1. You have been provided with large CSV files containing census data about Melbourne. Unfortunately, the data is 'noisy': some people have made mistakes on the census form, or intentionally entered incorrect data. Your first task as a programmer-analyst is to clean up the noisy data for later analysis.

There are a few particular errors in this data:

- People have accidentally included too many zeroes in their salary, or entered salaries which are too large (or even negative!) as a joke. All salaries should be greater than or equal to $0, and less than or equal to **MAX_SALARY**, a variable which is given to you.
- People have mischievously entered a suburb which doesn't exist. The valid suburbs are listed in a variable called **VALID_SUBURBS**, which is given to you.
- Some people have entered their ages as words instead of digits, e.g. inputting 'twenty' instead of '20' – others have entered negative ages.

Write a function **clean_data(data)** which takes one argument, a dictionary of data in the format returned by **read_data**. This data has been read directly from a CSV file, and is noisy! Your function should construct and return a new data dictionary which is identical to the input dictionary, except that invalid data values have been replaced with **None**. **You should not modify the argument dictionary, data.**
For example, let's look at the data contained in noisy_sample.csv:

```
>>> data_noisy = read_data('noisy_sample.csv')
>>> for key, value in sorted(data_noisy.items()):
...     print(key)
...     print(value)
P1
{'age': 'eighty two', 'language': 'English', 'suburb': 'Toorak', 'salary': '60196.0'}
P2
{'age': '49', 'language': 'Chinese', 'suburb': 'St. Kilda', 'salary': '1694514.0'}
P3
{'age': '54', 'language': 'Italian', 'suburb': 'Neverland', 'salary': '49775.0'}
```


Clearly some of the values are invalid! Let's call **clean_data** on the data, and look at the result:

```
>>> data_cleaned = clean_data(data_noisy)
>>> for key, value in sorted(data_cleaned.items()):
...     print(key)
...     print(value)
P1
{'age': None, 'language': 'English', 'suburb': 'Toorak', 'salary': '60196.0'}
P2
{'age': '49', 'language': 'Chinese', 'suburb': 'St. Kilda', 'salary': None}
P3
{'age': '54', 'language': 'Italian', 'suburb': None, 'salary': '49775.0'}
```

Notice the **None** values in the nested dictionaries of the cleaned data.
You can assume the following:

- the input data dictionary does not contain **None** values;
- all salaries (both valid and invalid) will be strings that can be cast to **float**;
- all valid ages will be strings that can be cast to non-negative **int**s.

Testing your function

We have included some code at the bottom of your file to make it easier for you to test your code. Don't worry if you don't understand all the details. When you open the Grok terminal, the code we've added will read the data from the file specified in `test_file` into a dictionary called `data_noisy`.

To manually test your code, change the value of `test_file` to the file you want to use to test your code, then execute the following in the Grok terminal:

```
>>> data_cleaned = clean_data(data_noisy)
```

program.py | header.py | noisy_sample.csv | noisy_data.csv

```
from header import read_data, VALID_SUBURBS, MAX_SALARY

def clean_data(data):
# replace 'pass' with your code
    pass




# to test your function with 'noisy_data.csv' or another CSV file,
# change the value of this variable
test_file = 'noisy_sample.csv'


# you don't need to modify the code below
if __name__ == '__main__':
    data_noisy = read_data(test_file)
```


2. Write a function called **average_salary(data, lower_age, upper_age)** which takes a dictionary containing census data and an age bracket (specified by two **float** parameters, **lower_age**and **upper_age**), and calculates the average salary (as a **float**) for individuals in that age bracket.

You may assume the census data in **data** is 'clean', that is all invalid values have been replaced by **None**. If a nested dictionary contains a **None** value for the **salary** key or **age** key, you should ignore it in your calculation. (If the dictionary has **None** for a different key, e.g. **suburb**, you should still include it in the calculation.)

You may assume that both **lower_age** and **upper_age** are positive. Note that **lower_age** and **upper_age** are *inclusive*bounds, meaning an individual is in the age bracket if their age is greater than or equal to **lower_age**, and less than or equal to **upper_age**.

If **lower_age > upper_age**, your function should return **0.0**. If there are no individuals in the age bracket, again you should return **0.0**.

Here are some examples of what your function should return for different datasets and age brackets:

```
>>> data_cleaned = read_data("cleaned_sample.csv")
>>> average_salary(data_cleaned, 40, 50)
0.0
>>> average_salary(data_cleaned, 40, 54)
49775.0
>>> data_cleaned = read_data("cleaned_data.csv")
>>> average_salary(data_cleaned, 30, 50)
77963.76392572944
>>> average_salary(data_cleaned, 50, 70)
122097.66666666667
```

program.py | header.py | cleaned_sample.csv | cleaned_data.csv

```
from header import read_data

def average_salary(data, lower_age, upper_age):
    # replace 'pass' with your code
    pass



# to test your function with 'cleaned_data.csv' or another CSV file,
# change the value of this variable
test_file = 'cleaned_sample.csv'
```

3. Your employers are interested in the distribution of wealth in the population. One way to analyse this is to divide the range of possible salaries into a number of equal-sized 'bins' – where a bin is just a subset of the overall range – then count the number of salaries falling into each bin (if you've ever worked with histograms before, this should be very familiar).

For example, we could divide the total salary range ($0–$200,000) into ten bins: $0–$20,000, $20,001–$40,000, $40,001–$60,000, and so on, up to $180,001–$200,000. The distribution of salaries would then be summarised by 10 integers corresponding to the ten bins. In general, we experiment with the number of bins to find the number that gives the most informative distribution.

Write a function called **wealth_distribution(data, n_bins, max_salary)**, which calculates the distribution of salaries greater than or equal to 0 and less than or equal to **max_salary**, by dividing that range into **n_bins** bins and counting the number of salaries that fall into each bin. **The bin width should be an integer.** Your function should return a **list** of **int**s, with each integer representing the number of salaries falling in the corresponding bin.

If a nested dictionary in **data** contains a **None** value for the **salary** key, you should ignore it in your calculation. (If the dictionary has **None** for a different key, you should still include it in the calculation.)

You may assume that both **n_bins** and **max_salary** are positive integers. Notice that the first bin will always start at 0, and that including **max_salary** in the last bin may make the last bin slightly 'wider' than the others. For example, if **max_salary == 100** and **n_bins == 5**, the bins would be 0–19, 20–39, 40–59, 60–79, and 80–100.

Here is an example of how your function should behave:

```
>>> data_cleaned = read_data("cleaned_data.csv")
>>> wealth_distribution(data_cleaned, 10, MAX_SALARY)
[309, 106, 160, 160, 129, 40, 3, 1, 0, 0]
>>> wealth_distribution(data_cleaned, 20, MAX_SALARY)
[294, 15, 42, 64, 67, 93, 74, 86, 77, 52, 28, 12, 3, 0, 1, 0, 0, 0, 0, 0]
>>> wealth_distribution(data_cleaned, 10, max_salary=105259)
[294, 19, 50, 69, 74, 91, 87, 85, 74, 41]
>>> wealth_distribution(data_cleaned, 20, max_salary=105259)
[293, 1, 7, 12, 19, 31, 32, 37, 31, 43, 47, 44, 43, 43, 48, 38, 42, 32, 21, 20]
```

program.py | header.py | cleaned_data.csv

```
from header import read_data, MAX_SALARY

def wealth_distribution(data, n_bins, max_salary):
    # replace 'pass' with your code
    pass



# to test your function with another CSV file,
# change the value of this variable
test_file = 'cleaned_data.csv'
```

4. One way in which suburbs vary is in the age profile of their residents. Some suburbs will be more popular with students and young people, others will be more attractive to families with young children, others will be more appropriate for elderly residents, and so on.

Write a function called **location_age_counts(data, lower_age, upper_age)** which returns a dictionary of the number of individuals in the given age bracket for each suburb. That is, each key in the dictionary should be a suburb name, and the value for that suburb should be an **int** corresponding to the number of individuals in the suburb who fall in the age bracket specified by **lower_age** and **upper_age** (as in question 2, the bracket *includes* **lower_age** and **upper_age**). Your dictionary should have a key for all suburbs in **VALID_SUBURBS**, even the ones that have no residents in the age bracket.
In this question, you should ignore any nested dictionary with a **None** value for the **age** key or the **suburb** key. **None** values for other keys are acceptable. As with question 2, you may assume that **lower_age** and **upper_age** are positive. If **lower_age > upper_age**, your function should return a dictionary with a value of 0 for every suburb.
Here are some examples of how your function should behave:


```
>>> data_cleaned = read_data("cleaned_data.csv")
>>> location_age_counts(data_cleaned, 0, 17)
{'Dandenong': 29, 'Docklands': 19, 'Frankston': 35, 'Collingwood': 17, 'Brunswick': 16, 'Parkville':
27, 'Fitzroy': 23, 'Footscray': 31, 'Kensington':
27, 'Richmond': 22, 'St. Kilda': 16, 'Toorak': 8, 'Caulfield': 22, 'Flemington': 18, 'Southbank': 26,
'Hawthorn': 23}
>>> location_age_counts(data_cleaned, 18, 34)
{'Dandenong': 17, 'Docklands': 8, 'Frankston': 15, 'Collingwood': 10, 'Brunswick': 11, 'Parkville': 8,
'Fitzroy': 13, 'Footscray': 14, 'Kensington': 12, 'Richmond': 10, 'St. Kilda': 3, 'Toorak': 8, 'Caulfield':
8, 'Flemington': 8, 'Southbank': 12, 'Hawthorn': 8}
>>> location_age_counts(data_cleaned, 35, 44)
{'Dandenong': 13, 'Docklands': 17, 'Frankston': 20, 'Collingwood': 12, 'Brunswick': 11, 'Parkville':
19, 'Fitzroy': 18, 'Footscray': 19, 'Kensington': 13, 'Richmond': 13, 'St. Kilda': 14, 'Toorak': 11,
'Caulfield': 23, 'Flemington': 21, 'Southbank': 18, 'Hawthorn': 14}
>>> location_age_counts(data_cleaned, 99, 100)
{'Dandenong': 0, 'Docklands': 0, 'Frankston': 0, 'Collingwood': 0, 'Brunswick': 0, 'Parkville': 0,
'Fitzroy': 0, 'Footscray': 0, 'Kensington': 0, 'Richmond': 0, 'St. Kilda': 0, 'Toorak': 0, 'Caulfield': 0,
'Flemington': 0, 'Southbank': 0, 'Hawthorn': 0}
```

program.py > header.py cleaned_data.csv

```python
from header import read_data, VALID_SUBURBS

def location_age_counts(data, lower_age, upper_age):
    # replace 'pass' with your code
    pass



# to test your function with another CSV file,
# change the value of this variable
test_file = 'cleaned_data.csv'

# you don't need to modify the code below
if __name__ == '__main__':
    data_cleaned = read_data(test_file)
```


5. A prestigious Victorian university has asked the ABS to produce a report on the financial status and living situation of 18–34 year olds in Melbourne. They have asked you to help them generate some of the data for this report.

Write a function called **main(datafile)** which takes a filename as an argument, which reads the census data contained in that file, cleans the data, and uses the data to print out some facts about 18–34 year olds. You should assume that the data in **datafile** is noisy. Your function should calculate and print out the following facts:

1. The average salary for 18–34 year olds.
2. A list of the suburbs in which 18–34 year olds live.

> **Note**
> You will probably find it useful to call `read_data`, `clean_data`, etc. in your `main` function. To help, we have provided implementations of all the functions from preceding questions. These are imported at the top of `program.py`. **You do not need to copy code from previous questions.**

The average salary should be printed to **two decimal places**. Use *string formatting* (either the **format** method or old-style **%**formatting) to do this. You do not need to round the **float** values themselves.
Suburbs should be listed in **alphabetical order**, and only listed if they have **at least one 18–34 year old resident**. Next to the suburb name, in brackets, print the number of 18–34 year olds that live in the suburb.

Here is an example of what your function should print. Make sure your function matches the format exactly.

```
>>> main("noisy_data.csv")
Average salary for 18-34 year olds: $50139.66
18-34 year olds live in the following suburbs:
Brunswick (11)
Caulfield (8)
Collingwood (10)
Dandenong (17)
Docklands (8)
Fitzroy (13)
Flemington (8)
Footscray (14)
Frankston (15)
Hawthorn (8)
Kensington (12)
Parkville (8)
Richmond (10)
Southbank (12)
St. Kilda (3)
Toorak (8)
>>> main("noisy_sample.csv")
Average salary for 18-34 year olds: $0.00
18-34 year olds live in the following suburbs:
```


program.py | header.py | noisy_sample.csv | noisy_data.csv

```
1  from header import read_data, VALID_SUBURBS, MAX_SALARY
2  from reference import mean_data, average_salary, wealth_distribution, location_age_counts
3
4  def main(datafile):
5      # replace 'pass' with your code
6      pass
```