[1. Fundamentals 1](#_Toc149729258)

[1.1. Number Beggars - Lists\_Basics 1](#_Toc149729259)

[1.2. Hkvh 2](#_Toc149729260)

[2. Advanced 2](#_Toc149729261)

[2.1. Paint Colors - Exercise: Stacks, Queues, Tuples, and Sets 2](#_Toc149729262)

[2.2. Summation Pairs - 02\_Tuples\_and\_Sets 4](#_Toc149729263)

[2.3. Present Delivery – List Multidimensional 5](#_Toc149729264)

[2.4. Recursive Power – Functions advanced 6](#_Toc149729265)

[3. OOP 7](#_Toc149729266)

[3.1. Formula 1 Manager - 06\_Polymorphism\_and\_Abstraction exercise 7](#_Toc149729267)

[4. aaOOP 11](#_Toc149729268)

[4.1. Paint Colors - Exercise: Stacks, Queues, Tuples, and Sets 11](#_Toc149729269)

Гордият човек е като локва - хвърли в нея камък и ще опръска всичко наоколо с мръсотия. А смиреният е като море - ще погълне безследно всеки камък и даже кръгове по водата няма да се образуват.

Дядо Добри

# Fundamentals

## Number Beggars - Lists\_Basics

# W:\1\_Python\1-Training\1\_Projects\1st\_Project

\02\_Fundamentals\_with\_Python

\03\_Lists\_Basics\Exercises\04\_number\_beggars\_a.py

W:\1\_Python\1-Training\1\_Projects\1st\_Project\02\_Fundamentals\_with\_Python\03\_Lists\_Basics\Exercises\04\_number\_beggars\_a.py

**4. Number Beggars**

You will receive **2 lines** of input. On the first line, you will receive a **single string of integers**, separated by a comma and a space **", "**. On the **second line,** you will receive a **count of beggars.** Your job is to print a **list with the sum** of what **each beggar** brings home, assuming they all take **regular turns**, from the first to the last number in the list.

For example: **[1, 2, 3, 4, 5]** for **2** beggars will return a result of **9** and **6**, as the first one takes **[1, 3, 5]**, the second one collects **[2, 4]**. The same list with **3 beggars** would produce a better outcome for the **second** beggar: **5**, **7** and **3**, as they will respectively take **[1, 4]**, **[2, 5]**, and **[3]**.

Also, note that not all beggars have to take the same amount of "offers", meaning that the length of the list is **not** necessarily a **multiple of n**. The list length could be even shorter - i.e., the last beggars will take nothing (0).

Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1, 2, 3, 4, 5  2 | [9, 6] |
| 3, 4, 5, 1, 29, 4  6 | [3, 4, 5, 1, 29, 4] |
| 100, 94, 24, 99  5 | [100, 94, 24, 99, 0] |

## Hkvh

# Advanced

## Paint Colors - Exercise: Stacks, Queues, Tuples, and Sets

G:\Other computers\My Computer\1-Training\1\_Projects\1st\_Project\03\_Advanced\03\_Stacks\_Queues\_Tuples\_and\_Sets\_Exercise\Recapitulate\Exercises\06\_paint\_colors\_c.py

**6. Paint Colors**

*You will have to find all possible color combinations that can be used.*

Write a program that finds colors in a string. You will be given a string on a **single line** containing **substrings** (separated by a **single space**) from which you will be able to form the following colors:

Main colors: **"red"**, **"yellow"**, **"blue"**

Secondary colors: **"orange"**, **"purple"**, **"green"**

To form a color, you should concatenate the **first** and the **last** **substrings** and check if you can get **any** of the **above colors'** **names**. If there is **only one substring left**, you should **use it** to do the same check.

You can only **keep** a **secondary color** if the **two main colors needed** for its creation could be **formed from the given substrings**:

* **orange** **= red + yellow**
* **purple = red + blue**
* **green = yellow + blue**

**Note:** You could find some of the main colors needed to keep a secondary color **after** it is found.

When you form a color, **remove both** substrings. Otherwise, you should **remove the last character** of **each** substring and **return** them in the **middle** of the **original string**. If the string contains an **odd number of substrings**, you should put the substrings **one position ahead**.

For example, if you are given the string **"re yellow bye"** you could not form a color with the substring **"re"** and **"bye"**, so you should remove the last character and return them in the middle of the string: **"r by yellow"**.

In the end, **print out the list with colors** in the order in which they are found.

Input

* Single line **string**

Output

* The **list** with the collected colors

Constrains

* You will not receive an empty string
* Please consider only the colors mentioned above
* There won't be any cases with repeating colors

Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| d yel blu e low redd | ['yellow', 'blue', 'red'] |
| **Comment** | |
| First, we take "**d"** and "**redd"**. After combining those substrings, we don't get any of the needed colors, so we remove the last characters from both substrings and return them in the middle of the original string, and it becomes "**yel blu red e low"**.  After that, we take "**yel"** and "**low"** so the first color we add to our list is yellow, and the string we are searching in looks as follows: "**blu red e"**  Then we take "**blu"** and "**e"**, and since this color is one of the searched ones (blue), we add it to our collection, and the state of the string is now "**red"**.  We should take the last substring and check if it matches some of the colors, and since it does, we add it (red) to our colors collection.  Finally, we print all the colors found: yellow, blue, and red in the format shown above. | |
| **Input** | **Output** |
| r ue nge ora bl ed | ['red', 'blue'] |
| **Comment** | |
| We don't keep orange because we don't have yellow in the final list with colors (combining red and yellow gives us orange). | |
| **Input** | **Output** |
| re ple blu pop e pur d | ['red', 'purple', 'blue'] |

## Summation Pairs - 02\_Tuples\_and\_Sets

W:\1\_Python\1-Training\1\_Projects\1st\_Project\03\_Advanced

\02\_Tuples\_and\_Sets\Lab\6\_summation\_pairs.py

**6. Summation Pairs**

***The task is not included in the Judge system.***

On the first line, you will receive a sequence of **numbers** separated by space. On the second line, you'll receive a **target** number. Your task is to **find** the **pairs of numbers** whose **sum** **equals** the **target number**. For each found pair print **"{number} + {number} = {target\_number}"**. You may **NOT** use the **same element twice to fulfill the condition above**.

Can you come up with an algorithm that has less time complexity?

Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 5 4 2 2 3 1 3 2  4 | 1 + 3 = 4  1 + 3 = 4  2 + 2 = 4 |
| 11 8 5 6 9 2 9 7 3 4  11 | 8 + 3 = 11  5 + 6 = 11  9 + 2 = 11  7 + 4 = 11 |

## Present Delivery – List Multidimensional

03\_Advanced\04\_Multidimensional\_Lists\Exercises\_2\07\_present\_delivery\_a.py

**7. Present Delivery**

*The presents are ready, and Santa has to deliver them to the kids.*

You will receive an integer **m** for the **number** of **presents** Santa has and an integer **n** for the **size** of the **neighborhood** with a **square** shape. On the following lines, you will receive the **matrix**, which represents the neighborhood.

Santa will be in a **random** **cell**, marked with the letter **"S"**. Each cell stands for a house where children may live. If the cell has **"X"** on it, that means there lives a **naughty** kid. Otherwise, if a **nice** kid lives there, the cell is marked by **"V"**. There can also be cells marked with **"C"** for cookies. **All of the empty positions** will be marked with **"-"**.

Santa can move "**up**", **"down"**, **"left"**, **"right"** with **one position** each time. These will be the **commands** that you receive. If he moves to a house with a **nice** kid, the kid **receives a present**, but if Santa reaches a house with a **naughty** kid, he **doesn't** drop a present. If the command sends Santa to a cell marked with **"C"**, Santa eats cookies and becomes happy and extra generous to **all the kids around him**\* (meaning all of them will receive presents - it doesn't matter if naughty or nice). If Santa has been to a house, the cell becomes **"-"**.

**Note**: \*around him means on his left, right, upwards, and downwards by one cell. In this case, **Santa** doesn't move to these cells, or if he does, he **returns** to the **cell** where the **cookie** was.

If Santa runs out of presents **or** receives the command "**Christmas morning**", you should end the program.

Keep in mind that you should check whether all the nice kids received presents.

**Input**

* On the first line, you are given the integer **m** - the count of presents
* On the second - integer **n** - the size of the neighborhood
* The **following n lines** hold the values for every **row**
* On each of the following lines you will get a command

**Output**

On the first line:

If Santa runs out of presents, **but** there are still some **nice** kids left print: "**Santa ran out of presents!**"

Next, print the matrix.

In the end, print one of these messages:

If he manages to give **all** the nice kids presents, print:  
"**Good job, Santa! {count\_nice\_kids} happy nice kid/s.**"

Otherwise, print:   
**"No presents for {count nice kids} nice kid/s."**

**Constraints**

* The size of the **square** matrix will be between **[2…10].**
* Santa's position will be marked with '**S**'.
* There will **always** be **at** **least** **1** **nice** kid.
* There **won't be a case** where the cookie is on the border of the matrix.

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  4  - X V -  - S - V  - - - -  X - - -  up  right  down  right  Christmas morning | - - - -  - - - S  - - - -  X - - -  Good job, Santa! 2 happy nice kid/s. | Santa has 5 presents. The size of the matrix is 4. After we receive the matrix, we start reading commands. The first one is "up". The "X" means there is a naughty kid, so Santa moves on without dropping any presents. Next, he reaches a nice kid and drops a present. The "down" command moves Santa to an empty cell. The last command before the "Christmas morning" message is "right". Again we have a nice kid. The count of nice kids reached 2, and we don't have any nice kids without presents left. So we print the appropriate message. |
| 3  4  - - - -  V - X -  - V C V  - - - S  left  up | Santa ran out of presents!  - - - -  V - - -  - - S -  - - - -  No presents for 1 nice kid/s. | The commands send Santa to a cell with a cookie, so all of the kids around him receive presents. He runs out of presents because we have 3 kids there and only 3 presents. The program ends, and we have 1 nice kid that hasn't received a present. |

**Examples**

## Recursive Power – Functions advanced

1st\_Project\03\_Advanced\05\_Functions\_Advanced\Lab\06\_recursive\_power.py

**Recursive Power**

Create a **recursive** function called recursive\_power() which should receive a **number** and a **power**. Using **recursion,** **return** the result of **number \*\* power**. Submit only the function in the judge system.

**Examples**

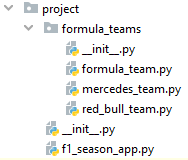
|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(recursive\_power(2, 10)) | 1024 |
| print(recursive\_power(10, 100)) | 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 |

# OOP

## Formula 1 Manager - 06\_Polymorphism\_and\_Abstraction exercise

**6. Formula 1 Manager**

For this task, you will be provided with a **skeleton** that includes all the folders and files you need.



***Note: You cannot change the folder and file structure and their names!***

**Judge Upload**

Create a **zip** file with the **project** **folder** and **upload it** to the judge system.

You do not need to include **in the zip file** your **venv**, **.idea**, **pycache**, and **\_\_MACOSX** (for Mac users), so you do not exceed **the maximum allowed size** of **16.00 KB**.

**Description**

You are the F1 manager of the two biggest teams in F1, "Red Bull" and "Mercedes". Your task is to create a program that calculates the revenue after every race for both teams. Your app should have the following structure and functionality.

**1. Class FormulaTeam**

In the **formula\_team.py** file, the class **FormulaTeam** should be implemented. It is a **base class** for any **type of formula team,** and it **should not be able to be instantiated**.

#### Structure

The class should have the following attribute:

* **budget: int**
* An integer that represents the **budget of the team**.
* If the budget is **less than 1 000 000**, raise **ValueError** with the message: **"F1 is an expensive sport, find more sponsors!"**

#### Methods

#### \_\_init\_\_(budget: int)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### calculate\_revenue\_after\_race(race\_pos: int)

* Each team should be able to calculate their revenue
* Each team has its unique sponsors
  + Sponsors give the team money if they finish in a certain position or better
* Each team has a different amount of expenses

**2. Class RedBullTeam**

In the **red\_bull\_team.py**, the class **RedBullTeam** should be implemented.

#### Methods

#### \_\_init\_\_(budget: int)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### calculate\_revenue\_after\_race(race\_pos: int)

* Red Bull sponsors:
  + Oracle:
    - 1st place – 1 500 000$
    - 2nd place – 800 000$
  + Honda:
    - 8th place – 20 000$
    - 10th place – 10 000$
* Red Bull expenses per race – 250 000$
* To **calculate the revenue** from the race, **sum the earned money** from the sponsors depending on the position in the race and **subtract the expenses**
* After that, **add the result** to the team's budget and **return** the following message: **"The revenue after the race is { revenue }$. Current budget { current budget }$"**

***Note: Each sponsor gives the money for the best position only. If you are 1st and the sponsor gives money for 1st and 2nd positions, you get the money only for the 1st position!***

**3. Class MercedesTeam**

In the **mercedes.py**, the class **MercedesTeam** should be implemented.

#### Methods

#### \_\_init\_\_(budget: int)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### calculate\_revenue\_after\_race(race\_pos: int)

* Mercedes sponsors:
  + Petronas:
    - 1st place – 1 000 000$
    - 3rd place – 500 000$
  + TeamViewer:
    - 5th place – 100 000$
    - 7th place – 50 000$
* Mercedes expenses per race – 200 000$
* To **calculate the revenue** from the race, **sum the earned money** from the sponsors depending on the position in the race and **subtract the expenses**
* After that, **add the result** to the team's budget and **return** the following message: **"The revenue after the race is { revenue }$. Current budget { current budget }$"**

***Note: Each sponsor gives the money for the best position only. If you are 1st and the sponsor gives money for 1st and 2nd positions, you get the money only for the 1st position!***

**4. Class F1SeasonApp**

In the **f1\_season\_app.py** file, the class **F1SeasonApp** should be implemented. It will contain all the functionality of the project.

#### Structure

The class should have the following attributes:

* **red\_bull\_team: RedBullTeam**
  + It should be **set to None on initialization**.
* **mercedes\_team: MercedesTeam**
  + It should be **set to None on initialization**.

#### Methods

#### \_\_init\_\_()

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### register\_team\_for\_season(team\_name: str, budget: int)

* Valid team names: **"Red Bull", "Mercedes"**
* If a **team name is valid**, register the team with the corresponding name and **return** the following message:

**"{ team name } has joined the new F1 season."**

* If a **team name is invalid**, raise **ValueError** with the message: **"Invalid team name!"**

***Note: There won't be a case where a valid team tries to register for a second time.***

#### new\_race\_results(race\_name: str, red\_bull\_pos: int, mercedes\_pos: int)

* If **Red Bull or Mercedes haven't registered yet**, raise an **Exception** with the following message: **"Not all teams have registered for the season."**
* Otherwise, find which team has the better position in the race, calculate every team's revenue, update their budget, and return the following message: **"Red Bull: { Red Bull revenue message }. Mercedes: { Mercedes revenue message }. { team with better position } is ahead at the { race name } race."**
* ***Note: Teams' positions will always be valid.***

**Examples**

|  |
| --- |
| **Input** |
| from project.f1\_season\_app import F1SeasonApp  f1\_season = F1SeasonApp()  print(f1\_season.register\_team\_for\_season("Red Bull", 2000000))  print(f1\_season.register\_team\_for\_season("Mercedes", 2500000))  print(f1\_season.new\_race\_results("Nurburgring", 1, 7))  print(f1\_season.new\_race\_results("Silverstone", 10, 1)) |
| **Output** |
| Red Bull has joined the new F1 season.  Mercedes has joined the new F1 season.  Red Bull: The revenue after the race is 1270000$. Current budget 3270000$. Mercedes: The revenue after the race is -150000$. Current budget 2350000$. Red Bull is ahead at the Nurburgring race.  Red Bull: The revenue after the race is -240000$. Current budget 3030000$. Mercedes: The revenue after the race is 900000$. Current budget 3250000$. Mercedes is ahead at the Silverstone race. |

# aaOOP

## Paint Colors - Exercise: Stacks, Queues, Tuples, and Sets