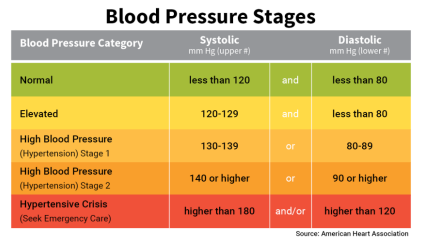
Python Lab Assignment 2

# Program 1: Blood pressure evaluator

Before starting, here are a few definitions. **Systolic blood pressure**, measures the pressure in your arteries when your heart beats. **Diastolic blood pressure** measures the pressure in your arteries when your heart rests between beats.

We are to write a python program to store the blood pressure test values of 10 patients, store the values in a dictionary (Systolic, Diastolic), and from the observation results, propose the stage of blood pressure they are in.

Here is a chart to indicate the blood pressure stages based on the systolic and diastolic blood pressure values…



To input 10 values, I have used a text file, named “patientData.txt”, where information on multiple patients is kept in a well-defined format: NAME, SYSTOLIC, DIASTOLIC <newline>

For due to the need to process and validate the data, I separated the code into two source files for convenience.

## Code (bpEvaluator.py) (main program)

from os.path import isfile

from bpDataProcessing import bpStage, processData

*# IMPORTANT VARIABLES*

dataFileName = "patientData.txt"

processedData = []

original = {} *# Empty dictionary or set... Dictionary in this case*

results = {} *# Empty dictionary or set... Dictionary in this case*

dataFile = "\*"

dataFileHeaderCount = 1

*# CHECKING IF DATA FILE EXISTS*

if isfile(dataFileName):

dataFile = open(dataFileName, "r") *# Opening the file stream.*

*# "r" mode is default. If no mode argument were passed, it would default to "r".*

else:

print("\nData file '" + dataFileName + "' not found.\n")

exit() *# Terminates main program.*

*# ENTERING THE STARTING LINE TO READ THE FILE FROM*

print("\nPATIENT BLOOD PRESSURE DATA RESULTS\n")

try:

start = int(input("Start reading from line... ")) - 1 + dataFileHeaderCount

*# We are dealing with lines by making a list of them.*

*# Hence, the 1st line will be at index 0.*

*# Hence, we subtract the inputted line number by 1.*

*# We then add to it the number of header lines expected in the file.*

*# Hence, if we input 1, we start at the 1st data line.*

except:

print("\nInvalid starting line. Defaulting to", dataFileHeaderCount, end = ".\n")

start = dataFileHeaderCount

*# Again accounting for the fact that the 1st index is 0, and there are "dataFileHeaderCount" expected header lines.*

*# STORING THE SPECIFIED 10 LINES OF THE FILE*

lines = dataFile.readlines()[start : start + 10]

*# There is no error if the upper bound exceeds the number of lines.*

*# Hence, "lines" will store less than or equal to 10 lines.*

print()

for line in lines:

*# PROCESSING THE LINE INTO A VALIDATED LIST*

processedData = processData(line)

if processedData == []:

print("Error in reading file!")

break

*# CREATING DICTIONARIES*

name = processedData[0]

systolic = processedData[1]

diastolic = processedData[2]

original.update({name : [systolic, diastolic]})

results.update({name : bpStage(systolic, diastolic)})

dataFile.close() *# Closing the file stream.*

*# FINAL DISPLAY*

if processedData != []:

print("File read successfully.")

print("------------------------------------")

print("ORIGINAL DATA\n")

for i, item in enumerate(original.items()):

print(i + start, item) *# i + start indexes the data by the line number in the file.*

print("------------------------------------")

print("EVALUATION RESULTS\n")

for i, item in enumerate(results.items()):

print(i + start, item) *# i + start indexes the data by the line number in the file.*

print()

else:

print("Nothing to display\n")

## Code (bpDataProcessing.py)

**def** bpStage(systolic, diastolic):

if systolic > 180 or diastolic > 120: return "Hypertensive Crisis"

if systolic >= 140 or diastolic >= 90: return "High Blood Pressure - Stage 2"

if systolic >= 130 or diastolic >= 80: return "High Blood Pressure - Stage 1"

if systolic >= 120 and diastolic < 80: return "Elevated"

if systolic < 120 and diastolic < 80: return "Normal"

**def** listify(line):

i = 0

max = len(line) - 1

processedData = []

while i < max:

myString = str()

while i < max and line[i].isspace(): i = i + 1

if line[i] != ",":

while i < max and line[i] != ",":

if (line[i] + line[i - 1]).isspace(): continue

*# The above statement skips excessive spaces.*

*# line[i] + line[i - 1] concatenates the current and previous characters of the line.*

myString = myString + line[i]

i = i + 1

processedData.append(myString)

else: i = i + 1

return processedData

**def** validateData(processedData):

errorMessage = "Non-numeric measurements!" *# Default error message*

try:

if len(processedData) != 3:

errorMessage = "Invalid length of data list!"

i = 1 / 0 *# Throws exception*

[processedData[1], processedData[2]] = [float(processedData[1]), float(processedData[2])]

*# Above is an example of packing variables and unpacking a list.*

if processedData[1] < 0 or processedData[2] < 0: i = 1 / 0

except:

print(errorMessage)

processedData = []

return processedData

**def** processData(line):

processedData = listify(line)

processedData = validateData(processedData)

return processedData

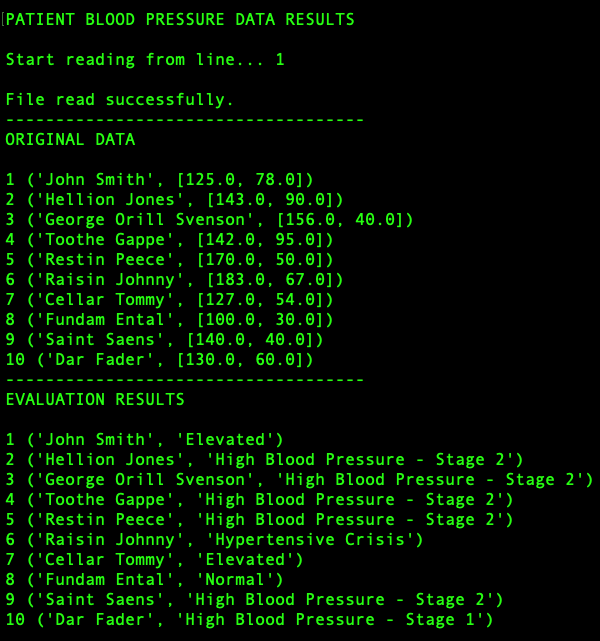
if \_\_name\_\_ == "\_\_main\_\_":

print("\nContains functions for data processing performed in 'bpEvaluator.py'\n")

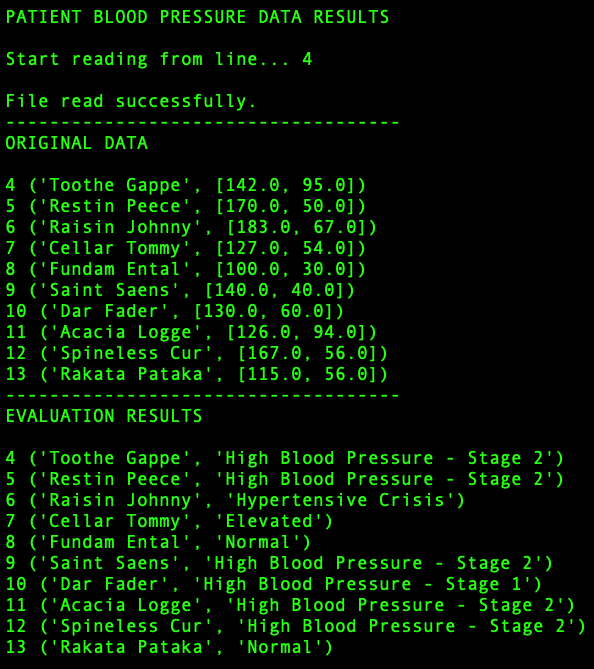
## Outputs

### No errors

#### From line 1…



#### From a different line…



### Errors in file data format

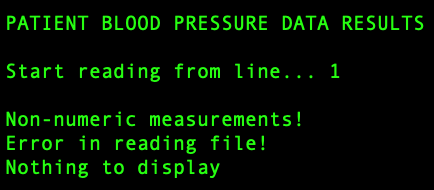
After changing the 1st record

**John Smith, 125, 78**

to

**John Smith, xyz, 78**

in the data file…



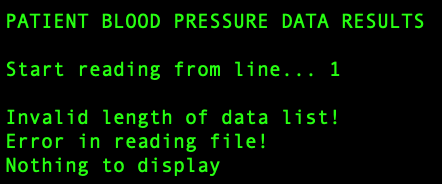
After changing 2nd record

**Hellion Jones, 143, 90**

to record

**Hellion Jones, 143,**

in the data file…



## Interpretation

It is impractical to store large amount of records on a run-time input basis. Furthermore, the utility of such input is limited. On the other hand, files are persistent, and if the data is properly formatted, it can be easy to read and store in the appropriate data structures.

Dictionaries are very effective in storing and displaying records with limited fields. However, for multiple fields, it may be easier or more effective to use multiple lists with corresponding indices. Sets are not a viable alternative, as their elements cannot be specifically referred to, and as they are completely unordered.

Separation of the code into modules, based on the functionalities of different parts of the code is extremely useful in organising your code in a logical and understandable manner.

Closing a file stream is less important in read-only mode, although it does clear the RAM as a copy of the file does not need to be maintained anymore. But it is vital in any form of write mode, since closing a stream commits the changes to persistent memory.

# Program 2: Set operation on lists

We are to take two lists, and write a python program (separate functions for each operation) that returns a list that contains only

* Elements two sets have in common. (Intersection)
* All the elements from both sets. (Union)
* Elements present on one set, but not on the other. (Difference)
* Elements from both sets, that are not present on the other. (Symmetric difference)

Make sure your program works on two lists of different sizes.

For this program, I previously used in-built set operations along with list and set constructor functions, and repackaging them as my own functions. However, to make things more interesting, I have commented this very concise code, and written a code that does not use sets or in-built set operations.

I used pre-defined lists, and included multiple options for the user to choose from, including operating on two lists at a time.

## Code

**def** union(list1, list2):

*#return list(set(list1 + list2))*

combined = list1 + list2

result = []

for i in combined:

if i not in result:

result.append(i)

return result

**def** intersection(list1, list2):

*#return list(set(list1).intersection(set(list2)))*

result = []

for i in list1:

if i in list2 and i not in result:

result.append(i)

return result

**def** difference(list1, list2):

*#return list(set(list1).difference(set(list2)))*

result = []

for i in list1:

if i not in list2 and i not in result:

result.append(i)

return result

**def** symmetricDifference(list1, list2):

*#return list(set(list1).symmetric\_difference(set(list2)))*

return union(difference(list1, list2), difference(list2, list1))

a = [1, 4, 2, 5, 4, 2, 6]

b = [5, 6, 2, 5, 4, 3]

c = [4, 5, 5, 8, 0, 45, 66, 73, 223]

d = [6, 7, 88, 45, 66, 23, 62, 22]

result = []

nameToListMap = {"a" : a, "b": b, "c" : c, "d" : d}

yesOptions = ["y", "yea", "yes", "yeah"]

**def** operate():

result = []

try:

list1 = nameToListMap[input("\nList 1: ")]

print("=", list1)

list2 = nameToListMap[input("List 2: ")]

print("=", list2)

except:

print("Unavailable list!")

return []

operation = input("Operation: ")

if operation.lower() == "u": result = union(list1, list2)

elif operation.lower() == "i": result = intersection(list1, list2)

elif operation.lower() == "d": result = difference(list1, list2)

elif operation.lower() == "s": result = symmetricDifference(list1, list2)

else: print("Invalid option!")

return result

**def** viewLists():

print()

for i in nameToListMap.keys():

print(i, "=", nameToListMap[i])

**def** viewOperations():

print("\nu : Union")

print("i : Intersection")

print("d : Difference")

print("s : Symmetric difference")

print("\nSET OPERATIONS ON LISTS", end = "")

while True:

print("\n------------------------")

print("1. Operate on lists")

print("2. View lists")

print("3. View operations")

print("x. Exit")

option = input("Enter option... ")

if option == "1": print("Result:", operate())

elif option == "2": viewLists()

elif option == "3": viewOperations()

elif option == "x": break

else: print("Invalid option!")

option = input("\nEncore? ")

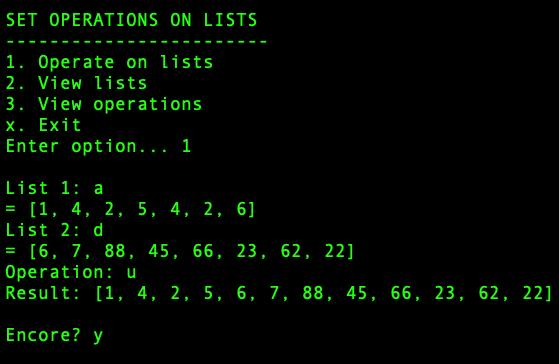
if option.lower() not in yesOptions: break

print("Program ended.\n")

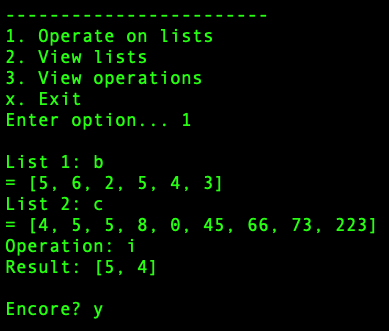
## Outputs

### No error

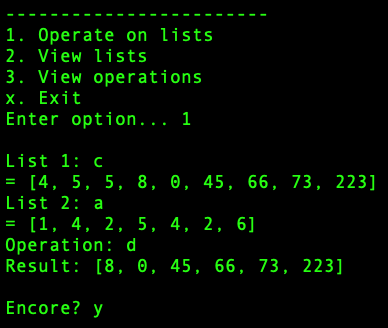
#### Union



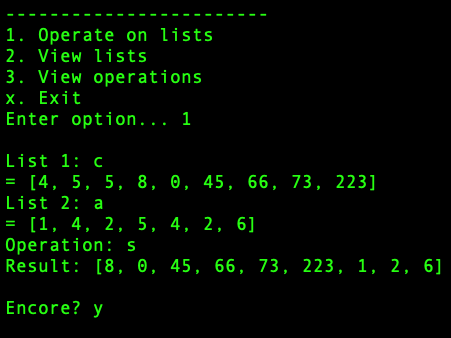
#### Intersection



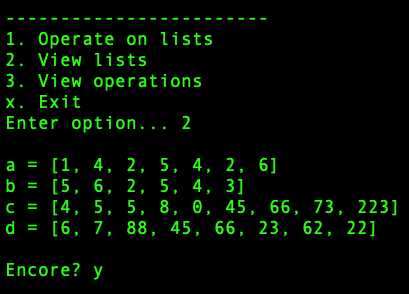
#### Difference



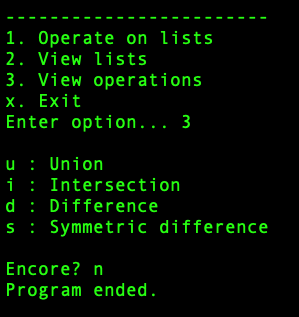
#### Symmetric difference



#### View lists



#### View operations



### Errors

#### Invalid option

### 

#### Unavailable list

### 

#### Invalid operation

### 

## Interpretation

Removal of duplicates is a constant concern when applying set operations to lists. Converting a list object to a set object automatically removes duplicates.

Symmetric difference is the union of the results of the difference applied both ways. This removes the need to include separate loops for this function.

Conversion functions from list to set types are in fact class constructors of the respective data types. If no arguments are passed, they return an empty instance of the respective data structure. These functions make it extremely easy and convenient to apply set operations on lists or other collection data types.