Group Fast Optimizer

Multiple Restriction Optimization Placement Problem For Class and Work Schedules

by

Furkan Eris
With Assoc. Prof. Faik Baskaya as Principle Investigator
Keywords: Optimization, Placement Problem, Multi Regression

05/20/2016

ABSTRACT

In the modern world we try to make life easier by optimizing and automating everything around us. In small ways we use computers to make an aggregate total towards making heaven around us. In this project I aim to help out in this process by automating and optimizing the school programs. Professors instead of using time and energy towards finding a program that is satisfactory towards all will instead be able to find a better program just by pressing a button.

This optimization process was done in Python using extensive algorithms that were done almost exclusively without the help of 3rd party libraries. Pandas was used for the systematic data frames instead of doing the reading from scratch, and a solver was used as the last step for the tree after all hard and soft constraints were done before.

The problem at heart was a very complex constraint problem with many different layers of constraints, all were taken into account and were put into order of hard and soft constraints with soft constraints taken as hard and removed as the code went on.

From this project I hoped to help people out in a small way that saves energy and time.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Background

Constraint satisfaction problems (**CSPs**) problems are problems defined as a group of objects that affect each other and must satisfy a group of rules or limitations called constraints. So they are made up of two parts the set and the constraints. These constraints are solved over the domains with constraint satisfaction methods. CSP problems are highly regarded in artificial intelligence and manufacturing problems. Thus they are very very important and will likely become more so over time. CSPs are often very complex and computing power is frequently not enough to get exact solutions, heuristics are used often and getting an exact solution is a very good feat for complex problems. The Boolean satisfiability problem (SAT), the satisfiability modulo theories (SMT) and answer set programming (ASP) Are some types of CSP problems.

Examples of CSP's are:

- Map coloring problem

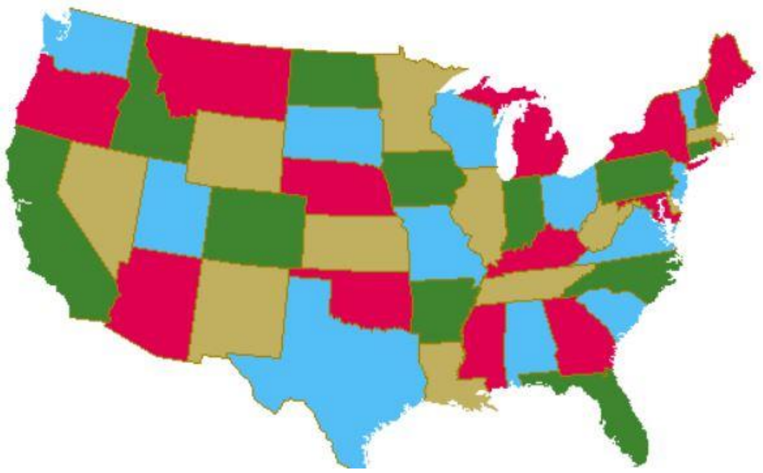


Figure 1.1: Map Coloring Problem (Coloring a map with boundaries with fewest different colors)

- Eight queens puzzle problem

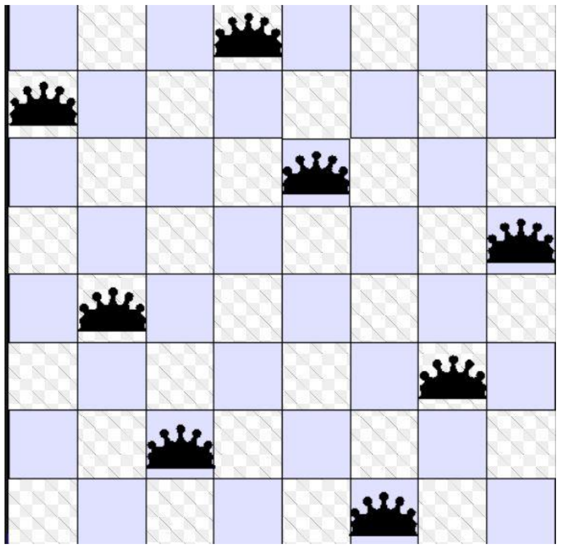
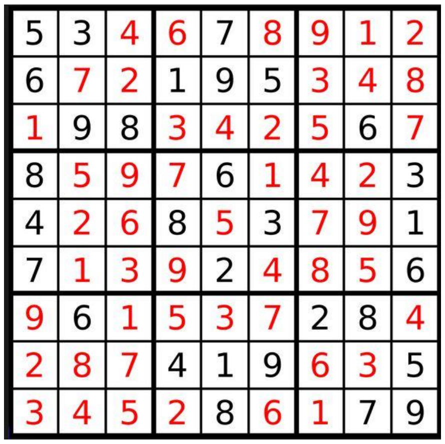


Figure 1.2: The famous 8 queens problem, queens are not to be able to eat each other

- Sudoku



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 1.3: A solved Sudoku again numbers effect one another, a perfect example of thinking in constraints

- Real life problems are: scheduling and decision making

Constrain problems are solved most frequently with search methods such variants of these techniques are variants of backtracking constraint propagation, and local search.

In this project I found that I was dealing with a constraint satisfaction problem after a couple weeks of research and stumbling upon the famous 8 queens problem, I discovered that while normally the 8 queens problem with brute force would take on the upwards of 3-4 hours on an average computer but with backtracking methods and domain reduction a 1000 queens problem was solvable within 20 30 mins.

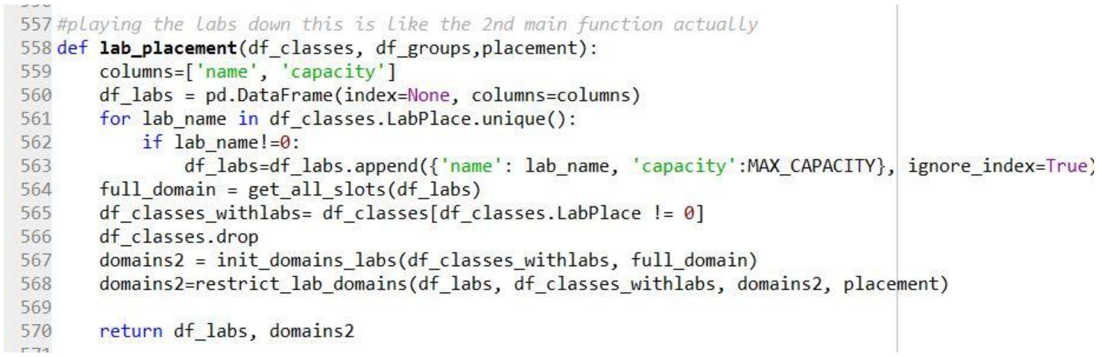
Thus I set out on this problem by trying to define everything within the confines of sets, domains and constraints hoping that at the end of the day I would have a problem that was solvable within a couple hours at worst. I had my some doubts but the 1000 queens problem gave me hope that my problem at worst would be 2 or 3 times more complex.

1.2 Main Function

To start off I would like to just put down my main function to show the general structure of the code.

First off I have 2 main like function one true main and a side main, because I decided very early on that since professors do not take part of the labs at Bogazici I could place them after placing the classes down (but with a constraint to get get classes that have lab sections at the beginning of the week and labs to the end of the week)

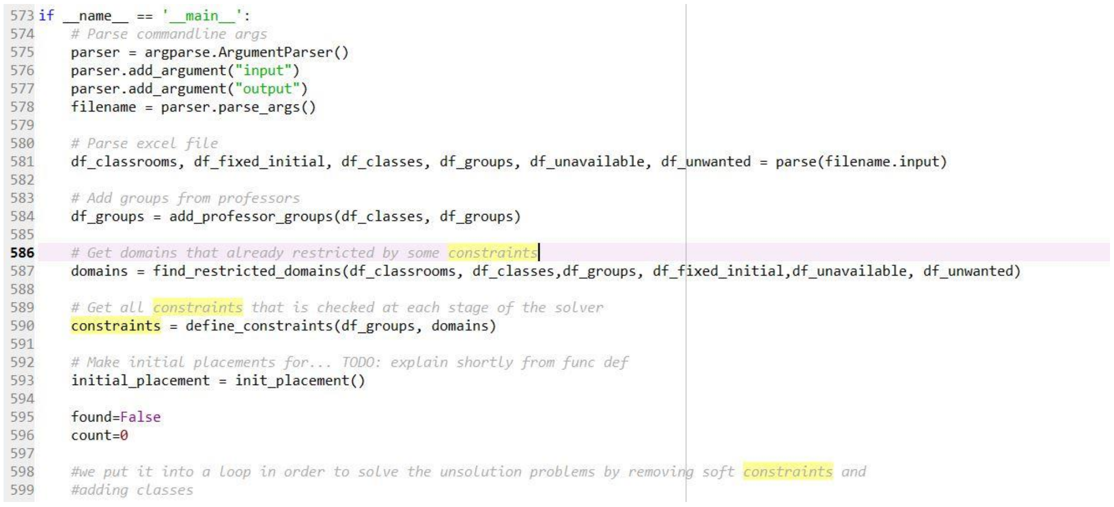
Lab main function:



```
#playing the labs down this is like the 2nd main function actually
def lab_placement(df_classes, df_groups,placement):
    columns=['name', 'capacity']
    df_labs = pd.DataFrame(index=None, columns=columns)
    for lab_name in df_classes.LabPlace.unique():
        if lab_name!=0:
            df_labs=df_labs.append({'name': lab_name, 'capacity':MAX_CAPACITY}, ignore_index=True
    full_domain = get_all_slots(df_labs)
    df_classes_withlabs= df_classes[df_classes.LabPlace != 0]
    df_classes.drop
    domains2 = init_domains_labs(df_classes_withlabs, full_domain)
    domains2=restrict_lab_domains(df_labs, df_classes_withlabs, domains2, placement)

    return df_labs, domains2
```

Code 1.1:The Lab Main Function

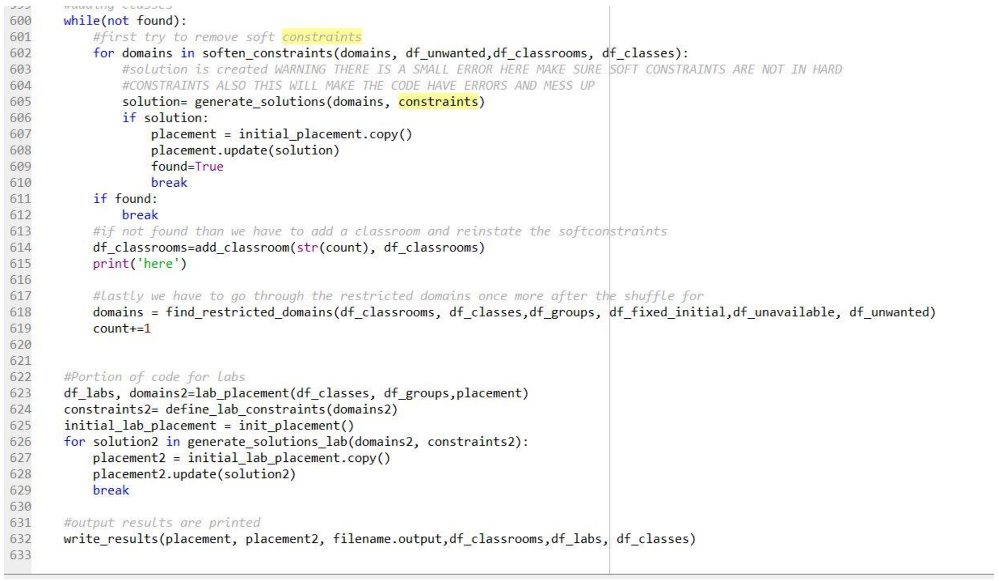
Lab placement function does a series of things it first creates a list of the labs available which for its counterpart is a list of classrooms, domains are created for the labs (of which I will go into in chapter 2 but it is basically the areas that a class or lab can occupy) domains are then dropped according to hard restrictions that were set at the beginnings of the project. These restrictions are later on in the main function given to a solver with constraints set up there also.



```python
if __name__ == '__main__':
    # Parse commandline args
    parser = argparse.ArgumentParser()
    parser.add_argument("input")
    parser.add_argument("output")
    filename = parser.parse_args()

    # Parse excel file
    df_classrooms, df_fixed_initial, df_classes, df_groups, df_unavailable, df_unwanted = parse(filename.input)

    # Add groups from professors
    df_groups = add_professor_groups(df_classes, df_groups)

    # Get domains that already restricted by some constraints
    domains = find_restricted_domains(df_classrooms, df_classes,df_groups, df_fixed_initial,df_unavailable, df_unwanted)

    # Get all constraints that is checked at each stage of the solver
    constraints = define_constraints(df_groups, domains)

    # Make initial placements for... TODO: explain shortly from func def
    initial_placement = init_placement()

    found=False
    count=0

    #we put it into a loop in order to solve the unsolution problems by removing soft constraints and
    #adding classes
```

Code 1.2: Main Function Part 1

The real main function has 4 portions to it (gone into in the chapters later on) First off is the parse function, next is initialization of domains and their pruning



```python
    while(not found):
        #first try to remove soft constraints
        for domains in soften_constraints(domains, df_unwanted,df_classrooms, df_classes):
            #solution is created WARNING THERE IS A SMALL ERROR HERE MAKE SURE SOFT CONSTRAINTS ARE NOT IN HARD
            #CONSTRAINTS ALSO THIS WILL MAKE THE CODE HAVE ERRORS AND MESS UP
            solution= generate_solutions(domains, constraints)
            if solution:
                placement = initial_placement.copy()
                placement.update(solution)
                found=True
                break
        if found:
            break
        #if not found than we have to add a classroom and reinstate the softconstraints
        df_classrooms=add_classroom(str(count), df_classrooms)
        print('here')

        #lastly we have to go through the restricted domains once more after the shuffle for
        domains = find_restricted_domains(df_classrooms, df_classes,df_groups, df_fixed_initial,df_unavailable, df_unwanted)
        count+=1


    #Portion of code for labs
    df_labs, domains2=lab_placement(df_classes, df_groups,placement)
    constraints2= define_lab_constraints(domains2)
    initial_lab_placement = init_placement()
    for solution2 in generate_solutions_lab(domains2, constraints2):
        placement2 = initial_lab_placement.copy()
        placement2.update(solution2)
        break

    #output results are printed
    write_results(placement, placement2, filename.output,df_classrooms,df_labs, df_classes)
```

Code 1.3: Main Function part 2

Next is the solver of dynamic constraints where there is a looping function that softens constraints as no solution turns up. Lastly is the writing portion of the code where I had to put the solved domains of the data frame back into a format that humans could understand.

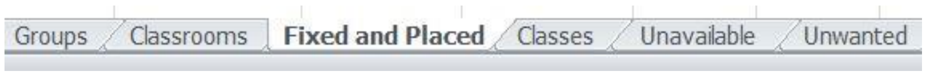
CHAPTER 2

READING, DOMAINS AND PANDAS

2.1 Input File Format

Early on I decided on using excel as my input data format, excel is user friendly and most users of computers have access to it. Excel also makes my task easier compared to CSV since I would have to read everything manually

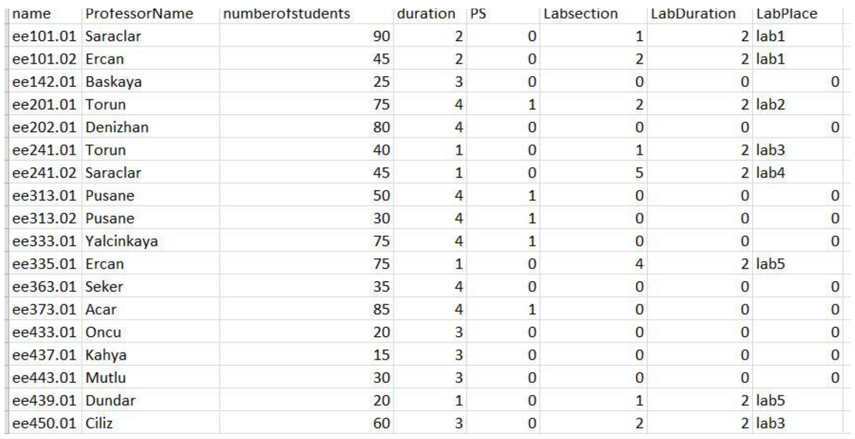
.



Groups | Classrooms | **Fixed and Placed** | Classes | Unavailable | Unwanted

Figure 2.1: The 6 data sheets of the input

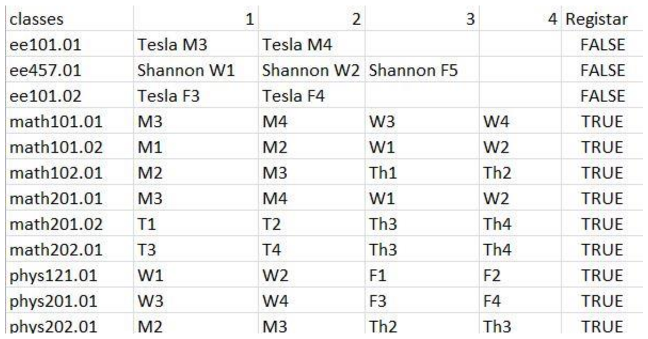
I will briefly introduce the sheets since input was designed in a very specific way to make it as easy as possible for the user but containing enough information to get all the data needed out of the data



name	ProfessorName	numberofstudents	duration	PS	Labsection	LabDuration	LabPlace
ee101.01	Saraclar	90	2	0	1	2	lab1
ee101.02	Ercan	45	2	0	2	2	lab1
ee142.01	Baskaya	25	3	0	0	0	0
ee201.01	Torun	75	4	1	2	2	lab2
ee202.01	Denizhan	80	4	0	0	0	0
ee241.01	Torun	40	1	0	1	2	lab3
ee241.02	Saraclar	45	1	0	5	2	lab4
ee313.01	Pusane	50	4	1	0	0	0
ee313.02	Pusane	30	4	1	0	0	0
ee333.01	Yalcinkaya	75	4	1	0	0	0
ee335.01	Ercan	75	1	0	4	2	lab5
ee363.01	Seker	35	4	0	0	0	0
ee373.01	Acar	85	4	1	0	0	0
ee433.01	Oncu	20	3	0	0	0	0
ee437.01	Kahya	15	3	0	0	0	0
ee443.01	Mutlu	30	3	0	0	0	0
ee439.01	Dundar	20	1	0	1	2	lab5
ee450.01	Ciliz	60	3	0	2	2	lab3

Figure 2.2: Classes Data Sheet

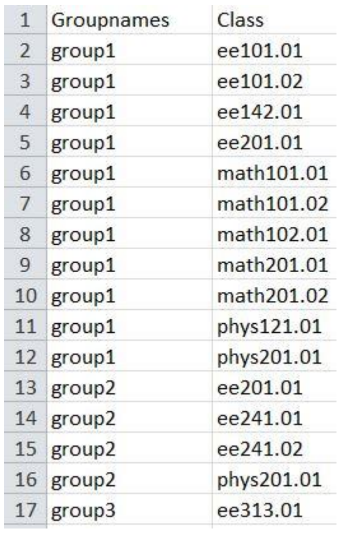
Classes is the main data sheet containing most of the information about the class program. Classes and the professors giving the classes duration and whether or not they have labs, PS are all in this data sheet.



classes	1	2	3	4	Registar
ee101.01	Tesla M3	Tesla M4			FALSE
ee457.01	Shannon W1	Shannon W2	Shannon F5		FALSE
ee101.02	Tesla F3	Tesla F4			FALSE
math101.01	M3	M4	W3	W4	TRUE
math101.02	M1	M2	W1	W2	TRUE
math102.01	M2	M3	Th1	Th2	TRUE
math201.01	M3	M4	W1	W2	TRUE
math201.02	T1	T2	Th3	Th4	TRUE
math202.01	T3	T4	Th3	Th4	TRUE
phys121.01	W1	W2	F1	F2	TRUE
phys201.01	W3	W4	F3	F4	TRUE
phys202.01	M2	M3	Th2	Th3	TRUE

Figure 2.3: Input Fixed and Placed

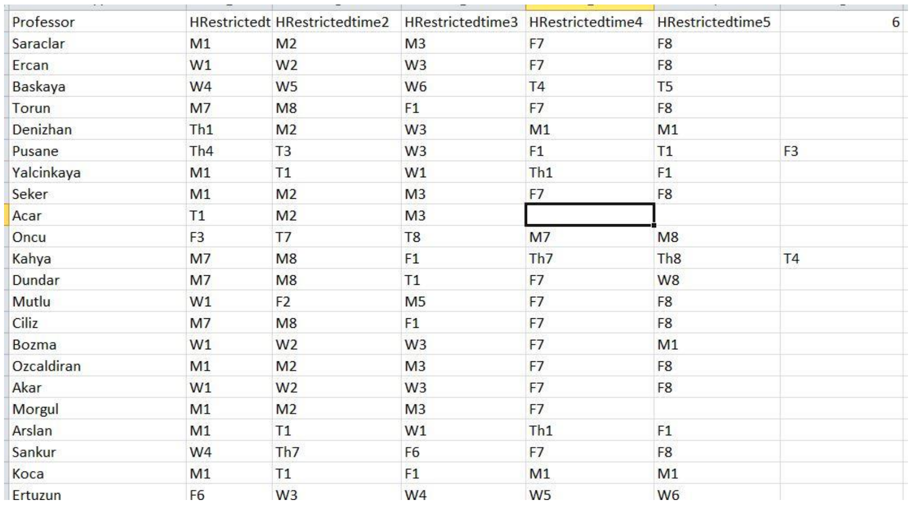
This input is for setting classes to specific domains, this function is unstable and will be discussed in later on chapters. This function is also for the register function that fixes times of classes that come from outside of the faculties control.



1	Groupnames	Class
2	group1	ee101.01
3	group1	ee101.02
4	group1	ee142.01
5	group1	ee201.01
6	group1	math101.01
7	group1	math101.02
8	group1	math102.01
9	group1	math201.01
10	group1	math201.02
11	group1	phys121.01
12	group1	phys201.01
13	group2	ee201.01
14	group2	ee241.01
15	group2	ee241.02
16	group2	phys201.01
17	group3	ee313.01

Figure 2.4: Groups Sheet

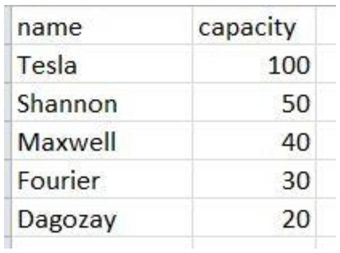
Groups sheet is also vital in the code because we have to separate the classes into groups that effect each other. Such as first year classes, electronics specialization classes etc.



Professor	HRestrictedt	HRestrictedtime2	HRestrictedtime3	HRestrictedtime4	HRestrictedtime5	6
Saraclar	M1	M2	M3	F7	F8	
Ercan	W1	W2	W3	F7	F8	
Baskaya	W4	W5	W6	T4	T5	
Torun	M7	M8	F1	F7	F8	
Denizhan	Th1	M2	W3	M1	M1	
Pusane	Th4	T3	W3	F1	T1	F3
Yalcinkaya	M1	T1	W1	Th1	F1	
Seker	M1	M2	M3	F7	F8	
Acar	T1	M2	M3			
Oncu	F3	T7	T8	M7	M8	
Kahya	M7	M8	F1	Th7	Th8	T4
Dundar	M7	M8	T1	F7	W8	
Mutlu	W1	F2	M5	F7	F8	
Ciliz	M7	M8	F1	F7	F8	
Bozma	W1	W2	W3	F7	M1	
Ozcaldiran	M1	M2	M3	F7	F8	
Akar	W1	W2	W3	F7	F8	
Morgul	M1	M2	M3	F7		
Arslan	M1	T1	W1	Th1	F1	
Sankur	W4	Th7	F6	F7	F8	
Koca	M1	T1	F1	M1	M1	
Ertuzun	F6	W3	W4	W5	W6	

Figure 2.5: The Professors Restriction

Each professor has 2 sheets where one is for hours that they CANNOT come to class, and one sheet where they DON'T WANT to come to class. The first is a hard restriction while the 2nd is soft. I will discuss a few problems about this function later on also.

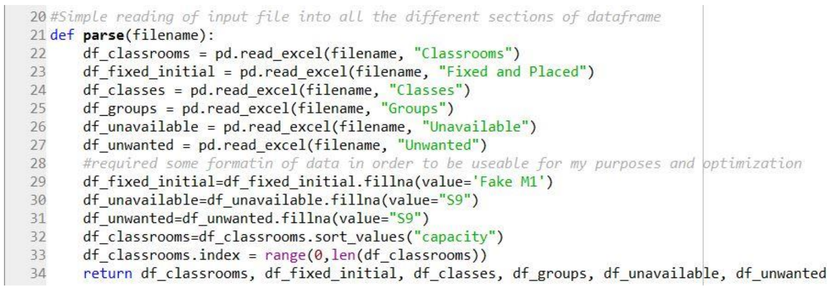


name	capacity
Tesla	100
Shannon	50
Maxwell	40
Fourier	30
Dagozay	20

Figure 2.6: Classrooms Sheet

The classrooms sheet is the list of places that we can use for the classes, if the classrooms are not enough as a last resort we request classrooms from the register.

2.2 Reading Setup

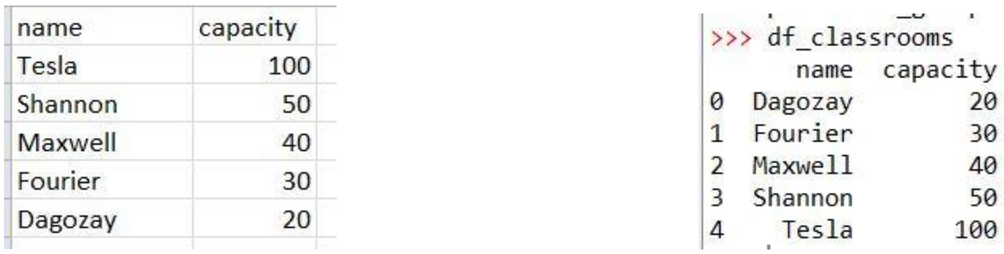


```python
#Simple reading of input file into all the different sections of dataframe
def parse(filename):
    df_classrooms = pd.read_excel(filename, "Classrooms")
    df_fixed_initial = pd.read_excel(filename, "Fixed and Placed")
    df_classes = pd.read_excel(filename, "Classes")
    df_groups = pd.read_excel(filename, "Groups")
    df_unavailable = pd.read_excel(filename, "Unavailable")
    df_unwanted = pd.read_excel(filename, "Unwanted")
    #required some formatin of data in order to be useable for my purposes and optimization
    df_fixed_initial=df_fixed_initial.fillna(value='Fake M1')
    df_unavailable=df_unavailable.fillna(value="S9")
    df_unwanted=df_unwanted.fillna(value="S9")
    df_classrooms=df_classrooms.sort_values("capacity")
    df_classrooms.index = range(0,len(df_classrooms))
    return df_classrooms, df_fixed_initial, df_classes, df_groups, df_unavailable, df_unwanted
```

Code 2.1: The Parse Function

The pandas library gives an easy reading code that takes in the excel sheets into structures called: Data Frames, I need to format these data frames into usable methods for later on, some of the sheets have to be arranged thus such as empty slots of fixed initial or the empty slots of unwanted/ unavailable(professor constraints) The classrooms for another constraint reason that I will explain in the next chapter had to be arranged from maximum capacity to min and the index had to be reset. Thus there is a lot of specific moves I had to do here for functions later on as most parse functions do indeed.

2.3 Pandas Data Frame and Domains



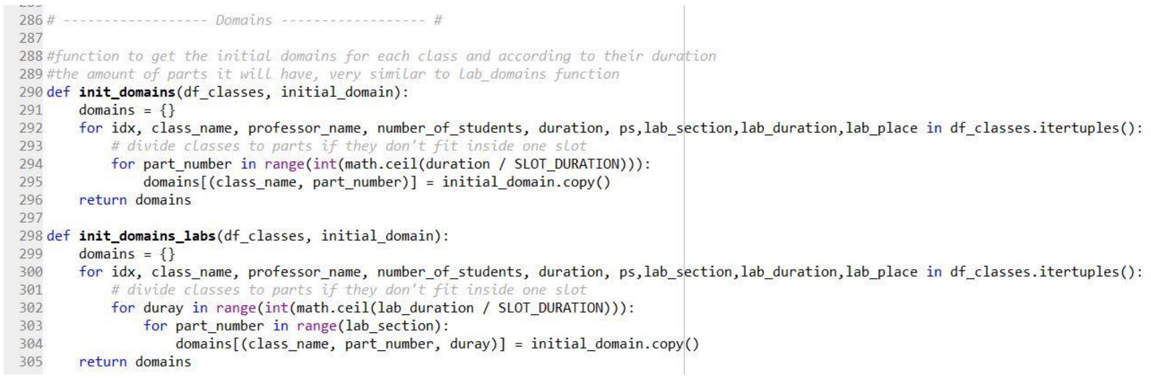
name	capacity
Tesla	100
Shannon	50
Maxwell	40
Fourier	30
Dagozay	20

```
>>> df_classrooms
      name  capacity
0  Dagozay        20
1  Fourier        30
2  Maxwell        40
3  Shannon        50
4    Tesla       100
```

Figure 2.7a and b: Input and After Reading i.e. Data Frame

As you can see that after reading pandas does a good job of getting all the input data into a very nice form. This is why after weeks of searching I choose pandas as my reader.

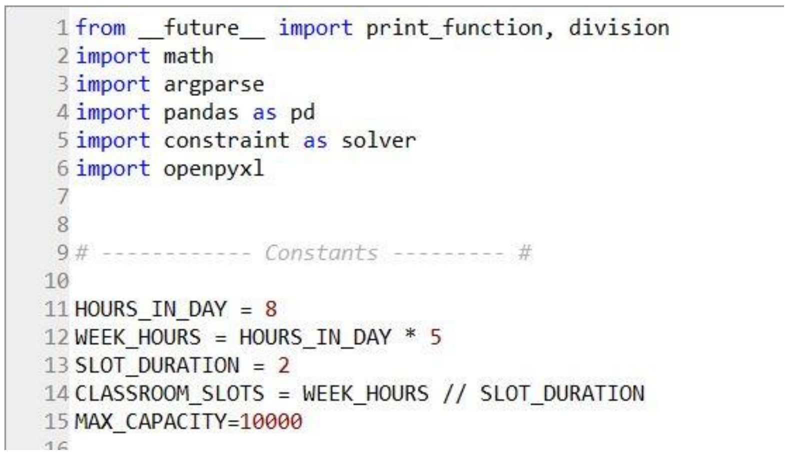
So after getting all the inputs as data frames like the one on the right and also changing some of the data frames for later on function I have to create the “domains” of the classes. I discussed this in chapter 1 where I talked about putting the problem into a format where I have domains and constraints at hand. So the first problem was deciding on how to create the domains of the classes. I decided that I would put each hour of each day into slots and the slots would differentiate for each class so it would be in some sense a 3D domain placed onto a 1D line by separating the line into portions.



```python
# ------------------ Domains ------------------ #

#function to get the initial domains for each class and according to their duration
#the amount of parts it will have, very similar to lab_domains function
def init_domains(df_classes, initial_domain):
    domains = {}
    for idx, class_name, professor_name, number_of_students, duration, ps,lab_section,lab_duration,lab_place in df_classes.itertuples():
        # divide classes to parts if they don't fit inside one slot
        for part_number in range(int(math.ceil(duration / SLOT_DURATION))):
            domains[(class_name, part_number)] = initial_domain.copy()
    return domains

def init_domains_labs(df_classes, initial_domain):
    domains = {}
    for idx, class_name, professor_name, number_of_students, duration, ps,lab_section,lab_duration,lab_place in df_classes.itertuples():
        # divide classes to parts if they don't fit inside one slot
        for duray in range(int(math.ceil(lab_duration / SLOT_DURATION))):
            for part_number in range(lab_section):
                domains[(class_name, part_number, duray)] = initial_domain.copy()
    return domains
```

Code 2.2: The initialization of the domains

The domains were made so that each classroom by their index would have a reserved portion of the number line while on each portion we would count the number of slots up for the week so that Monday Hour1 Classroom1 would be domain 1.



```python
from __future__ import print_function, division
import math
import argparse
import pandas as pd
import constraint as solver
import openpyxl


# ------------ Constants --------- #

HOURS_IN_DAY = 8
WEEK_HOURS = HOURS_IN_DAY * 5
SLOT_DURATION = 2
CLASSROOM_SLOTS = WEEK_HOURS // SLOT_DURATION
MAX_CAPACITY=10000
```

Code 2.3: Initializations

In order to determine the domains I put in some global indexes that could be changed by the user. The slot duration of 2 was chosen since the classes at Bogazici university are preferred to be 2 hours or 1 hour and not more. The number of hours in a day can also be increased but again 8 hours is the norm in Bogazici.

With this setup we get a nice 20 slot per classroom totaling 100 slots for 5 classrooms. So each class has 1-100 slots they can occupy each at the beginning before any constraints. The initialization functions given in *code 2.2* Show the way that each classes is also separated into parts if their duration is longer than the slot duration so that each class longer than 2 hours has 2 or more parts to place on the domain map. In this setup I had 31 classes separated into 55 parts in total. These 55 parts have to be placed into 1-100 slots.

The labs have a separate domain setup since they have labs to occupy and not classes. The labs in my setup were also 5 labs but they could have been 6 or any other number, the lab domains also have another factor in that they have multiple section that can be in the same day with also durations that could possibly split them into parts the same way classes are split up.

This is the main idea behind my whole code and is the part that I spent the longest on developing. In the next chapter I will discuss how I prepped these 1-100 domains for each part.

CHAPTER 3

HARD CONSTRAINTS AND DOMAIN REDUCTION (HARD SET)

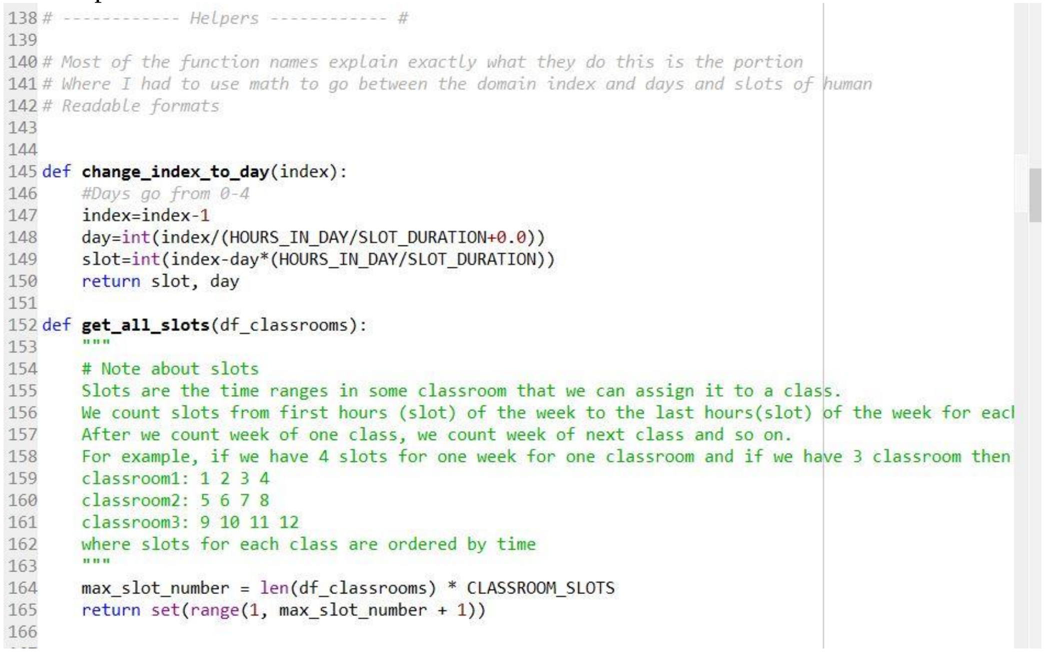
In this section I will talk about what I did with the domains next (the ones that were introduced in chapter 2)

The domains that were created now have slots open to them from 1-100 but they can be reduced by some hard set constraints that are not dynamic. The first step of this process was to find out what constraints I had and which of these constraints were hard-set or in other words reduced the size of the domain of a class irrespective of the domains of another class (to be more correct *the domains of a part of a class* since the parts effect each other in the more general fashion)

Thus I found out all the constraints by talking over the setup over many weeks with Prof. Faik. I then decided on the hard constraints that I talked about earlier and figured out the type of equations I would need in these functions. So the following subchapter talks about and shows the types of function I created for the purpose of reducing the size of domains.

As a last note before I get into the code I want to note that the 1000 queens problem was also solved in this way by domain reduction, it sped up the code by near 3 orders of magnitude so this portion was very important in the main problem of CSP (computation not being enough)

3.1 Helpers for Hard Reduction



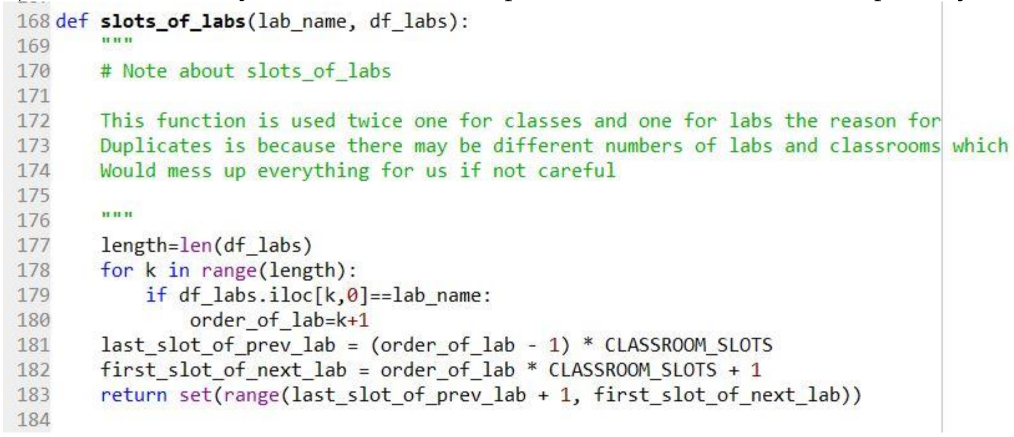
```
# ------------ Helpers ------------ #

# Most of the function names explain exactly what they do this is the portion
# Where I had to use math to go between the domain index and days and slots of human
# Readable formats


def change_index_to_day(index):
    #Days go from 0-4
    index=index-1
    day=int(index/(HOURS_IN_DAY/SLOT_DURATION+0.0))
    slot=int(index-day*(HOURS_IN_DAY/SLOT_DURATION))
    return slot, day

def get_all_slots(df_classrooms):
    """
    # Note about slots
    Slots are the time ranges in some classroom that we can assign it to a class.
    We count slots from first hours (slot) of the week to the last hours(slot) of the week for eacl
    After we count week of one class, we count week of next class and so on.
    For example, if we have 4 slots for one week for one classroom and if we have 3 classroom then
    classroom1: 1 2 3 4
    classroom2: 5 6 7 8
    classroom3: 9 10 11 12
    where slots for each class are ordered by time
    """
    max_slot_number = len(df_classrooms) * CLASSROOM_SLOTS
    return set(range(1, max_slot_number + 1))

```

Code 3.1: The Helpers Part1

The helper functions don't have an order of importance they are all equally important, so I will talk about each just in the order I coded them down and this section will be a little bit more straightforward since the helper functions are used in the section for elimination functions.

1) Change_index_to_day takes in the index (the number of the slot of 1-100 that I talked about) and outputs which day and time that index corresponds to irrelevant of the classroom so between 1 and 20 for this setup.

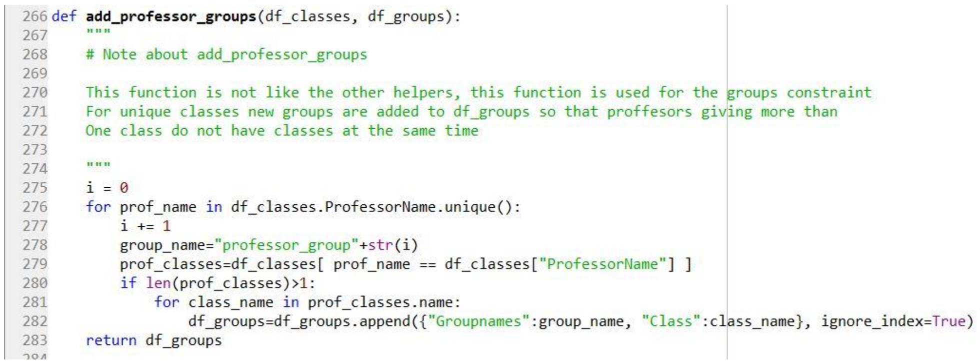
2) Get_all_slots is just a function that outputs all slots that a domain can possibly have from 1-100



```python
def slots_of_labs(lab_name, df_labs):
    """
    # Note about slots_of_labs

    This function is used twice one for classes and one for labs the reason for
    Duplicates is because there may be different numbers of labs and classrooms which
    Would mess up everything for us if not careful

    """
    length=len(df_labs)
    for k in range(length):
        if df_labs.iloc[k,0]==lab_name:
            order_of_lab=k+1
    last_slot_of_prev_lab = (order_of_lab - 1) * CLASSROOM_SLOTS
    first_slot_of_next_lab = order_of_lab * CLASSROOM_SLOTS + 1
    return set(range(last_slot_of_prev_lab + 1, first_slot_of_next_lab))
```

Code 3.2: The Helpers Part2

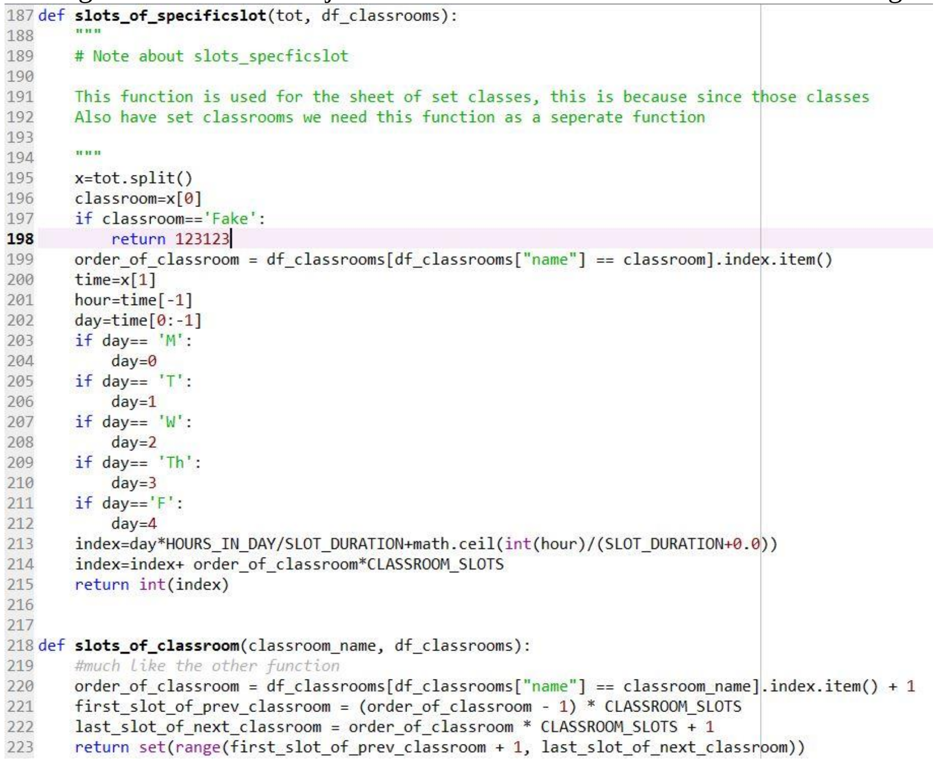
3) Slots_of_labs takes in all the labs and the specific lab we are looking for and outputs all the slots of that specific lab, so it is a more smaller form of all_slots and for labs this time



```python
def add_professor_groups(df_classes, df_groups):
    """
    # Note about add_professor_groups

    This function is not like the other helpers, this function is used for the groups constraint
    For unique classes new groups are added to df_groups so that proffesors giving more than
    One class do not have classes at the same time

    """
    i = 0
    for prof_name in df_classes.ProfessorName.unique():
        i += 1
        group_name="professor_group"+str(i)
        prof_classes=df_classes[ prof_name == df_classes["ProfessorName"] ]
        if len(prof_classes)>1:
            for class_name in prof_classes.name:
                df_groups=df_groups.append({"Groupnames":group_name, "Class":class_name}, ignore_index=True)
    return df_groups
```

Code 3.3: The Helpers Part3

4) Add_professor_groups Is a helper for a constraint function, this is talked about more expansively in chapter 4 but basically since some professors give more than one class the classes should not be during the same time duration so they act as if they are in the same group, so I decided instead of making a new data frame I just wrote a function to add all said classes into groups together.

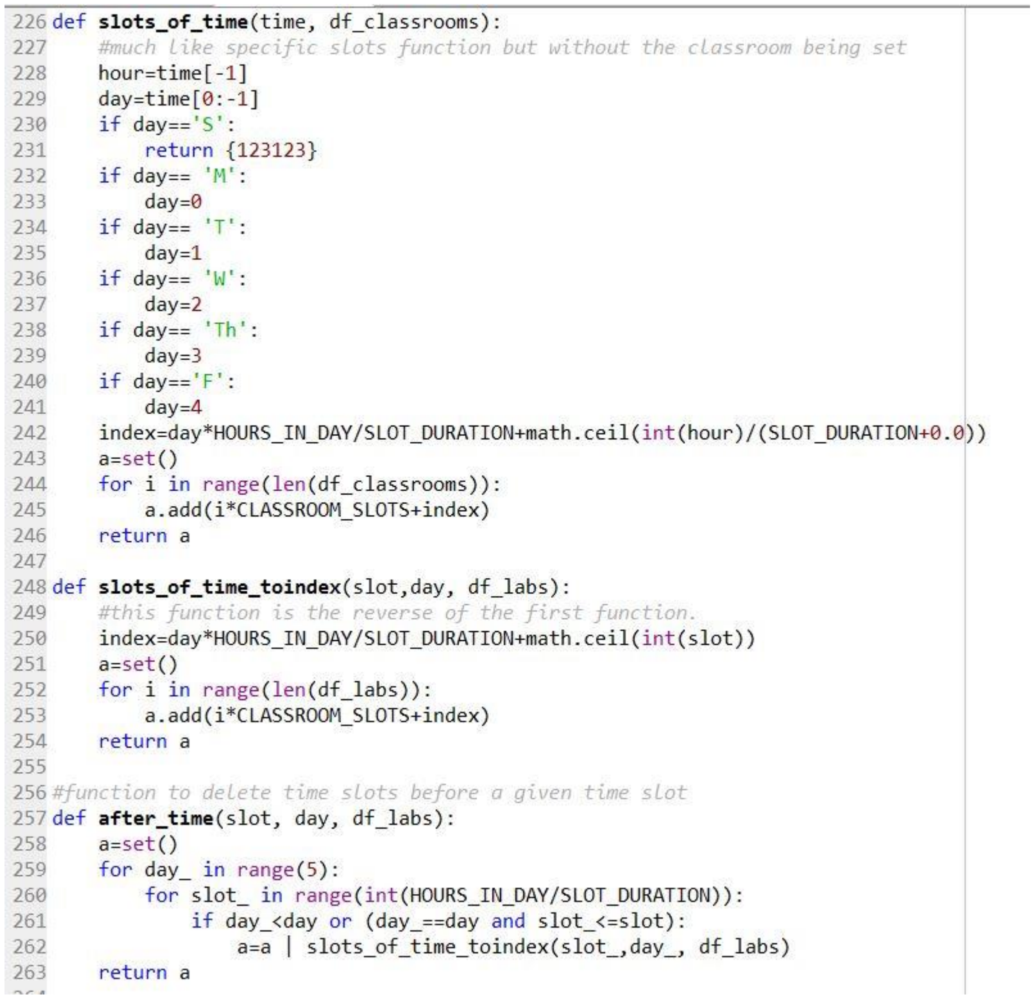


```python
def slots_of_specificslot(tot, df_classrooms):
    """
    # Note about slots_specficslot

    This function is used for the sheet of set classes, this is because since those classes
    Also have set classrooms we need this function as a seperate function

    """
    x=tot.split()
    classroom=x[0]
    if classroom=='Fake':
        return 123123
    order_of_classroom = df_classrooms[df_classrooms["name"] == classroom].index.item()
    time=x[1]
    hour=time[-1]
    day=time[0:-1]
    if day== 'M':
        day=0
    if day== 'T':
        day=1
    if day== 'W':
        day=2
    if day== 'Th':
        day=3
    if day=='F':
        day=4
    index=day*HOURS_IN_DAY/SLOT_DURATION+math.ceil(int(hour)/(SLOT_DURATION+0.0))
    index=index+ order_of_classroom*CLASSROOM_SLOTS
    return int(index)


def slots_of_classroom(classroom_name, df_classrooms):
    #much like the other function
    order_of_classroom = df_classrooms[df_classrooms["name"] == classroom_name].index.item() + 1
    first_slot_of_prev_classroom = (order_of_classroom - 1) * CLASSROOM_SLOTS
    last_slot_of_next_classroom = order_of_classroom * CLASSROOM_SLOTS + 1
    return set(range(first_slot_of_prev_classroom + 1, last_slot_of_next_classroom))
```

Code 3.4: The Helpers Part4

5) Slots_of_specificslot is a helper that outputs the exact index of a given class, time and day this function is used as part of the setclasses frame and is discussed in chapter6 as something to improve on to make more rigorous in the future, part of the function focuses on incoming NaN (where there are empty excel boxes because of durations changing from classes to class) The rest of the code is just simple math and manipulation

6) Slots_of_classroom is just like slots_of_labs it just outputs all the slots given to a specific classroom so a set of 20 indexes in this case for each classroom (since there are 20 slots to each classroom every week)



```
def slots_of_time(time, df_classrooms):
    #much like specific slots function but without the classroom being set
    hour=time[-1]
    day=time[0:-1]
    if day=='S':
        return {123123}
    if day== 'M':
        day=0
    if day== 'T':
        day=1
    if day== 'W':
        day=2
    if day== 'Th':
        day=3
    if day=='F':
        day=4
    index=day*HOURS_IN_DAY/SLOT_DURATION+math.ceil(int(hour)/(SLOT_DURATION+0.0))
    a=set()
    for i in range(len(df_classrooms)):
        a.add(i*CLASSROOM_SLOTS+index)
    return a

def slots_of_time_toindex(slot,day, df_labs):
    #this function is the reverse of the first function.
    index=day*HOURS_IN_DAY/SLOT_DURATION+math.ceil(int(slot))
    a=set()
    for i in range(len(df_labs)):
        a.add(i*CLASSROOM_SLOTS+index)
    return a

#function to delete time slots before a given time slot
def after_time(slot, day, df_labs):
    a=set()
    for day_ in range(5):
        for slot_ in range(int(HOURS_IN_DAY/SLOT_DURATION)):
            if day_<day or (day_==day and slot_<=slot):
                a=a | slots_of_time_toindex(slot_,day_, df_labs)
    return a
```

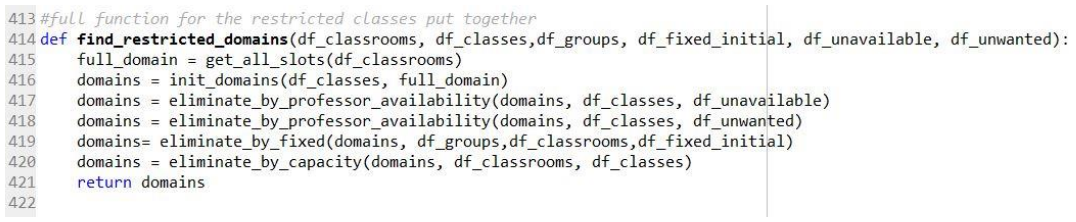
Code 3.5 The Helpers Part5

7) Slots_of_time is a less specific version of slots_of_specificslot; the input time gives all the slots for all classes of that time so for 5 classrooms we get an output of 5 slots.

8) slots_of_time_toindex This function is the same as slots_of_time except it takes in the hour and day instead of them as a string together, writing this function made the main functions more clean trying to change the input way every other function

3.2 Hard Set Elimination

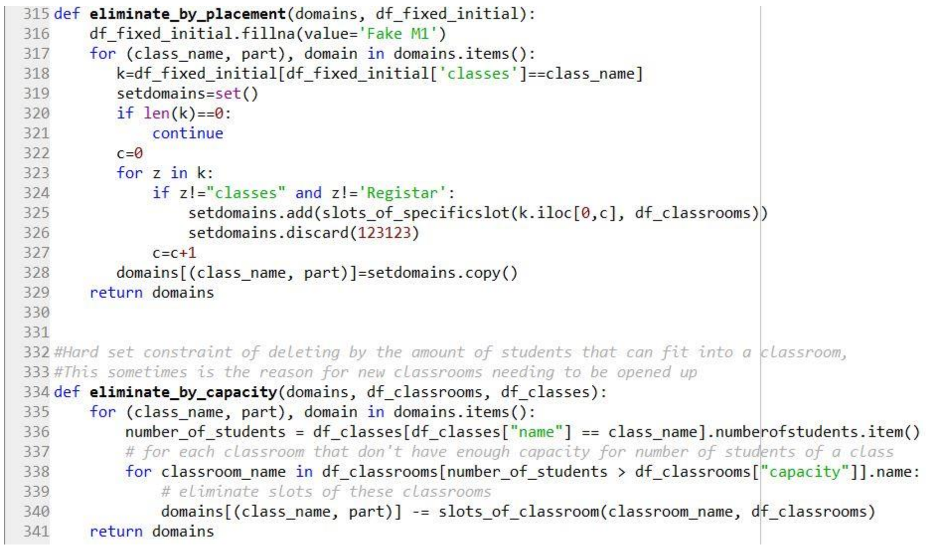
Now is the time to eliminate domain members according to stable constraints.



```python
#full function for the restricted classes put together
def find_restricted_domains(df_classrooms, df_classes,df_groups, df_fixed_initial, df_unavailable, df_unwanted):
    full_domain = get_all_slots(df_classrooms)
    domains = init_domains(df_classes, full_domain)
    domains = eliminate_by_professor_availability(domains, df_classes, df_unavailable)
    domains = eliminate_by_professor_availability(domains, df_classes, df_unwanted)
    domains= eliminate_by_fixed(domains, df_groups,df_classrooms,df_fixed_initial)
    domains = eliminate_by_capacity(domains, df_classrooms, df_classes)
    return domains

```

Code 3.6: Restricted Domains Full Setup

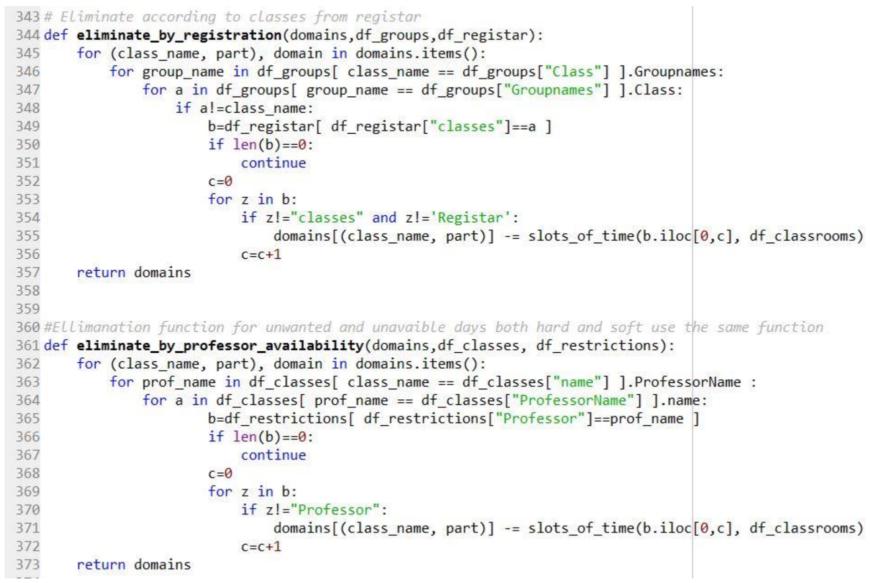
The domains are reduced one by one function each static reductions function. This function also initializes the domains first (sets them up)



```python
def eliminate_by_placement(domains, df_fixed_initial):
    df_fixed_initial.fillna(value='Fake M1')
    for (class_name, part), domain in domains.items():
        k=df_fixed_initial[df_fixed_initial['classes']==class_name]
        setdomains=set()
        if len(k)==0:
            continue
        c=0
        for z in k:
            if z!="classes" and z!='Registar':
                setdomains.add(slots_of_specificslot(k.iloc[0,c], df_classrooms))
                setdomains.discard(123123)
            c=c+1
        domains[(class_name, part)]=setdomains.copy()
    return domains


#Hard set constraint of deleting by the amount of students that can fit into a classroom,
#This sometimes is the reason for new classrooms needing to be opened up
def eliminate_by_capacity(domains, df_classrooms, df_classes):
    for (class_name, part), domain in domains.items():
         number_of_students = df_classes[df_classes["name"] == class_name].numberofstudents.item()
         # for each classroom that don't have enough capacity for number of students of a class
         for classroom_name in df_classrooms[number_of_students > df_classrooms["capacity"]].name:
             # eliminate slots of these classrooms
             domains[(class_name, part)] -= slots_of_classroom(classroom_name, df_classrooms)
    return domains
```

Code 3.7: Hard Set Elimination Part1

1) Eliminate_by_placement This code is discussed in chapter 6; this code gets a specific slot from the helper and sets the domain of the fixed classes to their set indexes. So this function reduces the 1-100 domain to just 1 specific domain and reduces the number of parts in the list of total parts (since that part now has to go to that one domain for certain)
2) Eliminate_by_capacity The capacity of a classroom is also a rigorous reduction that has to be done to the domains of classes, since a class has an expected number of students that have to occupy the classroom, some classrooms are too small for some classes, and the classes that are too large have their domains reduced, 20 by 20. So for example ee101.01 may have 90 expected students so 4 out of the 5 classrooms cannot house the classes so these classroom slots are reduced from the domain of the ee101.01 parts.

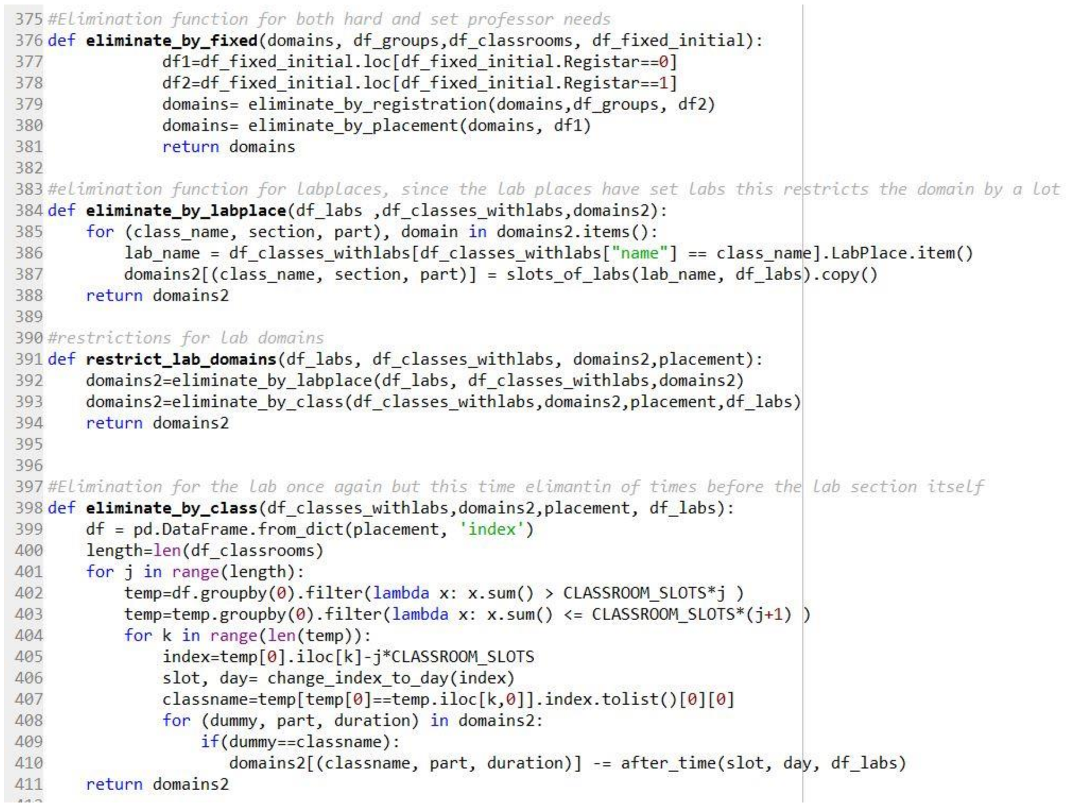


```python
# Eliminate according to classes from registar
def eliminate_by_registration(domains,df_groups,df_registar):
    for (class_name, part), domain in domains.items():
        for group_name in df_groups[ class_name == df_groups["Class"] ].Groupnames:
            for a in df_groups[ group_name == df_groups["Groupnames"] ].Class:
                if a!=class_name:
                    b=df_registar[ df_registar["classes"]==a ]
                    if len(b)==0:
                        continue
                    c=0
                    for z in b:
                        if z!="classes" and z!='Registar':
                            domains[(class_name, part)] -= slots_of_time(b.iloc[0,c], df_classrooms)
                        c=c+1
    return domains


#Ellimanation function for unwanted and unavaible days both hard and soft use the same function
def eliminate_by_professor_availability(domains,df_classes, df_restrictions):
    for (class_name, part), domain in domains.items():
        for prof_name in df_classes[ class_name == df_classes["name"] ].ProfessorName :
            for a in df_classes[ prof_name == df_classes["ProfessorName"] ].name:
                    b=df_restrictions[ df_restrictions["Professor"]==prof_name ]
                    if len(b)==0:
                        continue
                    c=0
                    for z in b:
                        if z!="Professor":
                            domains[(class_name, part)] -= slots_of_time(b.iloc[0,c], df_classrooms)
                        c=c+1
    return domains
```

Code 3.8: Hard Set Elimination Part2

3) Elimination_by_registration some classes are set outside the faculty by the register, the register has math and phys like classes that reduce the domains of classes in the same group as them, and it reduces by the same times as those register classes. So for example math101 occupying Monday 1 and 2 forces ee101 to not occupy Monday 1 and 2 sections.

4) Elimination_by_professor_availability this reduces the domains of classes by professors having hours that they are not able to give classes, a professor cannot give his or her class at a time that they cannot or do not want to give a class (simple enough) So I reduce the time slot from all classes from the domain of that class.



```python
#Elimination function for both hard and set professor needs
def eliminate_by_fixed(domains, df_groups,df_classrooms, df_fixed_initial):
            df1=df_fixed_initial.loc[df_fixed_initial.Registar==0]
            df2=df_fixed_initial.loc[df_fixed_initial.Registar==1]
            domains= eliminate_by_registration(domains,df_groups, df2)
            domains= eliminate_by_placement(domains, df1)
            return domains

#elimination function for labplaces, since the lab places have set labs this restricts the domain by a lot
def eliminate_by_labplace(df_labs ,df_classes_withlabs,domains2):
    for (class_name, section, part), domain in domains2.items():
        lab_name = df_classes_withlabs[df_classes_withlabs["name"] == class_name].LabPlace.item()
        domains2[(class_name, section, part)] = slots_of_labs(lab_name, df_labs).copy()
    return domains2

#restrictions for lab domains
def restrict_lab_domains(df_labs, df_classes_withlabs, domains2,placement):
    domains2=eliminate_by_labplace(df_labs, df_classes_withlabs,domains2)
    domains2=eliminate_by_class(df_classes_withlabs,domains2,placement,df_labs)
    return domains2


#Elimination for the lab once again but this time elimantin of times before the lab section itself
def eliminate_by_class(df_classes_withlabs,domains2,placement, df_labs):
    df = pd.DataFrame.from_dict(placement, 'index')
    length=len(df_classrooms)
    for j in range(length):
        temp=df.groupby(0).filter(lambda x: x.sum() > CLASSROOM_SLOTS*j )
        temp=temp.groupby(0).filter(lambda x: x.sum() <= CLASSROOM_SLOTS*(j+1) )
        for k in range(len(temp)):
            index=temp[0].iloc[k]-j*CLASSROOM_SLOTS
            slot, day= change_index_to_day(index)
            classname=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][0]
            for (dummy, part, duration) in domains2:
                if(dummy==classname):
                   domains2[(classname, part, duration)] -= after_time(slot, day, df_labs)
    return domains2
```

Code 3.9: Hard Set Elimination Part 3

5) Eliminate_by_fixed this is the umbrella function for elimination by register and elimination by fixed, this has to be done because the data for both are on the same sheet, the register column helps separate the data frame into two parts, since they behave differently in domain reduction this has to be done.

6) Eliminate_by_labplace reduces the domains into the lab place that each lab is placed into; this is a separate function than reduction by capacity since this function has to make the domains of the lab into just that lab instead of reducing the domains of the class by classrooms that don’t fit. So this is a fitting function instead of a reduction function.

7) Restrict_lab_domains is the umbrella function for the reduction of lab domains, this is like the setup of the reduction of domains of the classes but there are only 2 reduction functions here since the labs are much more rigorously set up (since the labs have specific labs they need to be in physically)

8) Eliminate_by_class eliminates the domains of the labs by their classes, since we want the classes to be before the lab sections the labs have to have the domains of their classes times (and the

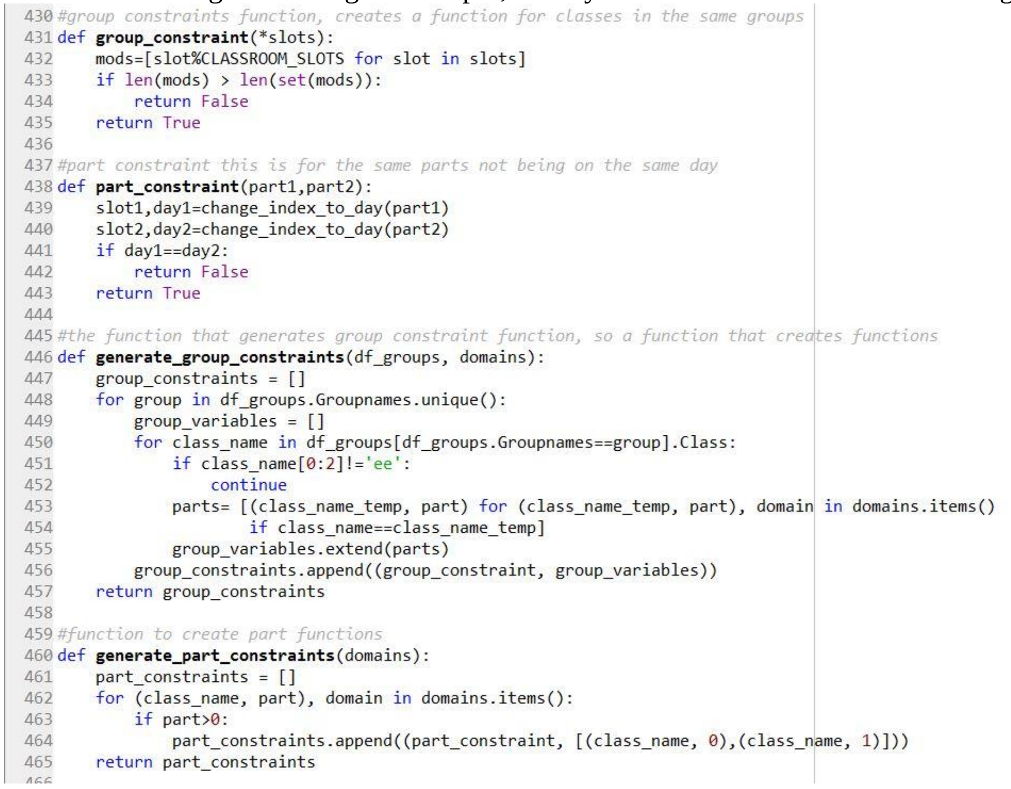
times before the class) Taken out of their domain. This is helped by also the way I set up the solver discussed in the next chapter.

CHAPTER 4
DYNAMIC CONSTRAINTS AND SOLVER (DYNAMIC)

Dynamic constraints were more difficult for me to write because I had never written something of this form beforehand. The dynamic constraints are different in that they change by the tree and the setup of the domains that enter the solver beforehand. The dynamic constraints but restrictions on the branches of the solver. When a class is placed down it effects the domains of some of the other classes, thus as the solution moves the next steps also change.

4.1 Constraint Generators

The constraint generators are functions that are creates functions, so these other functions are dynamic functions that change according to the input, so they churn out functions with the changing situations.

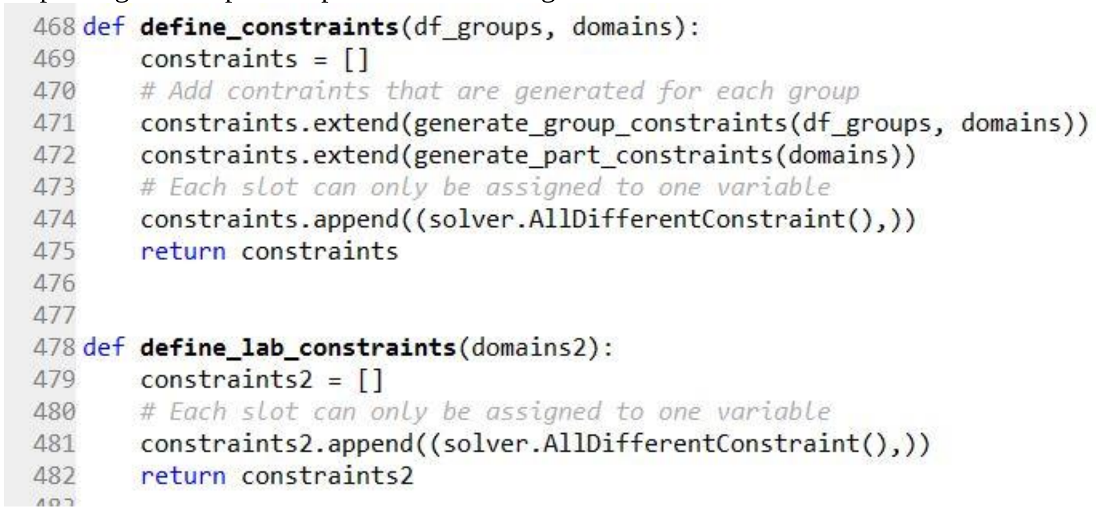


```python
#group constraints function, creates a function for classes in the same groups
def group_constraint(*slots):
    mods=[slot%CLASSROOM_SLOTS for slot in slots]
    if len(mods) > len(set(mods)):
        return False
    return True

#part constraint this is for the same parts not being on the same day
def part_constraint(part1,part2):
    slot1,day1=change_index_to_day(part1)
    slot2,day2=change_index_to_day(part2)
    if day1==day2:
        return False
    return True

#the function that generates group constraint function, so a function that creates functions
def generate_group_constraints(df_groups, domains):
    group_constraints = []
    for group in df_groups.Groupnames.unique():
        group_variables = []
        for class_name in df_groups[df_groups.Groupnames==group].Class:
            if class_name[0:2]!='ee':
                continue
            parts= [(class_name_temp, part) for (class_name_temp, part), domain in domains.items()
                    if class_name==class_name_temp]
            group_variables.extend(parts)
        group_constraints.append((group_constraint, group_variables))
    return group_constraints

#function to create part functions
def generate_part_constraints(domains):
    part_constraints = []
    for (class_name, part), domain in domains.items():
        if part>0:
            part_constraints.append((part_constraint, [(class_name, 0),(class_name, 1)]))
    return part_constraints
```

Code 4.1: Constraint Generators and Their Generated Functions

There are 3 dynamic constraints for the classes; the first is the groups constraint. Classes within the same groups cannot be on the same time as the other classes in that group. So the generate group constrains function finds which classes are in the same groups and feeds those 2 classes into the groups constraint function that does not allow the two classes to have the mods (which means being on the same time but not certainly the same classroom) So each pair of classes have this constraint but between each other and the group constrain generator feeds the function with these changing two classes by two classes.

The parts constraint function meanwhile has a different constraint. The parts cannot be on the same day (since we do not want more than 2 hours of the same class on the same day) Thus we use another helper to get these pairs of parts constrained against one another.



```
def define_constraints(df_groups, domains):
    constraints = []
    # Add contraints that are generated for each group
    constraints.extend(generate_group_constraints(df_groups, domains))
    constraints.extend(generate_part_constraints(domains))
    # Each slot can only be assigned to one variable
    constraints.append((solver.AllDifferentConstraint(),))
    return constraints


def define_lab_constraints(domains2):
    constraints2 = []
    # Each slot can only be assigned to one variable
    constraints2.append((solver.AllDifferentConstraint(),))
    return constraints2
```

Code 4.2: Their Generated Functions

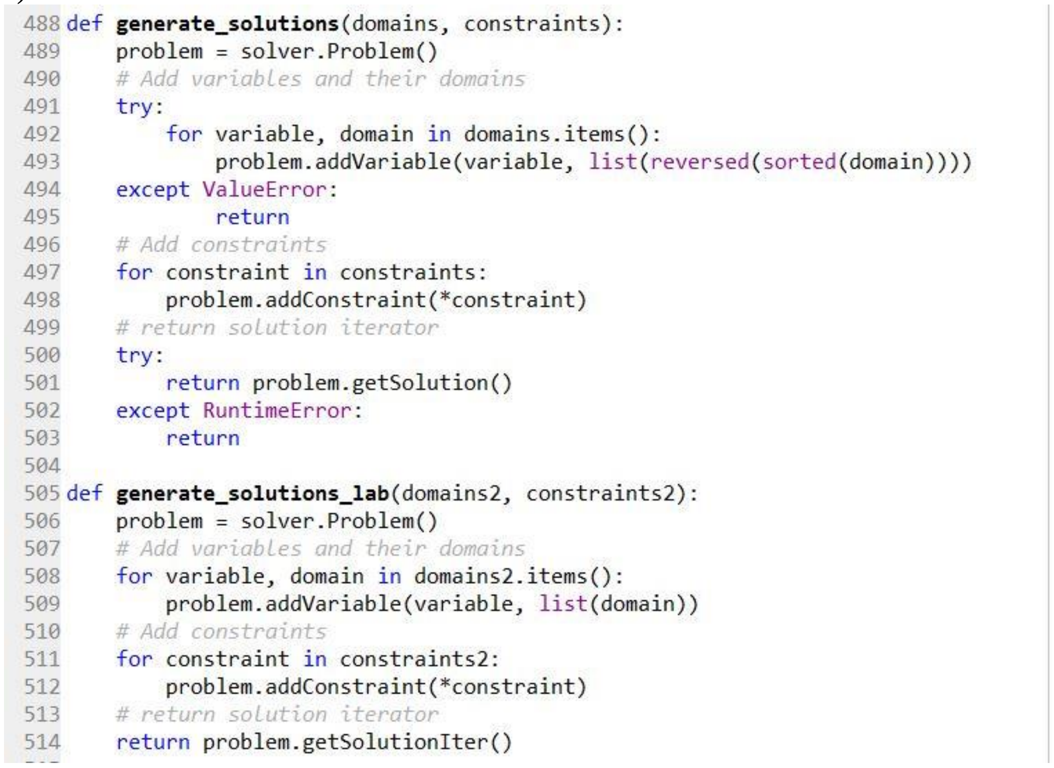
The last constraint for the classes is a little bit more straightforward as the solver library has a function for it (seeing as it is something that could be used for a lot of situations) the different classes cannot occupy the same domain so they have the AllDifferentConstraint applied to them. So the class’s domains thus have 3 constraints generated for their domains.

Meanwhile the labs have 1 constraint generated for them which is AllDifferentConstraint. This is much easier because the only other dynamic constraint of them being after their lab classes is taken care of by placing the lab classes first and turning the dynamic constraint into a hard-set one. (If there were many

many more labs this would not have worked most likely and we would have had to solve the classes considering the labs at the same time)

4.2 Solver and Constraint Lifters

Now that we have our constraints and our domains the only thing left is to feed them into the solver. The solver was a third party code that I found by the library constraint.py the logic is simple enough though for what I am using the library for and it is that there is a tree propagation that uses the domains (sets) and checks them against the constraints at each step and continues trying to propagate down the tree until it finds a legal solution. Since I set the code up as taking all soft constraints as hard the first pass through we know that all legal solutions are equal in their worth. So I am awarded the luxury of just finding ONE solution by getsolution instead of all solutions by getsolutions which when tried took a LOT of time to reach the end (actually it didn't end even after about 12 hours I will talk about this more in chapter 6)



```
def generate_solutions(domains, constraints):
    problem = solver.Problem()
    # Add variables and their domains
    try:
        for variable, domain in domains.items():
            problem.addVariable(variable, list(reversed(sorted(domain))))
    except ValueError:
            return
    # Add constraints
    for constraint in constraints:
        problem.addConstraint(*constraint)
    # return solution iterator
    try:
        return problem.getSolution()
    except RuntimeError:
        return

def generate_solutions_lab(domains2, constraints2):
    problem = solver.Problem()
    # Add variables and their domains
    for variable, domain in domains2.items():
        problem.addVariable(variable, list(domain))
    # Add constraints
    for constraint in constraints2:
        problem.addConstraint(*constraint)
    # return solution iterator
    return problem.getSolutionIter()
```

Code 4.3: Solver for the labs and Classes

The setup for the solver is that it takes in constraints and domains and outputs a solution or solutions depending on the function used. So I have done everything needed for a solution up to this point.

Side Note: One thing I would like to point out is that the solver is also stacked in two ways the classes are stacked in a reversed manner the labs are stacked in a normal manner. This is because the solver is situated to solve the branches by going in a top to bottom solution. So by stacking the domains in these two different ways we make it so that the labs are stacked from Friday to Monday and the classes Monday to Friday so that the labs are at the end of the week and the classes at the beginning.

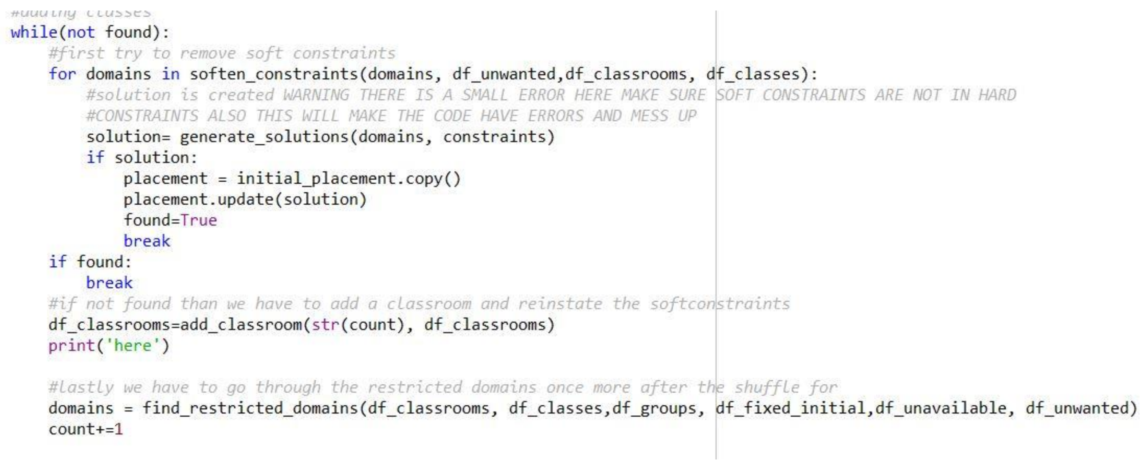
There is one more thing I did at the very beginning for the solvers sake and it is that with added classes the added classes is pushed to fill up first with the bottom to top approach that we have here, so I reshuffle the ordering of the classrooms from least capacity to most (with the added classes having 10,000 capacity so it is almost certainly at the end all the time) This forces the other classes to try and get filled up first before trying out the new classroom thus only the class having a problem fitting in is pushed to the new class.

Both of these 2 adjustments are small yet critical in the solution of the problem.

Now moving on:

There is the question of what happens if there is no solution that can be achieved? The answer is we have to reduce the number of constraints hard-set or dynamic on the domains or else there is no way that we would be able to get a legal solution. Since the dynamic constraints are hard set there is no way to let them go the only constraints we have the luxury of letting go of are the soft constraints of professors or the notion of not needing another classroom. Needing another classroom was decided as a last resort by Prof. Faik and me so we continue by reducing the soft constraints of the professors first.

But there is the problem of which direction we should go with reduction and again decided at the beginning we continue at this problem in a fair manner by reducing the soft constraints of professors beginning by the professors who have inputted more constraints than the ones who have few of them.

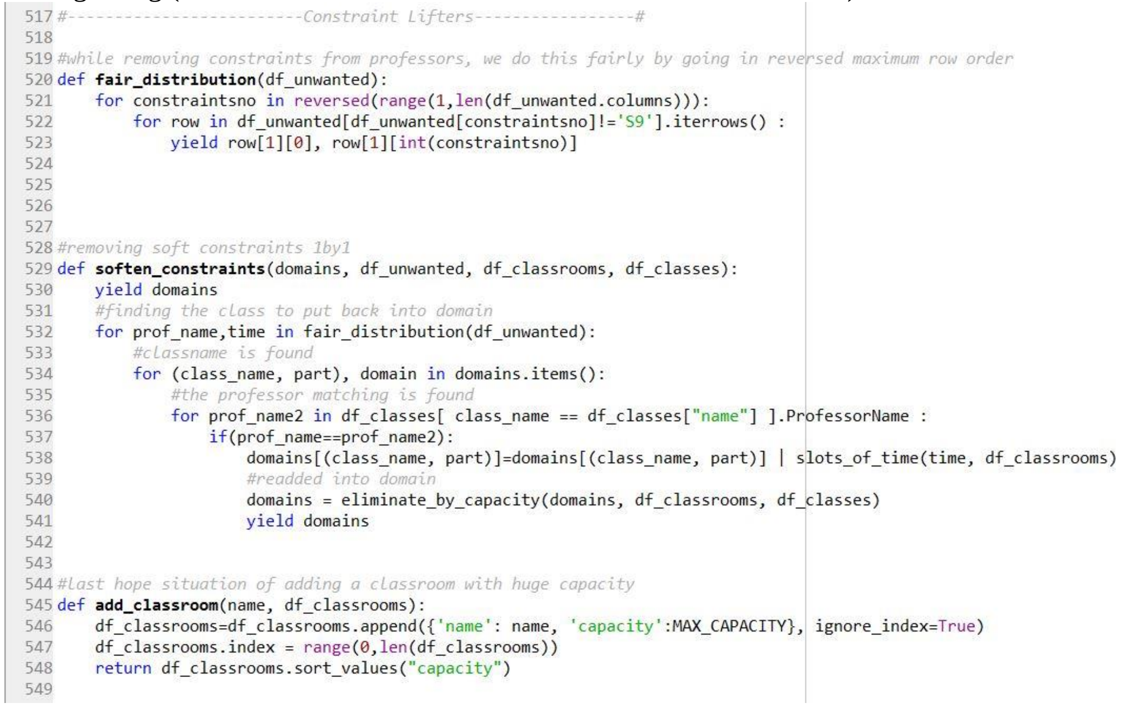


```
#adding classes
while(not found):
    #first try to remove soft constraints
    for domains in soften_constraints(domains, df_unwanted,df_classrooms, df_classes):
        #solution is created WARNING THERE IS A SMALL ERROR HERE MAKE SURE SOFT CONSTRAINTS ARE NOT IN HARD
        #CONSTRAINTS ALSO THIS WILL MAKE THE CODE HAVE ERRORS AND MESS UP
        solution= generate_solutions(domains, constraints)
        if solution:
            placement = initial_placement.copy()
            placement.update(solution)
            found=True
            break
    if found:
        break
    #if not found than we have to add a classroom and reinstate the softconstraints
    df_classrooms=add_classroom(str(count), df_classrooms)
    print('here')

    #lastly we have to go through the restricted domains once more after the shuffle for
    domains = find_restricted_domains(df_classrooms, df_classes,df_groups, df_fixed_initial,df_unavailable, df_unwanted)
    count+=1
```

Code 4.4: Constraint Lifters Loop

So with the solution iterator we check for a solution first and if there are empty domains for some of the classes an error is given by the solver, I put exceptions into 2 points in the solver which push us back into this loop but by using the soften_constraints function first. I had to use yield points to continue with the softened constraints where the trees left off saving time and data.

After all softened professors constraints are used up in the loop if we still have no found solution (which breaks out) We move on to add_classroom function and start the loop all over with the domains reset to the beginning (with the softened constraints now reinstated as hard ones)



```
#-------------------------Constraint Lifters-----------------#

#while removing constraints from professors, we do this fairly by going in reversed maximum row order
def fair_distribution(df_unwanted):
    for constraintsno in reversed(range(1,len(df_unwanted.columns))):
        for row in df_unwanted[df_unwanted[constraintsno]!='S9'].iterrows() :
            yield row[1][0], row[1][int(constraintsno)]




#removing soft constraints 1by1
def soften_constraints(domains, df_unwanted, df_classrooms, df_classes):
    yield domains
    #finding the class to put back into domain
    for prof_name,time in fair_distribution(df_unwanted):
        #classname is found
        for (class_name, part), domain in domains.items():
            #the professor matching is found
            for prof_name2 in df_classes[ class_name == df_classes["name"] ].ProfessorName :
                if(prof_name==prof_name2):
                    domains[(class_name, part)]=domains[(class_name, part)] | slots_of_time(time, df_classrooms)
                    #readded into domain
                    domains = eliminate_by_capacity(domains, df_classrooms, df_classes)
                    yield domains


#last hope situation of adding a classroom with huge capacity
def add_classroom(name, df_classrooms):
    df_classrooms=df_classrooms.append({'name': name, 'capacity':MAX_CAPACITY}, ignore_index=True)
    df_classrooms.index = range(0,len(df_classrooms))
    return df_classrooms.sort_values("capacity")

```

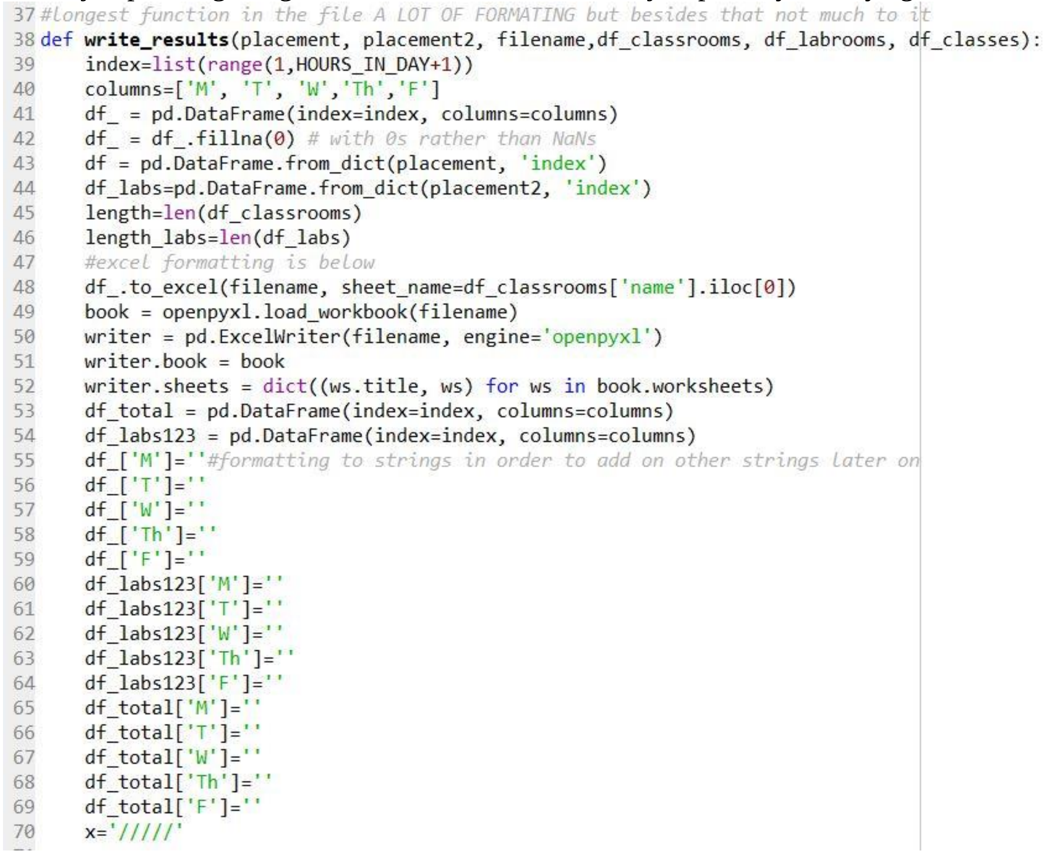
Code 4.5: Constraint Lifters

Yield function was hard to get a hang of on how to use it and took a while to understand how to use it correctly by many trial and error nights

CHAPTER 5
WRITING FUNCTION

So after all this we now have domains that have only one index for each class and each lab so this is our solution but humans cannot understand the indexes easily so we need to morph those into a readable format.

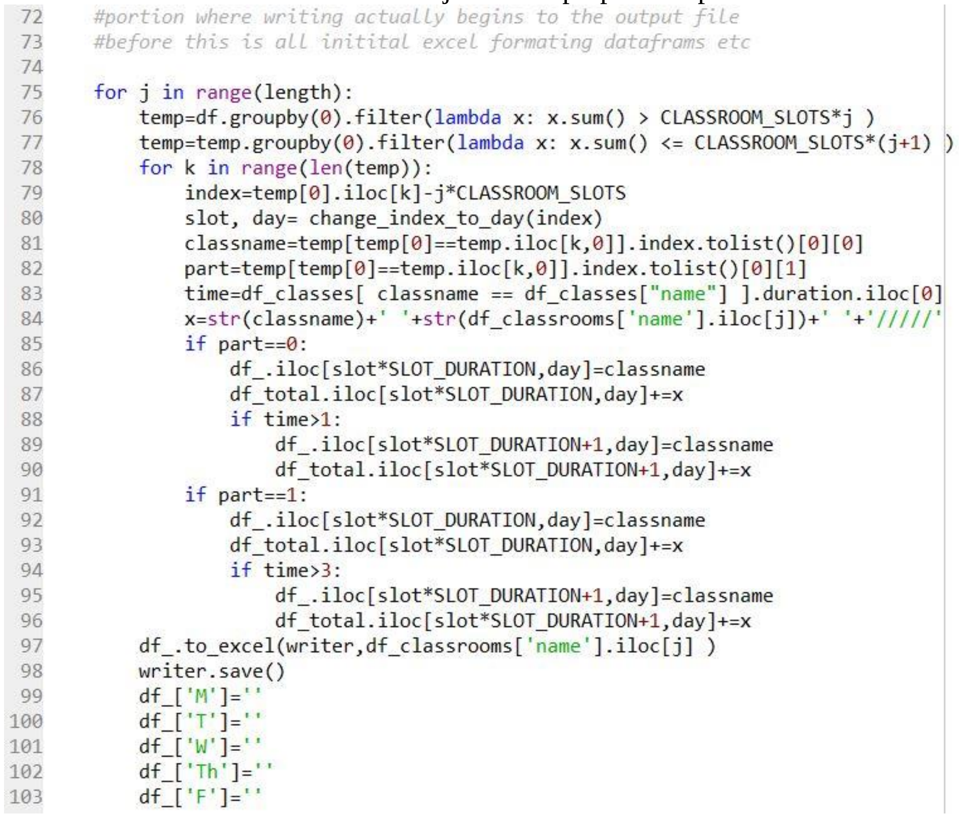
The writer function was actually the hardest function to write because of the pure indexes and hoops I had to jump through to get a correct readout. So it was just pure try and try again.



```python
#longest function in the file A LOT OF FORMATING but besides that not much to it
def write_results(placement, placement2, filename,df_classrooms, df_labrooms, df_classes):
    index=list(range(1,HOURS_IN_DAY+1))
    columns=['M', 'T', 'W','Th','F']
    df_ = pd.DataFrame(index=index, columns=columns)
    df_ = df_.fillna(0) # with 0s rather than NaNs
    df = pd.DataFrame.from_dict(placement, 'index')
    df_labs=pd.DataFrame.from_dict(placement2, 'index')
    length=len(df_classrooms)
    length_labs=len(df_labs)
    #excel formatting is below
    df_.to_excel(filename, sheet_name=df_classrooms['name'].iloc[0])
    book = openpyxl.load_workbook(filename)
    writer = pd.ExcelWriter(filename, engine='openpyxl')
    writer.book = book
    writer.sheets = dict((ws.title, ws) for ws in book.worksheets)
    df_total = pd.DataFrame(index=index, columns=columns)
    df_labs123 = pd.DataFrame(index=index, columns=columns)
    df_['M']=''#formatting to strings in order to add on other strings later on
    df_['T']=''
    df_['W']=''
    df_['Th']=''
    df_['F']=''
    df_labs123['M']=''
    df_labs123['T']=''
    df_labs123['W']=''
    df_labs123['Th']=''
    df_labs123['F']=''
    df_total['M']=''
    df_total['T']=''
    df_total['W']=''
    df_total['Th']=''
    df_total['F']=''
    x='/////'
```

Code 5.1: Writing Function 1

This portion of the writing function is opening a bunch of data frames that will be used for excel formatting. The setup data frames need to be initialized to empty data frames. So that when they are first pressed into the excel. All of the code here is just excel preps and opens etc.

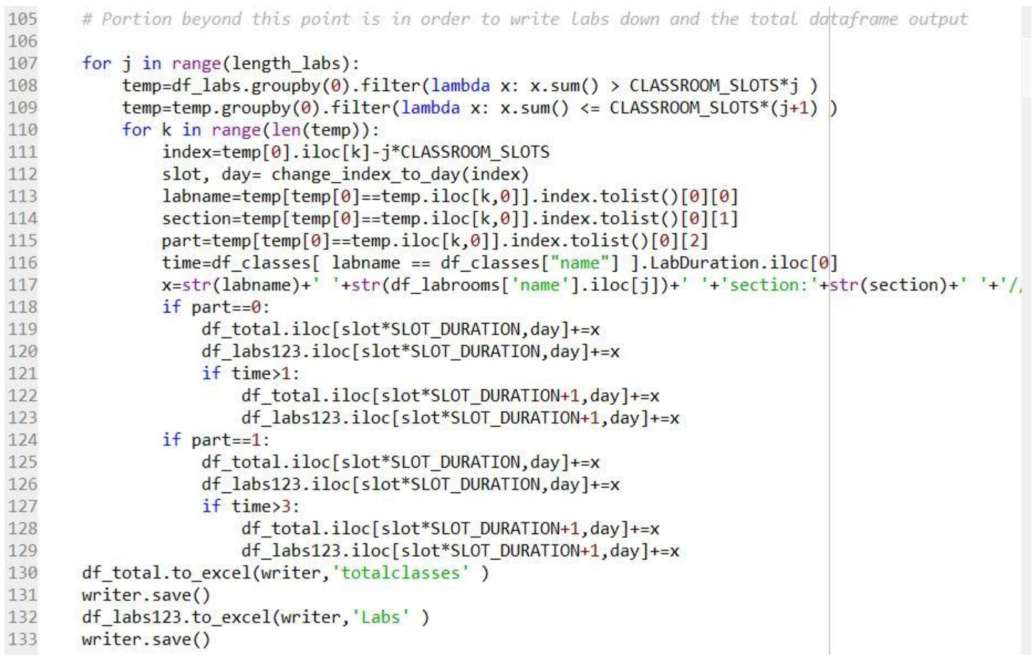


```
#portion where writing actually begins to the output file
#before this is all initital excel formating dataframs etc

for j in range(length):
    temp=df.groupby(0).filter(lambda x: x.sum() > CLASSROOM_SLOTS*j )
    temp=temp.groupby(0).filter(lambda x: x.sum() <= CLASSROOM_SLOTS*(j+1) )
    for k in range(len(temp)):
        index=temp[0].iloc[k]-j*CLASSROOM_SLOTS
        slot, day= change_index_to_day(index)
        classname=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][0]
        part=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][1]
        time=df_classes[ classname == df_classes["name"] ].duration.iloc[0]
        x=str(classname)+' '+str(df_classrooms['name'].iloc[j])+' '+'/////'
        if part==0:
            df_.iloc[slot*SLOT_DURATION,day]=classname
            df_total.iloc[slot*SLOT_DURATION,day]+=x
            if time>1:
                df_.iloc[slot*SLOT_DURATION+1,day]=classname
                df_total.iloc[slot*SLOT_DURATION+1,day]+=x
        if part==1:
            df_.iloc[slot*SLOT_DURATION,day]=classname
            df_total.iloc[slot*SLOT_DURATION,day]+=x
            if time>3:
                df_.iloc[slot*SLOT_DURATION+1,day]=classname
                df_total.iloc[slot*SLOT_DURATION+1,day]+=x
    df_.to_excel(writer,df_classrooms['name'].iloc[j] )
    writer.save()
    df_['M']=''
    df_['T']=''
    df_['W']=''
    df_['Th']=''
    df_['F']=''
```

Code 5.2: Writing Function 2

This portion of the code writes each class to separate sheets by separating the domains into the right classes into a temp set, the set is then written into the right classes.

Temp is separated by a filter function and the length of the temp is the amount of classes written into the temp data frame so that it can be saved into the writer after it exits the loop. Because of the way the slots are setup I needed to write the slots completely. This function is also discussed in chapter 6 since it is not a rigorous function setup.



```
# Portion beyond this point is in order to write labs down and the total dataframe output

for j in range(length_labs):
    temp=df_labs.groupby(0).filter(lambda x: x.sum() > CLASSROOM_SLOTS*j )
    temp=temp.groupby(0).filter(lambda x: x.sum() <= CLASSROOM_SLOTS*(j+1) )
    for k in range(len(temp)):
        index=temp[0].iloc[k]-j*CLASSROOM_SLOTS
        slot, day= change_index_to_day(index)
        labname=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][0]
        section=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][1]
        part=temp[temp[0]==temp.iloc[k,0]].index.tolist()[0][2]
        time=df_classes[ labname == df_classes["name"] ].LabDuration.iloc[0]
        x=str(labname)+' '+str(df_labrooms['name'].iloc[j])+' '+'section:'+str(section)+' '+'/,
        if part==0:
            df_total.iloc[slot*SLOT_DURATION,day]+=x
            df_labs123.iloc[slot*SLOT_DURATION,day]+=x
            if time>1:
                df_total.iloc[slot*SLOT_DURATION+1,day]+=x
                df_labs123.iloc[slot*SLOT_DURATION+1,day]+=x
        if part==1:
            df_total.iloc[slot*SLOT_DURATION,day]+=x
            df_labs123.iloc[slot*SLOT_DURATION,day]+=x
            if time>3:
                df_total.iloc[slot*SLOT_DURATION+1,day]+=x
                df_labs123.iloc[slot*SLOT_DURATION+1,day]+=x
df_total.to_excel(writer,'totalclasses' )
writer.save()
df_labs123.to_excel(writer,'Labs' )
writer.save()
```

Code 5.3: Writing Function 3

The next portion is the same but this is for the next two sheets, these two sheets have all labs and all the classes written together (df_labs123 for labs and df_total for everything together)

This portion is just as the other portion but does not have multiple rolls through it only goes in once.

CHAPTER 6
CONCLUSION

6.1 Results and Output

The results were outstanding most of the time.

The normal output (normal being no classes added) took about 10-15secs to output

The output with a class added takes about 30-40 secs worse but not worse by far.

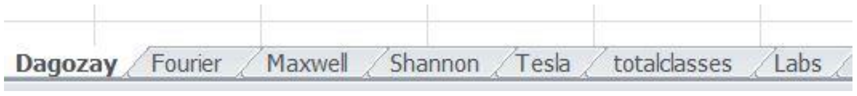
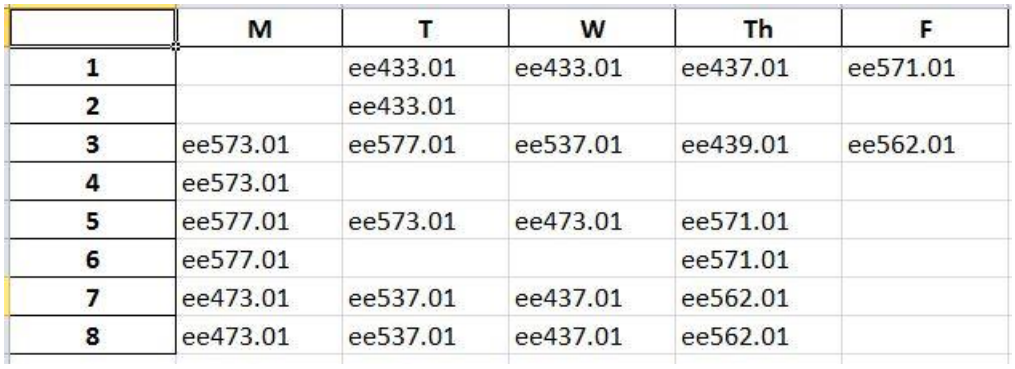


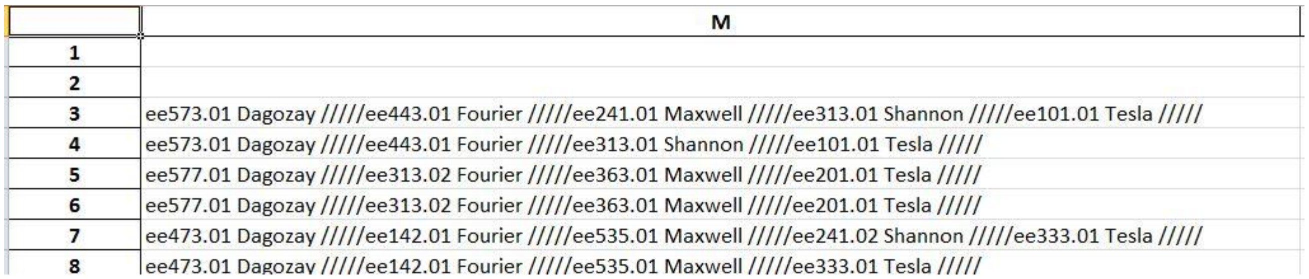
Figure 6.1: The Output Excel Sheets

This excel is the output of the code, if a class had been needed for some reason that class would also have been here.



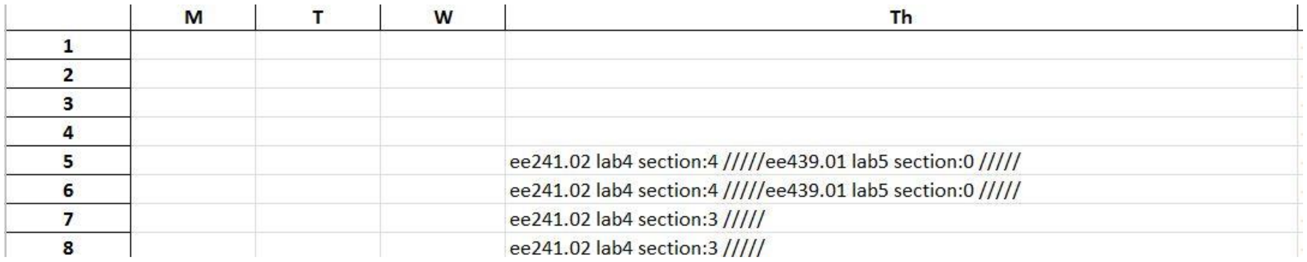
	M	T	W	Th	F
1		ee433.01	ee433.01	ee437.01	ee571.01
2		ee433.01			
3	ee573.01	ee577.01	ee537.01	ee439.01	ee562.01
4	ee573.01				
5	ee577.01	ee573.01	ee473.01	ee571.01	
6	ee577.01			ee571.01	
7	ee473.01	ee537.01	ee437.01	ee562.01	
8	ee473.01	ee537.01	ee437.01	ee562.01	

Figure 6.2: One of the output classes in this case Dagozay



	M
1	
2	
3	ee573.01 Dagozay /////ee443.01 Fourier /////ee241.01 Maxwell /////ee313.01 Shannon /////ee101.01 Tesla /////
4	ee573.01 Dagozay /////ee443.01 Fourier /////ee313.01 Shannon /////ee101.01 Tesla /////
5	ee577.01 Dagozay /////ee313.02 Fourier /////ee363.01 Maxwell /////ee201.01 Tesla /////
6	ee577.01 Dagozay /////ee313.02 Fourier /////ee363.01 Maxwell /////ee201.01 Tesla /////
7	ee473.01 Dagozay /////ee142.01 Fourier /////ee535.01 Maxwell /////ee241.02 Shannon /////ee333.01 Tesla /////
8	ee473.01 Dagozay /////ee142.01 Fourier /////ee535.01 Maxwell /////ee333.01 Tesla /////

Figure 6.3: The output of a day of all classes together



	M	T	W	Th
1				
2				
3				
4				
5				ee241.02 lab4 section:4 /////ee439.01 lab5 section:0 /////
6				ee241.02 lab4 section:4 /////ee439.01 lab5 section:0 /////
7				ee241.02 lab4 section:3 /////
8				ee241.02 lab4 section:3 /////

Figure 6.4: The output of a day of the labs

Thus I have all the constraints that come to mind taken care of and a legal situation for the classes without needing an extra classroom from outside of the electrical engineering building!

6.2 Discussion and Future Work

There are several things I would like to discuss here mixed together with things I plan to do in the future for the code.

1) Prof. Faik wished me to add a mean distribution constraint over the professors having less classes distributed over days but I was not able to do this with the time limits and it being a more complex constraint than the ones I already did. This is the first thing I plan on adding to the code in future work with more time.
2) The next thing that I would like to discuss that is weighing down on the health of the code is human error. My code is very fast and I am very proud of the outcome. It finds a correct

solution to most situations, my saying most is because it is weak against human error. Let me give a list of human errors I myself committed while testing the code extensively. A) Putting down a professor constraint as both hard and soft can force the code to go into an infinite loop if it so happens to stumble on that constraint to remove. As it tries to remove the soft constraint the hard constraint forces it back on and the code goes into an infinite loop without a solution. B) Putting anything about a fixed class wrong is a big mistake and this function is very fragile. Putting in a wrong capacity or putting down 3 hours with the class having duration of 2 both can force the code into a loop. I plan on making the code and especially this function sturdier to human errors by putting error outputs and checks along the way for loops that may occur.

3) Some situations can occur where the code is forced very deep into the tree for seemingly no reason at all (although very very rare maybe 1 setup out of 100) This is something I need to research more it may be a very complex interaction of domains forcing some constraints to battle it out and having a draw. I may need to put some type of precedence of hard constraints for situations as this one with a solution of lifting some soft constraints of getting a new class.
4) Next I wish to make the slots more flexible with the setup in this way the write function breaks with the slots duration changed. I need to make it so that the write function is not written just for the situation of slot_duration=2, so the write function needs to be generated as a function of the slot_duration. This means making the concept of slot_duration more flexible.
5) Lastly because of time constraints I was not able to integrate problem sections into the framework even though it is in the input, but it is simple in this situation since there are almost no restrictions to problem sections other than it being after the class it is appointed to. This can be easily added with time.

I plan to fix these problems in version 1.1 with this code being version 1.0.

I plan to make a user interface in version 1.2 that is friendly to anyone that wants to use it and making the system much more flexible for other schools by getting feedback from friends from other universities.

I plan to make the system work for high schools and elementary in version 1.3 with a highly flexible code.

6.3 Social, Environmental and Economic Impact

This project has a social impact on many professors at Bogazici University by making their lives much easier in a simple yet time consuming situation.

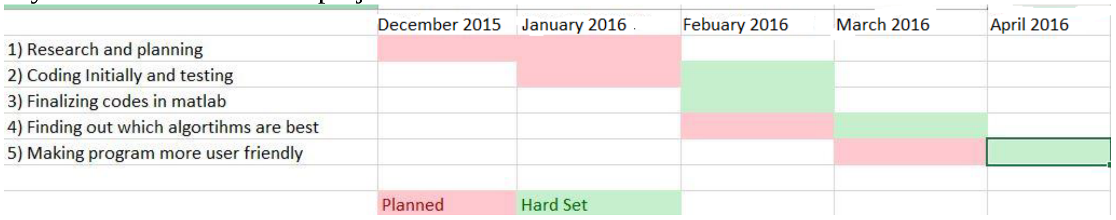
It does not have much of an environmental impact but could possibly have an economical one with time saved and maybe in the future versions of the code with errors fixed and a friendlier user interface this could be sold to other schools.

6.4 Cost Analysis and Time Schedule

I have increased the working hour's fee because the task turned out to be much more difficult than originally thought.

Product	Number	Cost
Laptop	1	2500 TL
Working hours	200	50000TL
Matlab Student Suite License Fee	1	222,5 TL
Total		52722,5 TL

My time Schedule for this project was



	December 2015	January 2016	Febuary 2016	March 2016	April 2016
1) Research and planning	🟥	🟥			
2) Coding Initially and testing		🟥	🟩		
3) Finalizing codes in matlab			🟩		
4) Finding out which algortihms are best			🟥	🟩	
5) Making program more user friendly				🟥	🟩
	🟥 Planned	🟩 Hard Set			

BIBLIOGRAPHY

[1] 10 Minutes to pandas¶. (n.d.). Retrieved October 1st 2015, from http://pandas.pydata.org/pandas-docs/stable/10min.html
[2] Python-constraint. (n.d.). Retrieved October 1st 2015, from https://labix.org/python-constraint
[3] Welcome to Python.org. (n.d.). Retrieved October 1st 2015, from https://www.python.org/
[4] Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning, Ian Miguel - slides.
[5] Solution reuse in dynamic constraint satisfaction problems, Thomas SchiexJ.