Artificial Intelligence

Algorithms and Applications with Python

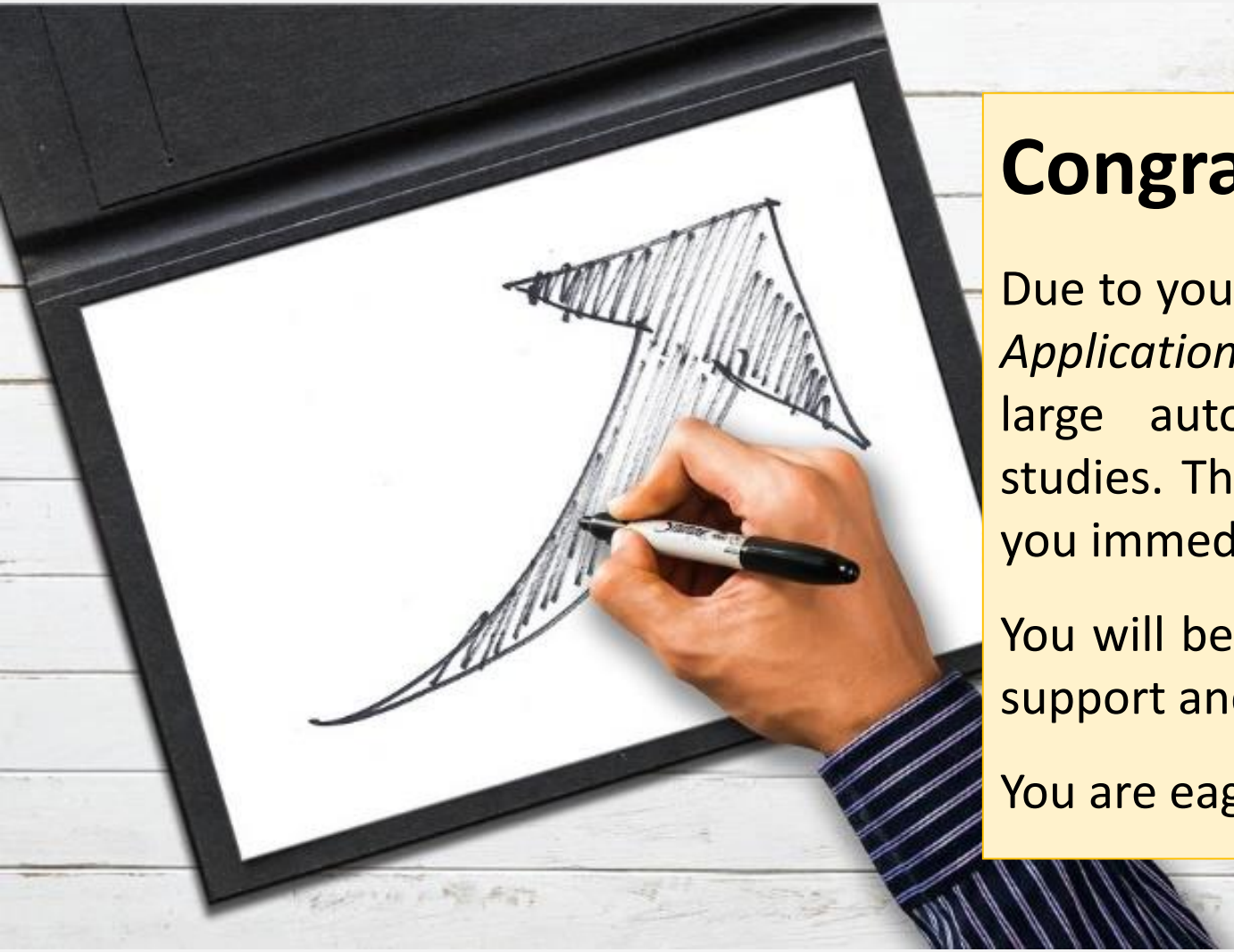
Lectorial

Dr. Dominik Jung
dominik.jung42@gmail.com

</> course material: ↗ https://github.com/dominikjung42

Image source: ↗ Pixabay (2019) / ↗ CC0

Scenario



Congratulations!

Due to your excellent grade in lecture "*AI Algorithms and Applications with Python*", you received a job offer at a large automobile manufacturer directly after your studies. The high salary and your gentle boss convinced you immediately to start.

You will be employed as an AI specialist and will have to support and advise various departments on issues.

You are eagerly awaiting your first job assignment!

Agenda

1.1 Basics and Repetition

1.2 Staff planning with genetic algorithms

Note

- This is a lectorial: I will explain/repeat the most important concepts and then you try to solve the programming task by your own
- You are explicitly encouraged to solve this task in groups. And I will help you and give suggestions. However, there is no perfect solution, you will get a possible solution.
- If the task is too hard for you at the moment – relaxe ☺. Just look at the task again at a later point in the course.

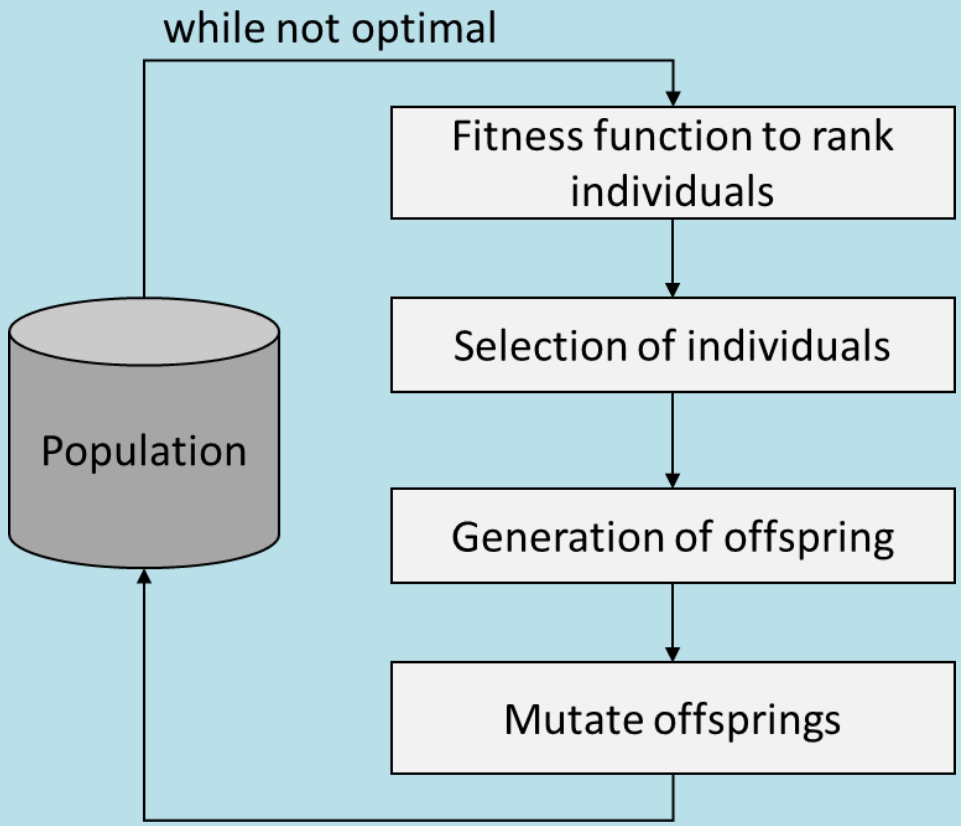
Recapitulation – Genetic Algorithms

Algorithm: Genetic Algorithm

Inputs: population, fitness_func

new_population <- empty set
Repeat
  For i = 1 to *SIZE(population)* ***do***
    X ← RANDOM-SELECTION(population, fitness_func)
    Y ← RANDOM-SELECTION(population, fitness_func)
    Child ← REPRODUCE(x,y)
    If(random prob) then *child ← MUTATE(child)*
      Add child to new_population
  Population ← new_population
Until *some individuals optimal or time elapse*

Return *the best individuals in pop, according fitness_func*



Read an Excel files into a pandas `DataFrame`

`read_excel()`

```
pandas.read_excel(open(), sheet_name)
```

Parameters

`open`(path, „rb“)	Input data as file object, supporting *xls*, *xlsx*, *xlsm*, *xlsb*, *odf*, *ods* and *odt* file extensions. In the file object you have to specify the modus of file acces (see lecture 3 - Reading Files).
`sheet_name`	Strings are used for sheet names. Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets
`header`	Row (0-indexed) to use for the column labels of the parsed DataFrame

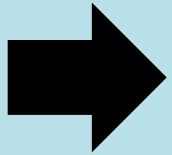
↗ https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Programming Concept: Masking

- Masking is a popular programming concept in data science
- Let us say, we want to calculate the sum of elements bigger than 10 and smaller than 99

```
nums = numpy.array([4, 100, 20, 1, 30, 900])
```

```
total = 0
for num in nums:
    if ((num>10) & (num<99)):
        total += num
```



```
mask  = ((nums > 10) & (nums<99))
nums[mask].sum()
```

Some other Practical Functions that are Good to Know

Generate random Integer

```
numpy.random.randint(low = 0, high = 10)
```

Generate random Sample

```
numpy.random.choice(range(0,50), size=25, replace=False)
```

Sum dataframes

```
df.sum(axis=1)
```

Apply mask on Pandas dataframe

```
df.mask(df > 0)
df.mask(dif < 0, 0)  # replace negative values with 0
```

Pandas DataFrames Indexing

```
import pandas as pd
speed = [290, 330, 345]
car = [718, 911, 918]
df = pd.DataFrame()
df["CAR"] = car
df["SPEED"] = speed
```

INDEX	CAR	SPEED
0	718	290
1	911	330
2	918	345

Nam	Type	Size	
car	list	3	[718, 911, 918]
df	DataFrame	(3, 2)	Column names: CAR, SPEED
speed	list	3	[290, 330, 345]

`df["SPEED"]`

290
330
345

`df[0:2]`

CAR	SPEED
718	290
911	330

Pandas offers many ways of multi-axis indexing. However, the most popular one is .loc-indexing. Check out the package documention for other indexing methods like iloc.

`df.loc[1:]` ← startindex ← stop

INDEX	CAR	SPEED
1	911	330
2	918	345

`df.loc[2]`

INDEX	CAR	SPEED
2	918	345

`df.loc[:, 'CAR']`

CAR
718
911
918

`df.loc[1:2, 'CAR':'SPEED']`

CAR	SPEED
911	330
918	345

Classroom Case

Case

Your first assignment is in the *Human Resources Department*. You will be needed there because they have difficulties to create a shift plan in the part handling factory. The problem is that whatever they do they can find an good plan. Immediately you remember the lecture and the topic *Genetic Algorithms*.

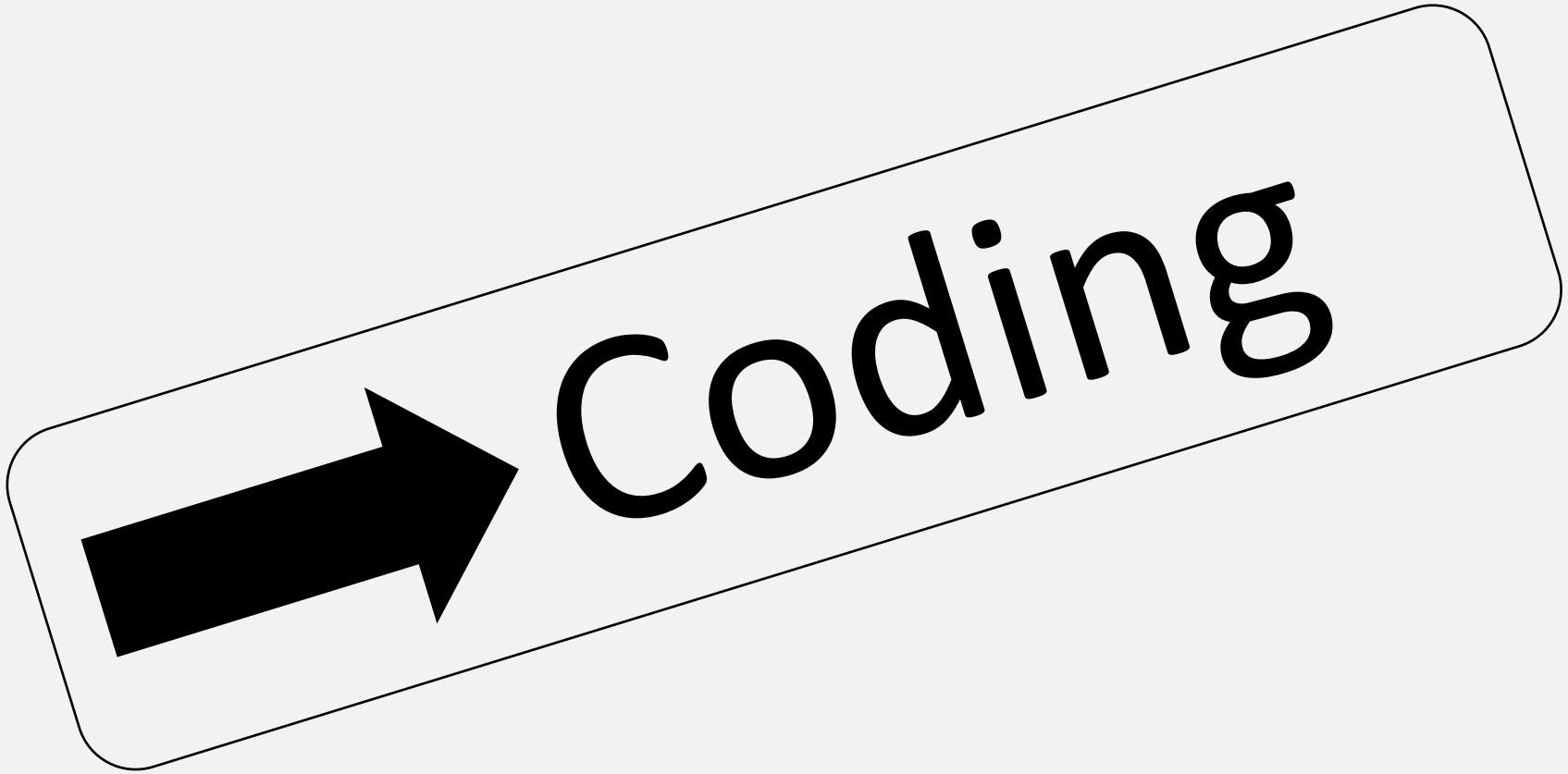
Please create a shift plan using genetic algorithms. Develop a solution in Python, so that you can help them with future planning easily.

Conditions:

- To facilitate modelling, lets say we do not work overnight (23:59 to 0:00 is closed)
- All workers have flexible start times, and can work up to the maximum hours per day of their contract
- Use the database extract `db_staffplanning.xlsx`, which contains further information about your staff and the needed capacities

You can find the solution of each lectorial online in the git repository if you need some starting help!

Further questions?



1 Before we start, we take a look at the data

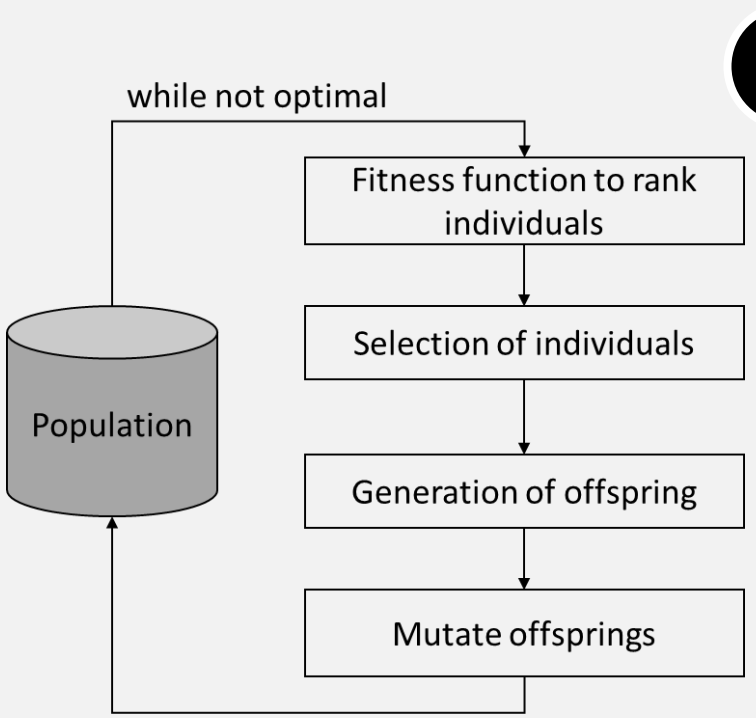
DAY	ID	START	CONTRACT
Monday	1	10	10
Monday	2	10	10
Monday	3	10	10
Monday	4	10	10
Monday	5	10	10
Monday	6	10	10
Monday	7	14	10
Monday	8	16	10
Monday	9	18	5
Monday	10	20	5
Tuesday	1	0	10
Tuesday	2	0	10
Tuesday	3	6	10
Tuesday	4	6	10
Tuesday	5	10	10
Tuesday	6	10	10
Tuesday	7	14	10
Tuesday	8	16	10
Tuesday	9	18	5
Tuesday	10	20	5
Wednesday	1	0	10
Wednesday	2	0	10

- The table "`staff`" is a list of the different workers, their preferred starting time and how much they are allowed to work (some have only a 50% contract)

DAY	0	1	2	3	4	5	6	7
Monday	0	0	0	0	1	1	2	4
Tuesday	0	0	0	0	1	1	2	4
Wednesday	0	0	0	0	1	1	2	4
Thursday	0	0	0	0	1	1	2	4
Friday	0	0	0	0	1	1	2	4

- The table "`workload`" is a timetable with the required manpower over the time e.g. on Monday at 4 o'clock one worker is needed

1 Then we make a Sketch of our Programming Concept



1

staff_list

Oli	7:00	10 h
Elon	16:00	4 h
Mary	12:00	10 h
Carlos	10:00	10 h
...	...	...

Compute availabilities

1:00-2:00
Yes
No
Yes
...

$\sum$

1:00-2:00
2

2

planning_table

	1:00-2:00	1:00-2:00
Mon	1	2
Thu	3	2
Wed	2	4

Fitness

workload

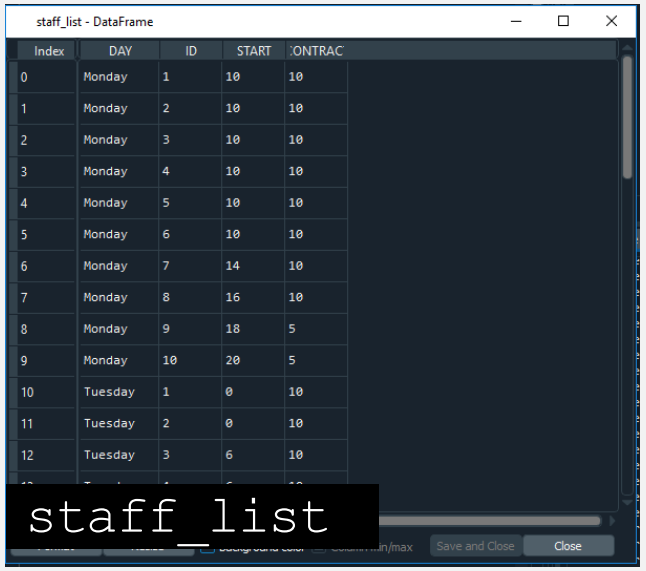
	1:00-2:00	1:00-2:00
Mon	1 ✓	2 ✓
Thu	2 ✓	4 ✗
Wed	2 ✓	4 ✓

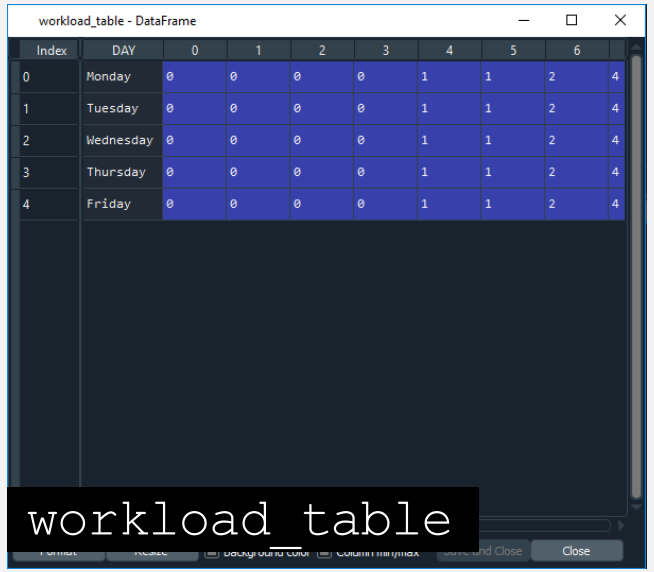
1. 1 Now we can start: Let us load the Data into our IDE

- In a first step we load the existing data into our IDE for further exploration

```
staff_list = pandas.read_excel(open("db_staffplanning.xlsx", "rb"),
                               sheet_name="staff") # this kind of data is termed 'long' data

workload_table = pandas.read_excel(open("db_staffplanning.xlsx", "rb"),
                               sheet_name="workload") # this kind of data is termed 'wide' data
```

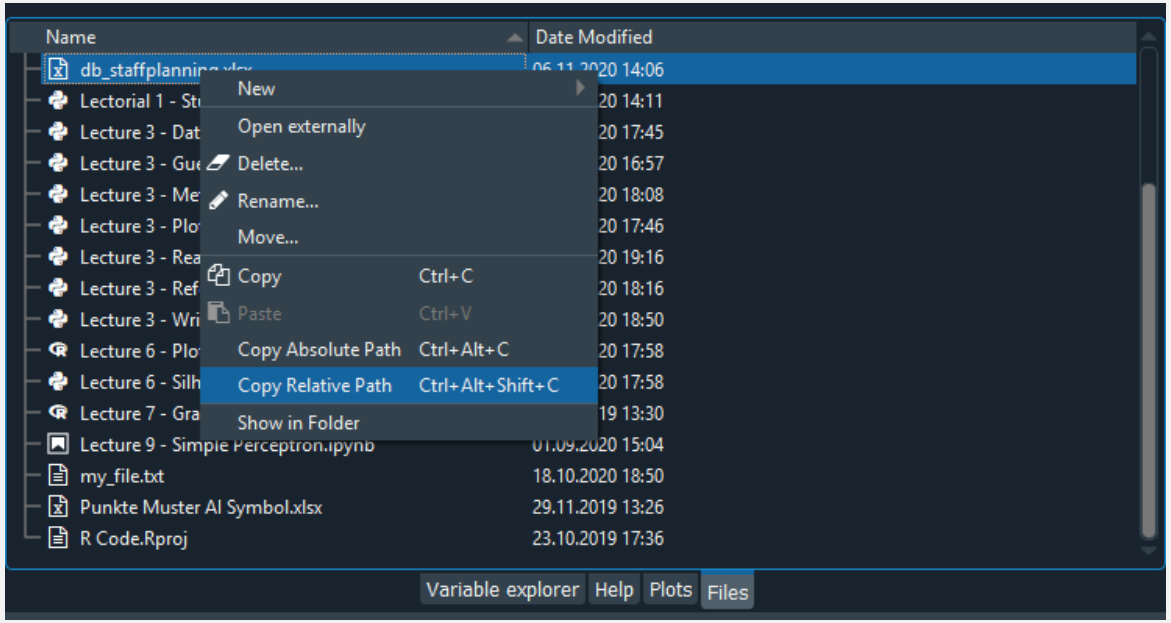
- Open the data with your IDE and take you some time and understand your data!





1.1 You have Difficulties to load the File?

```
Traceback (most recent call last):

  File "<ipython-input-371-21a17716bbf2>", line 1, in <module>
    pandas.read_excel(open("db_staffplanning2.xlsx", "rb"), sheet_name="staff")

FileNotFoundError: [Errno 2] No such file or directory: 'db_staffplanning2.xlsx'
```



Navigate from the file pane to your file and select “Copy relative Path”. Or setup an project and copy the file into the project folder!

1.2 Format staff_list

- Then we have to aggregate `staff_list` to `planing_table` to compare with `workload_table`

```
def format_staff_list(staff_list, workload_table):
    planing_table = workload_table.copy() # use workload as a template for timetable
comparison

    for day in planing_table["DAY"]:
        for time in range(0, 25):
            workers_available = 0
            for worker in range(1, 11):
                availability = is_worker_available(staff_list, worker, day, time)
                if(availability == True):
                    workers_available = (workers_available + 1)
            planing_table.loc[(planing_table["DAY"] == day),time] = workers_available

    return(planing_table)
```

1.2 Helper Function

- To make our code more feasible we make use of a `is_worker_available` function

```
def is_worker_available(staff_list, worker, day, time):
    condition = ((staff_list["ID"] == worker) & (staff_list["DAY"] == day))
    start_time = staff_list.loc[condition, "START"]
    work_time = staff_list.loc[condition, "CONTRACT"]
    end_time = start_time + work_time

    availability = ((time >= start_time) & (time < end_time)).item()

    return availability
```

- To test our coding so far, we use the following print statement

```
print(is_worker_available(staff_list, 1, "Monday", 1))
```


1.2 Common Mistake: Do not forget Python starts couting from 0

```
for worker in range(1, 11):
```

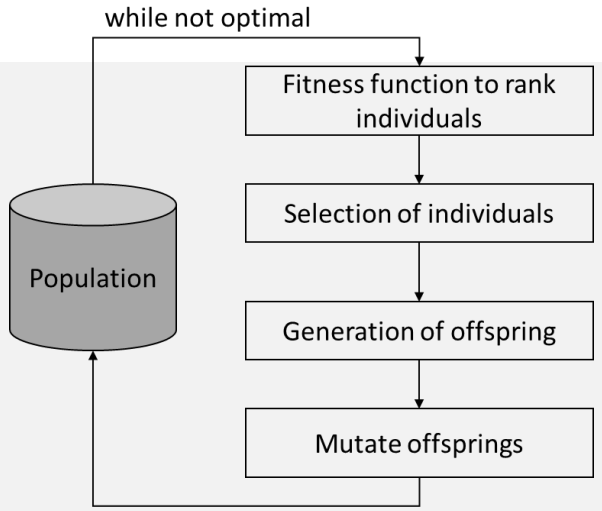
Do you have some trouble with the range function?

Do not forget that our worker IDs start from 1 and not from 0 (that's why we did data exploration. Furthermore Python starts counting from 0, and the last digit in range() is exclusive!

2.1 Genetic steps - Generate population of parents

- We build a function that generates multiple staff lists, otherwise you have to use the list from the db and starte there

```
def generate_staff_list_population(num_lists, staff_list):
    staff_list_population = []

    for i in range(num_lists):
        new_staff_list = generate_staff_list(staff_list)
        staff_list_population.append(new_staff_list)

    return staff_list_population
```



2.1 Genetic steps - Generate population of parents

- Then we generate new random starting times based on the conditions in the original staff_list

```
def generate_staff_list(staff_list):
    new_staff_list = staff_list.copy() # template

    for day in new_staff_list["DAY"]:
        for worker in new_staff_list["ID"]:
            condition = ((new_staff_list["ID"] == worker) & (new_staff_list["DAY"] == day))
            new_staff_list.loc[(condition), "START"] = numpy.random.randint(0, 23)

    return(new_staff_list)
```

2.2 Genetic steps - Combination / Crossover

- And further function to combine two different start_time lists

```
def crossover(parents, num_childs):
    num_parents = len(parents)
    childs = []

    for i in range(num_childs):
        mom = parents[numpy.random.randint(low = 0, high = num_parents - 1)]
        dad = parents[numpy.random.randint(low = 0, high = num_parents - 1)]
        child = mom.copy()

        selection = numpy.random.choice(range(0,50), size=25, replace=False)
        child.loc[selection] = dad.loc[selection]
        childs.append(child)

    return childs
```

2.3 Genetic steps - Mutation

- We implement the mutation step

```
def mutate(parents, num_mutations):
    num_parents = len(parents)

    for i in range(num_parents):
        selection = numpy.random.choice(range(0,50), size=num_mutations, replace=False)
        mutations = numpy.random.randint(low = 0, high = 23, size=len(selection))
        parents[i].loc[selection, "START"] = mutations

    return parents
```

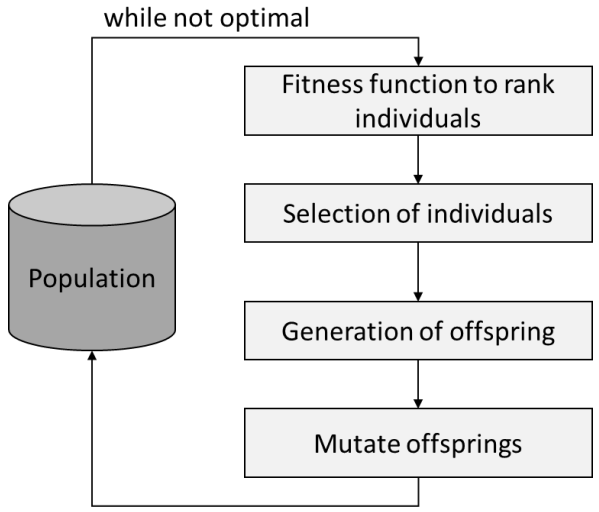

2.4 Genetic steps - Compute fitness of the current planning

- And two functions to rank and compute fitness

```
def fitness_function(planing_table, workload_table):
    dif = workload_table.set_index("DAY").subtract(planing_table.set_index("DAY"),
fill_value=0)
    dif = dif.mask(dif < 0, 0)
    dif_sum = dif.sum(axis=1).sum(axis=0)

    max_workload = workload_table.set_index("DAY").sum(axis=1).sum(axis=0)
    fitness = (1 - dif_sum/max_workload)

    return fitness

def rank_staff_lists(staff_list_population, workload_table):
    results = []
    for individual in staff_list_population:
        planing_table = format_staff_list(individual, workload_table)
        fitness = fitness_function(planing_table, workload_table)
        results.append([fitness, individual])

    results = sorted(results, reverse=True)
    return(results)
```

2. Final Genetic Algorithm

```
def evolution(staff_list, workload_table, num_iterations, pop_size):
    population = generate_staff_list_population(num_lists = pop_size, staff_list = staff_list)
    if(pop_size< 4):
        pop_size = 4

    for i in range(num_iterations):
        # compute fitness
        ranked_staff_lists = rank_staff_lists(staff_list_population = population, workload_table = workload_table)

        # keep only top 2
        population = []
        for j in range(2):
            population.append(ranked_staff_lists[j][1])

        # crossover & mutate
        childs = crossover(parents = population, num_childs = pop_size-2)
        childs = mutate(parents = childs, num_mutations = 10)

        # new population
        population.extend(childs)


    winner = rank_staff_lists(staff_list_population = population, workload_table = workload_table)[0]
    print(winner)
```

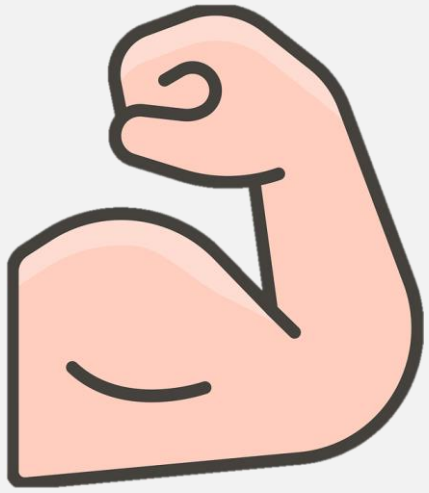


Understand Results

- Now, we can run our algorithm

```
evolution(staff_list = staff_list, workload_table = workload_table, num_iterations = 10, pop_size=4)
```

```
[0.5577464788732394,
          DAY  ID  START  CONTRACT
0      Monday   1      7        10
1      Monday   2     12        10
2      Monday   3      8        10
3      Monday   4      6        10
4      Monday   5     13        10
5      Monday   6      6        10
6      Monday   7      9        10
7      Monday   8     13        10
8      Monday   9     10         5
9      Monday  10      1         5
10    Tuesday   1      5        10
11    Tuesday   2      6        10
12 …
```



GOOD JOB!

Classroom Case

Case

Now get more familiar with genetic algorithms:

- Play with the different parameters e.g. num_childs, how does it influence the results?
- Try to implement another cost_function, how does it influence the results?
- Remember the national park roadtrip problem from lecture 2? Take a look at the work of Nathan Brixius (↗ here) and Randal Olson (↗here, and ↗ here)

Just

{ Keep }

Coding