Appendix

This section is included to assist you in performing the activities present in the book. It includes detailed steps that you can perform to complete and achieve the objectives of the book.

1. Vital Fundamentals – Math, Strings, Conditionals, and Loops

Activity 1 – assigning values to variables

Solution:

1. We begin with the first step, where `x` has been assigned the value of `14`:

   ```
   x = 14
   ```

2. Now we use the `+=` operator to set `x` equal to `x + 1` in the same step:

   ```
   x += 1
   ```

3. In this step, `x` is divided by `5`, and the result is squared:

   ```
   (x/5) ** 2
   ```

   You will get the following output:

   ```
   9.0
   ```

With this activity, you have learned how to perform multiple mathematical operations on a variable. This is very common in Python. For example, in machine learning, covered in *Chapter 11*, *Machine Learning*, the input may be a matrix, `X`, and multiple mathematical operations will be performed on the `X` matrix until predictive results are obtained. Although the mathematics behind machine learning is more sophisticated, the core ideas are the same.

Activity 2: finding the area of a triangle

Solution:

1. Open your Jupyter Notebook.
2. In this step, you need to write a docstring that describes the code as follows:

```
"""
This document shows to determine the area of a triangle.
The base and height are given, and the area is converted
to an integer
"""
```

3. Now, in the following code snippet, you have set `base` and `height` equal to `2` and `4`:

```
# Initialize variables
base, height = 2, 4
```

4. In the following step, you determine the area of a triangle by multiplying `1/2` by the `base` and `height` as follows:

```
# Formula for the area of a triangle
area_triangle = 1/2 * base * height
```

5. Now, you convert the area to an integer as in the following cod snippet :

```
# Convert the area to an integer
area_triangle = int(area_triangle)
```

6. To print the area of the triangle, enter the `area_triangle` variable inside the `print()` function as follows:

```
#Show the area
print(area_triangle)
```

You will get the following output:

```
4
```

In this activity, you have written a mini-program that determines the area of a triangle given the base and the height. Significantly, you have added a docstring and comments to clarify the code. There is never a correct answer for comments. It's up to the writer to determine how much information to give. A general goal is to be terse but informative. Comprehension is the most important thing. Adding comments and docstrings will always make your code look more professional and easier to read.

Activity 3 – using the input() function to rate your day

Solution:

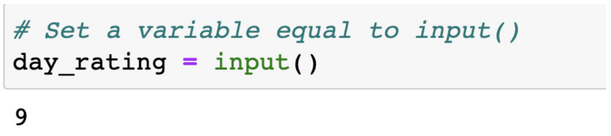
1. We begin this activity by opening up a new Jupyter Notebook.
2. In this step, a question is displayed, prompting a user to rate their day on a number scale:

```
# Choose a question to ask
print('How would you rate your day on a scale of 1 to 
10?')
```

3. In this step, the user input is saved as a variable:

```
# Set a variable equal to input()
day_rating = input()
```

You will get the following output:



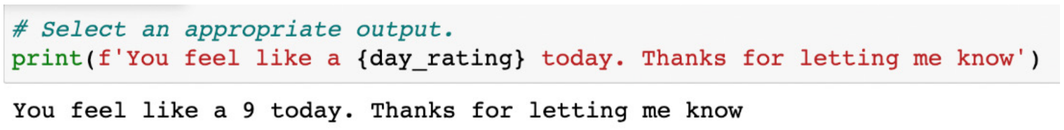
```
# Set a variable equal to input()
day_rating = input()

9
```

Figure Appendix.1 - Output asking the user for an input value

4. In this step, a statement is displayed that includes the provided number:

```
# Select an appropriate output.
print(f'You feel like a {day_rating} today. Thanks for 
letting me know')
```

You will get the following output:



```
# Select an appropriate output.
print(f'You feel like a {day_rating} today. Thanks for letting me know')

You feel like a 9 today. Thanks for letting me know
```

Figure Appendix.2 - Output displaying the user's day rating on a scale of 1 to 10

In this activity, you prompted the user for a number and used that number to display a statement back to the user that includes the number. Communicating directly with users depending upon their input is a core developer skill.

Activity 4 – finding the least common multiple (LCM)

Solution:

1. We begin by opening a new Jupyter Notebook.
2. Here, you begin by setting the variables equal to `24` and `36`:

```
# Find the Least Common Multiple of Two Divisors
first_divisor = 24
second_divisor = 36
```

3. In this step, you have initialized a `while` loop based on the counting Boolean, which is `True`, with an iterator, `i`:

```
counting = True
i = 1
while counting:
```

4. This step sets up a conditional to check whether the iterator divides both numbers:

```
   if i % first_divisor == 0 and i % second_divisor == 0:
```

5. This step breaks the `while` loop:

```
       break
```

6. This step increments the iterator at the end of the loop:

```
   i += 1
print('The Least Common Multiple of', first_divisor,
'and', second_divisor, 'is', i, '.')
```

The aforementioned code snippet prints the results.

You will get the following output:

```
The Least Common Multiple of 24 and 36 is 72.
```

In this activity, you used a `while` loop to run a program that computes the LCM of two numbers. Using `while` loops to complete tasks is an essential ability for all developers.

Activity 5 – building conversational bots using Python

Although you may use your own questions and answers, a full sample solution for the first bot is as follows:

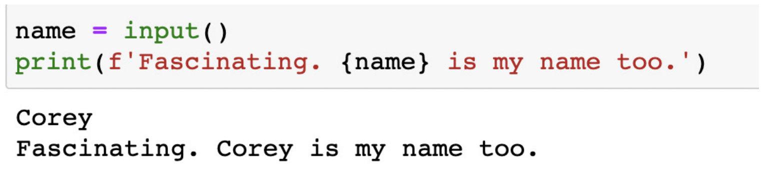
1. This step shows the first question asked of the user:

```
print('What is your name?')
```

2. This step shows the response with the answer:

```
name = input()
print(f'Fascinating. {name} is my name too.')
```

Once you enter the value, in this case, your name, the output will be as follows:



```
name = input()
print(f'Fascinating. {name} is my name too.')

Corey
Fascinating. Corey is my name too.
```

Figure Appendix.3 - Output once the user has entered the values

3. This step shows the second question asked of the user:

```
print('Have you thought about black holes today?')
```

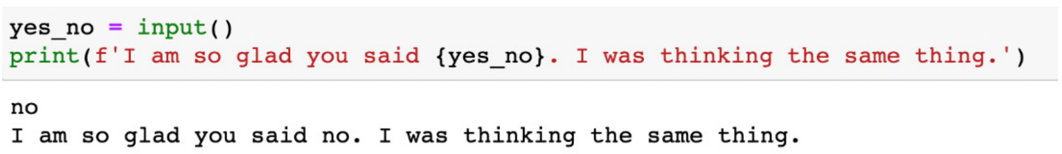
You will get the following output:

```
Have you thought about black holes today?
```

4. This step shows the response with the answer:

```
yes_no = input()
print('I am so glad you said', yes_no, '. I was thinking
the same thing.')
```

You will get the following output:



```
yes_no = input()
print(f'I am so glad you said {yes_no}. I was thinking the same thing.')

no
I am so glad you said no. I was thinking the same thing.
```

Figure Appendix.4 - Using the input() function asks the user to enter the value

5. This step shows the response with the answer:

```
print('We\'re kindred spirits,', name, '. Talk later.')
```

You will get the following output:

```
We're kindred spirits, Corey.Talk later.
```

Now, moving on to the second bot:

6. Create an `input()` function for a `smart` variable and change the type to `int`:

```
print('How intelligent are you? 0 is no intelligence. And
10 is a genius')
smarts = input()
smarts = int(smarts)
```

7. Create an `if` loop so that if the user enters a value equal to or less than `3`, we print `I don't believe you`. If not, then print the next statement:

```
if smarts <= 3:
   print('I don\'t believe you.')
   print('How bad of a day are you having? 0 is the
worst, and 10 is the best.')
```

8. Create an `input()` function for the `day` variable and change the type to `int`:

```
   day = input()
   day = int(day)
```

9. Now, create an `if else` loop where if the user entered a value less than or equal to 5, we `print` the output statement. If not, then we `print` the output `else` statement:

```
   if day <= 5:
       print('If I was human, I would give you a hug.')
   else:
       print('Maybe I should try your approach.')
```

10. Continue the loop using `elif`, also called `else-if`, where if the user enters a value less than or equal to `6`, we print the corresponding statement:

```
elif smarts <= 6:
   print('I think you\'re actually smarter.')
   print('How much time do you spend online? 0 is none
and 10 is 24 hours a day.')
```

11. Now, build another `input()` function for the `hours` variable and change the type to `int`. Use the `if-else` loop so that if the user enters a value less than or equal to `4`, we print the corresponding statement. If not, print the other statement:

```
    hours = input()
    hours = int(hours)
    if hours <= 4:
        print('That\'s the problem.')
    else:
       print('And I thought it was only me.')
```

12. Using the `elif` loop, we check the `smart` variable and output the corresponding `print` statement. We also use `if-else` to find out whether the user has entered a value less than or equal to `5`:

```
elif smarts <= 8:
   print('Are you human by chance? Wait. Don\'t answer 
that.')
   print('How human are you? 0 is not at all and 10 is 
human all the way.')
   human = input()
   human = int(human)
   if human <= 5:
       print('I knew it.')
   else:
       print('I think this courtship is over.')
```

13. We continue with the `else` loop from the `if-else` from step 7 and set the appropriate conditions and `print` statements:

```
else:
   print('I see... How many operating systems do you 
run?')
```

14. Set the `input()` functions once again to the `os` variable and change the type to `int`, after which we output the corresponding `print` statement depending on the user's input values:

```
    os = input()
    os = int(os)
    if os <= 2:
       print('Good thing you\'re taking this course.')
    else:
        print('What is this? A competition?')
```

You will get the following output:

```
How intelligent are you? 0 is no intelligence. And 10 is
a genius
8
Are you human by chance? Wait. Don't answer that.
How human are you? 0 is not at all and 10 is human all
the way.
10
I think this courtship is over.
```

Congratulations! By completing this activity, you have created two conversational bots using nested conditionals and `if-else` loops where we also used changing types, using the `input()` function to get values from the user and then respectively displaying the output.

2. Python Structures

Activity 6 – using a nested list to store employee data

Solution:

1. Begin by creating a list, adding data, and assigning it to `employees`:

```
employees = [['John Mckee', 38, 'Sales'], ['Lisa
Crawford', 29, 'Marketing'], ['Sujan Patel', 33, 'HR']]
print(employees)
```

You will get the following output:

```
[['John Mckee', 38, 'Sales'], ['Lisa Crawford', 29,
'Marketing'], ['Sujan Patel', 33, 'HR']]
```

2. Next, we can utilize the `for..in` loop to print each of the record's data within `employee`:

```
for employee in employees:
    print(employee)
```

Output:

```
['John Mckee', 38, 'Sales']
['Lisa Crawford', 29, 'Marketing']
['Sujan Patel', 33, 'HR']
```

3. To have the data presented in a structured version of the `employee` record, add the following lines of code:

```
for employee in employees:
    print("Name:", employee[0])
    print("Age:", employee[1])
    print("Department:", employee[2])
    print('-' * 20)
```

Output:

```
Name: John Mckee
Age: 38
Department: Sales
--------------------
Name: Lisa Crawford
Age: 29
Department: Marketing
--------------------
Name: Sujan Patel
Age: 33
Department: HR
--------------------
```

4. Lastly, if we were to print the details of `Lisa Crawford`, we would need to use the indexing method. Lisa's record is in position 1, so we would write:

```
employee = employees[1]
print(employee)
print("Name:", employee[0])
print("Age:", employee[1])
print("Department:", employee[2])
print('-' * 20)
```

The output will be as follows:

```
['Lisa Crawford', 29, 'Marketing']
Name: Lisa Crawford
Age: 29
Department: Marketing
--------------------
```

Having successfully completed this activity, you will be able to work with lists and nested lists. As mentioned in the activity, this is just one instance where this concept could come in handy, that is, to store data in lists and then access them as required.

Activity 7 – storing company employee table data using a list and a dictionary

Solution:

1. Open a Jupyter Notebook and enter the following code in it:

```
employees = [
    {"name": "John Mckee", "age":38,
"department":"Sales"},
    {"name": "Lisa Crawford", "age":29,
"department":"Marketing"},
    {"name": "Sujan Patel", "age":33, "department":"HR"}
]
print(employees)
```

   You will get the following output:

```
[{'name': 'John Mckee', 'age': 38, 'department':
'Sales'}, {'name': 'Lisa Crawford', 'age': 29,
'department': 'Marketing'}, {'name': 'Sujan Patel',
'age': 33, 'department': 'HR'}]
```

   In this step, we created a list, `employee`, and added values to it, such as `name`, `age`, and `department`.

2. Now, we will be adding a `for` loop to our `employee` list using the `*` operator. To do this, we will use a dictionary to `print` the `employee` details in a presentable structure:

```
for employee in employees:
    print("Name:", employee['name'])
    print("Age:", employee['age'])
    print("Department:", employee['department'])
    print('-' * 20)
```

   You should get the following output:

```
Name: John Mckee
Age: 38
Department: Sales
```

```
--------------------
Name: Lisa Crawford
Age: 29
Department: Marketing
--------------------
Name: Sujan Patel
Age: 33
Department: HR
--------------------
```

Note

You can compare this method with the previous activity, where we printed from a nested list. Using a dictionary gives us a more concise syntax, as we access the data using a key instead of a positional index. This is particularly helpful when we are dealing with objects with many keys.

3. The final step is to print the `employee` details of `Sujan Patel`. To do this, we will access the `employees` dictionary and will only `print` the value of one `employee` from a list of `employee` names:

```
for employee in employees:
    if employee['name'] == 'Sujan Patel':
        print("Name:", employee['name'])
        print("Age:", employee['age'])
        print("Department:", employee['department'])
        print('-' * 20)
```

You will get the following output:

```
Name: Sujan Patel
Age: 33
Department: HR
--------------------
```

Having completed this activity, you are able to work with lists and dictionaries. As you have seen, lists are very useful for storing and accessing data, which very often comes in handy in the real world when handling data in Python. Using dictionaries along with lists proves to be very useful, as you have seen in this activity.

3. Executing Python – Programs, Algorithms, Functions

Activity 8 – what's the time?

Solution:

Here, you will find the solution code to *Activity 8 – what's the time?*

To make it easier to understand, the code has been broken down with explanations:

current_time.py:

```
"""
This script returns the current system time.
"""
```

1. Firstly, we import the `datetime` library, which contains a range of useful utilities for working with dates:

```
import datetime
```

2. Using the `datetime` library, we can get the current `datetime` stamp, and then call the `time()` function in order to retrieve the time:

```
time = datetime.datetime.now().time()
```

3. If the script is being executed, this `if` statement will be true, and, therefore, the time will be printed:

```
if __name__ == '__main__':
    print(time)
```

You will get the following output:

```
16:48:22.416000
```

At the end of this activity, you are able to import the `datetime` module and execute the Python script to tell the time. Additionally, you are able to import the time to use it elsewhere in your code if necessary.

Activity 9 – formatting customer names

Solution:

The `customer.py` file should look like the steps mentioned next. Note that there are many different valid ways to write this function:

1. The `format_customer` function takes two required positional arguments, `first_name` and `last_name`, and one optional keyword argument, `location`:

```
def format_customer(first, last, location=None):
```

2. It then uses the `%` string formatting notation to create a `full_name` variable:

```
    full_name = '%s %s' % (first, last)
```

3. The third line checks whether a location has been specified and, if so, appends the location details to the full name. If no location was specified, just the full name is returned:

```
    if location:
        return '%s (%s)' % (full_name, location)
    else:
        return full_name
```

By the end of this activity, you are able to create a function that takes in various arguments for names and returns a string as you require.

Activity 10 – the Fibonacci function with an iteration

Solution:

1. This `fibonacci_iterative` function starts with the first two values of the Fibonacci sequence, `0` and `1`:

```
def fibonacci_iterative(n):
    previous = 0
    current = 1
```

2. For each loop in the iteration, it updates these values to represent the previous two numbers in the sequence. After reaching the final iteration, the loop terminates, and returns the value of the `current` variable:

```
    for i in range(n - 1):
        current_old = current
        current = previous + current
```

```
            previous = current_old
        return current
```

3. Now you can try running a few examples in Jupyter Notebook by importing the `fibonacci_iterative` function:

```
from fibonacci import fibonacci_iterative
fibonacci_iterative(3)
```

You will get the following output:

```
2
```

4. Let's try another example:

```
fibonacci_iterative(10)
```

You will get the following output:

```
55
```

In this activity, you were able to work with iterations and return the *n*th value in the Fibonacci sequence.

Activity 11 – the Fibonacci function with recursion

Solution:

1. Open the `fibonacci.py` file.
2. Define a `fibonacci_recursive` function that takes a single input named `n`:

```
def fibonacci_recursive(n):
```

3. Now check whether the value of `n` is equal to 0 or 1. If the condition is satisfied, the value of `n` is returned. Write the following code to implement this step:

```
    if n == 0 or n == 1:
        return n
```

4. Otherwise, in case the condition is not satisfied it will return the same function, but the argument will be decremented by 2 and 1 and the respective differences will be added:

```
    else:
        return fibonacci_recursive(n - 2) + fibonacci_
recursive(n - 1)
```

5. Once the function is created, try running the examples in the compiler using the following command:

```
from fibonacci import fibonacci_recursive
fibonacci_recursive(3)
```

You will get the following output:

```
2
```

You can now work with recursive functions. We implemented this on our `fibonnacci.py` file to get the expected output. Recursive functions are helpful in many cases in order to reduce the lines of code that you will be using if it is repetitive.

Activity 12 – the Fibonacci function with dynamic programming

Solution:

1. We begin by keeping a dictionary of Fibonacci numbers in the `stored` variable. The keys of the dictionary represent the index of the value in the sequence (such as the first, second, and fifth number), and the value itself:

```
stored = {0: 0, 1: 1}  # We set the first 2 terms of the
Fibonacci sequence here.
```

2. When calling the `fibonacci_dynamic` function, we check to see whether we have already computed the result; if so, we simply return the value from the dictionary:

```
def fibonacci_dynamic(n):
    if n in stored:
        return stored[n]
```

3. Otherwise, we revert to the recursive logic by calling the function to compute the previous two terms:

```
    else:
        stored[n] = fibonacci_dynamic(n - 2) + fibonacci_
dynamic(n - 1)
        return stored[n]
Now, run the following:
from fibonacci import fibonacci_recursive
fibonacci_dynamic(100)
```

You will get the following output:

```
354224848179261915075
```

In this activity, we used a function with dynamic programming that takes a single positional argument representing the number term in the sequence that we want to return.

4. Extending Python, Files, Errors, and Graphs

Activity 13 – visualizing the Titanic dataset using a pie chart and bar plots

Solution:

1. Import all the lines from the `csv` file in the `titanic_train.csv` dataset file and store it in a list:

```
import csv
lines = []
with open('titanic_train.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for line in csv_reader:
        lines.append(line)
```

2. Generate a collection of `passengers` objects. This step is designed to facilitate the subsequent steps where we need to extract values of different properties into a list for generating charts:

```
data = lines[1:]
passengers = []
headers = lines[0]
```

3. Create a simple `for` loop for the `d` variable in `data`, which will store the values in a list:

```
for d in data:
    p = {}
    for i in range(0,len(headers)):
        key = headers[i]
        value = d[i]
        p[key] = value
    passengers.append(p)
```

4. Extract the `survived`, `pclass`, `age`, and `gender` values of survived passengers into respective lists. We need to utilize list comprehension in order to extract the values; for the passengers who survived, we will need to convert `survived` into an integer and filter `survived == 1`, that is, passengers who survived:

```
survived = [p['Survived'] for p in passengers]
pclass = [p['Pclass'] for p in passengers]
age = [float(p['Age']) for p in passengers if p['Age'] !=
'']
gender_survived = [p['Sex'] for p in passengers if
int(p['Survived']) == 1]
```

5. Now, `import` all the necessary libraries, such as `matplotlib`, `seaborn`, and `numpy`, and draw a pie chart using `plt.pie` to visualize the passengers who survived:

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from collections import Counter
plt.title("Survived")
plt.pie(Counter(survived).values(),
labels=Counter(survived).keys(), autopct='%1.1f%%',
        colors=['lightblue', 'lightgreen', 'yellow'])
plt.show()
```

6. Execute the cell twice, and you will get the following output:

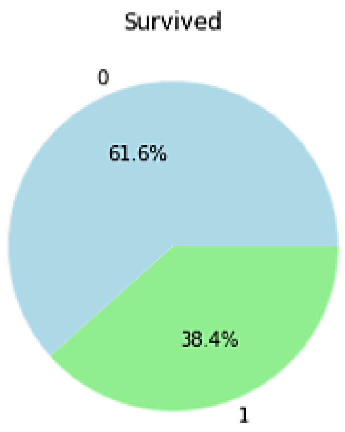


Figure Appendix.5 - Pie chart showing the survival rate of the passengers

7. Draw a column bar plot using `plt.bar` to visualize the passengers who survived based on their gender:

```
plt.title("surviving passengers count by gender")
plt.bar(Counter(gender_survived).keys(), Counter(gender_
survived).values())
plt.show()
```

You will get the following output:

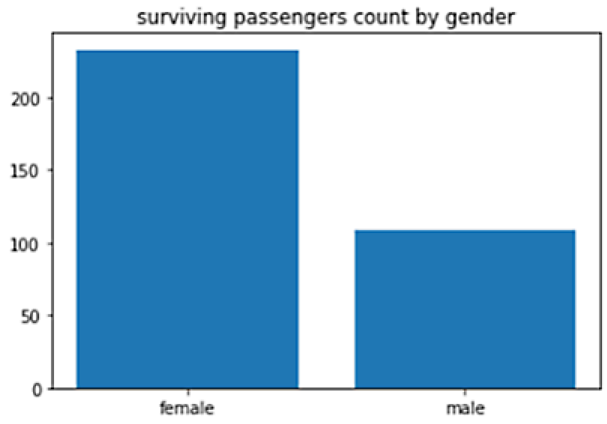


Figure Appendix.6: A bar plot showing the variation in gender of those who survived the incident

In this activity, we have used the interesting Titanic dataset to visualize data. We imported the dataset and stored the data in a list. Then, we used the `matplotlib`, `seaborn`, and `numpy` libraries to plot the various data and get the outputs as needed using the two plotting techniques: pie charts and bar plots.

5. Constructing Python – Classes and Methods

Activity 14 – creating classes and inheriting from a parent class

Solution:

1. Firstly, define the parent class, `Polygon`. We add an `init` method that allows the user to specify the lengths of the sides when creating the polygon:

```
class Polygon():
    """A class to capture common utilities for dealing
with shapes"""
    def __init__(self, side_lengths):
        self.side_lengths = side_lengths

    def __str__(self):
        return 'Polygon with %s sides' % self.num_sides
```

2. Add two properties to the `Polygon` class – one that computes the number of sides of the polygon, and another that returns the perimeter:

```
class Polygon():
    """A class to capture common utilities for dealing 
with shapes"""
    def __init__(self, side_lengths):
        self.side_lengths = side_lengths

    def __str__(self):
        return 'Polygon with %s sides' % self.num_sides

    @property
    def num_sides(self):
        return len(self.side_lengths)

    @property
    def perimeter(self):
            return sum(self.side_lengths)
```

3. Create a child class of `Polygon` called `Rectangle`. Add an `init` method that allows the user to specify the height and the width of the rectangle. Add a property that computes the area of the rectangle:

```
class Rectangle(Polygon):
    def __init__(self, height, width):
        super().__init__([height, width, height, width])

    @property
    def area(self):
        return self.side_lengths[0] * self.side_
lengths[1]
```

4. Test your `Rectangle` class by creating a new rectangle and checking the value of its properties – the `area` and the `perimeter`:

```
r = Rectangle(1, 5)
r.area, r.perimeter
```

   You will get the following output:

```
(5, 12)
```

5. Create a child class of `Rectangle` called `Square` that takes a single `height` parameter in its initialization:

```
class Square(Rectangle):
    def __init__(self, height):
        super().__init__(height, height)
```

6. Test your `Square` class by creating a new square and checking the value of its properties – the `area` and the `perimeter`:

```
s = Square(5)
s.area, s.perimeter
```

   You should get the following output:

```
(25, 20)
```

6. The Standard Library

Activity 15 – calculating the time elapsed to run a loop

Solution:

1. We begin by opening a new Jupyter file and importing the `random` and `time` modules:

```
import random
import time
```

2. Then, we use the `time.time` function to get the `start` time:

```
start = time.time()
```

3. Now, by using the aforementioned code, we will find the time in nanoseconds. Here, the range is set from 1 to 999:

```
l = [random.randint(1, 999) for _ in range(10 * 3)]
```

4. Now, we record the finish time and subtract this time to get the delta:

```
end = time.time()
print(end - start)
```

You will get the following output:

```
0.0019025802612304688
```

5. But this will give us a float. For measurements higher than 1 second, the precision might be good enough, but we can also use `time.time_ns` to get the time as the number of nanoseconds elapsed. This will give us a more precise result, without the limitations of floating-point numbers:

```
start = time.time_ns()
l = [random.randint(1, 999) for _ in range(10 * 3)]
end = time.time_ns()
print(end - start)
```

You should get the following output:

```
187500
```

Note

This is a good solution when using the `time` module and for common applications.

Activity 16 – Testing Python code

Solution:

The line, `compile("1" + "+1" * 10 ** 6, "string", "exec")`, will crash the interpreter; we will need to run it with the following code:

1. First, import the `sys` and `subprocess` modules as we are going to use them in the following steps:

```
import sys
import subprocess
```

2. We save the code that we were given in the `code` variable:

   ```
   code = 'compile("1" + "+1" * 10 ** 6, "string", "exec")'
   ```

3. Run the code by calling `subprocess.run` and `sys.executable` to get the Python interpreter we are using:

   ```
   result = subprocess.run([
       sys.executable,
       "-c", code
   ])
   ```

   The preceding code takes a code line, which compiles Python code that will crash and runs it in a subprocess by executing the same interpreter (retrieved via `sys.executable`) with the `-c` option to run Python code inline.

4. Now, we print the final result using `result.resultcode`. This will return the value `-11`, which means the process has crashed:

   ```
   print(result.returncode)
   ```

   The output will be as follows:

   ```
   -11
   ```

   This line of code just prints the return code of the `subprocess` call.

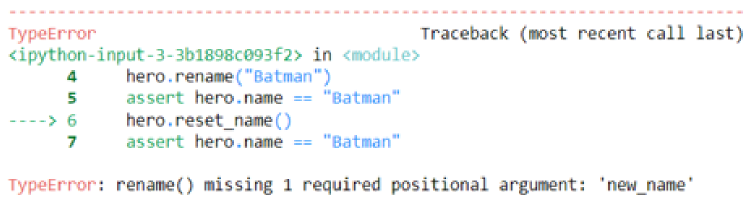
In this activity, we have executed a small program that can run the requested code line and checked whether it would crash without breaking the current process. It did end up crashing, hence outputting the value `-11`, which corresponded to an abort in the program.

Activity 17 – using partial on class methods

Solution:

You need to explore the `functools` module and realize a specific `helper` for methods, which can be used as explained in the following steps:

1. When you execute the mentioned code, you will get the following error message:



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-3b1898c093f2> in <module>
      4 hero.rename("Batman")
      5 assert hero.name == "Batman"
----> 6 hero.reset_name()
      7 assert hero.name == "Batman"

TypeError: rename() missing 1 required positional argument: 'new_name'
```

Figure Appendix.7 - Error output with a missing required positional argument

Now, to fix this, let's check for an alternative by observing the following steps.

2. Import the `functools` module:

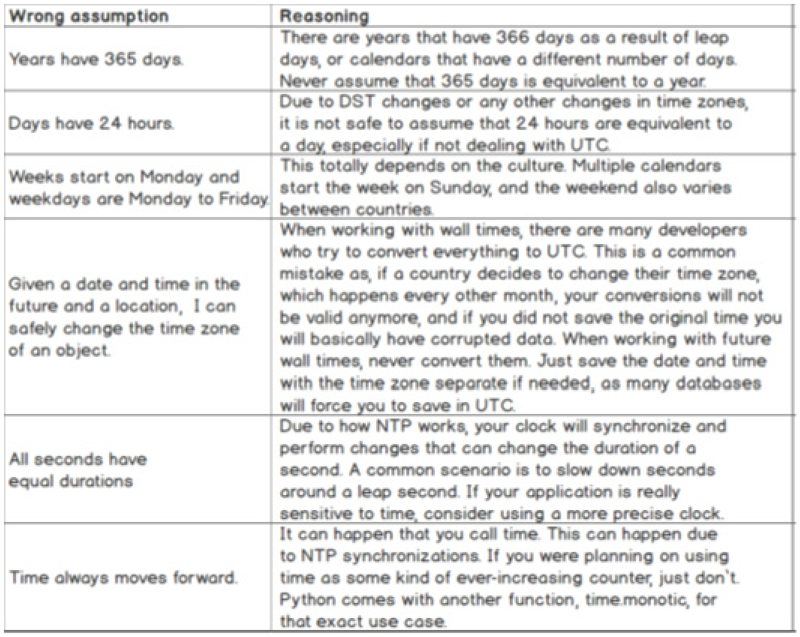
```
import functools
```

3. Create the `Hero` class, which uses `partialmethod` to set `reset_name`:

```
class Hero:
    DEFAULT_NAME = "Superman"
    def __init__(self):
        self.name = Hero.DEFAULT_NAME

    def rename(self, new_name):
        self.name = new_name

    reset_name = functools.partial(rename, DEFAULT_NAME)

    def __repr__(self):
        return f"Hero({self.name!r})"
```

The code makes use of a different version of `partial`, `partialmethod`, which allows the creation of `partial` for a method class. By using this utility on the `rename` method and setting the name to the default name, we can create `partial`, which will be used as a method. The name of the method is the one that is set in the scope of the `Hero` class definition, which is `reset_name`.

Wrong assumptions with date and time



Wrong assumption	Reasoning
Years have 365 days.	There are years that have 366 days as a result of leap days, or calendars that have a different number of days. Never assume that 365 days is equivalent to a year.
Days have 24 hours.	Due to DST changes or any other changes in time zones, it is not safe to assume that 24 hours are equivalent to a day, especially if not dealing with UTC.
Weeks start on Monday and weekdays are Monday to Friday.	This totally depends on the culture. Multiple calendars start the week on Sunday, and the weekend also varies between countries.
Given a date and time in the future and a location, I can safely change the time zone of an object.	When working with wall times, there are many developers who try to convert everything to UTC. This is a common mistake as, if a country decides to change their time zone, which happens every other month, your conversions will not be valid anymore, and if you did not save the original time you will basically have corrupted data. When working with future wall times, never convert them. Just save the date and time with the time zone separate if needed, as many databases will force you to save in UTC.
All seconds have equal durations	Due to how NTP works, your clock will synchronize and perform changes that can change the duration of a second. A common scenario is to slow down seconds around a leap second. If your application is really sensitive to time, consider using a more precise clock.
Time always moves forward.	It can happen that you call time. This can happen due to NTP synchronizations. If you were planning on using time as some kind of ever-increasing counter, just don't. Python comes with another function, time.monotic, for that exact use case.

Figure Appendix.8 - Wrong assumptions of date and time along with reasoning

7. Becoming Pythonic

Activity 18 – building a chess tournament

Solution:

1. Open the Jupyter Notebook.
2. Define the list of player names in Python:

```
names = ["Magnus Carlsen", "Fabiano Caruana", "Yifan
Hou", "Wenjun Ju"]
```

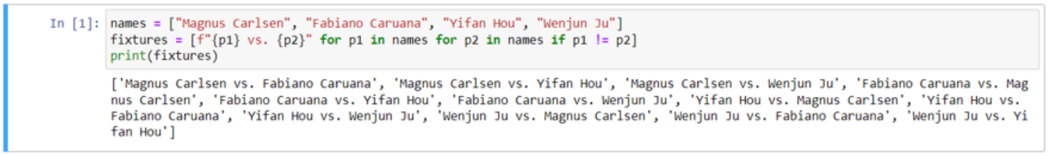
3. The list comprehension uses the list of names twice because each person can either be player 1 or player 2 in a match (that is, they can play with the white or the black pieces). Because we don't want the same person to play both sides in a match, add an `if` clause that filters out the situation where the same name appears in both elements of the comprehension:

```
fixtures = [f"{p1} vs. {p2}" for p1 in names for p2 in
names if p1 != p2]
```

4. Finally, print the resulting list so that the match officials can see who will be playing whom:

```
print(fixtures)
```

You will get the following output:



```
In [1]: names = ["Magnus Carlsen", "Fabiano Caruana", "Yifan Hou", "Wenjun Ju"]
        fixtures = [f"{p1} vs. {p2}" for p1 in names for p2 in names if p1 != p2]
        print(fixtures)

        ['Magnus Carlsen vs. Fabiano Caruana', 'Magnus Carlsen vs. Yifan Hou', 'Magnus Carlsen vs. Wenjun Ju', 'Fabiano Caruana vs. Mag
        nus Carlsen', 'Fabiano Caruana vs. Yifan Hou', 'Fabiano Caruana vs. Wenjun Ju', 'Yifan Hou vs. Magnus Carlsen', 'Yifan Hou vs.
        Fabiano Caruana', 'Yifan Hou vs. Wenjun Ju', 'Wenjun Ju vs. Magnus Carlsen', 'Wenjun Ju vs. Fabiano Caruana', 'Wenjun Ju vs. Yi
        fan Hou']
```

Figure Appendix.9 - The sorted fixtures' output using a list comprehension

In this activity, we used list comprehension to sort out players and create a fixture that was in the form of a string.

Activity 19 – building a scorecard using dictionary comprehensions and multiple lists

Solution:

1. The solution is to iterate through both collections at the same time, using an index. First, define the collections of names and their scores:

```
students = ["Vivian", "Rachel", "Tom", "Adrian"]
points = [70, 82, 80, 79]
```

Now build the dictionary. The comprehension is actually using the third collection; that is, the range of integers from 0 to 100.

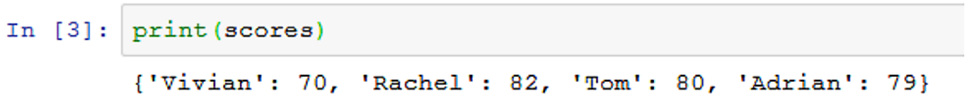
2. Each of these numbers can be used to index into the list of names and scores so that the correct name is associated with the correct `points` value:

```
scores = { students[i]:points[i] for i in range(4) }
```

3. Finally, print out the dictionary you just created:

```
print(scores)
```

You will get the following output:



```
In [3]: print(scores)

        {'Vivian': 70, 'Rachel': 82, 'Tom': 80, 'Adrian': 79}
```

Figure Appendix.10 - A dictionary indicating names and scores as a key-value pair

In this activity, we worked on dictionary comprehension and multiple lists. We executed the code to print out a scorecard with two separate lists of names and scores and outputted their values.

Activity 20 – using random numbers to find the value of Pi

Solution:

1. You will need the `math` and `random` libraries to complete this activity:

```
import math
import random
```

2. Define the `approximate_pi` function:

```
def approximate_pi():
Set the counters to zero:
    total_points = 0
    within_circle = 0
```

3. Calculate the approximation multiple times:

```
    for i in range (10001):
```

4. Here, `x` and `y` are random numbers between `0` and `1`, which, together, represent a point in the unit square:

```
        x = random.random()
        y = random.random()
        total_points += 1
```

5. Use Pythagoras' Theorem to work out the distance between the point and the origin, (0,0):

```
        distance = math.sqrt(x**2+y**2)
        if distance < 1:
```

If the distance is less than 1, then this point is both inside the square and inside a circle of radius 1, centered on the origin:

```
        within_circle += 1
```

6. Yield a result every 1,000 points. There's no reason why this couldn't yield a result after each point, but the early estimates will be very imprecise, so let's assume that users want to draw a large sample of random values:

```
        if total_points % 1000 == 0:
```

7. The ratio of points within the circle to total points generated should be approximately π/4 because the points are uniformly distributed across the square. Only some of the points are both in the square and the circle, and the ratio of areas between the circle segment and the square is π/4:

```
        pi_estimate = 4 * within_circle / total_points
        if total_points == 10000:
```

8. After `10000` points are generated, return the estimate to complete the iteration. Using what you have learned about `itertools` in this chapter, you could turn this generator into an infinite sequence if you want to:

```
                return pi_estimate
        else:
```

   Yield successive approximations to π:

```
                yield pi_estimate
```

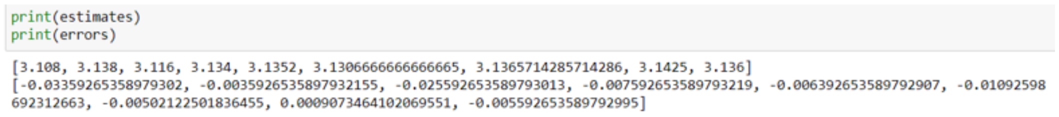
9. Use the generator to find estimates for the value of π. Additionally, use a list comprehension to find the errors: the difference between the estimated version and the "actual" value in Python's `math` module ("actual" is in scare quotes because it too is only an approximate value). Approximate values are used because Python cannot be exactly expressed in the computer's number system without using infinite memory:

```
estimates = [estimate for estimate in approximate_pi()]
errors = [estimate - math.pi for estimate in estimates]
```

10. Finally, print out our values and the errors to see how the generator performs:

```
print(estimates)
print(errors)
```

   You will get the following output:



```
print(estimates)
print(errors)

[3.108, 3.138, 3.116, 3.134, 3.1352, 3.1306666666666665, 3.1365714285714286, 3.1425, 3.136]
[-0.03359265358979302, -0.0035926535897932155, -0.025592653589793013, -0.007592653589793219, -0.006392653589792907, -0.01092598
692312663, -0.00502122501836455, 0.0009073464102069551, -0.005592653589792995]
```

Figure Appendix.11 - The output showing the generator yielding successive estimates of π

By completing this activity, you are now able to explain the working of generators. You successfully generated a plot of points, using which you were able to calculate the value of π. In the following section, we will learn about regular expressions.

Activity 21 – finding a winner for The X-Files

Solution:

1. First, create the list of names:

```
names = ["Xander Harris", "Jennifer Smith", "Timothy
Jones", "Amy Alexandrescu", "Peter Price", "Weifung Xu"]
```

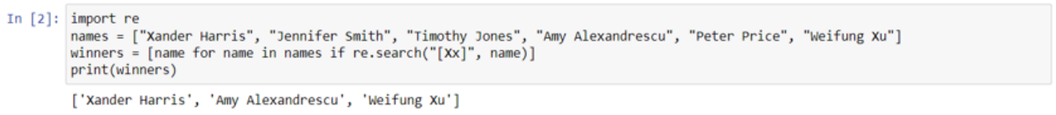
2. Using the list comprehension syntax from this chapter makes finding the winners as easy as a single line of Python:

```
winners = [name for name in names if re.search("[Xx]",
name)]
```

3. Finally, print the list of winners:

```
print(winners)
```

You will get the following output:



```
In [2]: import re
        names = ["Xander Harris", "Jennifer Smith", "Timothy Jones", "Amy Alexandrescu", "Peter Price", "Weifung Xu"]
        winners = [name for name in names if re.search("[Xx]", name)]
        print(winners)

        ['Xander Harris', 'Amy Alexandrescu', 'Weifung Xu']
```

Figure Appendix.12 - The output showing the winners list indicating the presence of "Xx" in the customer name

In this activity, we used regular expressions and Python's `re` module to find customers from a list whose name contains the value of `Xx`.

8. Software Development

Activity 22 – debugging sample Python code for an application

Solution:

1. First, you need to copy the source code, as demonstrated in the following code snippet:

```
DEFAULT_INITIAL_BASKET = ["orange", "apple"]
def create_picnic_basket(healthy, hungry,   initial_
basket=DEFAULT_INITIAL_BASKET):
    basket = initial_basket
    if healthy:
        basket.append("strawberry")
    else:
        basket.append("jam")
    if hungry:
        basket.append("sandwich")
    return basket
```

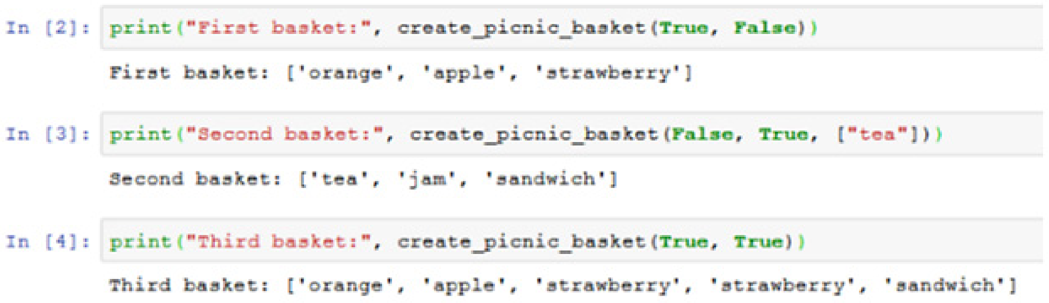
For the first step, the code creates a list of food that is based on an initial list that can be passed as an argument. There are then some flags that control what gets added. When `healthy` is true, a strawberry will get added. On the other hand, if it is false, the jam will be added instead. Finally, if the `hungry` flag is set to true, a sandwich will be added as well.

2. Run the code in your Jupyter Notebook, along with the reproducers, as demonstrated in the following code snippet:

```
# Reproducer
print("First basket:", create_picnic_basket(True, False))
print("Second basket:", create_picnic_basket(False, True,
["tea"]))
print("Third basket:", create_picnic_basket(True, True))
```

3. Observe the output; the issue will show up in the third basket, where there is one extra strawberry.

You will get the following output:



```
In [2]: print("First basket:", create_picnic_basket(True, False))
        First basket: ['orange', 'apple', 'strawberry']

In [3]: print("Second basket:", create_picnic_basket(False, True, ["tea"]))
        Second basket: ['tea', 'jam', 'sandwich']

In [4]: print("Third basket:", create_picnic_basket(True, True))
        Third basket: ['orange', 'apple', 'strawberry', 'strawberry', 'sandwich']
```

Figure Appendix.13 - The output with the additional item in the third basket

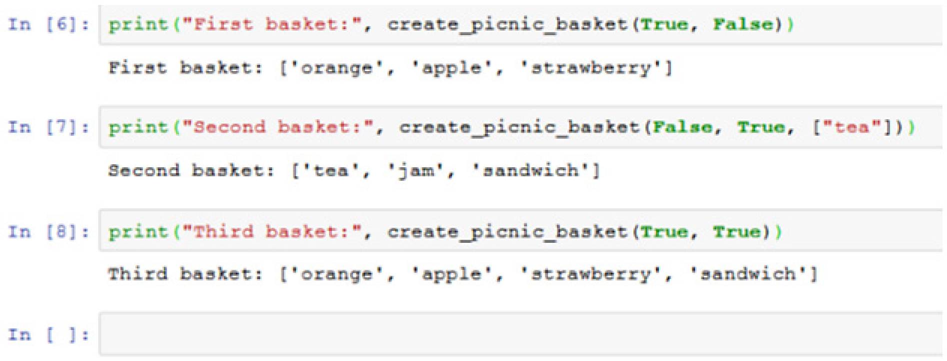
4. You will need to fix this by setting the basket value to `None` and using the `if-else` logic, as demonstrated in the following code snippet:

```
def create_picnic_basket(healthy, hungry, basket=None):
    if basket is None:
        basket = ["orange", "apple"]
    if healthy:
        basket.append("strawberry")
    else:
        basket.append("jam")
    if hungry:
        basket.append("sandwich")
    return basket
```

Note that default values in functions should not be mutable, as the modifications will persist across calls. The default basket should be set to `None` in the function declaration, and the constant should be used within the function. This is a great exercise to debug.

5. Now, run the reproducers once again, and it will be fixed.

The debugged output is as follows:



```
In [6]: print("First basket:", create_picnic_basket(True, False))

        First basket: ['orange', 'apple', 'strawberry']

In [7]: print("Second basket:", create_picnic_basket(False, True, ["tea"]))

        Second basket: ['tea', 'jam', 'sandwich']

In [8]: print("Third basket:", create_picnic_basket(True, True))

        Third basket: ['orange', 'apple', 'strawberry', 'sandwich']

In [ ]:
```

Figure Appendix.14 - Debugging the activity with the correct output

In this activity, you have implemented debugging to understand the source code, after which you were able to print the test cases (reproducers) and find the issue. You were then able to debug the code and fix it to achieve the desired output.

9. Practical Python – Advanced Topics

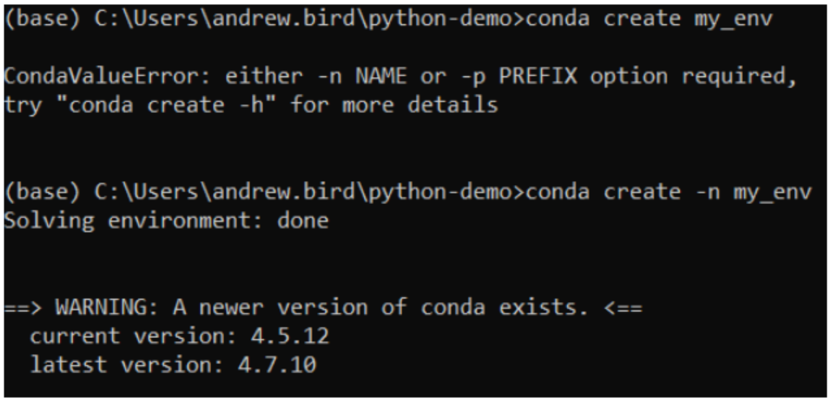
Activity 23 – generating a list of random numbers in a Python virtual environment

Solution:

1. Create a new `conda` environment called `my_env`:

   ```
   conda create -n my_env
   ```

   You will get the following output:



```
(base) C:\Users\andrew.bird\python-demo>conda create my_env

CondaValueError: either -n NAME or -p PREFIX option required,
try "conda create -h" for more details


(base) C:\Users\andrew.bird\python-demo>conda create -n my_env
Solving environment: done


==> WARNING: A newer version of conda exists. <==
  current version: 4.5.12
  latest version: 4.7.10
```

Figure Appendix.15 - Creating a new conda environment (truncated)

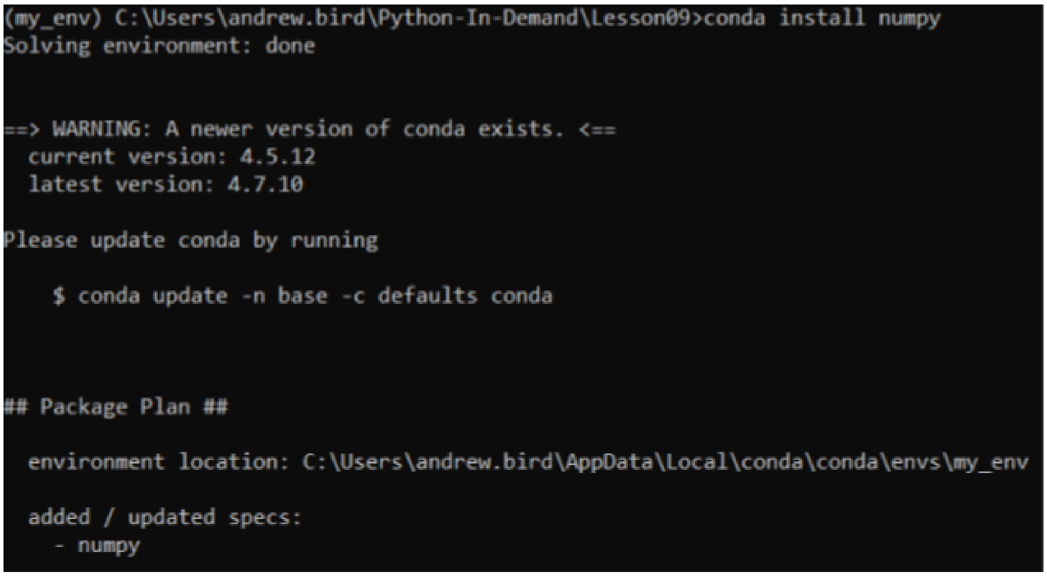
2. Activate the `conda` environment:

   ```
   conda activate my_env
   ```

3. Install `numpy` in your new environment:

   ```
   conda install numpy
   ```

   You will get the following output:



```
(my_env) C:\Users\andrew.bird\Python-In-Demand\Lesson09>conda install numpy
Solving environment: done


==> WARNING: A newer version of conda exists. <==
  current version: 4.5.12
  latest version: 4.7.10

Please update conda by running

    $ conda update -n base -c defaults conda



## Package Plan ##

  environment location: C:\Users\andrew.bird\AppData\Local\conda\conda\envs\my_env

  added / updated specs:
    - numpy
```

Figure Appendix.16 - Installing numpy (truncated)

4. Next, install and run a `jupyter` Notebook from within your virtual environment:

   ```
   conda install jupyter
   jupyter notebook
   ```

5. Create a new `jupyter` Notebook and start with the following imports:

   ```
   import threading
   import queue
   import cProfile
   import itertools
   import numpy as np
   ```

6. Create a function that uses the `numpy` library to generate an array of random numbers. Recall that when threading, we need to be able to send a signal for the `while` statement to terminate:

   ```
   in_queue = queue.Queue()
   out_queue = queue.Queue()
   def random_number_threading():
       while True:
           n = in_queue.get()
   ```

```
        if n == 'STOP':
            return
        random_numbers = np.random.rand(n)
        out_queue.put(random_numbers)
```

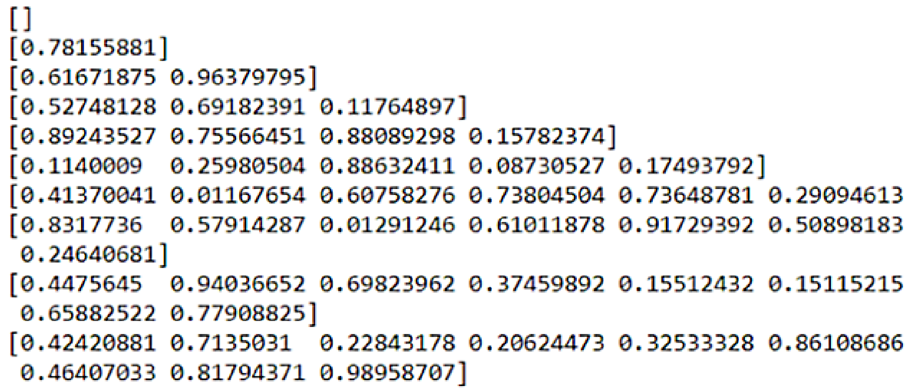
7. Next, let's add a function that will start a thread and put integers into the `in_queue` object. We can optionally print the output by setting the `show_output` argument to `True`:

```
def generate_random_numbers(show_output, up_to):
    thread = threading.Thread(target=random_number_
threading)
    thread.start()
    for i in range(up_to):
        in_queue.put(i)
        random_nums = out_queue.get()
        if show_output:
            print(random_nums)
    in_queue.put('STOP')
    thread.join()
```

8. Run the numbers on a small number of iterations to test and see the output:

```
generate_random_numbers(True, 10)
```

You will get the following output:



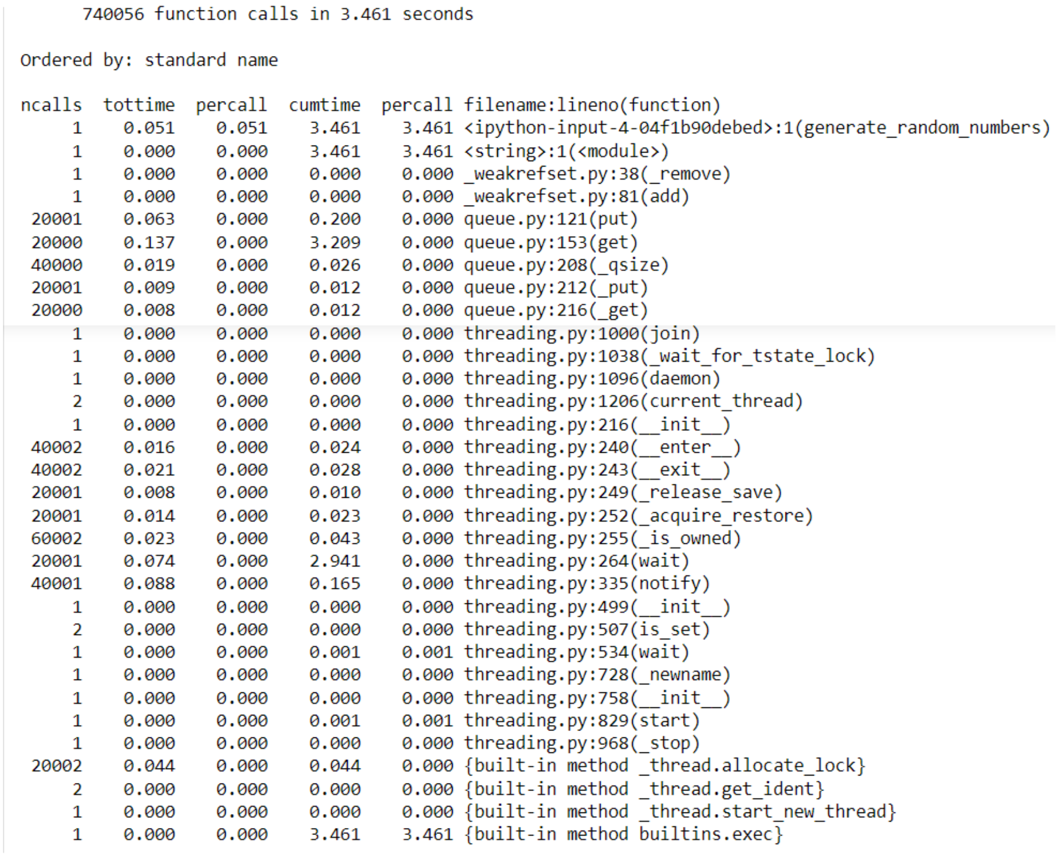
```
[]
[0.78155881]
[0.61671875 0.96379795]
[0.52748128 0.69182391 0.11764897]
[0.89243527 0.75566451 0.88089298 0.15782374]
[0.1140009  0.25980504 0.88632411 0.08730527 0.17493792]
[0.41370041 0.01167654 0.60758276 0.73804504 0.73648781 0.29094613
[0.8317736  0.57914287 0.01291246 0.61011878 0.91729392 0.50898183
 0.24640681]
[0.4475645  0.94036652 0.69823962 0.37459892 0.15512432 0.15115215
 0.65882522 0.77908825]
[0.42420881 0.7135031  0.22843178 0.20624473 0.32533328 0.86108686
 0.46407033 0.81794371 0.98958707]
```

Figure Appendix.17 - Generating lists of random numbers with numpy

9. Rerun the numbers with a large number of iterations and use `cProfile` to view a breakdown of what is taking time to execute:

```
cProfile.run('generate_random_numbers(False, 20000)')
```

You will get the following output:



```
         740056 function calls in 3.461 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.051    0.051    3.461    3.461 <ipython-input-4-04f1b90debed>:1(generate_random_numbers)
        1    0.000    0.000    3.461    3.461 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 _weakrefset.py:38(_remove)
        1    0.000    0.000    0.000    0.000 _weakrefset.py:81(add)
    20001    0.063    0.000    0.200    0.000 queue.py:121(put)
    20000    0.137    0.000    3.209    0.000 queue.py:153(get)
    40000    0.019    0.000    0.026    0.000 queue.py:208(_qsize)
    20001    0.009    0.000    0.012    0.000 queue.py:212(_put)
    20000    0.008    0.000    0.012    0.000 queue.py:216(_get)
        1    0.000    0.000    0.000    0.000 threading.py:1000(join)
        1    0.000    0.000    0.000    0.000 threading.py:1038(_wait_for_tstate_lock)
        1    0.000    0.000    0.000    0.000 threading.py:1096(daemon)
        2    0.000    0.000    0.000    0.000 threading.py:1206(current_thread)
        1    0.000    0.000    0.000    0.000 threading.py:216(__init__)
    40002    0.016    0.000    0.024    0.000 threading.py:240(__enter__)
    40002    0.021    0.000    0.028    0.000 threading.py:243(__exit__)
    20001    0.008    0.000    0.010    0.000 threading.py:249(_release_save)
    20001    0.014    0.000    0.023    0.000 threading.py:252(_acquire_restore)
    60002    0.023    0.000    0.043    0.000 threading.py:255(_is_owned)
    20001    0.074    0.000    2.941    0.000 threading.py:264(wait)
    40001    0.088    0.000    0.165    0.000 threading.py:335(notify)
        1    0.000    0.000    0.000    0.000 threading.py:499(__init__)
        2    0.000    0.000    0.000    0.000 threading.py:507(is_set)
        1    0.000    0.000    0.001    0.001 threading.py:534(wait)
        1    0.000    0.000    0.000    0.000 threading.py:728(_newname)
        1    0.000    0.000    0.000    0.000 threading.py:758(__init__)
        1    0.000    0.000    0.001    0.001 threading.py:829(start)
        1    0.000    0.000    0.000    0.000 threading.py:968(_stop)
    20002    0.044    0.000    0.044    0.000 {built-in method _thread.allocate_lock}
        2    0.000    0.000    0.000    0.000 {built-in method _thread.get_ident}
        1    0.000    0.000    0.000    0.000 {built-in method _thread.start_new_thread}
        1    0.000    0.000    3.461    3.461 {built-in method builtins.exec}
```

Figure Appendix.18 - cProfile output (truncated)

Having completed this activity, you now know how to execute programs in a `conda` virtual environment and get the final output as a set amount of time to execute the code. You also used `CProfiling` to analyze the time taken by various parts of your code, giving you the opportunity to diagnose which parts of your code were the least efficient.

10. Data Analytics with pandas and NumPy

Activity 24 – performing data analysis to find the outliers in pay versus the salary report in the UK statistics dataset

Solution:

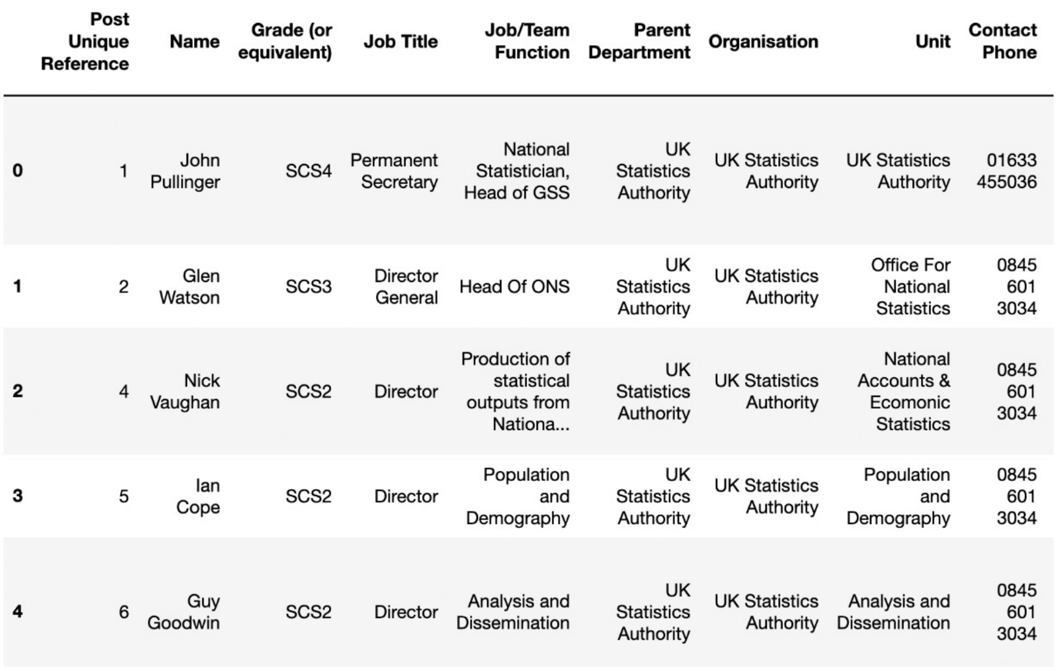
1. Copy the UK Statistics dataset file into a specific folder where you will be performing this activity.
2. Begin with a new Jupyter Notebook, and import the necessary data visualization packages, which include `pandas` as `pds`, `matplotlib` as `plt`, and `seaborn` as `sns`:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
# Set up seaborn dark grid
sns.set()
```

3. Choose a variable to store `DataFrame` and place the `UKStatistics.csv` file within the folder of your Jupyter Notebook to display the descriptive statistics and find information about each column as follows:

```
df = pd.read_csv('UKStatistics.csv')
df.head()
```

The truncated output will be as follows:



	Post Unique Reference	Name	Grade (or equivalent)	Job Title	Job/Team Function	Parent Department	Organisation	Unit	Contact Phone
0	1	John Pullinger	SCS4	Permanent Secretary	National Statistician, Head of GSS	UK Statistics Authority	UK Statistics Authority	UK Statistics Authority	01633 455036
1	2	Glen Watson	SCS3	Director General	Head Of ONS	UK Statistics Authority	UK Statistics Authority	Office For National Statistics	0845 601 3034
2	4	Nick Vaughan	SCS2	Director	Production of statistical outputs from Nationa...	UK Statistics Authority	UK Statistics Authority	National Accounts & Ecomonic Statistics	0845 601 3034
3	5	Ian Cope	SCS2	Director	Population and Demography	UK Statistics Authority	UK Statistics Authority	Population and Demography	0845 601 3034
4	6	Guy Goodwin	SCS2	Director	Analysis and Dissemination	UK Statistics Authority	UK Statistics Authority	Analysis and Dissemination	0845 601 3034

Figure Appendix.19 - Dataset output to view

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 51 entries, 0 to 50
Data columns (total 19 columns):
 #   Column                             Non-Null Count  Dtype
---  ------                             --------------  -----
 0   Post Unique Reference              51 non-null     object
 1   Name                               51 non-null     object
 2   Grade (or equivalent)              51 non-null     object
 3   Job Title                          51 non-null     object
 4   Job/Team Function                  49 non-null     object
 5   Parent Department                  51 non-null     object
 6   Organisation                       51 non-null     object
 7   Unit                               49 non-null     object
 8   Contact Phone                      46 non-null     object
 9   Contact E-mail                     49 non-null     object
 10  Reports to Senior Post             51 non-null     object
 11  Salary Cost of Reports (£)         51 non-null     int64
 12  FTE                                51 non-null     float64
 13  Actual Pay Floor (£)               51 non-null     int64
 14  Actual Pay Ceiling (£)             51 non-null     int64
 15  Unnamed: 15                        0 non-null      float64
 16  Professional/Occupational Group    51 non-null     object
 17  Notes                              27 non-null     object
 18  Valid?                             51 non-null     int64
dtypes: float64(2), int64(4), object(13)
memory usage: 7.7+ KB
```

Figure Appendix.20 - Info of the various columns

```
df.describe()
```

	Salary Cost of Reports (£)	FTE	Actual Pay Floor (£)	Actual Pay Ceiling (£)	Unnamed: 15	Valid?
count	5.100000e+01	51.000000	51.000000	51.000000	0.0	51.0
mean	1.651590e+06	0.982157	22745.098039	23823.313725	NaN	1.0
std	2.656579e+06	0.101771	45114.451404	47133.322796	NaN	0.0
min	0.000000e+00	0.300000	0.000000	0.000000	NaN	1.0
25%	0.000000e+00	1.000000	0.000000	0.000000	NaN	1.0
50%	1.105736e+06	1.000000	0.000000	0.000000	NaN	1.0
75%	2.175629e+06	1.000000	0.000000	0.000000	NaN	1.0
max	1.672047e+07	1.000000	150000.000000	154999.000000	NaN	1.0

Figure Appendix.21 - Statistics of the various columns

4. Now, to plot a histogram of the data for `Actual Pay Floor (£)`, we use the `.hist` method, as mentioned in the following code snippet. Here, you will see the difference in `Pay Floor` in the histogram:

```
title = 'Pay Floor Histogram'
plt.figure(figsize=(14,8))
plt.hist(df['Actual Pay Floor (£)'], color='green',
alpha=0.6)
plt.title(title, fontsize=15)
plt.xlabel('Actual Pay Floor (£)')
plt.ylabel('Count')
plt.savefig(title, dpi=300)
plt.show()
```

The output will be as follows:

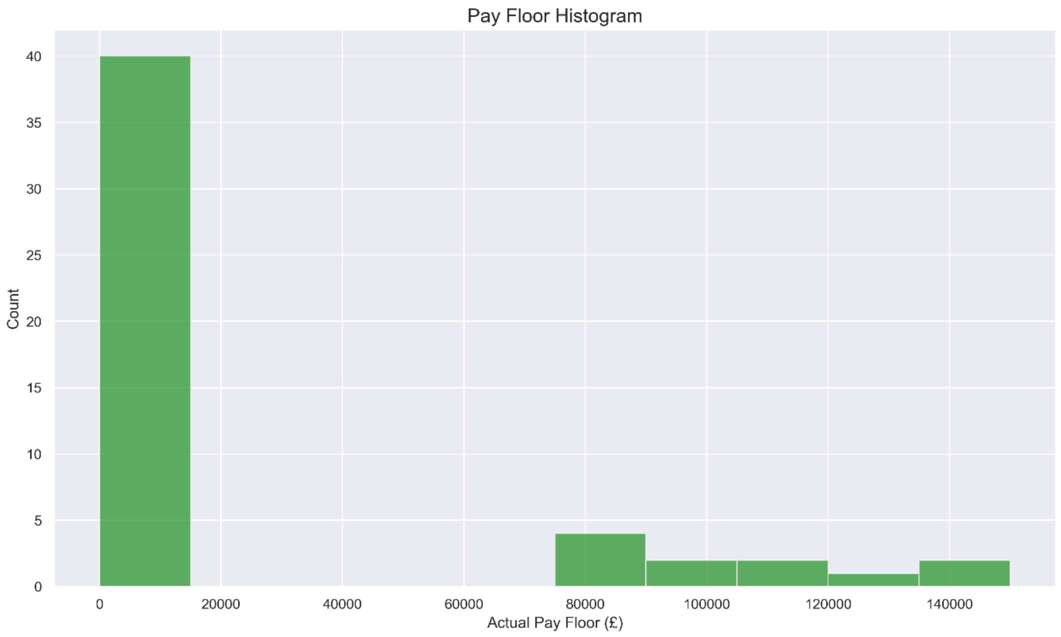


Figure Appendix.22 - Output as a histogram

5. To plot the scatter plot, we use the `.scatter` method, and we will be comparing the x values as `Salary Cost of Reports (£)`, and y as `Actual Pay Floor (£)`:

```
plt.figure(figsize=(16,10))
my_title='Salary Cost and Pay Floor'
```

```
plt.title(my_title, size=15)
sns.scatterplot(x=df['Salary Cost of Reports (£)'],
                y=df['Actual Pay Floor (£)'],
                hue=df['FTE'],
                size=df['Actual Pay Ceiling (£)'],
                sizes=(100, 500),
                palette='Greens')
plt.savefig(my_title, dpi=300)
plt.show()
```

The output will be as follows:

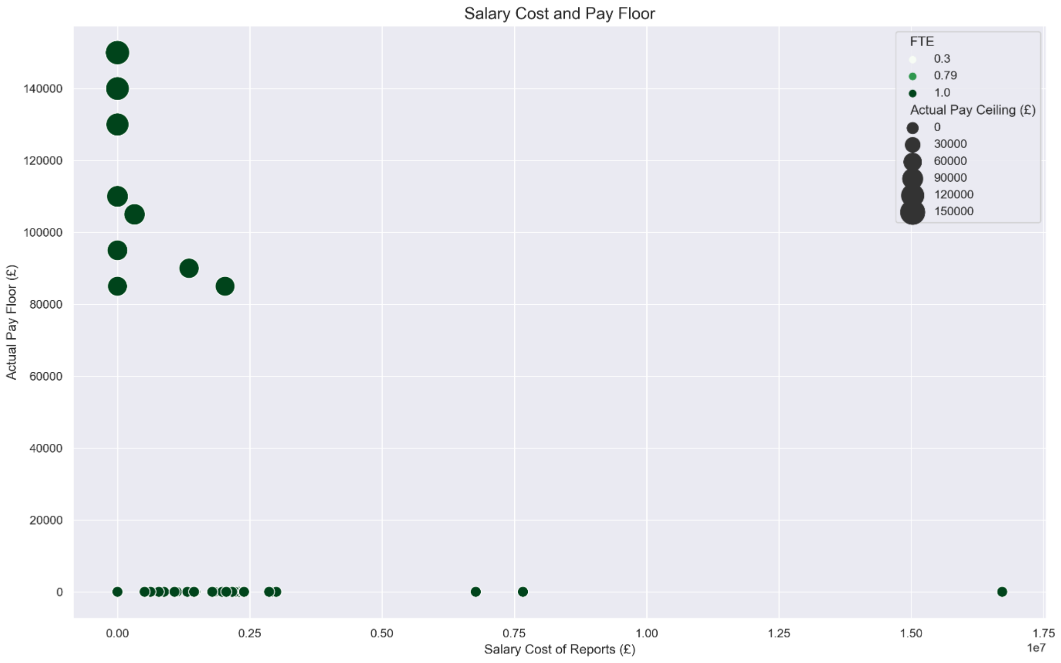


Figure Appendix.23 - Output of the scatter plot

6. Next, you need to create a box plot the `Salary Cost of Reports (£)` and `Actual Pay Floor (£)` using the box plot graphs as follows:

```
plt.figure(figsize=(14, 8))
title='Salary Cost Box Plot'
plt.title(title, size=15)
sns.boxplot(x=df['Salary Cost of Reports (£)'])
```

```
plt.savefig(title, dpi=300)
plt.show()
```

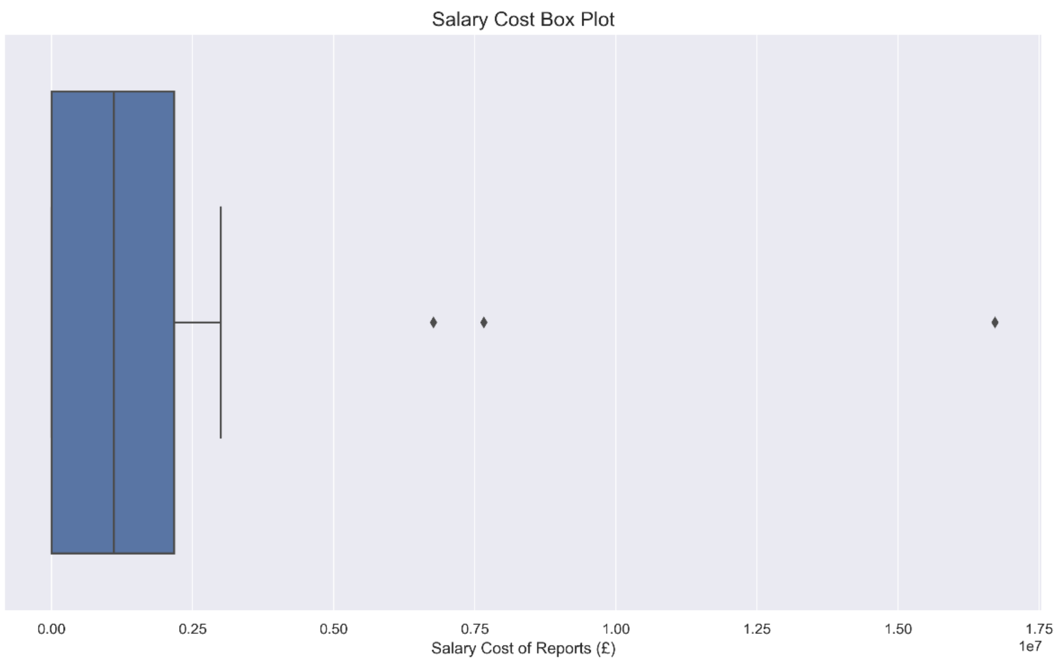


Figure Appendix.24 - Output of the box plot for Salary Cost

```
plt.figure(figsize=(14, 8))
title='Pay Floor Box Plot'
plt.title(title, size=15)
sns.boxplot(x=df['Actual Pay Floor (£)'])
plt.savefig(title, dpi=300)
plt.show()
```

The output will be as follows:

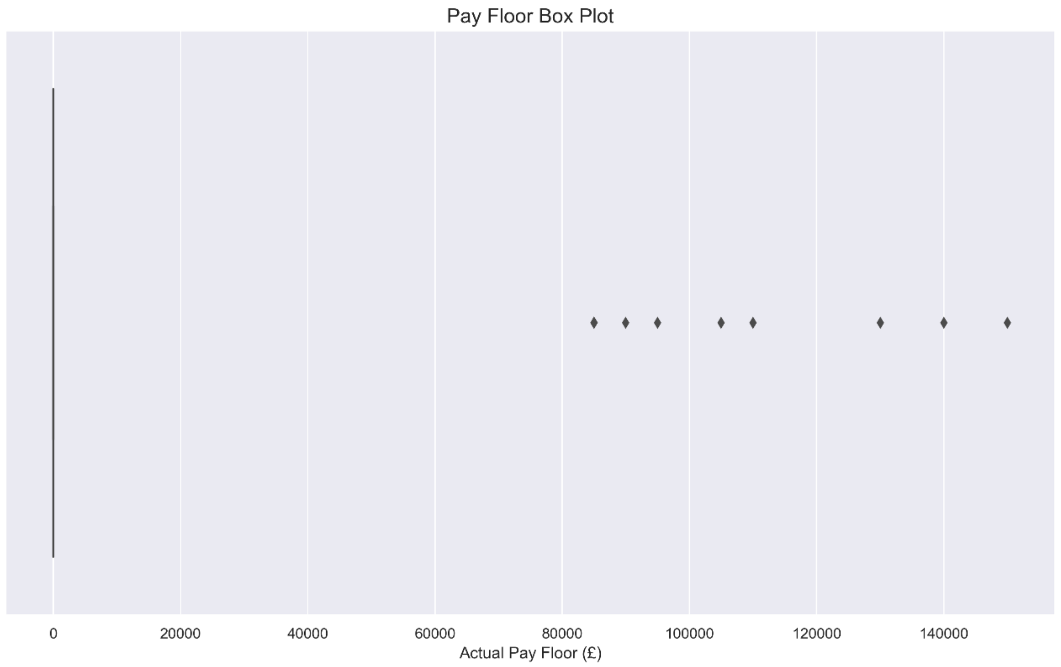


Figure Appendix.25 - Output of the box plot for Actual Pay Floor

In this activity, we compared two specific pieces of data from the dataset, that is, `Salary Cost of Reports (£)` and `Actual Pay Floor (£)`. We used histograms, scatter plots, and box plots to learn about and observe the differences between `Salary Cost of Reports` and `Actual Pay Floor`. The box plot graph clearly shows us that there are three outliers in the data that make it prone to inconsistencies regarding pay from the government.

11. Machine Learning

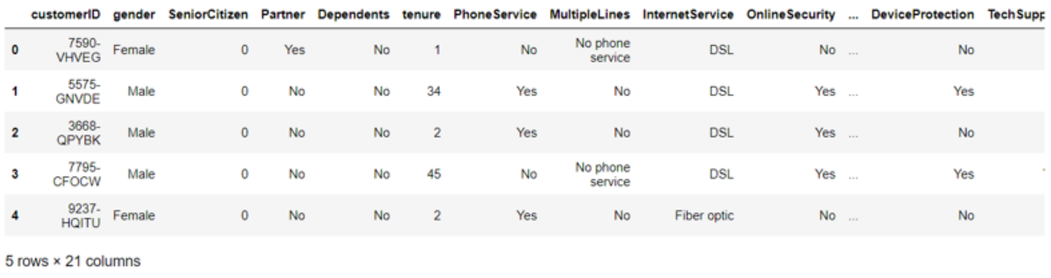
Activity 25 – using ML to predict customer return rate accuracy

Solution:

1. The first step asks you to download the dataset from a provided link.
2. Next, load the `CHURN.csv` file in a Jupyter Notebook and show the first 5 rows.

```
import pandas as pd
df = pd.read_csv('CHURN.csv')
df.head()
```

You will get the following output:



	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	...	DeviceProtection	TechSupp
0	7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	...	No	
1	5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	...	Yes	
2	3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	...	No	
3	7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	...	Yes	
4	9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	...	No	

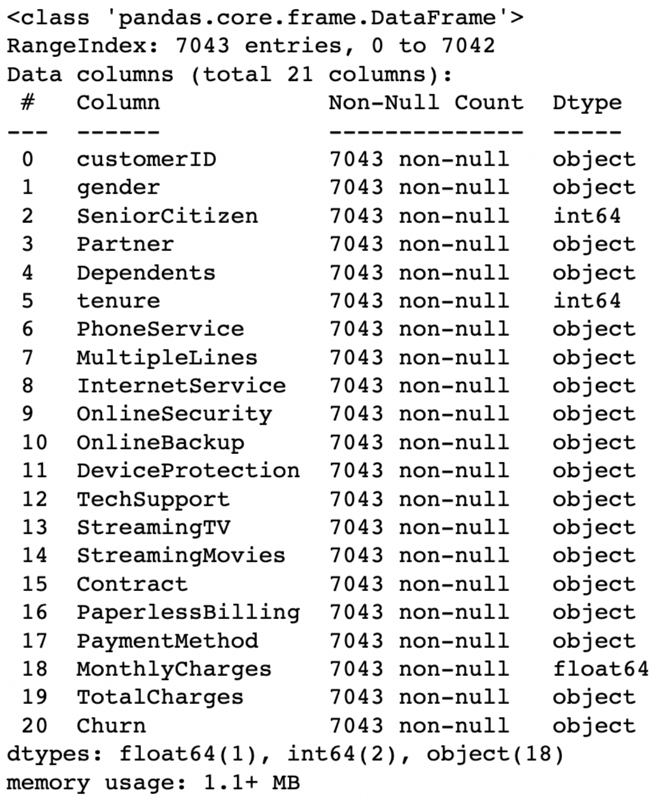
5 rows × 21 columns

Figure Appendix.26 - Dataset displaying the data as output

3. The next step asks you to check for `NaN` values. The following code reveals that there are none:

```
df.info()
```

You will get the following output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        7043 non-null   object 
 1   gender            7043 non-null   object 
 2   SeniorCitizen     7043 non-null   int64  
 3   Partner           7043 non-null   object 
 4   Dependents        7043 non-null   object 
 5   tenure            7043 non-null   int64  
 6   PhoneService      7043 non-null   object 
 7   MultipleLines     7043 non-null   object 
 8   InternetService   7043 non-null   object 
 9   OnlineSecurity    7043 non-null   object 
 10  OnlineBackup      7043 non-null   object 
 11  DeviceProtection  7043 non-null   object 
 12  TechSupport       7043 non-null   object 
 13  StreamingTV       7043 non-null   object 
 14  StreamingMovies   7043 non-null   object 
 15  Contract          7043 non-null   object 
 16  PaperlessBilling  7043 non-null   object 
 17  PaymentMethod     7043 non-null   object 
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object 
 20  Churn             7043 non-null   object 
dtypes: float64(1), int64(2), object(18)
memory usage: 1.1+ MB
```

Figure Appendix.27 - Information on the dataset

4. The next step is done for you. The following code converts `'No'` and `'Yes'` into `0` and `1`:

```
df['Churn'] = df['Churn'].replace(to_replace=['No',
'Yes'], value=[0, 1])
```

5. The next step asks you to correctly define `X` and `y`. The correct solution is as follows. Note that the first column is eliminated because a customer ID would not be useful in making predictions:

```
X = df.iloc[:,1:-1]
y = df.iloc[:, -1]
```

6. In order to transform all of the predictive columns into numeric columns, the following code will work:

```
X = pd.get_dummies(X)
```

7. This step asks you to write a classifier function with `cross_val_score`. This is done as follows:

```
from sklearn.model_selection import cross_val_score
def clf_model (model, cv=3):
    clf = model
    scores = cross_val_score(clf, X, y, cv=cv)
    print('Scores:', scores)
    print('Mean score', scores.mean())
```

8. The following code and output show the implementation of six classifiers (you are asked to choose 5), as required in this step:

   By using logistic regression:

```
from sklearn.linear_model import LogisticRegression
clf_model(LogisticRegression(max_iter=1000))
```

   You will get the following output:

```
Scores: [0.80238501 0.80238501 0.80400511]
Mean score 0.802925043315291
```

   By using KNeighborsClassifier:

```
from sklearn.neighbors import KNeighborsClassifier
clf_model(KNeighborsClassifier())
```

   You will get the following output:

```
Scores: [0.78109029 0.76192504 0.77290158]
Mean score 0.7719723028927428
```

By using `GaussianNB`:

```
from sklearn.naive_bayes import GaussianNB
clf_model(GaussianNB())
```

You will get the following output:

```
Scores: [0.27725724 0.28109029 0.27652322]
Mean score 0.2782902503153228
```

By using RandomForestClassifier:

```
from sklearn.ensemble import RandomForestClassifier
clf_model(RandomForestClassifier())
```

You will get the following output:

```
Scores: [0.79045997 0.78407155 0.78994461]
Mean score 0.7881587087748638
```

By using AdaBoostClassifier:

```
from sklearn.ensemble import AdaBoostClassifier
clf_model(AdaBoostClassifier())
```

You will get the following output:

```
Scores: [0.80366269 0.80451448 0.80059651]
Mean score 0.8029245594131428
```

By using XGBClassifier:

```
from xgboost import XGBClassifier
clf_model(XGBClassifier())
```

You will get the following output:

```
Scores: [0.7879046  0.77172061 0.78824031]
Mean score 0.782621839907265
```

You may or may not have the same warning as us in your notebook files. Generally speaking, warnings that do not interfere with code are okay. They are often used to warn the developer of future changes. The top three performing models, in this case, are `AdaBoostClassifer`, `RandomForestClassifier`, and `Logistic Regression`.

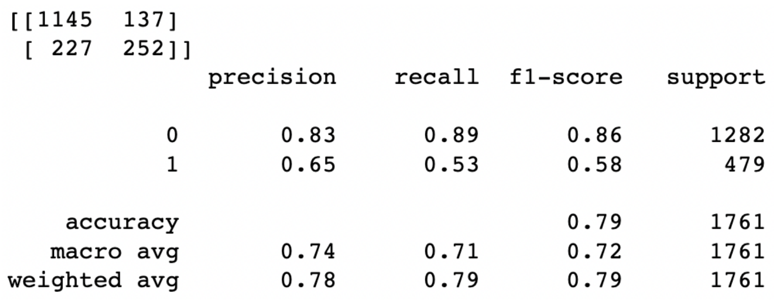
9. In this step, you are asked to build a function using the confusion matrix and the classification report and run it on your top three models. The following code does just that:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
X_train, X_test ,y_train, y_test = train_test_split(X, y,
test_size = 0.25)
def confusion(model):
    clf = model
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print('Confusion Matrix:', confusion_matrix(y_test,
y_pred))
    print('Classfication Report:', classification_
report(y_test, y_pred))
    return clf
```

10. Now, build a function for the confusion matrix using `AdaBoostClassifier`:

```
confusion(AdaBoostClassifier())
```

You should get the following output:



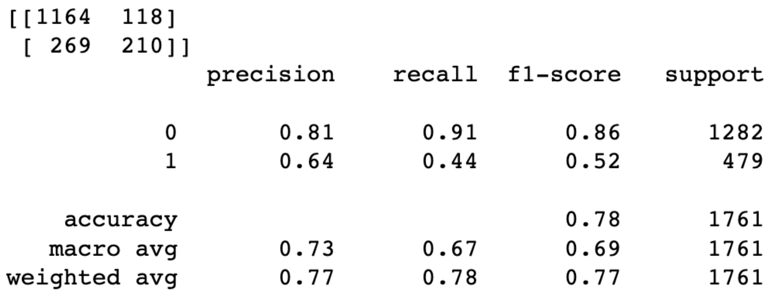
```
[[1145  137]
 [ 227  252]]
              precision    recall  f1-score   support

           0       0.83      0.89      0.86      1282
           1       0.65      0.53      0.58       479

    accuracy                           0.79      1761
   macro avg       0.74      0.71      0.72      1761
weighted avg       0.78      0.79      0.79      1761
```

Figure Appendix.28 - Output of the confusion matrix on AdaBoostClassifier

Confusion matrix using `RandomForestClassifier`:

```
confusion(RandomForestClassifier())
```

You will get the following output:



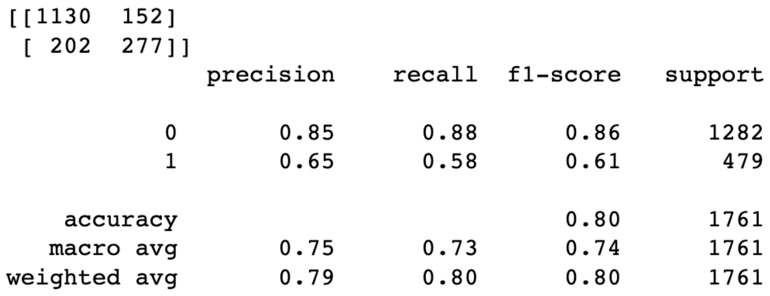
```
[[1164  118]
 [ 269  210]]
              precision    recall  f1-score   support

           0       0.81      0.91      0.86      1282
           1       0.64      0.44      0.52       479

    accuracy                           0.78      1761
   macro avg       0.73      0.67      0.69      1761
weighted avg       0.77      0.78      0.77      1761
```

Figure Appendix.29 - Output of the confusion matrix on RandomForestClassifier

Confusion matrix using `LogisticRegression`:

```
confusion(LogisticRegression())
```

You will get the following output:



```
[[1130  152]
 [ 202  277]]
              precision    recall  f1-score   support

           0       0.85      0.88      0.86      1282
           1       0.65      0.58      0.61       479

    accuracy                           0.80      1761
   macro avg       0.75      0.73      0.74      1761
weighted avg       0.79      0.80      0.80      1761
```

Figure Appendix.30 - Output of the confusion matrix on LogisticRegression

11. In this step, you are asked to optimize one hyperparameter for your best model. We looked up `AdaBoostClassifier()` and discovered the `n_estimators` hyperparameter, similar to the `n_estimators` of Random Forests. We tried several out and came up with the following result for `n_estimators=35`:

```
confusion(AdaBoostClassifier(n_estimators=35))
```

You will get the following output:

```
[[1143  139]
 [ 224  255]]
              precision    recall  f1-score   support

           0       0.84      0.89      0.86      1282
           1       0.65      0.53      0.58       479

    accuracy                           0.79      1761
   macro avg       0.74      0.71      0.72      1761
weighted avg       0.78      0.79      0.79      1761
```

Figure Appendix.31 - Output of the confusion matrix using n_estimators=250

As you can see by the end of this activity, when it comes to predicting user churn, `AdaBoostClassifier(n_estimators = 35)` gives the best predictions.

12. Deep Learning with Python

Activity 26 – Build your own neural network to predict whether a patient has heart disease

Solution:

1. The first step asks you to download the data via the provided link.
2. The next step asks you to set X and y which may be accomplished as follows:

```
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

3. Splitting the data into a training and test set may be done as follows:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

4. Initializing a Sequential model is shown here, with additional imports to be used later:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.callbacks import EarlyStopping
num_cols = X_train.shape[1]
model = Sequential()
```

5. Your first dense layer may be coded as such:

```
model.add(Dense(100, input_shape=(num_
cols,),activation='relu'))
```

6. A subsequent additional layer, if included, may be as follows:

```
model.add(Dense(10,activation='relu'))
```

7. Since this is classification, the final layer does require a sigmoid activation:

```
model.add(Dense(1, activation='sigmoid'))
```

8. Here is an early stopping monitor:

```
early_stopping_monitor = EarlyStopping(patience=10)
```

9. Here is the code to fit the data on the training set:

```
model.fit(X_train, y_train, epochs=1000,
     validation_split=0.2,
callbacks=[early_stopping_monitor])
```

10. Obtain, your score, as follows, making changes to the above code to find improvements:

```
model.evaluate(X_test, y_test)
```

A Colab Notebook with a full solution is here: `https://colab.research.google.com/drive/1O-F_0NwTlV3zMt6TrU4bUMeVhlsjLMY9?usp=sharing`

Activity 27 – Classify MNIST Fashion images using CNNs

Solution:

1. Download the MNIST Fashion dataset using the provided code.
2. Import your libraries as follows:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout, Conv2D,
MaxPool2D
```

3. Reshape the data, and set X and y as follows:

```
X = X/255
X_test = X_test/255
```

```
X = X.reshape((X.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
from keras.utils.np_utils import to_categorical
y = to_categorical(y, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

4. Initialize your Sequential model, and set your first convolutional layer as follows:

```
model = Sequential()
model.add(Conv2D(64, kernel_size=3, activation='relu',
input_shape=(28, 28, 1)))
```

5. Add additional layers as necessary; a sample combination is shown in the following code:

```
model.add(Dropout(0.2))
model.add(MaxPool2D(2))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPool2D(2))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

6. Evaluate your model, as shown in the following code.

```
model.compile(optimizer='adam', loss='categorical_
crossentropy',
metrics=['accuracy'])
model.fit(X, y, epochs=20)
model.evaluate(X_test, y_test)
```

7. Make any adjustments to the above code as needed.

A Colab Notebook with a full solution is here: `https://colab.research.google.com/drive/13uuqqXUhLrZ2N_Wj2KAFej8KSoVXZA7C?usp=sharing`