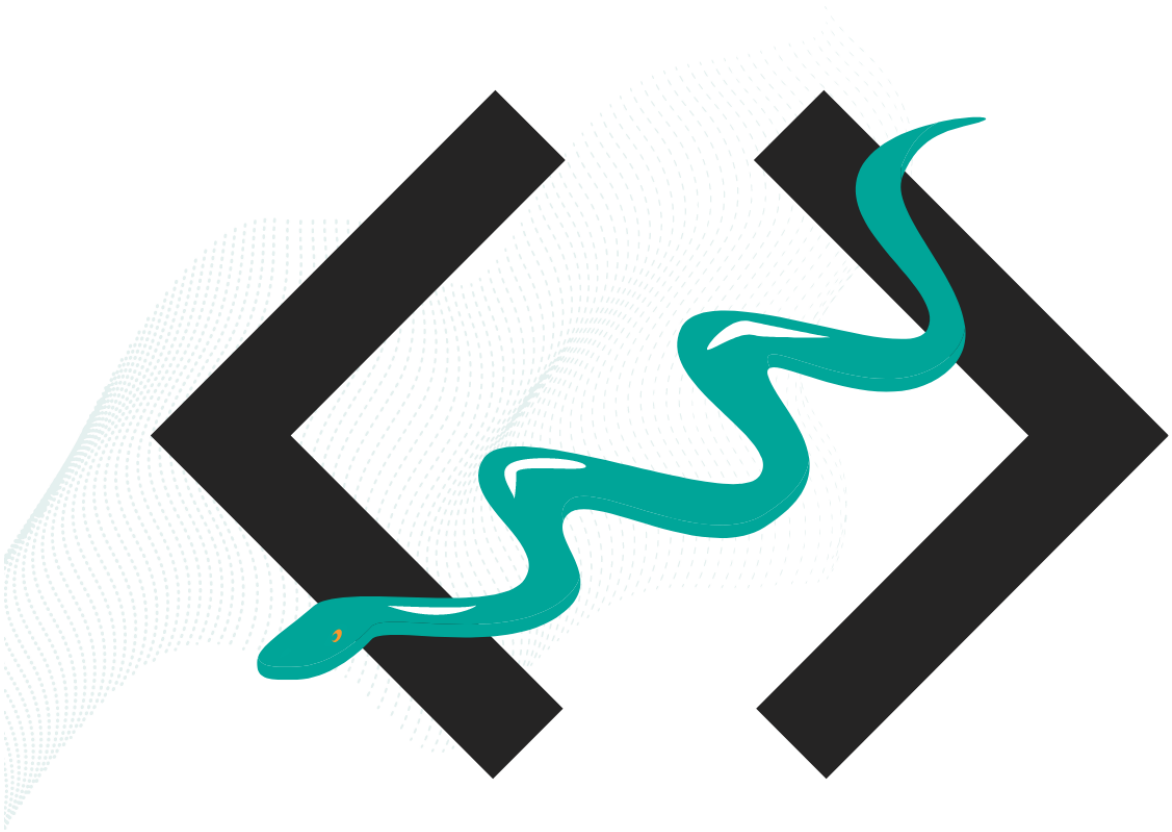
Python Cheat Sheet – Technical Concepts for the Job Interview

A quick reference Python cheat sheet for learning the common tasks. Data structures, functions, data wrangling – the basics any data scientist should know.



Python is among the most popular programming languages for Data Science. Along with SQL and R, most data science roles will require an above-average proficiency in Python. As a data analyst, you might be able to do most of your work in SQL. However, if you want to move to higher-paying roles like a Data Engineer or Data Scientist and work with complex tasks like building data pipelines or deal with Machine Learning Algorithms and Statistics, you must master Python. To understand the kind of questions asked in Data Science Interviews, have a

look at our comprehensive article on Python Interview Questions and Answers. We also have a framework and step-by-step guide to solving Python Coding Interview questions.

In this Python cheat sheet, we will use basic Python data structures, control program flow, and define and use functions. We will perform basic data wrangling tasks like joining data sets, splitting data into groups, and aggregating data to generate summary statistics. You should ***NOT*** use Pandas or NumPy libraries to answer these questions. Later on, we will solve these exercises using the Pandas (and NumPy) libraries. We will be avoiding Pandas initially to make sure that you understand how even basic Python data structures like lists, dictionaries, tuples, and sets can be used for day-to-day data wrangling tasks. Further, there might be cases when one may not have access to the Pandas library because of admin permissions. In such a scenario, too, one should be able to perform these tasks.

Datasets

We will be using the EU referendum results data. There are three files. You can make use of the starter notebook to try these problems.

The first file, named `uk_demo.json`, contains information about the area, the region the area belongs to, and the electorate in the area. The data is presented in the following manner.

Field	Description
Area_Code	Unique identifier for the area
Area	Area Name
Region_Code	Unique identifier for the region
Region	Region Name
Electorate	Number of registered voters in the area

The second file, named `uk_results.json`, contains the results of the voting for each area. The data therein is arranged thus.

Field	Description
Area_Code	Unique identifier for an area
Votes_Cast	Number of Votes Cast in the area
Valid_Votes	Number of Valid Votes cast in the area
Remain	Number of votes for Remain
Leave	Number of votes for Leave

We also have additional info about the rejected votes in each region in the third dataset - `uk_rejected_ballots.json`

The data is presented in the following manner.

Field	Description
Area_Code	Unique identifier for an area
Rejected_Ballots	Number of Rejected Ballots in the Area
No_official_mark	Number of ballots rejected because they did not have any official mark
Voting_for_both_answers	Number of ballots rejected because they voted for both answers
Writing_or_mark	Number of ballots rejected because they had a writing instead of check

Reading Data Into Python

Before we start solving these problems in this python cheat sheet, we need to read data into Python. One way to do this is to download the file locally and then read it. A simpler way is to use the requests module. The requests module is the most widely used Python module for HTTP. We can transmit data securely using security protocols and user authentication easily using the requests module. For the purpose of this exercise, we will simply use the requests module to read the JSON file and load it into a Python list.

```
import requests
```

```
uk_demo_src =
r"https://raw.githubusercontent.com/viveknest/statascratch-solutions/main/UK%20Referendum%20Data/uk_demo.json"
f = requests.get(uk_demo_src)
uk_demo = f.json()
uk_demo[:5]
```


```
[{'Area_Code': 'E06000031',
  'Area': 'Peterborough',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 120892},
 {'Area_Code': 'E06000032',
  'Area': 'Luton',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 127612},
 {'Area_Code': 'E06000033',
  'Area': 'Southend-on-Sea',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 128856},
 {'Area_Code': 'E06000034',
  'Area': 'Thurrock',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 109897},
 {'Area_Code': 'E06000055',
  'Area': 'Bedford',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 119530}]
```


As you can see, the output is a list of dictionaries. We can do the same for the other two files as well.

```
uk_results_src =
r"https://raw.githubusercontent.com/viveknest/statascratch-solutions/main/UK%20Referendum%20Data/uk_results.json"
f = requests.get(uk_results_src)
uk_results = f.json()
uk_results[:5]
```

```
[{'Area_Code': 'E06000031',
  'Votes_Cast': 87469,
  'Valid_Votes': 87392,
  'Remain': 34176,
  'Leave': 53216},
 {'Area_Code': 'E06000032',
  'Votes_Cast': 84616,
  'Valid_Votes': 84481,
  'Remain': 36708,
  'Leave': 47773},
 {'Area_Code': 'E06000033',
  'Votes_Cast': 93939,
  'Valid_Votes': 93870,
  'Remain': 39348,
  'Leave': 54522},
 {'Area_Code': 'E06000034',
  'Votes_Cast': 79950,
  'Valid_Votes': 79916,
  'Remain': 22151,
  'Leave': 57765},
 {'Area_Code': 'E06000055',
  'Votes_Cast': 86135,
  'Valid_Votes': 86066,
  'Remain': 41497,
  'Leave': 44569}]
```

```
uk_rejected_src =
r"https://github.com/viveknest/statascratch-solutions/raw/main/UK%20Referendum%20Data/uk_rejected_ballots.json"
f = requests.get(uk_rejected_src)
uk_rejected = f.json()
uk_rejected[:5]
```

```
[{'Area_Code': 'E06000031',
  'Rejected_Ballots': 77,
  'No_official_mark': 0,
  'Voting_for_both_answers': 32,
  'Writing_or_mark': 7,
  'Unmarked_or_void': 38},
 {'Area_Code': 'E06000032',
  'Rejected_Ballots': 135,
  'No_official_mark': 0,
  'Voting_for_both_answers': 85,
  'Writing_or_mark': 0,
  'Unmarked_or_void': 50},
 {'Area_Code': 'E06000033',
  'Rejected_Ballots': 69,
  'No_official_mark': 0,
  'Voting_for_both_answers': 21,
  'Writing_or_mark': 0,
  'Unmarked_or_void': 48},
 {'Area_Code': 'E06000034',
  'Rejected_Ballots': 34,
  'No_official_mark': 0,
  'Voting_for_both_answers': 8,
  'Writing_or_mark': 3,
  'Unmarked_or_void': 23},
 {'Area_Code': 'E06000055',
  'Rejected_Ballots': 69,
  'No_official_mark': 0,
  'Voting_for_both_answers': 26,
  'Writing_or_mark': 1,
  'Unmarked_or_void': 42}]
```

Join the Datasets

First, in this Python cheat sheet, we need to “merge” the three lists using the `Area_Code` as the key. To accomplish this merging, we created a nested dictionary keeping the merge field as the key for the top-level dictionary. Since we will do this for multiple datasets, we create a simple function.

```
# Convert lists into a dictionary of dictionaries
def dict_convert(data, key = 'Area_Code'):
    out_dict = {}
    for row in data:
        out_dict.update({
            row[key]: row
        })
    return out_dict
```

We can simply pass each of the lists to this function and get a nested dictionary.

```
uk_demo_dict = dict_convert(uk_demo)
uk_demo_dict
```

```
{'E06000031': {'Area_Code': 'E06000031',
  'Area': 'Peterborough',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 120892},
 'E06000032': {'Area_Code': 'E06000032',
  'Area': 'Luton',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 127612},
 'E06000033': {'Area_Code': 'E06000033',
  'Area': 'Southend-on-Sea',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 128856},
 'E06000034': {'Area_Code': 'E06000034',
  'Area': 'Thurrock',
  'Region_Code': 'E12000006',
```

We repeat this process for the other lists as well.

```
uk_results_dict = dict_convert(uk_results)
uk_rejected_dict = dict_convert(uk_rejected)
```

We now have the key of each of these dictionaries as the merge key. We can simply create a merged dictionary by accessing the nested dictionary using Area_Code, which is the key of the outer dictionary in each of the three dictionaries.

```
merged_dict = {}

for key in uk_demo_dict.keys():
    merged_dict.update({
        key: {**uk_demo_dict[key], **uk_results_dict[key],
**uk_rejected_dict[key]}
    })
merged_dict
```

```
{'E06000031': {'Area_Code': 'E06000031',
  'Area': 'Peterborough',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 120892,
  'Votes_Cast': 87469,
  'Valid_Votes': 87392,
  'Remain': 34176,
  'Leave': 53216,
  'Rejected_Ballots': 77,
  'No_official_mark': 0,
  'Voting_for_both_answers': 32,
  'Writing_or_mark': 7,
  'Unmarked_or_void': 38},
 'E06000032': {'Area_Code': 'E06000032',
  'Area': 'Luton',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 127612,
```

And there we have the three datasets merged into a single dictionary. If we want to get a list again, we can use “unnest” the dictionary into a list.

```
merged_list = [v for k,v in merged_dict.items()]
merged_list
```

```
[{'Area_Code': 'E06000031',
  'Area': 'Peterborough',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 120892,
  'Votes_Cast': 87469,
  'Valid_Votes': 87392,
  'Remain': 34176,
  'Leave': 53216,
  'Rejected_Ballots': 77,
  'No_official_mark': 0,
  'Voting_for_both_answers': 32,
  'Writing_or_mark': 7,
  'Unmarked_or_void': 38},
 {'Area_Code': 'E06000032',
  'Area': 'Luton',
  'Region_Code': 'E12000006',
  'Region': 'East',
  'Electorate': 127612,
```

Summary Statistics

The next in this Python cheat sheet is to find the maximum, minimum, median, 25th, and 75th percentile values for

- Electorate
- Number of Votes Cast
- Number of Valid Votes
- Number of Remain Votes
- Number of Leave Votes
- Number of Rejected Votes

Since we have to repeat the same process multiple times, we will create a function that generates the summary statistics. We use a variable parameter to specify the variable for which we need to find the summary statistic. Let us start off by defining the function. We would be passing the merged_dict that we had generated earlier as the source data.

```
def summ_stats(source_data, variable):
    return None
```

We need to extract the data points for the relevant variable. Since we would like to perform mathematical operations on these data points, we will use a list. We also sort the list so as to calculate the median, 25th, and 75th percentiles easily.

```
def summ_stats(source_data, variable):
    # Get the variable data
    out_list = []
    for k,v in source_data.items():
        out_list.append(v[variable])
    out_list.sort()

    return out_list

summ_stats(merged_dict, 'Electorate')
```

```
[1799,
 5987,
 16658,
 17375,
 21259,
 24119,
 27478,
 29390,
 36418,
 36794,
 37273,
 37841,
 38527,
 39176,
```

We start calculating the summary statistics now. We will output a dictionary with all these statistics. Finding minimum and maximum values is pretty easy. We simply use the built-in min() and max() functions.

```
def summ_stats(source_data, variable):
    # Get the variable data
    out_list = []
    for k,v in source_data.items():
        out_list.append(v[variable])
    out_list.sort()
    max_value = max(out_list)
    min_value = min(out_list)


    out_dict = {
        'Max': max_value
        , 'Min': min_value
    }

    return out_dict

summ_stats(merged_dict, 'Electorate')
```

```
{'Max': 1260955, 'Min': 1799}
```

Median

Median is the middle value of a dataset. If the number of elements (n) is odd, then the median is the following element.

$$(\frac{n+1}{2})^{th}$$

If the n is even, then the median is the average of the following elements

$$(\frac{n}{2})^{th}$$

And

$$(\frac{n}{2}+1)^{th}$$

To implement this in Python, we first need to find the number of elements in the list. We can do this by finding the length of the list using the len() function. Then we check if the number is odd or even. Here is a simple implementation of the odd / even checker.

```
def odd_even(number):
    # This function will return True if the number is even, otherwise it
will return false
    if number // 2 == number / 2:
        return True
    return False
```

```
odd_even(4)
```

```
odd_even(4)
```

```
True
```

```
odd_even(11)
```

```
odd_even(11)
```

```
False
```

We use the same logic to check if the number of elements is odd or even and then calculate the median as described above.

```
def summ_stats(source_data, variable):
    # Get the variable data
    out_list = []
    for k,v in source_data.items():
        out_list.append(v[variable])
    out_list.sort()
    max_value = max(out_list)
    min_value = min(out_list)
    num_items = len(out_list)
    # Median Value
    if num_items //2 == num_items / 2:
        mid_value1 = num_items //2 - 1
        mid_value2 = num_items //2
        median_value = (out_list[mid_value1] + out_list[mid_value2]) / 2
    else:
        mid_value = (num_items + 1) // 2 - 1
        median_value = out_list[mid_value]


    out_dict = {
        'Max': max_value
        , 'Min': min_value
        , 'Median' : median_value
    }

    return out_dict

summ_stats(merged_dict, 'Electorate')
```

```
{'Max': 1260955, 'Min': 1799, 'Median': 96425.5}
```

Note: We subtract 1 from each of the positions since Python is zero-indexed

Quartiles

Calculating the percentiles is a bit tricky. There are multiple options to choose from. We will be using the nearest rank method to calculate the percentiles. The location of the pth percentile value in an ordered dataset is given by

$$\lceil \frac{P}{100} x N \rceil$$

Where the below represents the ceiling function.

$$\lceil x \rceil$$

In simple terms, it rounds up a decimal to the least integer greater than or equal to x. To find the ceiling in Python, we can use the built-in function associated with all floating point values __ceil__()

The function can now be modified to incorporate the two quartiles.

```
def summ_stats(source_data, variable):
    # Get the variable data
    out_list = []
    for k,v in source_data.items():
        out_list.append(v[variable])
    out_list.sort()
    max_value = max(out_list)
    min_value = min(out_list)
    num_items = len(out_list)
    # Median Value
    if num_items //2 == num_items / 2:
        mid_value1 = num_items //2 - 1
        mid_value2 = num_items //2
        median_value = (out_list[mid_value1] + out_list[mid_value2]) / 2
    else:
        mid_value = (num_items + 1) // 2 - 1
        median_value = out_list[mid_value]
```

```
    # Quartiles, using the nearest rank method
    q1_rank = (num_items * 0.25).__ceil__() - 1
    q3_rank = (num_items * 0.75).__ceil__() - 1

    q1 = out_list[q1_rank]
    q3 = out_list[q3_rank]

    out_dict = {
        'N': num_items
        , 'Max': max_value
        , 'Min': min_value
        , 'Median' : median_value
        , 'q1' : q1
        , 'q3' : q3
    }

    return out_dict

summ_stats(merged_dict, 'Electorate')
```

```
{'N': 382,
 'Max': 1260955,
 'Min': 1799,
 'Median': 96425.5,
 'q1': 72487,
 'q3': 141486}
```

We can find the summary statistics for the other variables as well.

```
summ_stats(merged_dict, 'Votes_Cast')
```

```
{'N': 382,
 'Max': 790523,
 'Min': 1424,
 'Median': 72544.5,
 'q1': 54864,
 'q3': 104809}
```

```
summ_stats(merged_dict, 'Valid_Votes')
```

```
{'N': 382,
 'Max': 790149,
 'Min': 1424,
 'Median': 72511.5,
 'q1': 54833,
 'q3': 104699}
```

```
summ_stats(merged_dict, 'Remain')
```

```
{'N': 382,
 'Max': 440707,
 'Min': 803,
 'Median': 33475.0,
 'q1': 23515,
 'q3': 48257}
```

```
summ_stats(merged_dict, 'Leave')
```

```
{'N': 382,
 'Max': 349442,
 'Min': 621,
 'Median': 37573.5,
 'q1': 28631,
 'q3': 54198}
```

```
summ_stats(merged_dict, 'Rejected_Ballots')
```

```
{'N': 382, 'Max': 614, 'Min': 0, 'Median': 46.5, 'q1': 33, 'q3': 74}
```

Find the Area with the highest and lowest electorates

The next is this Python cheat sheet is to find the area with the highest and lowest electorates, we will use the object properties for sorting. Let us use a simple example to illustrate the process. Suppose we have the following list.

```
sample_list = ['Alpha', 'Gamma', 'Beta', 'Zeta', 'Delta', 'Epsilon', 'Nu',
'Mu']
sample_list
```

```
['Alpha', 'Gamma', 'Beta', 'Zeta', 'Delta', 'Epsilon', 'Nu', 'Mu']
```

We can sort the list in alphabetical order

```
sorted(sample_list)
```

```
['Alpha', 'Beta', 'Delta', 'Epsilon', 'Gamma', 'Mu', 'Nu', 'Zeta']
```

Or reverse alphabetical order

```
sorted(sample_list, reverse = True)
```

```
['Zeta', 'Nu', 'Mu', 'Gamma', 'Epsilon', 'Delta', 'Beta', 'Alpha']
```

Suppose we want to sort by the properties of the list elements, let's say we want to sort by the length of the string. We can use the key parameter and define a lambda function.

```
sorted(sample_list, key = lambda x: len(x))
```

```
['Nu', 'Mu', 'Beta', 'Zeta', 'Alpha', 'Gamma', 'Delta', 'Epsilon']
```

We can also sort by multiple levels. The following will sort based on the length of the string and then by alphabetical order.

```
sorted(sample_list, key = lambda x: (len(x), x))
```

```
['Mu', 'Nu', 'Beta', 'Zeta', 'Alpha', 'Delta', 'Gamma', 'Epsilon']
```

We will now implement this on the merged list we had created earlier.

```
sorted(merged_list, key = lambda x:x['Electorate'])
```


```
[{'Area_Code': 'E06000053',
  'Area': 'Isles of Scilly',
  'Region_Code': 'E12000009',
  'Region': 'South West',
  'Electorate': 1799,
  'Votes_Cast': 1424,
  'Valid_Votes': 1424,
  'Remain': 803,
  'Leave': 621,
  'Rejected_Ballots': 0,
  'No_official_mark': 0,
  'Voting_for_both_answers': 0,
  'Writing_or_mark': 0,
  'Unmarked_or_void': 0},
 {'Area_Code': 'E09000001',
  'Area': 'City of London',
  'Region_Code': 'E12000007',
  'Region': 'London',
```


Since we need the one with the lowest electorate, we take the first element

```
sorted(merged_list, key = lambda x:x['Electorate'])[0]
```

```
{'Area_Code': 'E06000053',
 'Area': 'Isles of Scilly',
 'Region_Code': 'E12000009',
 'Region': 'South West',
 'Electorate': 1799,
 'Votes_Cast': 1424,
 'Valid_Votes': 1424,
 'Remain': 803,
 'Leave': 621,
 'Rejected_Ballots': 0,
 'No_official_mark': 0,
 'Voting_for_both_answers': 0,
 'Writing_or_mark': 0,
 'Unmarked_or_void': 0}
```

From this we output the Area.

```
sorted(merged_list, key = lambda x:x['Electorate'])[0]['Area']
```

```
'Isles of Scilly'
```

To get the Area with the highest electorate, we simply take the last element in the sorted list.

```
sorted(merged_list, key = lambda x:x['Electorate'])[-1]['Area']
```

```
'Northern Ireland'
```

Aggregation

Find the Region wise totals

We start off by getting all the regions. We do this by using a Python set.

```
regions = set([area['Region'] for area in merged_list])
regions
```

```
{'East',
 'East Midlands',
 'London',
 'North East',
 'North West',
 'Northern Ireland',
 'Scotland',
 'South East',
 'South West',
 'Wales',
 'West Midlands',
 'Yorkshire and The Humber'}
```

Now we iterate over the entire list and add totals region wise.

```
region_dict = {}
for region in regions:
    reg_sum = sum([area['Electorate'] for area in merged_list if
area['Region'] == region])
    region_dict.update({region: reg_sum})

region_dict
```

```
{'Northern Ireland': 1260955,
 'East Midlands': 3384299,
 'Scotland': 3987112,
 'North West': 5241568,
 'South East': 6465404,
 'Wales': 2270272,
 'West Midlands': 4116572,
 'Yorkshire and The Humber': 3877780,
 'London': 5424768,
 'East': 4398796,
 'North East': 1934341,
 'South West': 4138134}
```

Contribution of each region to the total

To calculate the contribution of each region, we first find the total. We can do this by adding up the region_dict values.

```
total = sum(region_dict.values())
total
```

```
46500001
```

We now calculate the contribution of the region's electorate as a percentage of the total.

```
cont_dict = {}
for k,v in region_dict.items():
    cont_dict.update({k: v/total * 100})
cont_dict
```

```
{'Northern Ireland': 2.711731124478901,
 'East Midlands': 7.278062209073931,
 'Scotland': 8.574434224205715,
 'North West': 11.272189004899161,
 'South East': 13.904094324643133,
 'Wales': 4.882305271348273,
 'West Midlands': 8.852842820368972,
 'Yorkshire and The Humber': 8.339311648616954,
 'London': 11.66616749105016,
 'East': 9.459776140649975,
 'North East': 4.159873028819935,
 'South West': 8.899212711844887}
```

There we have it. We have performed the most common dataframe operations in Pandas without using the Pandas library. You can find the entire solved notebook here. Let us perform these operations using the Pandas library.

Pandas

The next is Pandas in this Python cheat sheet. The Pandas library provides us with numerous functions to manipulate structured data. It is the go-to library for data science operations and is also very useful for transitioning to Python if one comes from the spreadsheet or SQL world. We start off by importing pandas as pd in Python.

```
import pandas as pd
```

Load the datasets

We can use the built-in read_json function to read the input datasets directly into a pandas dataframe.

```
demo_df =
pd.read_json(r"https://raw.githubusercontent.com/viveknest/statascratch-sol
```

```
utions/main/UK%20Referendum%20Data/uk_demo.json")
demo_df
```

	Area_Code	Area	Region_Code	Region	Electorate
0	E06000031	Peterborough	E12000006	East	120892
1	E06000032	Luton	E12000006	East	127612
2	E06000033	Southend-on-Sea	E12000006	East	128856
3	E06000034	Thurrock	E12000006	East	109897
4	E06000055	Bedford	E12000006	East	119530
...	...	...	...	...	...
377	E08000032	Bradford	E12000003	Yorkshire and The Humber	342817
378	E08000033	Calderdale	E12000003	Yorkshire and The Humber	149195
379	E08000034	Kirklees	E12000003	Yorkshire and The Humber	307081
380	E08000035	Leeds	E12000003	Yorkshire and The Humber	543033
381	E08000036	Wakefield	E12000003	Yorkshire and The Humber	246096

382 rows × 5 columns

We can do the same for the other datasets as well.

```
results_df =
pd.read_json(r"https://raw.githubusercontent.com/viveknest/statascratch-sol
utions/main/UK%20Referendum%20Data/uk_results.json")
rejected_df =
pd.read_json(r"https://github.com/viveknest/statascratch-solutions/raw/main
/UK%20Referendum%20Data/uk_rejected_ballots.json")
```

Join the datasets

Pandas provides multiple methods to work with multiple datasets. We can use the merge method to perform an SQL-style join.

```
merged_df = pd.merge(left = demo_df, right = results_df, on =
'Area_Code').merge(rejected_df, on = 'Area_Code')
merged_df
```

	Area_Code	Area	Region_Code	Region	Electorate	Votes_Cast	Valid_Votes	Remain	Leave	Rejected_Ballots	No_official_mark	Voting_for_both
0	E06000031	Peterborough	E12000006	East	120892	87469	87392	34176	53216	77	0	
1	E06000032	Luton	E12000006	East	127612	84616	84481	36708	47773	135	0	
2	E06000033	Southend-on-Sea	E12000006	East	128856	93939	93870	39348	54522	69	0	
3	E06000034	Thurrock	E12000006	East	109897	79950	79916	22151	57765	34	0	
4	E06000055	Bedford	E12000006	East	119530	86135	86066	41497	44569	69	0	
...	...	...	...	...	...	...	...	...	...	...	...	
377	E08000032	Bradford	E12000003	Yorkshire and The Humber	342817	228727	228488	104575	123913	239	0	
378	E08000033	Calderdale	E12000003	Yorkshire and The Humber	149195	106004	105925	46950	58975	79	0	
379	E08000034	Kirklees	E12000003	Yorkshire and The Humber	307081	217428	217240	98485	118755	188	0	
380	E08000035	Leeds	E12000003	Yorkshire and The Humber	543033	387677	387337	194863	192474	340	39	
381	E08000036	Wakefield	E12000003	Yorkshire and The Humber	246096	175155	175042	58877	116165	113	0	

382 rows × 14 columns

Summary Statistics

Pandas comes with a built-in function to generate summary statistics quickly. We can actually generate all the summary statistics in one go using the describe method.

```
merged_df.describe()
```

	Electorate	Votes_Cast	Valid_Votes	Remain	Leave	Rejected_Ballots	No_official_mark	Voting_for_both_answers	Writing
count	3.820000e+02	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	38
mean	1.217278e+05	87898.801047	87832.416230	42254.557592	45577.858639	66.384817	0.607330	23.780105	
std	9.706175e+04	63603.499333	63554.624962	35622.620732	31308.892098	59.451306	2.851117	27.232427	
min	1.799000e+03	1424.000000	1424.000000	803.000000	621.000000	0.000000	0.000000	0.000000	
25%	7.252375e+04	54875.500000	54844.250000	23535.250000	28668.500000	33.250000	0.000000	10.000000	
50%	9.642550e+04	72544.500000	72511.500000	33475.000000	37573.500000	46.500000	0.000000	16.000000	
75%	1.413798e+05	104436.500000	104332.000000	48245.500000	54137.500000	74.000000	0.000000	27.000000	
max	1.260955e+06	790523.000000	790149.000000	440707.000000	349442.000000	614.000000	39.000000	311.000000	3

Since we need only limited statistics and only for a subset of the columns, we use the .loc method to retrieve the relevant fields.

```
merged_df.describe().loc[['max', 'min', '50%','25%', '75%']][['Electorate',
```

```
'Votes_Cast', 'Valid_Votes', 'Remain', 'Leave', 'Rejected_Ballots' ]]
```

	Electorate	Votes_Cast	Valid_Votes	Remain	Leave	Rejected_Ballots
max	1260955.00	790523.0	790149.00	440707.00	349442.0	614.00
min	1799.00	1424.0	1424.00	803.00	621.0	0.00
50%	96425.50	72544.5	72511.50	33475.00	37573.5	46.50
25%	72523.75	54875.5	54844.25	23535.25	28668.5	33.25
75%	141379.75	104436.5	104332.00	48245.50	54137.5	74.00

Find the Area with the highest and lowest electorates

There are multiple ways of doing this. In out Python cheat sheet, let us try a couple of methods.

Sorting

As we did in the non-Pandas method above in this Python cheat sheet, we can sort the values in ascending or descending order and output the first or the last row to get the lowest and highest values.

```
merged_df.sort_values(by = 'Electorate').iloc[0]['Area']
```

```
'Isles of Scilly'
```

```
merged_df.sort_values(by = 'Electorate').iloc[-1]['Area']
```

```
'Northern Ireland'
```

Using the idxmax() and idxmin() functions.

The idxmax() function returns the dataframe index of the highest value. We can pass this index back to the main dataframe to find the row containing the highest value and subset the relevant column.

```
merged_df['Electorate'].idxmax()
```

```
171
```

```
merged_df.loc[merged_df['Electorate'].idxmax()]['Area']
```

```
'Northern Ireland'
```

The idxmin() function will give the index of the lowest value.

```
merged_df.loc[merged_df['Electorate'].idxmin()]['Area']
```

```
'Isles of Scilly'
```

Find the Region wise totals

Again there are numerous ways in Pandas to aggregate data. Let us use the two most common ways.

Using groupby

Those from the SQL world can use the groupby method to achieve aggregation.

```
merged_df.groupby(by = 'Region').agg({'Electorate': sum})
```

Region	Electorate
East	4398796
East Midlands	3384299
London	5424768
North East	1934341
North West	5241568
Northern Ireland	1260955
Scotland	3987112
South East	6465404
South West	4138134
Wales	2270272
West Midlands	4116572
Yorkshire and The Humber	3877780

Pivot Table

Those from the spreadsheet world can use the pivot table function which works exactly as the name suggests.

```
merged_df.pivot_table(index = ['Region'], aggfunc = {'Electorate': sum})
```

Region	Electorate
East	4398796
East Midlands	3384299
London	5424768
North East	1934341
North West	5241568
Northern Ireland	1260955
Scotland	3987112
South East	6465404
South West	4138134
Wales	2270272
West Midlands	4116572
Yorkshire and The Humber	3877780

This gives an identical solution as the previous step.

Contribution of each region to the total

To find the contribution of each region, we simply divide the aggregate values from above with the sum of the column values.

```
merged_df.groupby(by = 'Region').agg({'Electorate': sum}) /
merged_df['Electorate'].sum() * 100
```

Region	Electorate
East	9.459776
East Midlands	7.278062
London	11.666167
North East	4.159873
North West	11.272189
Northern Ireland	2.711731
Scotland	8.574434
South East	13.904094
South West	8.899213
Wales	4.882305
West Midlands	8.852843
Yorkshire and The Humber	8.339312

You can find the entire solved solutions using Pandas here. Also, check out this *Pandas cheat sheet* that explains the functions any data scientist should know and includes interview questions from Forbes, Meta, Google, and Amazon.

Conclusion

In this Python Cheat Sheet, we solved data wrangling problems in Python with and without the Pandas library. If you want to succeed in Data Science, you need to be versatile with your skills. Depending on the resources available, one must be able to get a solution. Luckily, Python is one of the easiest languages to learn. If you have a good understanding of SQL or spreadsheets, Pandas - one of the most popular Python libraries for Data Science, is very easy to learn. Also, have a look at our Python vs R article. On the StrataScratch platform, you will find more than 500 questions from Data Science interviews from top companies like Netflix, Amazon, Microsoft, Google, et al. Sign up on StrataScratch and join a community of over 20,000 aspirants and mentors helping make your dream of getting top data science jobs a reality.