Now go away or I will taunt you a second time

Histogram Example

Let’s look at an example that combines the use of dictionaries and exception handling. This example is about statistical analysis of data. Specifically, we want to read data from a file (one floating point number per line) and produce a histogram of the data. To do this we want to subdivide numbers into ‘buckets’ and count how many times data values fall in each bucket. For the program, we will ask the user for the name of the file containing the data to be processed, and the width of each bucket. For this example we will use the file `data1.txt`.

Aside: Constructing randomised data sets

This file was constructed as follows.

```
>>> import random
>>> f = open('data1.txt', 'w')
>>> for i in range(1000) :
        f.write(str(random.normalvariate(0, 10)) + '\n')

>>> f.close()
```

The `random` module contains functions for producing pseudo-random numbers from different distributions. In this case, we use a normal distribution with mean 0 and standard deviation 10. We write 1000 generated random numbers into the file. Note we need to add a newline character otherwise all the numbers will be on a single line.

For this problem, we will produce a stand-alone program. The program consists of three parts: getting input from the user, computing the histogram, and writing out the results. Later we will write a GUI version of the same program.

We start with the main part — a function that takes a file name and a bucket size and returns a dictionary of bucket counts. The key for the dictionary will be the ‘bucket position’ — i.e. how many buckets away from 0 this bucket is. So, for example, if bucket size = 10, a value between 0 and 10 will be in bucket 0 and a value between -10 and 0 will be in bucket -1.

Here is the function definition.

```
def make_histogram(filename, bucketsize) :
    """Compute the histogram of the data in 'filename' with given bucket size.

    An error message will be produced if either the file cannot be opened
    for reading or an invalid data value is found.

    Parameters:
        filename (str): Name of the file from which data is read.
        bucketsize (int): Size of the data bucket.
    Return:
        dict<int, float>: Histogram of occurrences of data in file.

    Preconditions:
        bucketsize > 0
    """
    try :
        file = open(filename, 'r')
    except IOError :
        print('Error: cannot open ' + filename + ' for reading')
        return {}

    hist = {}
    for data_element in file :
        try :
            val = float(data_element)
        except ValueError :
            print('Error: cannot convert "' + data_element + '" to a float')
            return {}
        if val < 0 :
            bucket = int((val-bucketsize) / bucketsize)
        else :
            bucket = int(val / bucketsize)
        hist[bucket] = hist.get(bucket, 0) + 1

    return hist
```

The first thing this function does is attempt to open the file. Unlike previous examples, we have surrounded this with a `try` statement. This allows us to catch the error raised if the file cannot be opened for reading. If an `IOError` is raised then we print a simple error message and return an empty dictionary. We then create an empty dictionary, `hist` to store our data in. We then go through every line in the file directly with a for loop. The first part of the body of the for loop is to attempt to convert the line to a float. This is also surrounded in a `try` statement to test if the line can be converted. If it cannot then the `ValueError` is caught and again a simple error message is printed and an empty dictionary is returned. We then check the sign of the number `val`. If it is negative, we use one formula for which bucket it would belong in; otherwise, we use a different formula. This is because we use rounding when we convert from a float to an integer, as Python always rounds down we need to treat the different signed numbers differently. The last part of the for loop is to increase the count of how many items are in that particular bucket. This is the same as used in the character frequency example above. Finally, we return `hist`.

Note the power of dictionaries here — we could have used lists to store the bucket counts BUT we need to know the range of data values first. This would have meant reading the file twice or loading all the data into another list. The processing would also have been more complicated as well. Now we can save our program so far, `histogram1.py`, and test the function. We also create a file `data2.txt` and test the function using this file (it is a good idea to test small first).

```
>>> make_histogram('data2.txt', 1.0)
{0: 2, 1: 1, -1: 1}
>>> make_histogram('data1.txt', 1.0)
{0: 41, 1: 29, 2: 35, 3: 33, 4: 45, 5: 35, 6: 43, 7: 27, 8: 33, 9: 23,
10: 17, 11: 15, 12: 10, 13: 12, 14: 23, 15: 10, 16: 6, 17: 6, 18: 7, 19: 4, 20: 8,
21: 4, 22: 4, 23: 1, 24: 5, 25: 2, 26: 2, 27: 1, 28: 1, 29: 2, 30: 1, 34: 1,
35: 1, -1: 36, -32: 1, -30: 1, -28: 2, -26: 1, -25: 2, -24: 6, -23: 5, -22: 4,
-21: 6, -20: 2, -19: 6, -18: 6, -17: 10, -16: 18, -15: 11, -14: 17, -13: 20,
-12: 24, -11: 30, -10: 23, -9: 25, -8: 39, -7: 40, -6: 28, -5: 33, -4: 50, -3: 35,
-2: 32}
```

The next step is to get user input. Given we want a positive number for the bucket size we write the following function that checks user input.

```
def get_bucketsize() :
    """Return the bucket size asked from the user.

    Ensures that a valid bucket size is entered.

    Return:
        float: Number entered for the bucket size.
    """
    while True :
        user_input = input("Bucket size: ")
        try :
            size = float(user_input)
        except ValueError :
            print('Not a number')
            continue
        if size > 0 :
            break
        print('Number must be positive')
    return size
```

This function starts with an infinite `while True` loop; this enables us to keep asking the user until correct input is given. Then the user is prompted for the bucket size. The function then attempts to convert the bucket size given by the user into a float. This is surrounded by a try statement to catch a `ValueError` in case the user does not input a number. If the user input is not a number then a message is printed and then on the next line is a `continue` statement. `continue` starts at the begining of the loop body. This effectively skips all the code after the `continue`, then since we are at the top of the loop again the function asks the user again for a bucket size. If the input is a number, then it is tested to see if it is positive. If it is positive we `break` out of the while loop and return the bucket size. If it is not then we print out a message and the loop starts again. Saving this code into `histogram2.py` we can do a few tests.

```
>>> get_bucketsize()
Bucket size: a
Not a number
Bucket size: -10
Number must be positive
Bucket size: 3
3.0
```

Loop with `continue`

If `continue` is used inside the body of a loop, when the `continue` line is executed the loop moves on to the next iteration of the loop immediately. For `while` loops, this simply means starting the loop from the beginning, effectively ignoring the code after the `continue`. For a `for` loop it works like the `while` loop except that it moves onto the next item in the object being iterated through.

The last part is to pretty print the resulting histogram. The next function (similar to the frequency count one) does the job.

```
def pp_histogram(histogram, bucketsize) :
    """ Pretty prints the histogram using the size of the buckets."""
    keys = histogram.keys()
    keys = sorted(keys)
    for key in keys :
        print('({0:7.2f}, {1:7.2f}) : {2:3}'.format(
            key*bucketsize, (key+1)*bucketsize, histogram[key]))
```

This function first gets the list of the keys from the dictionary, `histogram`, using the `keys` method of dictionaries. This enables us to perform the next line, which is to sort this keys dictionary so that it is in order from smallest to largest. We then iterate over this list and print out the histogram information. The first two substitutions of the format string use two methods of the extra format string options discussed in the notes about the dictionary data structure. These segments of the format string look like `:7.2f`. The 7 means to have spacing of 7 characters. The .2f means to print as floats to 2 decimal places. The last substitution area contains the index number with a spacing of 3 characters. As the item to be printed here is just an integer it needs no float formatting.

Here is the result of saving our `histogram3.py` code and applying this function to the histogram for `data1.txt`.

```
>>> pp_histogram(make_histogram('data1.txt', 5.0), 5.0)
( -35.00,  -30.00) :   1
( -30.00,  -25.00) :   4
( -25.00,  -20.00) :  23
( -20.00,  -15.00) :  42
( -15.00,  -10.00) : 102
( -10.00,   -5.00) : 155
(  -5.00,    0.00) : 186
(   0.00,    5.00) : 183
(   5.00,   10.00) : 161
(  10.00,   15.00) :  77
(  15.00,   20.00) :  33
(  20.00,   25.00) :  22
(  25.00,   30.00) :   8
(  30.00,   35.00) :   2
(  35.00,   40.00) :   1
```

To complete the stand-alone program we just need to add the following code to the end of the code that we have written so far.

```
print('Print a histogram\n')
filename = input('File name: ')
bucketsize = get_bucketsize()
print('\n\n------------------------------------------\n\n')
pp_histogram(make_histogram(filename, bucketsize), bucketsize)
```

The complete code for this example is in `histogram.py`. If we run the module from IDLE we get the following output in the interpreter window.

```
Print a histogram
```

```
File name: data1.txt
Bucket size: 5.0


--------------------------------------------


( -35.00,  -30.00) :   1
( -30.00,  -25.00) :   4
( -25.00,  -20.00) :  23
( -20.00,  -15.00) :  42
( -15.00,  -10.00) : 102
( -10.00,   -5.00) : 155
(  -5.00,    0.00) : 186
(   0.00,    5.00) : 183
(   5.00,   10.00) : 161
(  10.00,   15.00) :  77
(  15.00,   20.00) :  33
(  20.00,   25.00) :  22
(  25.00,   30.00) :   8
(  30.00,   35.00) :   2
(  35.00,   40.00) :   1
```

What is going on? Well, all the expressions in the file are evaluated in the interpreter. The definitions are evaluated and as a consequence are added to the interpreter. The other expressions are then evaluated — this is really the program being executed.