file I/O, Benford's Law, sets

Ben Bolker

20 October 2019

Benford's law

Benford's law describes the (surprising) distribution of first (*leading*) digits of many different sets of numbers:

> Benford's law states that in listings, tables of statistics, etc., the digit 1 tends to occur with probability ~30%, much greater than the expected 11.1% (i.e., one digit out of 9). Benford's law can be observed, for instance, by examining tables of logarithms and noting that the first pages are much more worn and smudged than later pages (Newcomb 1881).

Read it about it on Wikipedia or MathWorld

We'll write a Python function `benford_count` that tabulates the occurrence of digits from a set of numbers.

- But where do we get our numbers from?

file I/O

A bunch of ways to read a file

Reference

- `f = open(filename,mode)` (mode = `r` for reading, `w` for writing, `a` for appending) - text by default; returns a **file object**

```
f = open("test.txt")
type(f)
print(f)
print(f.closed)
f.close()
print(f.closed)
```

`.closed` is a *data attribute* (e.g. see here): it's a value rather than a function (unlike a *method attribute* like `list.append()`): no parentheses!

- To free up resources, it's good practice to `close(f)` to close the file once you're done with it.
- By default, `open()` looks in the **current working directory**: if you want to get a file from somewhere else, you need to specify the full **path**, e.g.

```
f = open('/Users/bolker/Documents/temp/temp.txt', 'r')
f.close()
```

- you can use `import os; os.getcwd()` to check the current working directory and `os.listdir()` to see what files are there.

Once a file has been opened, its entire contents can be read in as a `str` using the `read()` method on the associated file object.

```
f = open('test.txt', 'r')
contents = f.read()
print(contents)
print(repr(contents))
f.close()
```

- Each line of the file ends in the **newline** character `\n`.

- You can use the `read()` method to read a specified number of characters from the file.
- `f.read(10)` will read the next 10 characters from the file.
- If the end of the file has been reached, this returns `''`, the empty string.

- the file object **keeps track of how much has been read**. Reopen if you want to start at the beginning again.

```
f = open('test.txt', 'r')
print('first 10 characters of the file:')
print(f.read(10))
print('next 6 characters (including the end of line character ‘\\n‘:')
print(f.read(6))
print("this is the rest of the file:")
contents = f.read()
print(contents)
f.close()
```

- Use the `readlines()` method file to read the file in a list of strings.

```
f = open('test.txt')
lines = f.readlines()
```

```
# print the list of lines.
# Each line ends with a new line character \n.
print(lines)
print(lines[2])
f.close()
```

- using `for` loop automatically reads lines too: `for line in f: do_something_with(L)`. This is probably the most common way to process a file. For example, if you know there is exactly one floating-point value per line and want to print the squares:

```
f = open("test.txt")
for line in f:
    print(line)
    line_u = line.upper()
    print(line_u)
f.close()
```

More I/O details

- getting rid of pesky newlines: `.strip()` method for strings (gets rid of leading and trailing *whitespace* [spaces, tabs, newlines ...])

```
f = open("test.txt")
L = f.readline() ## read one line
print(repr(L))
print(repr(L.strip()))
```

- breaking lines into words: `.split()` method for strings

```
f = open("test.txt")
L = f.readline() ## read one line
LL = L.strip().split(" ")
print(LL)
```

And even more

- `import os` in order to find out working directory (`os.getcwd()`), or set the directory `os.chdir(newpath)`; use full path or use (e.g.) `..` to go up one level
- opening a URL from the web `import urllib.request as ur; ur.urlopen(url)`
- to read a text file: `io.TextIOWrapper()`
- to read a CSV file: `import csv`, use `csv.reader()`

`next()`, and more flow control

- what if we want to a little more control? **next()** function, works on any **iterable** object (tuples, lists, ranges, files ...)

```
f = open("test.txt")
line = next(f) ## read one line
print(line)
```

- when you get to the end of a file (or a list or whatever) and try to use `next()` you get a `StopIteration` error; use `try/except` to handle it safely

```
f = open("test.txt")
finished = False
while not finished:
   try:
      line = next(f)
   except StopIteration:
      finished = True
```

- `try` is a *general* way to handle errors safely
- `except` tells Python what to do if `try` doesn't work. You can specify the particular **type** of error you're looking for, or use `except:` to catch any kind of error
- a standard idiom for doing something until it works uses `break`:

```
while True:
   try:
      x = input("enter a number: ")
      print(int(x))
      break
   except ValueError:
      print("Try again!")
```

Benford's Law

- **Project:** Create a python function `benford_count` that tabulates the occurrence of leading digits from a set of numbers.
- The set of numbers should be read in from a file.
- The filename should be an argument to the function.
- The function should return a tuple `digits_count` of length 10 (one for each digit) such that `digits_count[i]` is equal to the number of occurrences of the digit i as the leading digit of some number from the file, for i = 1, 1, . . . , 9.
- So `digits_count[2]` is the count of the number of times the digit 2 occurred as the leading digit of some number from the file.

- Note that the digit 0 doesn't occur as the leading digit of a number, except for the number 0, so we really don't need to count the number of 0's that occur.

File format

To properly process a data file, we need to make some assumptions about its format. - We'll assume that each line contains a number of words, for example, the name of a town, followed by its province/state and country, etc., and that the last word will be a number that represents the size of the quantities being counted in the file.

Steps:

1. Initialize a `digits_count` **list** of length 10, filled with zeros (why a list??)
2. Open the file
3. For each line in the file,
    - Retrieve the last word from the line
    - If it is a string of digits that doesn't start with 0, get the leading digit and update `digits_count`.
4. return `tuple(digits_count)`

Other considerations

- `http://www.testingbenfordslaw.com/` has some interesting data sets.
- Modify the code so that it will process a list of files.
- Improve the code so that it can correctly deal with a general number format of the form: 123,456,789.123456 . . . .
- Enable the program to directly access suitable files from the internet.
- The file that we created contains several function definitions. This file could be imported by other files in order to make use of the functions.
- To prevent the code at the bottom of the file to be executed when the file is imported, first test to see if the interpreter's name for the file is `__main__`. This will be the case when the file itself is being executed and not imported.
- Use a python dictionary instead of a tuple or list to keep track of the digit counts.

Sets

- A **set** is an unordered collection of distinct immutable objects.
- It is similar to a list, but the elements of a set are not ordered.
- `sets` are mutable objects and so can be altered.
- Sets are created using braces (`{` and `}`).
- Duplicate items in a set are ignored.

```
vowels = {'a', 'e', 'i', 'o', 'u'}
print(type(vowels))
print(vowels)
vowels = {'a', 'e', 'i', 'o', 'u', 'e', 'a'}
print(vowels)
print({1, 2, 3, 'a'} == {'a', 1, 1, 3, 2, 'a'})
```

- sets can also be created with the function `set`.
- `set()` produces the empty set.
- if `L` is a list and `S` is a `str`, `set(L)` and `set(S)` produce the set of items/characters from the list/string.
- sets can also be produced from a range.
- sets also support iteration, or looping, but the order is arbitrary.

```
print(set())
print(set([1, 2, 0, -1, 3, 1, 1, 2]))
print(set('hello world!'))
print(set(range(0, 10, 2)))

---

- a number of operations and methods are available to set
objects.
- ‘add‘, ‘remove‘, ‘intersection‘, ‘union‘, ...
```

```
small = {0, 1, 2, 3}
mid = {3, 4, 5, 6, 7}
big = {7, 8, 9}
big.add(10)
small.remove(0)
print(small, big)
print(small.intersection(mid))
print(small.union(big))
```

---

sets can be compared using methods or set operators.

```
d = {0,1}
print(d.issubset({0, 1, 2, 3}))
print(d <= {0, 1, 2, 3})

---

Code for testing whether a string is hexadecimal:

hex_char = "0123456789abcdef"
word = "12aac"
for char in word:
    if not char in hex_char:
       return False
    return True
```

this can be replaced by:

```
set(word) <= hex_char
```