6 Exercise

```
from tkinter import *

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

    def quit_button_pressed(self):
        self.parent.destroy()

    def add_button_pressed(self):
        self.view.set_label_text(self.view.get_entry_text())
        self.model.add_to_list(self.view.get_entry_text())

    def remove_button_pressed(self):
        selected_items = self.view.listbox.curselection() # Get
selected items in the Listbox.

        if selected_items:
            # Iterate though remove corresponding items.
            for index in selected_items:
                self.model.remove_from_list(index)
        else:
            # Display a message if no item is selected.
            print("No item selected for removal")

    def list_changed_delegate(self):
        items = self.model.get_list()
        self.view.update_listbox(items) # Updated to call the
update_listbox method
        print(items)
```

```
class MyView(Frame):
    def __init__(self, controller):
        super().__init__()
        self.controller = controller
        self.entry_text = StringVar()
        self.label_text = StringVar()

        self.load_view()

    def load_view(self):
        self.grid(row=0, column=0)

        Button(self, text='Quit',
command=self.controller.quit_button_pressed).grid(row=0, column=0)
        Button(self, text="Add",
command=self.controller.add_button_pressed).grid(row=0, column=1)
        Button(self, text="Remove",
command=self.controller.remove_button_pressed).grid(row=0, column=2) #
Create a 'Remove' button that calls the 'remove_button_pressed' method
in the controller when clicked.

        Entry(self, textvariable=self.entry_text).grid(row=1,
column=0, columnspan=3, sticky=EW)
        Label(self, textvariable=self.label_text).grid(row=2,
column=0, columnspan=3, sticky=EW)

        self.listbox = Listbox(self)
        self.listbox.grid(row=3, column=0, columnspan=3, sticky=EW)

    def update_listbox(self, items):
        self.listbox.delete(0, END)  # Clear the current items in the
listbox.
        for item in items:
            self.listbox.insert(END, item)

    def get_entry_text(self):
        return self.entry_text.get()

    def set_entry_text(self, text):
        self.entry_text.set(text)

    def get_label_text(self):
        return self.label_text.get()

    def set_label_text(self, text):
        self.label_text.set(text)


class MyModel:
    def __init__(self, vc):
```

```
        self.vc = vc
        self.my_list = []

    def add_to_list(self, item):
        self.my_list.append(item)
        self.vc.list_changed_delegate()

    def remove_from_list(self, index):
        if index < len(self.my_list):
            # Remove the item at the specified from the list.
            del self.my_list[index]
            # Notify the controller that the list has changed
            self.vc.list_changed_delegate()

    def get_list(self):
        return self.my_list


root = Tk()
root.title("MVC Application")
controller = MyController(root)
root.mainloop()
```

```
['Add to Label']
['Add to Label', 'dw']
['Add to Label', 'dw', 'hello']
['Add to Label', 'dw']
```



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(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.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, startangle=90)
    ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn
as a circle.
    plt.show()
```

Unit 7 Exercise

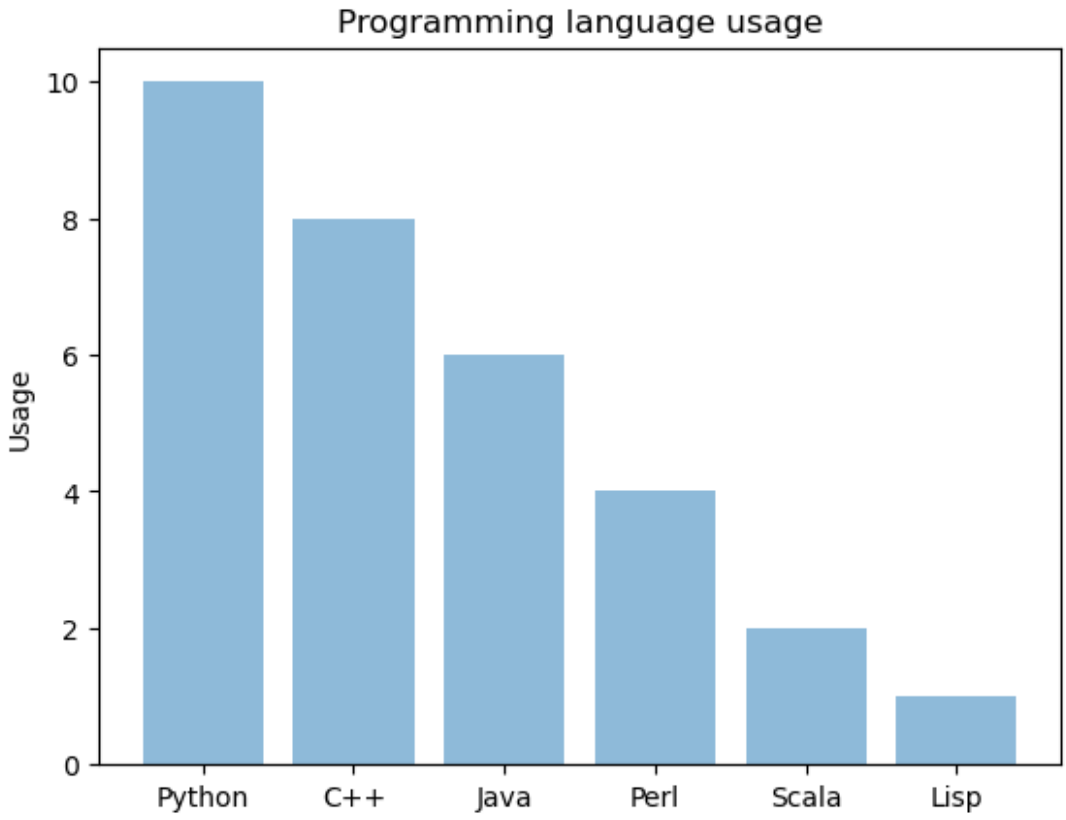
```
# sizes = subject.sizes didnt seem to be correct but sizes =
subject.data worked

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
```

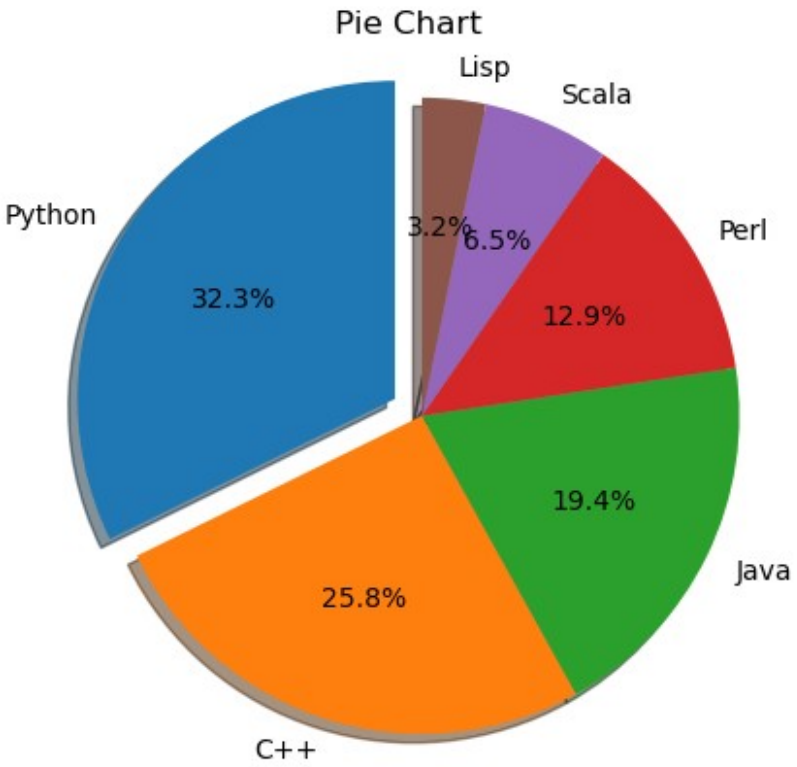
```
            self._observers.append(observer)  # append the observer to
the list

    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()
```

```
class PieView(View):
    def update(self, subject):
        sizes = subject.data
        labels = subject.labels  # Update the labels when the Model
notifies a change
        explode = (0.1, 0, 0, 0, 0, 0)  # Example explode values
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%
%', shadow=True, startangle=90)
        ax1.axis('equal')
        plt.title('Pie Chart')
        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("3: 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]

Generalised Viewer '1: standard text viewer' has:
Name = Model 1;
Labels = ['Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp'];
Data = [10, 8, 6, 4, 2, 1]
```

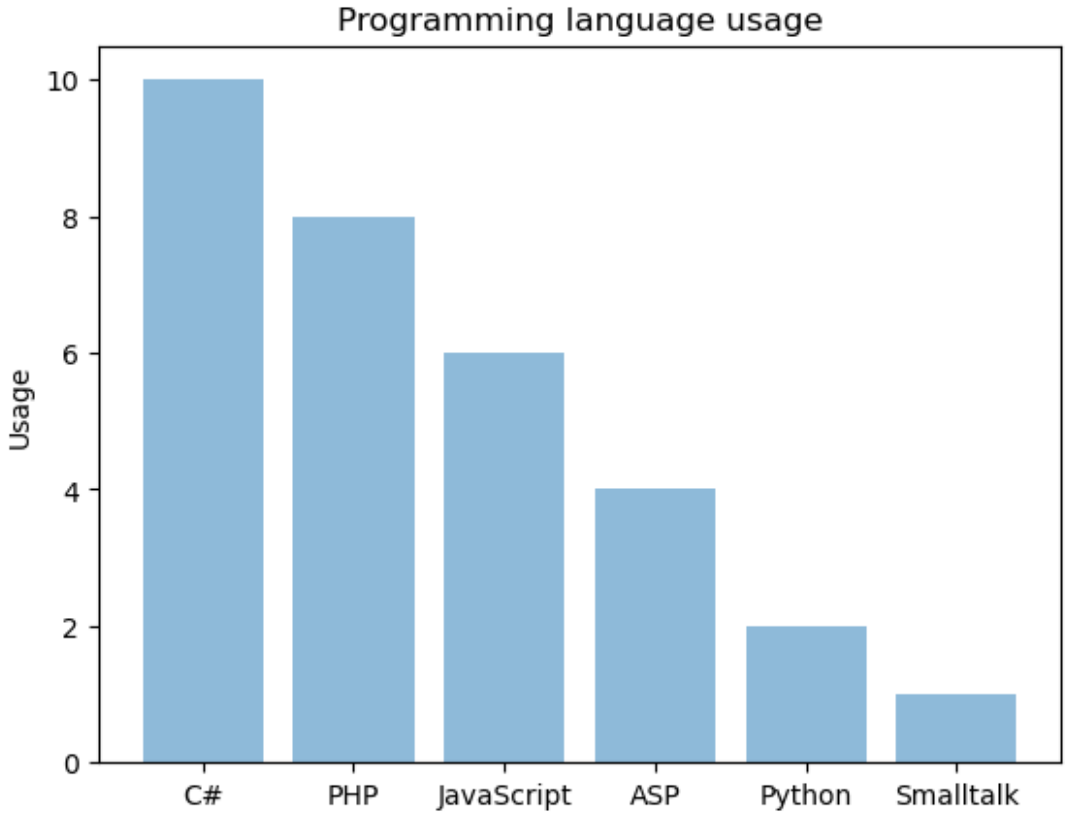
Python	10
C++	8
Java	6
Perl	4
Scala	2
Lisp	1

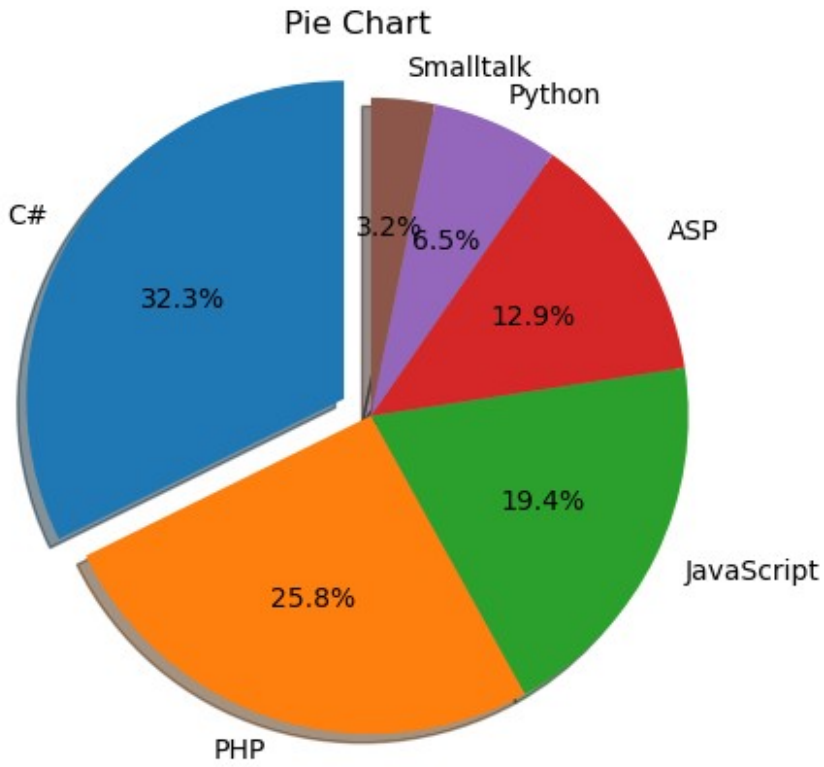





```
Generalised Viewer '1: standard text viewer' has:
Name = Model 1;
Labels = ['C#', 'PHP', 'JavaScript', 'ASP', 'Python', 'Smalltalk'];
Data = [10, 8, 6, 4, 2, 1]
```

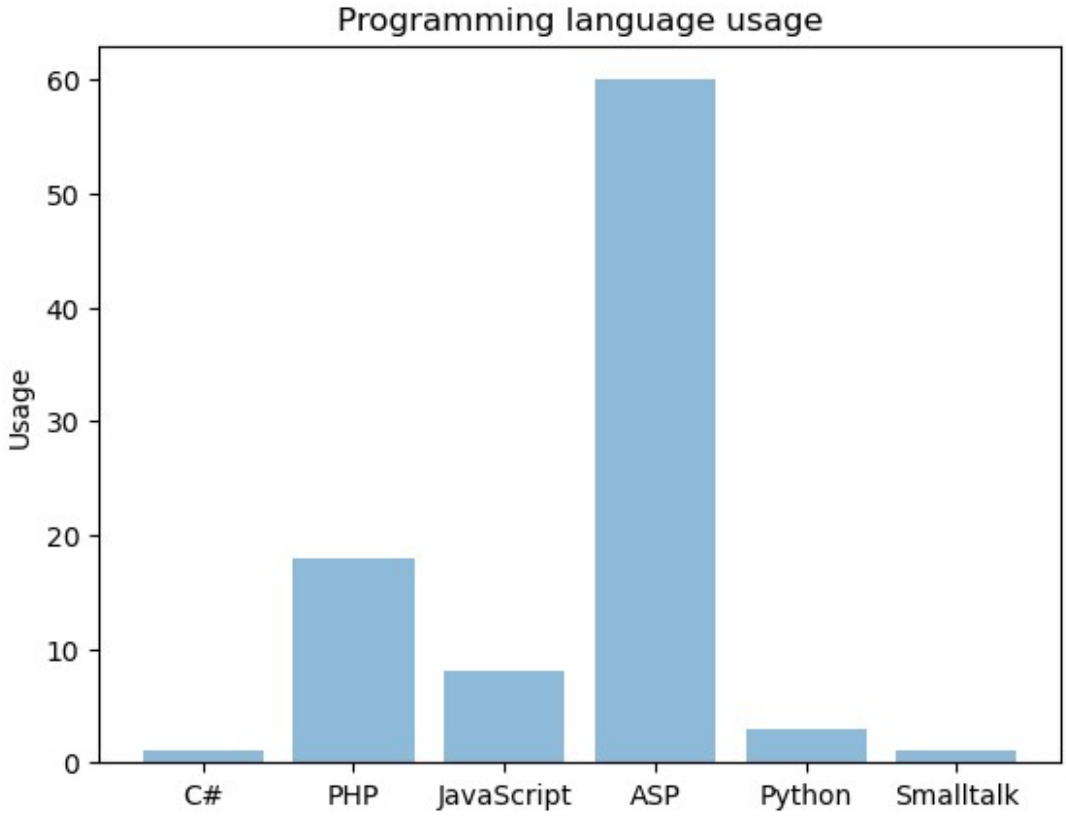
C#	10
PHP	8
JavaScript	6
ASP	4
Python	2
Smalltalk	1

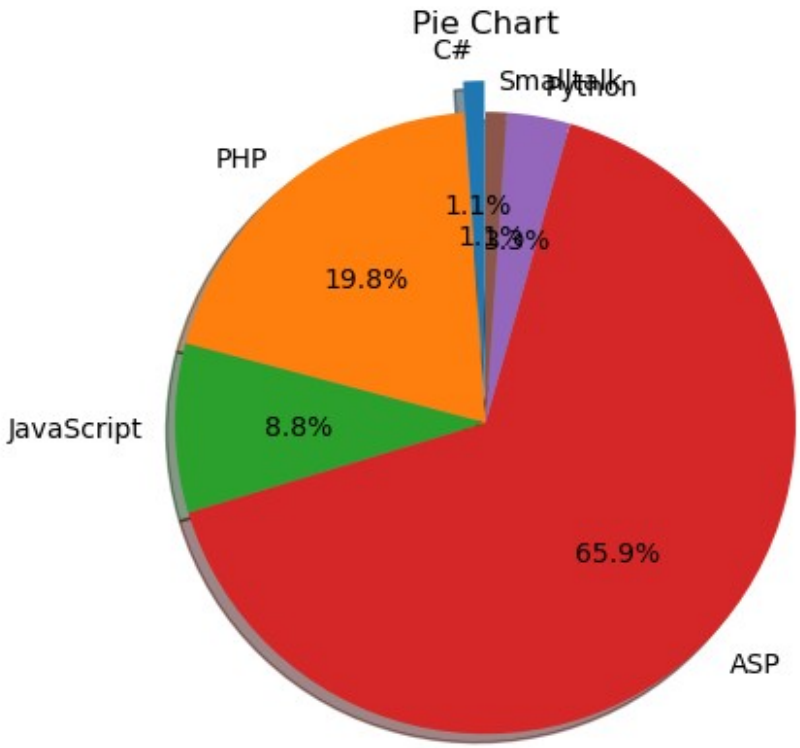





```
Generalised Viewer '1: standard text viewer' has:
Name = Model 1;
Labels = ['C#', 'PHP', 'JavaScript', 'ASP', 'Python', 'Smalltalk'];
Data = [1, 18, 8, 60, 3, 1]
```

C#	1
PHP	18
JavaScript	8
ASP	60
Python	3
Smalltalk	1





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 ...

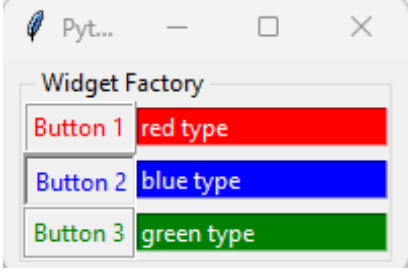
Challenge question

- 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)

```
# Challenge question to do this I would define an interface/abstract
class that is the functionality
# of the products called vehicle as an example. I would then implement
concrete classes for each vehicle
# which will extend the vehicle interface. I would then define the
interface/abstract class for the factory
# which will instance he vehicle called vehicle factory. I would then
implement a class concrete class for the
# factory called ConcretevehicleFactory which extends the
VehicleFactory which will be used for creating buttons.
# I will then modify the client code to use the abstract factory
"Vehicle  factory" and abstract product "vehicle".

import tkinter as tk
from tkinter import ttk

class WidgetFactory():
    def createWidget(self, type_):
        return self.widgetTypes[type_]()

class Button(WidgetFactory):
    def __init__(self, text, relief, foreground):
        self.text = text
        self.relief = relief
        self.foreground = foreground

class EntryField(WidgetFactory):
    def __init__(self, textvariable, background):
        self.textvariable = textvariable
        self.background = background

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

    def createWidgets(self):
        self.widgetFactory = ttk.LabelFrame(text=' Widget Factory ')
```

```
        self.widgetFactory.grid(column=0, row=0, padx=8, pady=4)

        self.createButtons()
        self.createEntryFields()

    def createButtons(self):
        # Button 1
        button1 = Button("Button 1", "ridge", "red")
        action = tk.Button(self.widgetFactory, text=button1.text,
relief=button1.relief, foreground=button1.foreground)
        action.grid(column=0, row=1)

        # Button 2
        button2 = Button("Button 2", "sunken", "blue")
        action = tk.Button(self.widgetFactory, text=button2.text,
relief=button2.relief, foreground=button2.foreground)
        action.grid(column=0, row=2)

        # Button 3
        button3 = Button("Button 3", "groove", "green")
        action = tk.Button(self.widgetFactory, text=button3.text,
relief=button3.relief, foreground=button3.foreground)
        action.grid(column=0, row=3)

    def createEntryFields(self):
        # Entry field 1
        entry1 = EntryField("red type", "red")
        sv1 = tk.StringVar()
        sv1.set(entry1.textvariable)
        action = tk.Entry(self.widgetFactory, textvariable=sv1,
background=entry1.background, foreground="white")
        action.grid(column=1, row=1)

        # Entry field 2
        entry2 = EntryField("blue type", "blue")
        sv2 = tk.StringVar()
        sv2.set(entry2.textvariable)
        action = tk.Entry(self.widgetFactory, textvariable=sv2,
background=entry2.background, foreground="white")
        action.grid(column=1, row=2)

        # Entry field 3
        entry3 = EntryField("green type", "green")
        sv3 = tk.StringVar()
        sv3.set(entry3.textvariable)
        action = tk.Entry(self.widgetFactory, textvariable=sv3,
background=entry3.background, foreground="white")
        action.grid(column=1, row=3)

#========================
```

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



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'!
```

Unit 9 Exercise

```
# Abstract Product #
class Animal:
    def speak(self):
        pass
    def __str__(self):
        pass

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

```
        return "Dog"

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

# Abstract Factory
class AnimalFactory:
    def get_pet(self):
        pass
    def get_food(self):
        pass

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

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

# The Client
class PetStore:
    def __init__(self, pet_factory=None):
        self._pet_factory = pet_factory
    def show_pet(self):
        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))

# Client using DogFactory
factory = DogFactory()
shop = PetStore(factory)
shop.show_pet()

# Client using CatFactory
factory = CatFactory()
shop = PetStore(factory)
shop.show_pet()
```

```
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'!
```

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

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.!

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_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()
```

Unit 10 Exercise

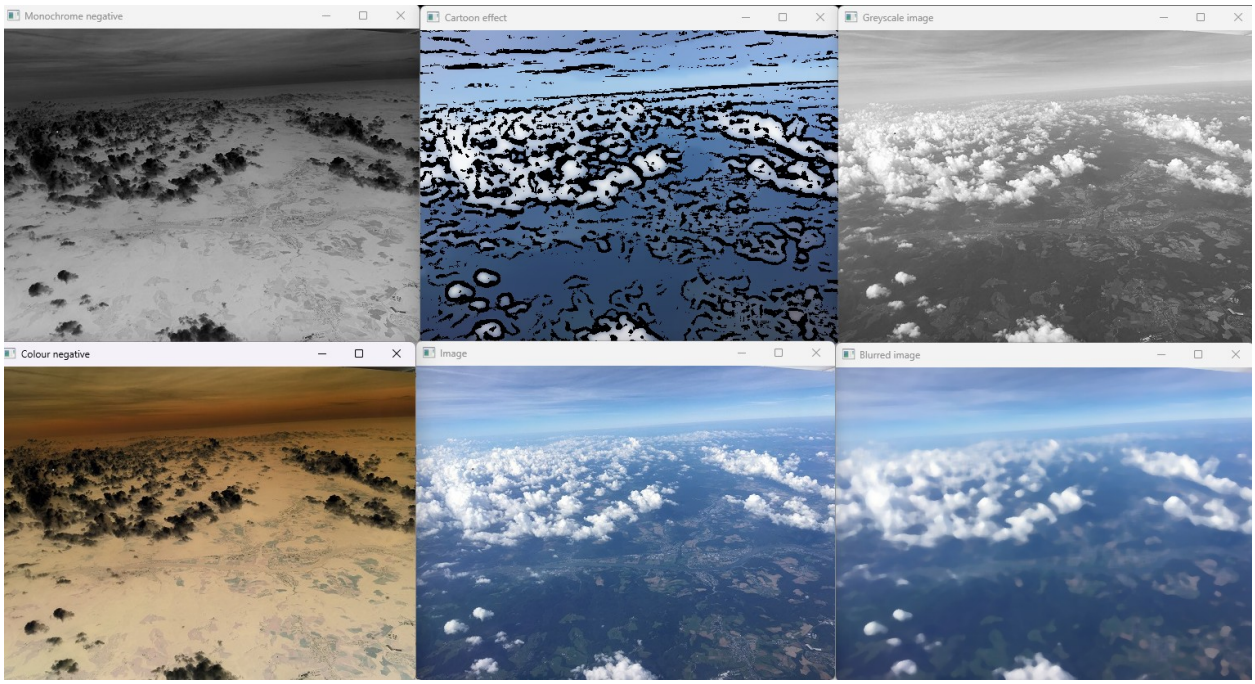
```
# Import OpenCV
import cv2
import numpy as np


import types
```

```
class Strategy:
    def __init__(self, function=None):
        self.image = "clouds.jpg"
        if function:
            self.execute = types.MethodType(function, self)

    def execute(self):
        print("The image {} is used to execute this
strategy".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):
    num_down = 2
    num_bilateral = 7
    img_rgb = cv2.imread(self.image)
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv2.pyrDown(img_color)
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=9,
sigmaSpace=7)
```

```
    for _ in range(num_down):
        img_color = cv2.pyrUp(img_color)
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
    img_edge = cv2.adaptiveThreshold(img_blur, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, blockSize=9, C=2)
    img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
    img_cartoon = cv2.bitwise_and(img_color, img_edge)
    cv2.imshow('Cartoon effect', img_cartoon)


s0 = Strategy()
s0.execute()

s1 = Strategy(strategy_greyscale)
s1.execute()

s2 = Strategy(strategy_blur)
s2.execute()

s3 = Strategy(strategy_colNegative)
s3.execute()

s4 = Strategy(strategy_greyNegative)
s4.execute()

s5 = Strategy(strategy_cartoon)
s5.execute()

cv2.waitKey(0)
```

```
The image clouds.jpg is used to execute this strategy
```

```
-1
```



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())
```

Unit 11 Exercise

```
import sys

class Top10BooksIterator:
    def __init__(self,data):
        self.data = data
        self.index = 0
```

```
    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
             result = self.data[self.index]
             self.index += 1
             return result
        else:
             raise StopIteration


top10books=["Anna Karenina by Leo Tolstoy", "Madame Bovary by Gustave
Flaubert", "War and Peace by Leo Tolstoy",
           "Lolita by Vladimir Nabokov", "The Adventures of
Huckleberry Finn by Mark Twain", "Hamlet by William Shakespeare",
           "The Great Gatsby by F. Scott Fitzgerald", "In Search of
Lost Time by Marcel Proust",
           "The Stories of Anton Chekhov by Anton Chekhov",
"Middlemarch by George Eliot"]


iter_top10books = Top10BooksIterator(top10books)
myiter = iter(iter_top10books)

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

```
Anna Karenina by Leo Tolstoy
Madame Bovary by Gustave Flaubert
War and Peace by Leo Tolstoy
Lolita by Vladimir Nabokov
```

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

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 ...

Unit 12 Exercise

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

class Korean: def **init**(self): self.name = "Korean"

```
def speak_korean(self):
    return "  ! (Annyeonghaseyo!)"
```

class British: def **init**(self): self.name = "British"

```
def say_english(self):
    return "Hello!"
```

class German: def **init**(self): self.name = "German"

```
def talk_a_little_german(self):
    return "Guten Tag!"
```

class Adapter: def **init**(self, obj, **adapted_methods): self.obj = obj self.__dict__.update(adapted_methods)

```
def __getattr__(self, attr):
    return getattr(self.obj, attr)
```

objects = []

korean = Korean() british = British() german = German()

objects.append(Adapter(korean, speak=korean.speak_korean)) objects.append(Adapter(british, speak=british.say_english)) objects.append(Adapter(german, speak=german.talk_a_little_german))

for obj in objects: print("{} says ... '{}'".format(obj.name, obj.speak()))

class PlateCarreeAdapter: def **init**(self): self.name = "PlateCarree" self.projection_obj = ccrs.PlateCarree()

```
def project(self, fig=None):
    if fig is None:
        fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection=self.projection_obj)
    return ax
```

class InterruptedGoodeHomolosineAdapter: def **init**(self): self.name = "InterruptedGoodeHomolosine" self.projection_obj = ccrs.InterruptedGoodeHomolosine()

```
def project(self, fig=None):
    if fig is None:
        fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection=self.projection_obj)
    return ax
```

class AlbersEqualAreaAdapter: def **init**(self): self.name = "AlbersEqualArea" self.projection_obj = ccrs.AlbersEqualArea()

```
def project(self, fig=None):
    if fig is None:
        fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection=self.projection_obj)
    return ax
```

class MollweideAdapter: def **init**(self): self.name = "Mollweide" self.projection_obj = ccrs.Mollweide()

```
def project(self, fig=None):
    if fig is None:
        fig = plt.figure()
```

```
    ax = fig.add_subplot(1, 1, 1, projection=self.projection_obj)
    return ax
```

class MapProjectionAdapter: def **init**(self): self.projections = {}

```
def add_projection(self, key, value):
    # Ensure that value is an instance of the provided class
    if isinstance(value, type):
        value = value()

    self.projections[key] = value

def show_all_projections(self):
    for key, obj in self.projections.items():
        fig = plt.figure()
        ax = obj.project(fig)
        if obj.name == "PlateCarree":
            ax.stock_img()
        else:
            ax.stock_img()
        ax.coastlines()
        print(obj.name)
        plt.show()
```

plateCarreeAdapter = PlateCarreeAdapter() interruptedGoodeAdapter = InterruptedGoodeHomolosineAdapter() albersEqualAreaAdapter = AlbersEqualAreaAdapter() mollweideAdapter = MollweideAdapter()

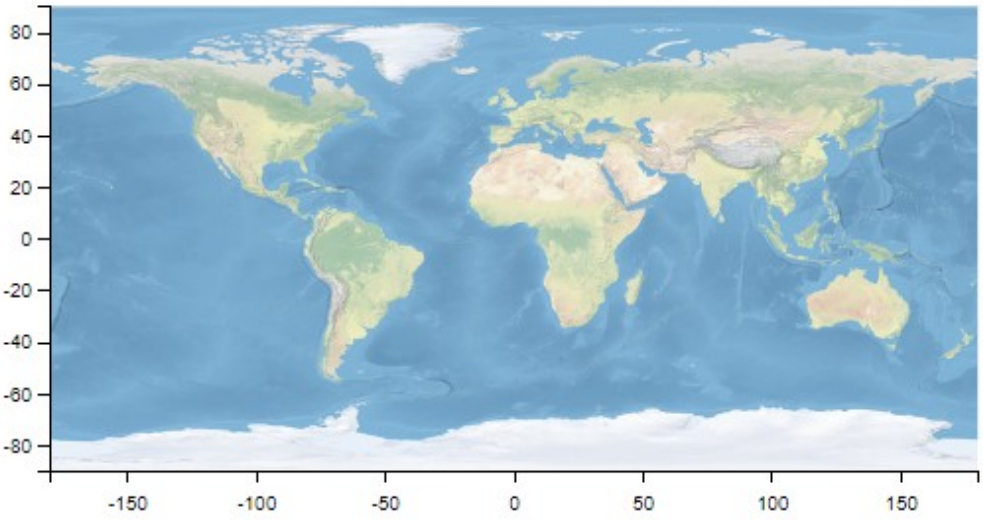
projectionAdapter = MapProjectionAdapter()

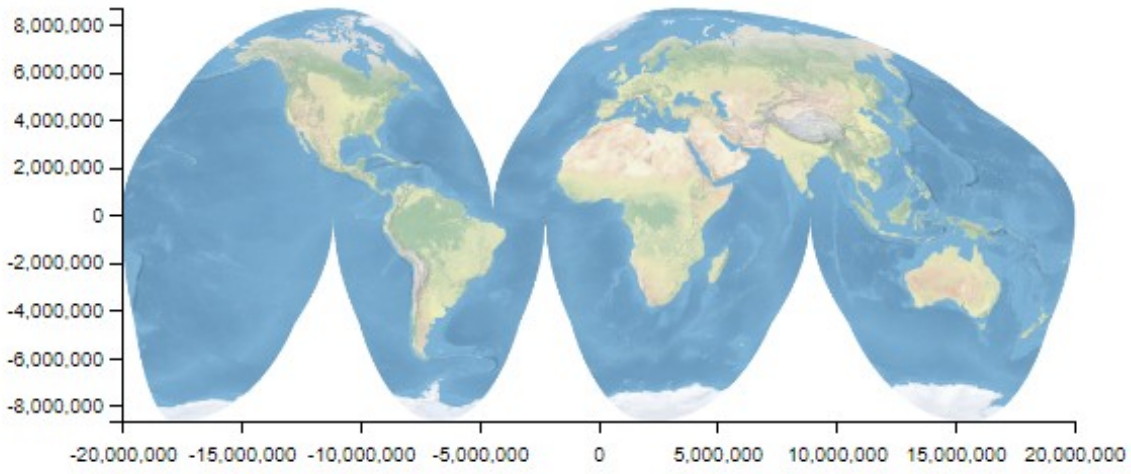
projectionAdapter.add_projection("PlateCarree", plateCarreeAdapter)
projectionAdapter.add_projection("InterruptedGoodeHomolosine", interruptedGoodeAdapter)
projectionAdapter.add_projection("AlbersEqualArea", albersEqualAreaAdapter)
projectionAdapter.add_projection("Mollweide", mollweideAdapter)

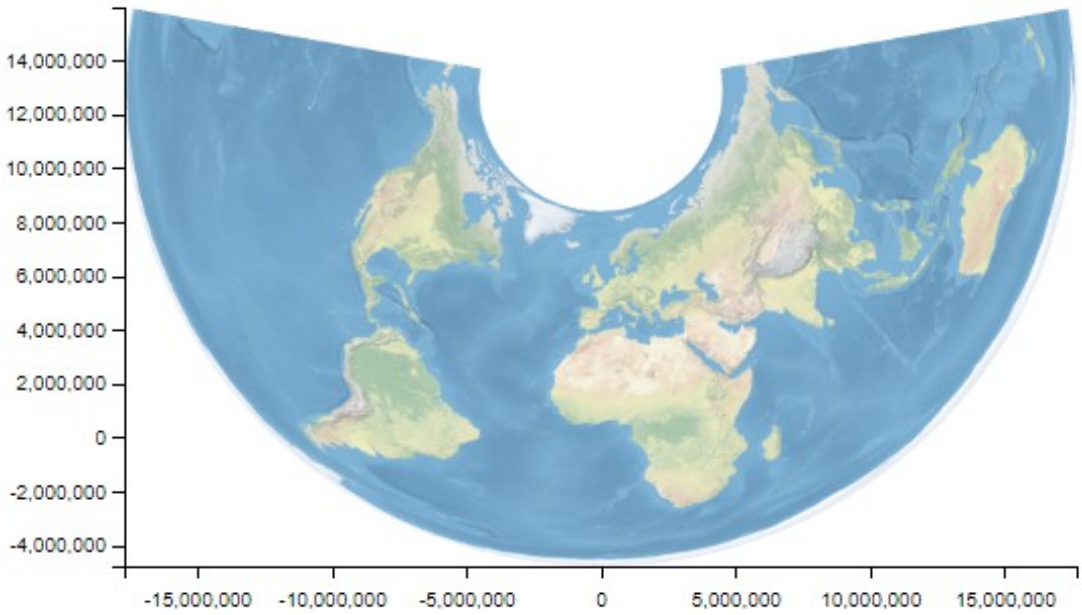
projectionAdapter.show_all_projections()

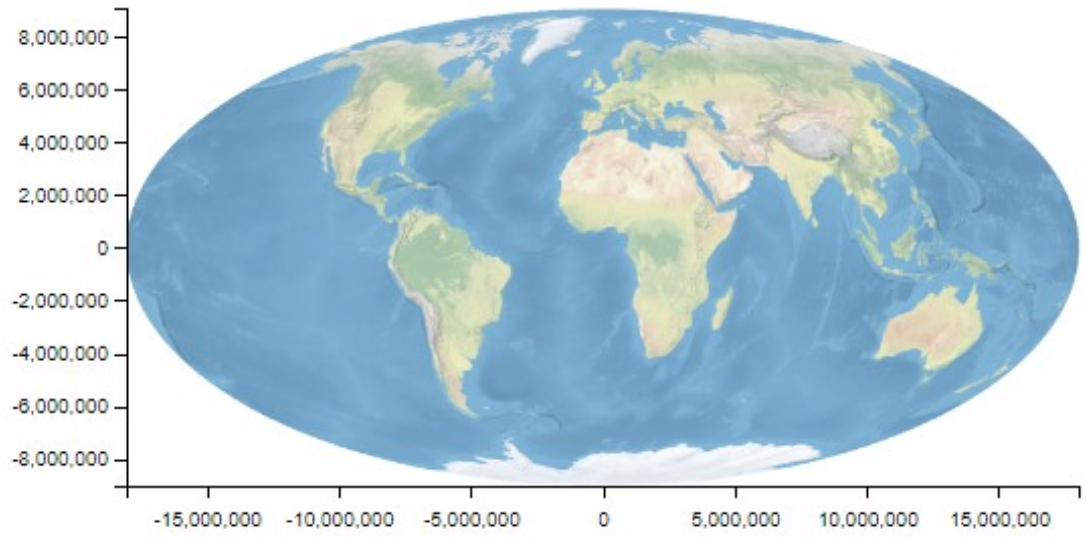
OutPut

The screens are of the code running in Pycharm for some reason in my environment I get an issue causing the code not to run in the notbook but works fine in Pycharm but it should fully run it may just be some sort of package conflict.


80
60
40
20
0
-20
-40
-60
-80
-150
-100
-50
0
50
100
150


8,000,000
6,000,000
4,000,000
2,000,000
0
-2,000,000
-4,000,000
-6,000,000
-8,000,000
-20,000,000
-15,000,000
-10,000,000
-5,000,000
0
5,000,000
10,000,000
15,000,000
20,000,000


14,000,000
12,000,000
10,000,000
8,000,000
6,000,000
4,000,000
2,000,000
0
-2,000,000
-4,000,000
-15,000,000
-10,000,000
-5,000,000
0
5,000,000
10,000,000
15,000,000



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 docStringDecorator  >>>
    What is your name? ... Buggy Code
    Hello ... Buggy Code
```

Unit 13 Exercise

```
def docStringDecorator(f):
    def wrapper(*args, **kwargs):
        print("<<< Name of the 'decorated' function ... ", f.__name__,
" >>> ")
        print("<<< Docstring for the 'decorated' function is ... ",
f.__doc__, " >>>")
        return f(*args, **kwargs)
    return wrapper

@docStringDecorator
def aTestMethod():
```

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

print(aTestMethod())
```

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

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

Unit 14 Exercise

My observations having tested the two objects are:

The Repr(1) and repr(2) outputs suggest that both p1 and p2 refer to the same instance of OneOnly class. the p1 == p2 returns true which confirms that p1 and p2 are the same object. the memory address of p1 and p2 are the same saying they have the same memory address.

```
## 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

p1 = OneOnly()
p2 = OneOnly()

# Check if p1 and p2 refer to the same object (identity check)
print("Check (p1 is p2):", p1 is p2)

# Check if p1 and p2 are equal (using == operator)
```

```
print("p1 == p2:", p1 == p1)

# Check if the memory addresses are the same
print("Memory address of p1", id(p1))
print("Memory address of p2", id(p2))

# Display repr()
print("p1:", repr(p1))
print("p2:", repr(p2))
```

```
Check (p1 is p2): True
p1 == p2: True
Memory address of p1 2134818688976
Memory address of p2 2134818688976
p1: <__main__.OneOnly object at 0x000001F10D1C0FD0>
p2: <__main__.OneOnly object at 0x000001F10D1C0FD0>
```

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

Unit 15 Exercise

My observations having tested the three objects are:

The **init** initializes the shared state as empty, and the **init** of the singleton class updates the state with values. the instances check, checks if they are the same instances. The memory address shows their address showing different memory addresses. the shared_state checks if each instance has its own state. the _shared_state is static but updated each instance with its own state. each instance of singleton has its own memory address as shown by checks. meaning instances share a common state but each has its own characteristics.

```
# 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

s1 = Singleton("s1")
s2 = Singleton("s2")
s3 = Singleton("s3")

# Check if instances are equal  (using == operator)
print("instances are equal: s1 == s2", s1 == s2)
print("instances are equal: s1 == s3", s1 == s3)
print("instances are equal: s2 == s3", s2 == s3)

print("Memory address of s1", id(s1))
print("Memory address of s2", id(s2))
print("Memory address of s3", id(s3))

print(s1._shared_state == s2._shared_state == s3._shared_state)
```

```
Value of self._shared_state is ...{}
Value of self._shared_state is ...{'val': 's1'}
Value of self._shared_state is ...{'val': 's2'}
instances are equal: s1 == s2 False
instances are equal: s1 == s3 False
instances are equal: s2 == s3 False
Memory address of s1 2134695958672
Memory address of s2 2134830101584
Memory address of s3 2134818746064
True
```

Logbook Exercises 16 and 17

- Per the assignment brief, for the third and final part of your assignment (copied below), note that it is necessary to write about TWO additional Design Patterns ...
  - *In you logbooks draw on a selection of Two Design Patterns, and document these using the following headings: 1 intent; 2 motivation; 3 structure (embed UML diagrams if applicable); 4 implementation; 5 sample code (your working example); 6 evaluation – raise any key points concerning programming language*

idioms, consequences of using the pattern, examples of appropriate uses with respect to application, architecture and implementation requirements [40 marks]

- Clearly it will be necessary to insert both markdown and code cells.
- Please make sure that all cells are properly titled with the excercise they represent
- Use and embed any screen capture that supports your assignment responses
- Suitable task subjects include:
- Demonstrating a new pattern (one which we haven't encountered)
- Presenting a pattern that we have addressed in a new/alternative/re-engineered format
- Converting a pattern to another programming language
- Identifying patterns in well known frameworks (e.g. for Python web development you might examine Django, Flask and/or Jinja2)

Unit 16 and 17 Exercise

Logbook Exercise 16: Command Pattern

Intent

The command pattern serves as a behavioural design pattern that turns a request into a stand-alone object that contains all information about the requests. This transformation lets you pass requests as method arguments, delay or queue a request's execution (Refactoring.guru, 2014), and support undoable operations. this specific design pattern promotes the use of decoupling of the sender and receiver of a command, allowing for greater flexability in managing the execution of operations.

Motivation

Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. The Command design pattern suggests encapsulating ("wrapping") in an object all (or some) of the following: an object, a method name, and some arguments as stated (Sourcemaking.com, 2024).

The motivation behind the command pattern is the need to address challenges in handling requests and operations within the program. by converting requests into standalone objects, allowing for modular and reusable component samples. it allows for the system to have more complex behaviours by using simple, independent objects focusing on maintainability.

Structure

Command(GameCommand) Declare the interface for executing a particular operation

ConcreteCommand(Hello, JumpCommand, AttackCommand, ReloadCommand) implements the execution by invoking the corresponding operation on the receiver.

Client(GameController) holds a reference to a command object and invokes execute.

Invoker(GameController) is the command to execute a request.

Receiver(GameCharacter) This knows how to perform the operation

https://github.com/Stefan-Allen/DesignPatterns The UML Diagram will be on GitHub it's stopped allowing embedding for some reason

Implementation

The implementation is a detailed structure and purpose of the strategy design pattern. it shows the specific role and task of each class allowing for a clearer understanding and visualisation of the system.

The implementation of the command design pattern involves creating a set of classes that encapsulate specific operations, allowing for various client requests, the ability to queue requests, and undoable operations.

Implementing into the system.

Step 1: Create an interface with a single execute method. Step 2: Create a class that inherits the command interface, each encapsulating a specific operation. Step 3: Define the class that will execute the operations(Receiver). Step 4: Implement an invoker that keeps a reference to a command and triggers execution. Step 5: Instance the Receiver, Concrete commands and the invoker setting the command in the invoker to trigger operations Step 6: Review to ensure it meets the system requirements.

Sample Code

```
# Base class for game commands
class GameCommand:
    def __init__(self, character):
        self.character = character

    def execute(self):
        pass

# Specialized command to display a welcome message
class Hello(GameCommand):
    def __init__(self):
        super().__init__(character=None)

    def execute(self):
        print("Welcome to the game!")

# Command for making a character jump
class JumpCommand(GameCommand):
```

```
    def __init__(self, character):
        super().__init__(character)

    def execute(self):
        print(f"{self.character.name} is Jumping!")

# Command for attacking a target character
class AttackCommand(GameCommand):
    def __init__(self, character, target):
        super().__init__(character)
        self.target = target

    def execute(self):
        print(f"{self.character.name} is Attacking
{self.target.name}!")
        self.target.state = "Dead"

# Command for reloading a character's weapon
class ReloadCommand(GameCommand):
    def __init__(self, character):
        super().__init__(character)

    def execute(self):
        print(f"{self.character.name} is Reloading!")

# Represents a game character with a name and state
class GameCharacter:
    def __init__(self, name):
        self.name = name
        self.state = "Alive"

    def set_state(self, new_state):
        self.state = new_state

# Game controller to handle button presses and execute corresponding
commands
class GameController:
    def __init__(self):
        self.button_a_command = None
        self.button_b_command = None
        self.button_x_command = None

    def set_button_a_command(self, command):
        self.button_a_command = command

    def set_button_b_command(self, command):
        self.button_b_command = command

    def set_button_x_command(self, command):
        self.button_x_command = command
```

```
    def press_button_a(self):
        self.button_a_command.execute()

    def press_button_b(self):
        self.button_b_command.execute()

    def press_button_x(self):
        self.button_x_command.execute()

# Create two game characters
if __name__ == "__main__":
    player1 = GameCharacter("Player1")
    player2 = GameCharacter("Player2")

    # Instantiate command objects for jump, attack, and reload
    jump_command = JumpCommand(character=player1)
    attack_command = AttackCommand(character=player1, target=player2)
    reload_command = ReloadCommand(character=player1)

    # Create a game controller and assign commands to buttons
    controller = GameController()
    controller.set_button_a_command(jump_command)
    controller.set_button_b_command(attack_command)
    controller.set_button_x_command(reload_command)

    # Simulate button presses during gameplay
    controller.press_button_a()  # Player 1 jumps
    controller.press_button_b()  # Player 1 attacks


    controller.press_button_x()  # Player 1 switches state to
PowerUpState

    # Simulate another button press after state change
    controller.press_button_a()  # Player 1 jumps in PowerUpState

    # Simulate an attack on Player 2
    controller.press_button_b()  # Player 1 attacks Player 2

    if player2.state == "Dead":
        print("Player 1 wins!")
```

```
Player1 is Jumping!
Player1 is Attacking Player2!
Player1 is Reloading!
Player1 is Jumping!
Player1 is Attacking Player2!
Player 1 wins!
```

Evaluation

The command design pattern is a behavioural design pattern that turns a request into stand-alone objects. the objects contain all information about the requests, allowing for different requests, queuing of requests and logging of the requests. The four main pattern components:

Command:

- defines the interface for executing a specific operation.
- typically houses the execute method.

ConcetreCommand:

- Implements command interface amd cliend actions.
- Holds a reference to the object to perform an act.

Involker:

- Ask the command to request the requests.
- Doesn't need to know specific commands or how to execute them.

Receiver:

- Know how to perform specific operations associated with particular commands.

The command design pattern, whilst it offers benefits such as flexibility, and modularity. it should also be taken into consideration of the requirements of the system. the command design pattern is great for applications that demand adaptability and evolving command structures but should be careful of the drawback of this method, especially in a system that is performance sensitive.

Advantages and disadvantages of Command patterns(and, 2024). Advantages of Command patterns:

- It decouples the classes that invoke the operation from the object that knows how to execute the operation.
- It allows you to create a sequence of commands by providing a queue system.
- Extensions to add a new commands are easy and can be done without changing the existing code.

Disadvantages of Command patterns

- There are a high number of classes and objects working together to achieve a goal. Application developers need to be careful in developing these classes correctly.
- Every individual command is a ConcreteCommand.

Key Points Concerning Command Design Pattern

- The use of Command and invoker aligns with OOP in that it supports abstraction and polymorphism.
- The pattern relies on object references, which is common in many programming languages to establish relations between objects.

Consequences of Using the Pattern

- Complexity the high number of classes and objects involved may lead to a complex system.
- That pattern may cause issues related to performance, due to multiple objects and misleading direction of command execution.

Examples of Appropriate Uses

- Adaptable system where the system needs an evolving command structure which changes current code.
- Logging and queuing of commands are essential, allowing for the use of undo/redo functionality.

Application, Architecture, and Implementation Requirements:

- when developing the system care and consideration of system performance is critically evaluating the impact on response time and resource usage.
- maintainability is achieved within the system by the design pattern allowing the creation of new commands without changing existing code but must have the correct design and documentation to manage the complexity.
- Command execution workflow user standing the workflow of command executions, ensuring that the workflow and execution meet the requirement of the application.

The Command design pattern offers advantages in flexibility and modularity but should be well thought-out, considering the specific requirements and constraints of the system, especially in terms of performance and complexity. to ensure the Command design pattern meets the needs of the system requirements.

References: Command Design Pattern

Refactoring.guru. (2014). Command. [online] Available at: https://refactoring.guru/design-patterns/command#:~:text=Intent,execution%2C%20and%20support%20undoable%20operations. [Accessed 22 Jan. 2024]. Sourcemaking.com. (2024). Design Patterns and Refactoring. [online] Available at: https://sourcemaking.com/design_patterns/command/java/2#:~:text=Motivation.,method%20name%2C%20and%20some%20arguments. [Accessed 22 Jan. 2024].

Sourcemaking.com. (2024). Design Patterns and Refactoring. [online] Available at: https://sourcemaking.com/design_patterns/command [Accessed 22 Jan. 2024].

and, A. (2024). Learning Python Design Patterns - Second Edition. [online] O'Reilly Online Learning. Available at: https://www.oreilly.com/library/view/learning-python-design/9781785888038/ch07s04.html [Accessed 22 Jan. 2024].

Logbook Exercise 17: State Design Pattern

Intent

The state design pattern Allows an object to alter its behaviour when its internal state changes. The object will appear to change its class as stated (Sourcemaking.com, 2024). This pattern is designed to encapsulate the state-specific functionality in distinct classes with a modular and maintainable code structure.

- Allows an object to adjust its behaviour as its state evolves.
- Encapsulates the state-specific login in separate classes preventing a complex structure.
- Permits the objects to transition between states dynamically during execution.
- Extensive switch or if-elf statements by state-specific code into divisional classes.
- Allows the addition of new states without modifying existing code.

Motivation

The motivation for using state design patterns is in the challenges when it comes to dynamic changes in an object's behaviour based on its internal state. Other patterns often don't handle complexities of state-dependent logic, leading to issues like reduced maintainability, code duplication and code readability.

Objectives

- Enables objects to adapt to new behaviours in runtime.
- Allows for the organization of state-specific functionality into specific classes for modular code structure.
- Enhanced readability and maintainability.

Rationale

- Breaking down state-specific login into separate classes enhances maintainability and allows future changes.
- State design pattern allows for dynamic changes without complex conditional statements, improving adaptability.
- Allows the addition of new states without modifying existing code.

The state design pattern is based on the internal state, offering a structured approach to handling state-dependent functionality. the state design pattern offers a solution for dynamic behaviour, code maintainability and modularity.

Structure

Components context:

- Represents the object by its internal state.
- Maintains a reference to the current state.
- Delegates state-specific tasks into the current state object.

State:

- Implements the State interface.
- Provides concrete implementations for state-specific methods.
- Handles behaviour associated with specific states.

ConcreteState:

- Implements the State interface.
- Provides concrete implementations for state-specific methods.
- Handles behaviour associated with specific states.

State machine:

- Encapsulation of the state machine's interface in the "wrapper" class.
- The wrapped hierarchy's interface mirrors the wrapper's interface with an additional parameter.
- The extra parameter, allowing derived classes to call back to the wrapper class.

UML Diagram https://github.com/Stefan-Allen/DesignPatterns The UML Diagram will be on GitHub it's stopped allowing embedding for some reason

Implementation

Classes OrderContext:

- Represents the (ordering system) influenced by its internal state.
- Maintains a reference to the current state.
- Contains a method to request a state transition.

OrderState:

- Interface for state-specific behaviour.

OrderPlacedState, OrderProcessingState, OrderShippedState (ConcreteState):

- Implements the OrderState interface.
- Provides implementation of state-specific methods.
- Handles behaviour associated with specific states Order Placed, Order Processing, Order Shipped

Relationships:

- OrderContext delegates state-specific tasks to the current OrderState.
- Each OrderState handles its state-specific behaviour.
- Switches between states managed by changing the current state.

Design Decisions/Considerations: Open/Closed Principle:

- Allows for new states without modifying existing code.

Separation of Concerns:

- State-specific behaviour is encapsulated in separate state classes, promoting modular and maintainability.

Flexibility:

- The state design pattern allows dynamic changes.

Sample code

```
# Base class representing the state of an account
class AccountState:
    def deposit(self, account, amount):
        pass

    def withdraw(self, account, amount):
        pass

# Concrete state class for an active account
class ActiveState(AccountState):
    def deposit(self, account, amount):
        print(f"Depositing £{amount} into your account")
        account.balance += amount

    def withdraw(self, account, amount):
        print(f"Withdrawing £{amount} from your account")
        if amount <= account.balance:
            account.balance -= amount
        else:
            print("Sorry, you don't have enough funds.")

# Concrete state class for a frozen account
class FrozenState(AccountState):
    def deposit(self, account, amount):
        print("Cannot deposit into a frozen account")

    def withdraw(self, account, amount):
        print("Cannot withdraw from a frozen account")

# Concrete state class for a closed account
class ClosedState(AccountState):
    def deposit(self, account, amount):
        print("Cannot deposit into a closed account")

    def withdraw(self, account, amount):
        print("Cannot withdraw from a closed account")

# Class representing an account with a state
class Account:
    def __init__(self, initial_balance):
        self.balance = initial_balance
        self.state = ActiveState()
```

```
    def change_state(self, new_state):
        self.state = new_state

    def deposit(self, amount):
        self.state.deposit(self, amount)

    def withdraw(self, amount):
        self.state.withdraw(self, amount)

#Example usage
if __name__ == "__main__":
    account = Account(initial_balance=1000)
    print("Account balance set to £1000")
    account.deposit(500)
    print("Current balance is ", account.balance)

    account.withdraw(200)
    print("Current balance is ", account.balance)

    account.change_state(FrozenState())
    print("Change state to FrozenState")
    account.withdraw(50)
    print("Trys to withdraw £50")
    account.deposit(50)
    print("Trys to deposit £50")

    account.change_state(ClosedState())
    print("Change state to ClosedState")
    account.withdraw(100)
    print("Trys to withdraw £100")
    account.deposit(100)
    print("Trys to deposit £100")

    account.change_state(ActiveState())
    print("Change state to ActiveState")
    account.deposit(10)
    print("Deposits £10")
    account.withdraw(10)
    print("Withdraws £10")

    print("Current balance is ", account.balance)
```

```
Account balance set to £1000
Depositing £500 into your account
Current balance is  1500
Withdrawing £200 from your account
Current balance is  1300
Change state to FrozenState
Cannot withdraw from a frozen account
```

```
Trys to withdraw £50
Cannot deposit into a frozen account
Trys to deposit £50
Change state to ClosedState
Cannot withdraw from a closed account
Trys to withdraw £100
Cannot deposit into a closed account
Trys to deposit £100
Change state to ActiveState
Depositing £10 into your account
Deposits £10
Withdrawing £10 from your account
Withdraws £10
Current balance is  1300
```

Evaluation

The state design pattern is a type of design pattern that focuses on how an object's behaviour can change when its state changes. It achieves this by separating the object's data into classes and assigning behaviours to those classes. This approach allows for runtime changes, in the object behaviour without modifying the class.

The key components of the State Design Pattern are as follows. The Context class maintains an instance of a Concrete State, which represents the state. The Context class delegates state behaviours to the state object. The State interface or abstract class defines an interface. Declares methods that represent state-specific behaviours. Concrete State classes implement the State interface providing specific implementations for behaviours associated with states.

In terms of programming language idioms: Dynamic polymorphisms are utilised in Python to achieve flexible behaviour. Interface classes, such as abstract classes are used to define state behaviours.

When using this pattern it is important to consider its consequences. It may result in an increased number of classes impacting code readability and maintainability. Managing complexity is crucial, in Python systems as unnecessary complexity should be avoided.

Here are some appropriate use cases, for the system.

Order Processing Systems. This is great for managing the stages of an order like when it's placed and when it's shipped.

Workflow Management Systems. It's a fit for systems that handle workflows and have states like "Order placed" and "Order shipped".

In terms of requirements here are some scenarios where this system works well.

Large Scale Applications:

- It's suitable for handling behaviours in large-scale systems that have states.

Modular Architectures:

- It integrates with architectures making them easier to maintain and extend.

Real-time Systems:

- It's can be used in applications requiring real-time state transitions and adaptability.

The state design pattern aligns with the OOP principles, showing its benefits in large-scale systems and intricate workflows. But it is critical to also take the question with the amount of class the system may have and balance the flexibility with simplicity in simpler applications. it's ideal for systems with dynamic state transitions and adaptability requirements. it also promotes the use of maintainability but isolates each code into its related state.

References: State Design Pattern

Sourcemaking.com. (2024). Design Patterns and Refactoring. [online] Available at: https://sourcemaking.com/design_patterns/state [Accessed 22 Jan. 2024].