First-Digit Law

> "Benford's law, also known as the Newcomb-Benford law, the law of anomalous numbers, or the first-digit law, is an observation that in many real-life sets of numerical data, the leading digit is likely to be small. In sets that obey the law, the number `1` appears as the leading significant digit about `30%` of the time, while `9` appears as the leading significant digit less than `5%` of the time. Uniformly distributed digits would each occur about `11.1%` of the time."
>
> -- *https://en.wikipedia.org/wiki/Benford's_law*

In particular, Wikipedia gives the following frequencies for each leading digit.

Leading Digit	Frequency
1	30.1%
2	17.6%
3	12.5%
4	9.7%
5	7.9%
6	6.7%
7	5.8%
8	5.1%
9	4.6%

The Data

Let's see if we can observe the pattern in real-world dataset: the populations of the cities, boroughs, and townships in Pennsylvania.

Wikipedia doesn't give us a way to export table data, but I found a good third-party tool that will generate CSV files from a Wikipedia article. I used this tool to extract and save the data in the file pa-cities.csv.

```
city,population
Philadelphia,1550542
Pittsburgh,303255
Allentown,124880
Reading city,94903
Erie,92957
Upper Darby,84893
Scranton,75805
Lower Merion,64157
Bensalem,62800
Abington,58451
Bethlehem city,58349
Lancaster city,57153
Lower Paxton,54807
Bristol township,53897
Millcreek township,53101
Haverford,50503
Harrisburg,50012
Middletown township,45634
York city,44867
Manheim township,44265
Wilkes-Barre city,44254
Altoona,42788
Hempfield township,41613
State College,40687
Northampton township,39872
```

Reading digits

We can use the `csv:` import scheme to import the population data as a Pointless table.

```
cities = import "csv:pa-cities.csv"
```

cities

city	population
Philadelphia	1550542
Pittsburgh	303255
Allentown	124880
Reading city	94903
Erie	92957
Upper Darby	84893
Scranton	75805
Lower Merion	64157
Bensalem	62800
Abington	58451
Bethlehem city	58349
Lancaster city	57153
Lower Paxton	54807
Bristol township	53897
Millcreek township	53101
Haverford	50503
Harrisburg	50012
Middletown township	45634
York city	44867
Manheim township	44265
Wilkes-Barre city	44254

x 2570

This table contains `2570` rows and has two columns: `name` and `population`. We can access the values in the population column as a list.

```
cities.population
```

```
[
  1550542,
  303255,
  124880,
  94903,
  92957,
  84893,
  75805,
  64157,
  62800,
  58451,
  58349,
  57153,
```

Our table is sorted by population, so the first value in the column is the population for Philadelphia, the largest city in the state.

```
cities.population[0]
```

```
1550542
```

Before we can get the first digit of the number, we need to convert it to a string.

```
str.of(cities.population[0])
```

```
"1550542"
```

Next, we'll convert our number string into a list of characters.

```
chars(str.of(cities.population[0]))
```

```
["1", "5", "5", "0", "5", "4", "2"]
```

And get the first digit character from the list.

```
chars(str.of(cities.population[0]))[0]
```

```
"1"
```

Calculating Frequencies

Let's take a moment and refactor our code into pipeline syntax using the `|` operator and `arg` keyword.

```
cities.population[0] | chars(str.of(arg))[0]
```

We'll tweak this code to use the mapping pipeline operator `$` to get the first digit for every population value in the list, rather than just the first population.

```
cities.population $ chars(str.of(arg))[0]
```

```
[
  "1",
  "3",
  "1",
  "9",
  "9",
  "8",
  "7",
  "6",
  "6",
  "5",
  "5",
  "5",
```

Finally, we'll use `list.counts` to get the occurrence count and share for each value.

```
cities.population
  $ chars(str.of(arg))[0]
  | list.counts
```

value	count	share
"1"	802	0.31206225680933850
"2"	463	0.18015564202334630
"3"	304	0.11828793774319066
"4"	257	0.10000000000000000
"5"	196	0.07626459143968872
"6"	149	0.05797665369649806
"8"	143	0.05564202334630350
"7"	136	0.05291828793774319
"9"	120	0.04669260700389105

x 9

Now we have the frequency information (in the `share` column) for each starting digit! We can see that the frequency decreases as the digits increase.

Tidying Up

Let's see if we can make our frequency values a little more readable. To start, we'll go back and store our table in a new variable.

```
digitStats = cities.population
  $ chars(str.of(arg))[0]
  | list.counts
```

We can use this new variable to access the values in the `share` column as a list.

```
digitStats.share
```

```
[
  0.3120622568093385,
  0.1801556420233463,
  0.11828793774319066,
  0.1,
  0.07626459143968872,
  0.05797665369649806,
  0.0556420233463035,
  0.05291828793774319,
  0.04669260700389105,
]
```

Let's use `$` to convert each of these values to a percent, rounded to a single decimal place.

```
digitStats.share $ roundTo(arg * 100, 1)
```

```
[31.2, 18, 11.8, 10, 7.6, 5.8, 5.6, 5.3, 4.7]
```

Finally, we'll put these values back into our table in place of the old `share` column.

```
digitStats.share $= roundTo(arg * 100, 1)
```

digitStats

value	count	share
"1"	802	31.2
"2"	463	18.0
"3"	304	11.8
"4"	257	10.0
"5"	196	7.6
"6"	149	5.8
"8"	143	5.6
"7"	136	5.3
"9"	120	4.7

x 9

Syntax Note

Note that the following two syntax forms are equivalent.

```
digitStats.share = digitStats.share $ roundTo(arg * 100, 1)
```

```
digitStats.share $= roundTo(arg * 100, 1)
```

Our calculated frequencies match the expected values quite well!

Leading Digit	Calculated	Expected
1	31.2%	30.1%
2	18.0%	17.6%
3	11.8%	12.5%
4	10.0%	9.7%
5	7.6%	7.9%
6	5.8%	6.7%
7	5.6%	5.8%
8	5.3%	5.1%
9	4.7%	4.6%

Wrapping Up

Here's the complete code.

```
cities = import "csv:pa-cities.csv"

digitStats = cities.population
  $ chars(str.of(arg))[0]
  | list.counts

digitStats.share $= roundTo(arg * 100, 1)

print(digitStats)
```