Ch6_Working_with_strings_notes

June 18, 2024

1 Chapter 6 - Working with strings

1.1 This chapter will cover

- UTF-8 encoding of Julia strings; byte versus character indexing
- Manipulating strings: interpolation, splitting, using regular expressions, parsing
- Working with symbols
- Using the InlineStrings.jl package to work with fixed-width strings
- Using the PooledArrays.jl package to compress vectors of strings

As an application of string processing, we will analyze movie genres that were given ratings by Twitter users. We want to understand which movie genre is most common and how the relative frequency of this genre changes with the movie year.

We will analyze the movie genre data according to the following steps, which are described in the subsequent sections of this chapter and depicted in figure 6.1:

1. Read in the data in Julia.

2. Parse the original data to extract the year and genre list for each analyzed movie.
3. Create frequency tables to find which movie genre is most common.
4. Create a plot of popularity of the most common genre by year.

An image depicting the steps that we'll be taking

1.2 Download the file

[2]:
```
url = "https://raw.githubusercontent.com/sidooms/MovieTweetings/
↪44c525d0c766944910686c60697203cda39305d6/snapshots/10K/movies.dat"
```

[2]:
```
"https://raw.githubusercontent.com/sidooms/MovieTweetings/44c525d0c766944910686c
60697203cda39305d6/snapshots/10K/movies.dat"
```

[3]:
```
download(url, "movies.dat")
```

[3]:
```
"movies.dat"
```

1.2.1 Basic characteristics of strings in julia

We can interpolate variable in strings by using the '$' operator `println("This is $price American dollers")`

If we are executing a function/operation we must wrap the code inside curly brackets as per `"This price is $(a + price) American dollers"`

The newline character can be embedded within the string literal to print a new line and divide the string up

[6]:
```
print("This is the first\nthis is the second\nthis is the third")
```

```
This is the first
this is the second
this is the third
```

To actually use the dollar sign '$' character so that it's not interpreted as an interpolation, we have to escape it

[8]:
```
print("The price is \$100")
```

```
The price is $100
```

1.2.2 Avoiding complicated escape combinations

In instances in which we want to print a string which contains multiple special characters, which should not be interpolated and interpreted as newlines, instead of embedding escape characters amongst the string literal, we can simply prefix the entire string with **raw**, turning the string into a raw literal. For example this would help greatly if we want to pring file paths and the like

[9]:
```
raw"C:\my_folder\my_file.txt"
```

[9]:
```
"C:\\my_folder\\my_file.txt"
```

The triple quote `"""something"""` is used to create multi-line strings

Let's read the data file line by line

[11]:
```
movies = readlines("movies.dat")
```

[11]:
```
3096-element Vector{String}:
 "0002844::Fantômas - À l'ombre de la guillotine (1913)::Crime|Drama"
 "0007264::The Rink (1916)::Comedy|Short"
 "0008133::The Immigrant (1917)::Short|Comedy|Drama|Romance"
 "0012349::The Kid (1921)::Comedy|Drama|Family"
 "0013427::Nanook of the North (1922)::Documentary"
 "0014142::The Hunchback of Notre Dame (1923)::Drama|Romance"
 "0014538::Three Ages (1923)::Comedy"
 "0014872::Entr'acte (1924)::Short"
 "0015163::The Navigator (1924)::Action|Comedy"
 "0015324::Sherlock Jr. (1924)::Comedy|Fantasy"
 "0015400::The Thief of Bagdad (1924)::Adventure|Family|Fantasy|Romance"
 "0017925::The General (1926)::Action|Adventure|Comedy|Romance|War"
 "0018773::The Circus (1928)::Comedy|Romance"
```

```
 "2638984::Teal Diva (2012)::Documentary|Short"
 "2645104::Romantik komedi 2: Bekarliga veda (2013)::Comedy"
 "2645164::The Hardy Bucks Movie (2013)::Comedy"
 "2646378::The Frankenstein Theory (2013)::Horror|Sci-Fi"
 "2649128::Metro (2013)::Thriller"
 "2670226::Jîn (2013)::Drama"
 "2700330::The Two Escobars (2010)::Documentary"
 "2735466::Pickett's Charge (2012)::Short|Comedy|Drama|Family"
 "2748368::Neil (2013)::Short|Comedy"
 "2750600::A Different Tree (2013)::Short|Drama|Family"
 "2763252::Broken Night (2013)::Short|Drama"
 "2769592::Kiss Shot Truth or Dare (2013)::Short"
```

The file used the "::" charactera as string delimiters, for columns, and then the '|' characters for a within column separator e.g. the genres. We should work on the file and format it to make it more consistent and tidy - data sanitation if you will. A few ideas * Keep the first column as ID * Split the release year from the movie title * Parse the genres into an array

Take the first line - subsampling it

[107]:
```
first_line = first(movies)
```

[107]:
```
"0002844::Fantômas - À l'ombre de la guillotine (1913)::Crime|Drama"
```

What is the type of the data?

[14]:
```
typeof(first_line)
```

[14]:
```
String
```

A string, so we can use split

[37]:
```
first_split = split(first_line, [':', '|', '(', ')']) # split using several␣
 ↪characters
```

[37]:
```
8-element Vector{SubString{String}}:
 "0002844"
 ""
 "Fantômas - À l'ombre de la guillotine "
 "1913"
 ""
 ""
 "Crime"
 "Drama"
```

"You might have noticed that the movies vector has the type Vector{String}, while the movie1_parts vector has the type Vector{SubString{String}}. This is because Julia, for efficiency, when splitting a string with the split function, does not copy the string but instead creates a SubString{String} object that points to the slice of the original string. Having this behavior is safe, as strings in Julia are immutable (we already talked about mutable and immutable types in chapter 4). Therefore, once the string is created, its contents cannot be changed. Creation of a substring of a string is guaranteed to be a safe operation. In your code, if you want to create a SubString{String}, you can use the view function or the @view macro on a String."

[59]:
```
second_split = []
for f in first_split
    if match(r"[A-Za-z0-9]+", f) != nothing
        push!(second_split, f)
    end
end
```

[66]:
```
for f in second_split[4:end]
    println(f)
end
```

```
Crime
Drama
```

[120]:
```
second_split
```

[120]:
```
5-element Vector{Any}:
 "0002844"
 "Fantômas - À l'ombre de la guillotine "
 "1913"
 "Crime"
 "Drama"
```

How might we parse the lines and get a nice map of each column? This is my own function, which doesn't rely on just using regex, as bogumils does, and so it is dirtier and less elegant.

[200]:
```
function parseline_1(line::AbstractString)
    first_split = split(line, [':', '|', '(', ')'])
    second_split = []
    for f in first_split
        if match(r"[A-Za-z0-9]+", f) != nothing
            push!(second_split, f)
        end
    end
    return (id=second_split[1],
            name=second_split[2],
            year=second_split[3],
            genre=second_split[4:end])
end
```

[200]:
```
parseline_1 (generic function with 1 method)
```

[136]:
```
parsed_columns = parseline(first_line)
```

[136]: (id = "0002844", name = "Fantômas - À l'ombre de la guillotine ", year = "1913", genre = Any["Crime", "Drama"])

[151]:
```
parsed_columns.id
```

[151]: "0002844"

Another variant - this will simply print the indices without assigning them to any variable

[152]:
```
function parseline_2(line::AbstractString)
    first_split = split(line, [':', '|', '(', ')'])
    second_split = []
    for f in first_split
        if match(r"[A-Za-z0-9]+", f) != nothing
            push!(second_split, f)
        end
    end
    return (second_split[1],
            second_split[2],
            second_split[3],
            second_split[4:end])
end
```

[152]: parseline_2 (generic function with 1 method)

[ ]:
```

```

[160]:
```
tmp_array = []
for lines in readlines("movies.dat")
    push!(tmp_array, parseline_2(lines))
end
tmp_array
```

[160]:
```
3096-element Vector{Any}:
 ("0002844", "Fantômas - À l'ombre de la guillotine ", "1913", Any["Crime",
"Drama"])
 ("0007264", "The Rink ", "1916", Any["Comedy", "Short"])
 ("0008133", "The Immigrant ", "1917", Any["Short", "Comedy", "Drama",
"Romance"])
 ("0012349", "The Kid ", "1921", Any["Comedy", "Drama", "Family"])
 ("0013427", "Nanook of the North ", "1922", Any["Documentary"])
 ("0014142", "The Hunchback of Notre Dame ", "1923", Any["Drama", "Romance"])
 ("0014538", "Three Ages ", "1923", Any["Comedy"])
 ("0014872", "Entr'acte ", "1924", Any["Short"])
 ("0015163", "The Navigator ", "1924", Any["Action", "Comedy"])
 ("0015324", "Sherlock Jr. ", "1924", Any["Comedy", "Fantasy"])
 ("0015400", "The Thief of Bagdad ", "1924", Any["Adventure", "Family",
"Fantasy", "Romance"])
```

```
 ("0017925", "The General ", "1926", Any["Action", "Adventure", "Comedy",
"Romance", "War"])
 ("0018773", "The Circus ", "1928", Any["Comedy", "Romance"])
 
 ("2638984", "Teal Diva ", "2012", Any["Documentary", "Short"])
 ("2645104", "Romantik komedi 2", " Bekarliga veda ", Any["2013", "Comedy"])
 ("2645164", "The Hardy Bucks Movie ", "2013", Any["Comedy"])
 ("2646378", "The Frankenstein Theory ", "2013", Any["Horror", "Sci-Fi"])
 ("2649128", "Metro ", "2013", Any["Thriller"])
 ("2670226", "Jîn ", "2013", Any["Drama"])
 ("2700330", "The Two Escobars ", "2010", Any["Documentary"])
 ("2735466", "Pickett's Charge ", "2012", Any["Short", "Comedy", "Drama",
"Family"])
 ("2748368", "Neil ", "2013", Any["Short", "Comedy"])
 ("2750600", "A Different Tree ", "2013", Any["Short", "Drama", "Family"])
 ("2763252", "Broken Night ", "2013", Any["Short", "Drama"])
 ("2769592", "Kiss Shot Truth or Dare ", "2013", Any["Short"])
```

1.2.3 Load this into a dataframe using some bogumil magic from

https://stackoverflow.com/questions/72957438/how-to-convert-a-vector-of-vectors-into-a-dataframe-in-julia-without-for-loop

[170]:
```
df_one = DataFrame([getindex.(tmp_array, i) for i in 1:4], :auto,␣
↪copycols=false)
```

[170]:

	x1	x2	x3	x4
	SubStrin…	SubStrin…	SubStrin…	Array…
1	0002844	Fantômas - À l'ombre de la guillotine	1913	["Crime", "Drama"]
2	0007264	The Rink	1916	["Comedy", "Short"]
3	0008133	The Immigrant	1917	["Short", "Comedy", "Drama", "Ro
4	0012349	The Kid	1921	["Comedy", "Drama", "Famil
5	0013427	Nanook of the North	1922	["Documentary"]
6	0014142	The Hunchback of Notre Dame	1923	["Drama", "Romance"]
7	0014538	Three Ages	1923	["Comedy"]
8	0014872	Entr'acte	1924	["Short"]
9	0015163	The Navigator	1924	["Action", "Comedy"]
10	0015324	Sherlock Jr.	1924	["Comedy", "Fantasy"]
11	0015400	The Thief of Bagdad	1924	["Adventure", "Family", "Fantasy", "
12	0017925	The General	1926	["Action", "Adventure", "Comedy", "Ron
13	0018773	The Circus	1928	["Comedy", "Romance"]
14	0019729	The Broadway Melody	1929	["Musical", "Romance"]
15	0019760	Chelovek s kino-apparatom	1929	["Documentary"]
16	0020530	Un chien andalou	1929	["Short", "Fantasy"]
17	0020629	All Quiet on the Western Front	1930	["Drama", "War"]
18	0020902	Free and Easy	1930	["Comedy", "Musical"]
19	0021577	L'âge d'or	1930	["Comedy", "Drama"]
20	0021746	Cimarron	1931	["Drama", "Western"]
21	0021884	Frankenstein	1931	["Horror", "Sci-Fi"]
22	0022958	Grand Hotel	1932	["Drama", "Romance"]
23	0023876	Cavalcade	1933	["Drama", "Romance", "War
24	0023969	Duck Soup	1933	["Comedy", "Musical"]
25	0025132	Forsaking All Others	1934	["Comedy", "Drama", "Roman
26	0025456	Man of Aran	1934	["Documentary"]
27	0025586	Of Human Bondage	1934	["Drama", "Romance"]
28	0025878	The Thin Man	1934	["Comedy", "Crime", "Myster
29	0026752	Mutiny on the Bounty	1935	["Adventure", "Drama", "Histo
30	0027698	The Great Ziegfeld	1936	["Biography", "Drama", "Musical", "
…	…	…	…	…

Jesus, it actually worked? How? Using the broadcasting funtion of get index, we create a comphrension where we iterate through the columns 1 to 4 of the vector, sequentially loading the column, and allowing DataFrames to automatically use these values are input **:auto** option

[89]:
```
column_names = ["id", "title", "year", "genre"]
```

[89]:
```
4-element Vector{String}:
 "id"
 "title"
 "year"
 "genre"
```

[171]:
```
df_two = rename(df_one, column_names)
```

[171]:

	id	title	year	genre
	SubStrin…	SubStrin…	SubStrin…	Array…
1	0002844	Fantômas - À l'ombre de la guillotine	1913	["Crime", "Drama"]
2	0007264	The Rink	1916	["Comedy", "Short"]
3	0008133	The Immigrant	1917	["Short", "Comedy", "Drama", "Ro
4	0012349	The Kid	1921	["Comedy", "Drama", "Famil
5	0013427	Nanook of the North	1922	["Documentary"]
6	0014142	The Hunchback of Notre Dame	1923	["Drama", "Romance"]
7	0014538	Three Ages	1923	["Comedy"]
8	0014872	Entr'acte	1924	["Short"]
9	0015163	The Navigator	1924	["Action", "Comedy"]
10	0015324	Sherlock Jr.	1924	["Comedy", "Fantasy"]
11	0015400	The Thief of Bagdad	1924	["Adventure", "Family", "Fantasy", "
12	0017925	The General	1926	["Action", "Adventure", "Comedy", "Ron
13	0018773	The Circus	1928	["Comedy", "Romance"]
14	0019729	The Broadway Melody	1929	["Musical", "Romance"]
15	0019760	Chelovek s kino-apparatom	1929	["Documentary"]
16	0020530	Un chien andalou	1929	["Short", "Fantasy"]
17	0020629	All Quiet on the Western Front	1930	["Drama", "War"]
18	0020902	Free and Easy	1930	["Comedy", "Musical"]
19	0021577	L'âge d'or	1930	["Comedy", "Drama"]
20	0021746	Cimarron	1931	["Drama", "Western"]
21	0021884	Frankenstein	1931	["Horror", "Sci-Fi"]
22	0022958	Grand Hotel	1932	["Drama", "Romance"]
23	0023876	Cavalcade	1933	["Drama", "Romance", "War
24	0023969	Duck Soup	1933	["Comedy", "Musical"]
25	0025132	Forsaking All Others	1934	["Comedy", "Drama", "Roman
26	0025456	Man of Aran	1934	["Documentary"]
27	0025586	Of Human Bondage	1934	["Drama", "Romance"]
28	0025878	The Thin Man	1934	["Comedy", "Crime", "Myster
29	0026752	Mutiny on the Bounty	1935	["Adventure", "Drama", "Histo
30	0027698	The Great Ziegfeld	1936	["Biography", "Drama", "Musical", "
…	…	…	…	…

[68]:
```julia
using DataFrames
```

[92]:
```julia
sample_df = DataFrame([name => [] for name in column_names])
```

[92]:

id	title	year	genre
Any	Any	Any	Any

1.2.4 How did Bogumil do it in the course?

[231]:
```julia
function parseline(line::AbstractString)
    parts = split(line, "::")
    m = match(r"(.+) \((\d{4})\)", parts[2])
    return (id=parts[1],
            name=m[1],
```

```
            year=parse(Int, m[2]),
            genres=split(parts[3], "|"))
end
```

[231]: parseline (generic function with 1 method)

Instead of creating loops like I did, he split the parts using the '::' separator, and then used a regex to separaterly split the year from the title from the first splitting procedure. After this he assigned the various parts to variables.

1.3 How String indexing is done

1.3.1 UTF-8 encoding of strings in Julia

UTF-8 is a character encoding standard which is most commonly used today. It defines the how the character in encoded in bytes of sizes 1:4, with some characters being encoded with 1 byte, and others with 2, 3 or even 4 bytes. Why is this important and practical? In julia, when performing indexing on a string, julia indexes based on the byte size, rather than on the character size itself. Here is an example

[173]:
```
z = "Fantômas"
```

[173]: "Fantômas"

This string contains a special character on 'o', so taking a look at the literal length of the string, we can see that it's 8 characters long. If we take a look at it's length in UTF-8 encoding, meaning, its byte length using the **codeunits()** function

[181]:
```
length(codeunits(z))
```

[181]: 9

We can see that it's 9 bytes long?! How come? The special 'o' character is represented by two bytes, and so it takes up bytes [5,6]. Try indexing from literal 1:6 and see if it let's you.

[183]:
```
z[1:6]
```

```
StringIndexError: invalid index [6], valid nearby indices [5]=>'ô', [7]=>'m'

Stacktrace:
 [1] string_index_err(s::String, i::Int64)
   @ Base ./strings/string.jl:12
 [2] getindex(s::String, r::UnitRange{Int64})
   @ Base ./strings/string.jl:470
 [3] top-level scope
   @ In[183]:1
```

Nope. Now try from 1:7 and 1:5

[185]:
```
z[1:7]
```

[185]:
```
"Fantôm"
```

[186]:
```
z[1:5]
```

[186]:
```
"Fantô"
```

We can imagine the inconsistencies and problems this can introduce to our code if we rely on indexing strings in this format. **"Therefore, you must always check when using a function to see whether it works with byte indices or character indices. You have already seen that indexing using square brackets uses byte indexing and that the function first uses character counts. In my blog post "The String, or There and Back Again" (http://mng.bz/XaW1), I have created a glossary of the most commonly used functions when working with strings, including the kinds of indexing they use."**

[189]:
```
for i in eachindex(z)
           println(i, ": ", z[i])
       end
```

```
1: F
2: a
3: n
4: t
5: ô
7: m
8: a
9: s
```

The function **first()** with alternatively use character-literal indexing on strings, rather than byte indexing

[191]:
```
first(z, 6)
```

[191]:
```
"Fantôm"
```

"In data science workflows, you will most commonly want to operate on strings using character counts, not byte indexing. Therefore, it is recommended that you do not index into strings using square brackets."

1.3.2 ASCII string

In one case, byte and character indexing are guaranteed to produce the same result. This happens when your string consists only of ASCII characters. The most important examples of such characters are the digits 0 to 9, lowercase letters a to z, uppercase letters A to Z, and common symbols like !, +, -, *,), and (. Generally, any character that can be typed without using meta keys on a standard US keyboard is an ASCII character.

An important feature of ASCII characters is that they are always represented by a single byte in UTF-8 encoding. In Julia, you can easily check whether your string consists of only ASCII characters by using the isascii function:

```
[192]: isascii("abcd")
```

```
[192]: true
```

```
[193]: isascii("  x: x 0")
```

```
[193]: false
```

1.4 Analyzing genre frequency in movies

We'll be creating a basic frequency table, as well and looking at how the top genres change from year to year.

1.4.1 Finding common movie genres

```
[228]: movies_vector = parseline.(movies)
```

```
[228]: 3096-element Vector{@NamedTuple{id::SubString{String}, name::SubString{String},
year::Int64, genres::Vector{SubString{String}}}}:
 (id = "0002844", name = "Fantômas - À l'ombre de la guillotine", year = 1913,
genres = ["Crime", "Drama"])
 (id = "0007264", name = "The Rink", year = 1916, genres = ["Comedy", "Short"])
 (id = "0008133", name = "The Immigrant", year = 1917, genres = ["Short",
"Comedy", "Drama", "Romance"])
 (id = "0012349", name = "The Kid", year = 1921, genres = ["Comedy", "Drama",
"Family"])
 (id = "0013427", name = "Nanook of the North", year = 1922, genres =
["Documentary"])
 (id = "0014142", name = "The Hunchback of Notre Dame", year = 1923, genres =
["Drama", "Romance"])
 (id = "0014538", name = "Three Ages", year = 1923, genres = ["Comedy"])
 (id = "0014872", name = "Entr'acte", year = 1924, genres = ["Short"])
 (id = "0015163", name = "The Navigator", year = 1924, genres = ["Action",
"Comedy"])
 (id = "0015324", name = "Sherlock Jr.", year = 1924, genres = ["Comedy",
"Fantasy"])
 (id = "0015400", name = "The Thief of Bagdad", year = 1924, genres =
["Adventure", "Family", "Fantasy", "Romance"])
 (id = "0017925", name = "The General", year = 1926, genres = ["Action",
"Adventure", "Comedy", "Romance", "War"])
 (id = "0018773", name = "The Circus", year = 1928, genres = ["Comedy",
"Romance"])

 (id = "2638984", name = "Teal Diva", year = 2012, genres = ["Documentary",
"Short"])
 (id = "2645104", name = "Romantik komedi 2: Bekarliga veda", year = 2013,
genres = ["Comedy"])
 (id = "2645164", name = "The Hardy Bucks Movie", year = 2013, genres =
```

```
["Comedy"])
 (id = "2646378", name = "The Frankenstein Theory", year = 2013, genres =
["Horror", "Sci-Fi"])
 (id = "2649128", name = "Metro", year = 2013, genres = ["Thriller"])
 (id = "2670226", name = "Jîn", year = 2013, genres = ["Drama"])
 (id = "2700330", name = "The Two Escobars", year = 2010, genres =
["Documentary"])
 (id = "2735466", name = "Pickett's Charge", year = 2012, genres = ["Short",
"Comedy", "Drama", "Family"])
 (id = "2748368", name = "Neil", year = 2013, genres = ["Short", "Comedy"])
 (id = "2750600", name = "A Different Tree", year = 2013, genres = ["Short",
"Drama", "Family"])
 (id = "2763252", name = "Broken Night", year = 2013, genres = ["Short",
"Drama"])
 (id = "2769592", name = "Kiss Shot Truth or Dare", year = 2013, genres =
["Short"])
```

Create a single vector containing all of the genres from the movies.dat

[229]:
```julia
all_genres = String[]
```

[229]:
```
String[]
```

[232]:
```julia
for lines in movies_vector
    append!(all_genres, lines.genres)
end
```

[233]:
```julia
all_genres
```

[233]:
```
8121-element Vector{String}:
 "Crime"
 "Drama"
 "Comedy"
 "Short"
 "Short"
 "Comedy"
 "Drama"
 "Romance"
 "Comedy"
 "Drama"
 "Family"
 "Documentary"
 "Drama"

 "Short"
 "Comedy"
 "Drama"
 "Family"
```

```
"Short"
"Comedy"
"Short"
"Drama"
"Family"
"Short"
"Drama"
"Short"
```

Create a basic frequency mapping using the StatsBase function **countmap()**

[226]:
```
using StatsBase
```

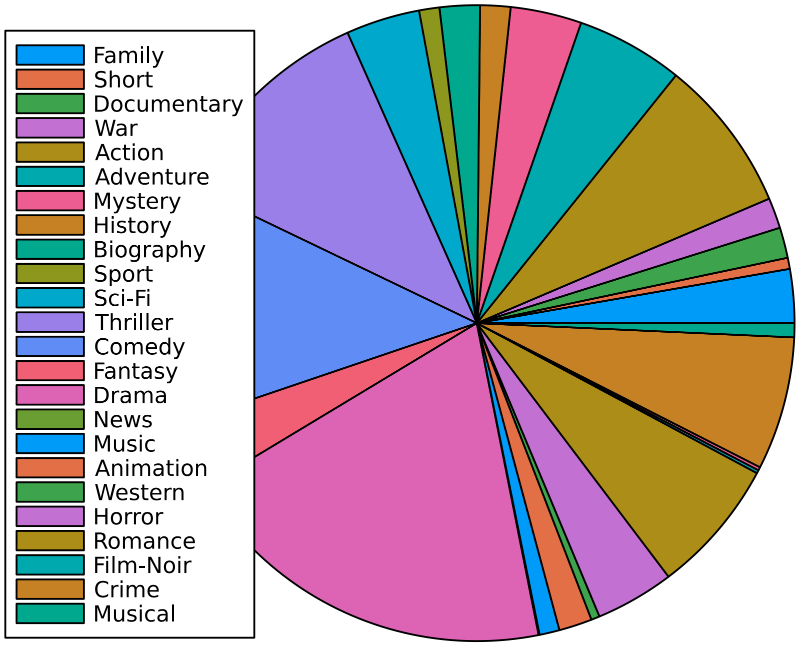
[234]:
```
countmap(all_genres)
```

[234]:
```
Dict{String, Int64} with 25 entries:
  "Family"      => 222
  "Short"       => 46
  "Documentary" => 126
  "War"         => 126
  "Action"      => 635
  "Adventure"   => 443
  "Mystery"     => 294
  "History"     => 125
  "Biography"   => 166
  "Sport"       => 84
  "Sci-Fi"      => 305
  "Thriller"    => 910
  "Comedy"      => 1001
  "Fantasy"     => 278
  "Drama"       => 1583
  "News"        => 4
  "Music"       => 83
  "Animation"   => 137
  "Western"     => 35
  "Horror"      => 325
  "Romance"     => 558
  "Film-Noir"   => 13
  ""            => 14
  "Crime"       => 550
  "Musical"     => 58
```

[235]:
```
using Plots
```

[239]:
```
pie(countmap(all_genres))
```

[239]:



Ugly!!!! Not acceptable at all.

Now create a frequency table using the FreqTable package

[ ]:
```
using Pkg; Pkg.add("FreqTables") ; using FreqTables
```

[248]:
```
sort(freqtable(all_genres))
```

[248]:
```
25-element Named Vector{Int64}
Dim1

News             4
Film-Noir       13
                14
Western         35
Short           46
Musical         58
Music           83
Sport           84
History        125
Documentary    126
War            126

Fantasy        278
```

```
Mystery        294
Sci-Fi         305
Horror         325
Adventure      443
Crime          550
Romance        558
Action         635
Thriller       910
Comedy        1001
Drama         1583
```

Pay attention to the frequency table Type - the Named Vector (NamedArray). Each type has it's own idiosyncracies and thus factor into design considerations

1.4.2 Does genre frequency change much across years?

Let's take a look. We'll look at creating binary variables for absence/presence of a movie in a particular year, and then proportionality tables which calculate percentages for us

Extract the year of each movie to a vector

[250]:
```julia
years = [line.year for line in movies_vector]
```

[250]:
```
3096-element Vector{Int64}:
 1913
 1916
 1917
 1921
 1922
 1923
 1923
 1924
 1924
 1924
 1924
 1926
 1928
 ⋮
 2012
 2013
 2013
 2013
 2013
 2013
 2010
 2012
 2013
 2013
 2013
```

```
2013
```

[252]:
```
has_drama = ["Drama" in line.genres for line in movies_vector]
```

[252]:
```
3096-element Vector{Bool}:
 1
 0
 1
 1
 0
 1
 0
 0
 0
 0
 0
 0
 0

 0
 0
 0
 0
 0
 1
 0
 1
 0
 1
 1
 0
```

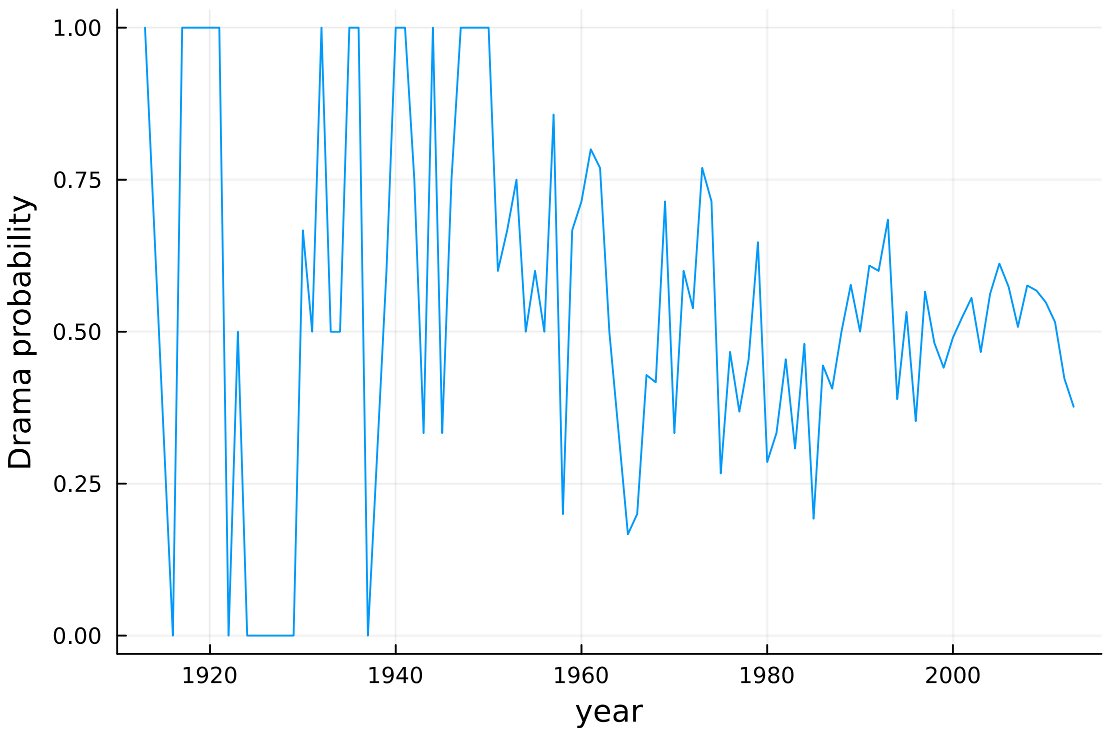
[255]:
```
drama_prop = proptable(years, has_drama; margins=1)
```

[255]:
```
93×2 Named Matrix{Float64}
Dim1   Dim2      false      true

1913               0.0       1.0
1916               1.0       0.0
1917               0.0       1.0
1921               0.0       1.0
1922               1.0       0.0
1923               0.5       0.5
1924               1.0       0.0
1926               1.0       0.0
1928               1.0       0.0
1929               1.0       0.0
```

```
1930          0.333333  0.666667

2003          0.533333  0.466667
2004          0.438095  0.561905
2005          0.387931  0.612069
2006          0.426357  0.573643
2007          0.492063  0.507937
2008          0.424051  0.575949
2009          0.432432  0.567568
2010          0.451754  0.548246
2011          0.484472  0.515528
2012          0.577017  0.422983
2013          0.623529  0.376471
```

[260]:
```
plot(names(drama_prop, 1), drama_prop[:, 2]; legend=false,
         xlabel="year", ylabel="Drama probability")
```

[260]:



1.4.3 Exercise 6.1

Create a plot of the number of movies by year, using the years variable.

- Create a dict mapping year to number of drama movies?
- Subset the original movies vector to include only rows containing drama as a genre? then just count number of times each years appears?

[278]:
```
#dict = Dict()
#for line in movies_vector
#    Dict#
#
#    countmap(movies_vector.year)
```

[272]:
```
all_years = Int64[]
```

[272]: Int64[]

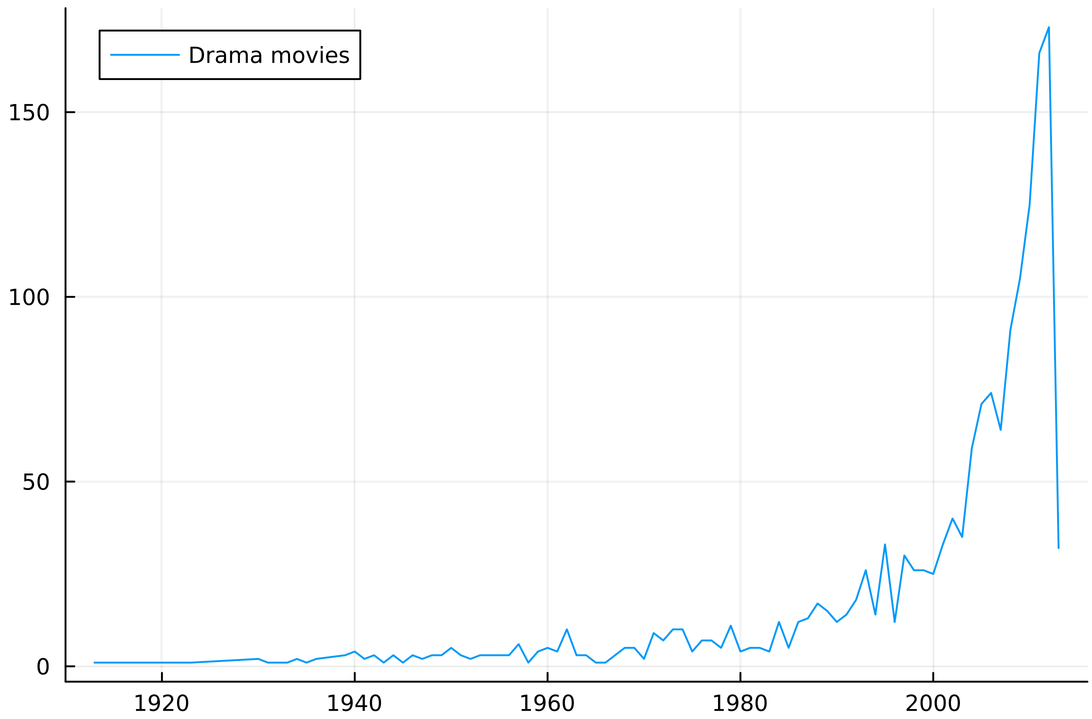
[273]:
```
for line in movies_vector
    if "Drama"   line.genres
        append!(all_years, line.year)
    end
end
```

[276]:
```
mapped_years = countmap(all_years)
```

[276]:
```
Dict{Int64, Int64} with 86 entries:
  1923 => 1
  1985 => 5
  1953 => 3
  2004 => 59
  1956 => 3
  2002 => 40
  1952 => 2
  1963 => 3
  1964 => 3
  1967 => 3
  1948 => 3
  2001 => 33
  1943 => 1
  2013 => 32
  1965 => 1
  1942 => 3
  1991 => 14
  1976 => 7
  1944 => 3
  1941 => 2
  1957 => 6
  1988 => 17
  2008 => 91
  1955 => 3
  1932 => 1
       =>
```

[304]:
```
plot(mapped_years, label = "Drama movies")
```

[304]:



Overall total movies produced on top of this graph

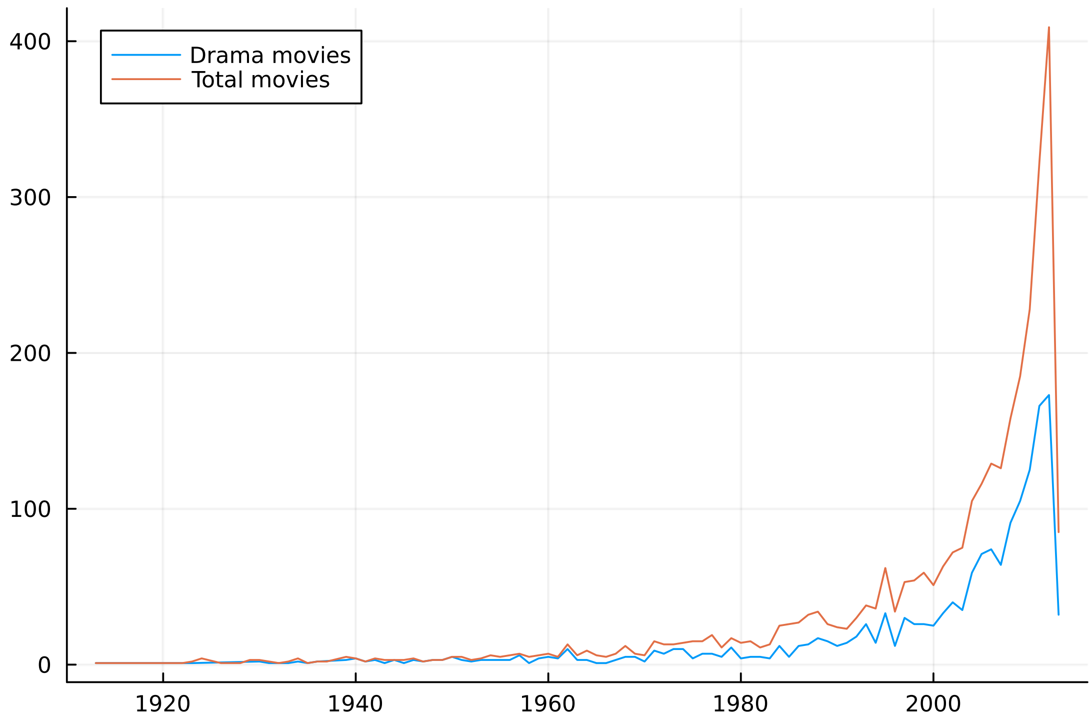
[301]:
```
all_movies = []
```

[301]: Any[]

[302]:
```
for lines in movies_vector
    append!(all_movies, lines.year)
end
```

[305]:
```
plot!(countmap(all_movies), labels = "Total movies")
```

[305]:



1.5 Using Symbols

Symbols are specific Types in Julia which have several performance advantages when used in the correct context. Symbols are stored within Julia in a 'pool' and indexed for the remainder of the session. We don't use them for much aside from comparisons for equality, and so they can be used conveniently in data science workflows as non-mutable labels-category tags and so on.

Create a simple symbol

[308]:
```
s1 = Symbol("x")
```

[308]: :x

Notice the ":" prefix of the symbol

This will concatenate the two elements into a single symbol

[309]:
```
s3 = Symbol("x", 1) #
```

[309]: :x1

We can also easily create one by simply writing `:x1`, but this wont work on intergers/numbers, and there must not be any spaces between strings

[312]:
```
:x1
```

[312]: :x1

[314]:
```
typeof(:1)
```

[314]: Int64

Symbols may look like Strings but they are not, their supertype is not AbstractString, so all the operations we can perform on Strings will not work.

[316]:
```
supertype(Symbol)
```

[316]: Any

"The point is that an equality comparison for symbols is fast, much faster than testing strings for equality. The next listing shows a simple benchmark, in which we look for a value in a vector of one million elements."

[317]:
```
using BenchmarkTools
```

[318]:
```
str = string.("x", 1:10^6)
```

[318]:
```
1000000-element Vector{String}:
 "x1"
 "x2"
 "x3"
 "x4"
 "x5"
 "x6"
 "x7"
 "x8"
 "x9"
 "x10"
 "x11"
 "x12"
 "x13"

 "x999989"
 "x999990"
 "x999991"
 "x999992"
 "x999993"
 "x999994"
 "x999995"
 "x999996"
 "x999997"
 "x999998"
 "x999999"
 "x1000000"
```

[319]:
```julia
symb = Symbol.(str)
```

[319]:
```
1000000-element Vector{Symbol}:
 :x1
 :x2
 :x3
 :x4
 :x5
 :x6
 :x7
 :x8
 :x9
 :x10
 :x11
 :x12
 :x13

 :x999989
 :x999990
 :x999991
 :x999992
 :x999993
 :x999994
 :x999995
 :x999996
 :x999997
 :x999998
 :x999999
 :x1000000
```

[320]:
```julia
@btime "x" in $str
```

```
  2.612 ms (0 allocations: 0 bytes)
```

[320]:
```
false
```

[321]:
```julia
@btime :x in $symb
```

```
  323.069  s (0 allocations: 0 bytes)
```

[321]:
```
false
```

Indexing into a symbol vector is much quicker - performing lookups and so forth.

You might ask how this is achieved. The trick is that Julia internally keeps a global pool of all symbols. If you introduce a new Symbol, Julia first checks whether it is already present in this pool, and if so, Julia reuses it. Therefore, when you compare two symbols, you can compare their address in memory without having to check their content.

Because the symbols are stored in this pool of memory until the end of the session,

if we create many many symbols, this can cause memory disturbances, and so this should be kept in mind

1.6 Fixed with string types for improved performance

I'm sure that we've all created a dataframe in which the values of a particular variable only take up a fixed number of characters, say, postcodes which are 5 digits long, or isbn10 numbers which are 10 long, or even addresses which maybe 120 characters long at most. In these instances, we can restrict the length of the String Type to this predefine BYTE limit, and thus maximize performance. Remember, it is indexed based on byte length in UTF-8, not literal character length.

There are several fixed-with String types: 1, 3, 7, 15, 31, 63, 127 and 255.

The package InlineStrings can help us work with these strings in a nice and clean way

[323]:
```
Pkg.add("InlineStrings")
```

```
   Resolving package versions…
    Updating `~/.julia/environments/v1.10/Project.toml`
  [842dd82b] + InlineStrings v1.4.0
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[324]:
```
using InlineStrings
```

Let's allow the **InlineString()** and **inlinestrings** function to automatically detect the appropriate StringN length

[325]:
```
s1 = InlineString("x")
```

[325]:
```
"x"
```

[326]:
```
typeof(s1)
```

[326]:
```
String1
```

[327]:
```
sv = inlinestrings(["The", "quick", "brown", "fox", "jumps",
                    "over", "the", "lazy", "dog"])
```

[327]:
```
9-element Vector{String7}:
 "The"
 "quick"
 "brown"
 "fox"
 "jumps"
 "over"
 "the"
 "lazy"
 "dog"
```

We can see that s1 requires only a String1, whereas sv requires a String7, due to the varying sizes of the different elements.

Let's do an experiment to showcase the performance advantages. We'll create a long vector of random strings of length 3, and then another vector of the same strings converted to String3 types, and then we'll sort the and do some benchmarks.

[328]:
```julia
using Random
```

[329]:
```julia
string_basic = [randstring(3) for i in 1:10^6]
```

[329]:
```
1000000-element Vector{String}:
 "HFC"
 "WV7"
 "PCf"
 "H2X"
 "h6Q"
 "gwL"
 "ZhG"
 "KRK"
 "opj"
 "0z0"
 "25v"
 "zbH"
 "647"
 
 "si1"
 "m1N"
 "Zgb"
 "xG8"
 "ayx"
 "iC0"
 "Z1m"
 "TkR"
 "Uqu"
 "WHB"
 "Uks"
 "hgq"
```

[330]:
```julia
string_three = inlinestrings(string_basic)
```

[330]:
```
1000000-element Vector{String3}:
 "HFC"
 "WV7"
 "PCf"
 "H2X"
 "h6Q"
 "gwL"
 "ZhG"
 "KRK"
 "opj"
```

```
 "OzO"
 "25v"
 "zbH"
 "647"
 ⋮
 "si1"
 "m1N"
 "Zgb"
 "xG8"
 "ayx"
 "iC0"
 "Z1m"
 "TkR"
 "Uqu"
 "WHB"
 "Uks"
 "hgq"
```

[334]:
```
@btime sort(string_three)
```

```
  6.740 ms (6 allocations: 7.65 MiB)
```

[334]:
```
1000000-element Vector{String3}:
 "000"
 "000"
 "000"
 "000"
 "000"
 "000"
 "001"
 "001"
 "001"
 "002"
 "002"
 "002"
 "003"
 ⋮
 "zzx"
 "zzx"
 "zzx"
 "zzy"
 "zzy"
 "zzy"
 "zzy"
 "zzz"
 "zzz"
 "zzz"
 "zzz"
```

```
"zzz"
```

[333]:
```
@btime sort(string_basic)
```

```
216.979 ms (4 allocations: 15.26 MiB)
```

[333]:
```
1000000-element Vector{String}:
 "000"
 "000"
 "000"
 "000"
 "000"
 "000"
 "001"
 "001"
 "001"
 "002"
 "002"
 "002"
 "003"

 "zzx"
 "zzx"
 "zzx"
 "zzy"
 "zzy"
 "zzy"
 "zzy"
 "zzz"
 "zzz"
 "zzz"
 "zzz"
 "zzz"
```

What is the memory footprint?

[336]:
```
Base.summarysize(string_basic)
```

[336]:
```
19000040
```

[335]:
```
Base.summarysize(string_three)
```

[335]:
```
4000040
```

19000040 vs 4000040

2 Exercise 6.2

Using the s1 vector from listing 6.7, create the s3 vector consisting of symbols representing the same strings contained in the s1 vector. Next, benchmark how fast you can sort the s3 vector. Finally, benchmark how fast you can de-duplicate the s1, s2, and s3 vectors by using the unique function.

```
string_basic = [randstring(3) for i in 1:10^6];
string_three = inlinestrings(string_basic)
```

[337]:
```
s3 = Symbol.(string_basic)
```

[337]:
```
1000000-element Vector{Symbol}:
 :HFC
 :WV7
 :PCf
 :H2X
 :h6Q
 :gwL
 :ZhG
 :KRK
 :opj
 :OzO
 Symbol("25v")
 :zbH
 Symbol("647")

 :si1
 :m1N
 :Zgb
 :xG8
 :ayx
 :iCO
 :Z1m
 :TkR
 :Uqu
 :WHB
 :Uks
 :hgq
```

[338]:
```
@btime sort(s3)
```

```
  165.970 ms (4 allocations: 15.26 MiB)
```

[338]:
```
1000000-element Vector{Symbol}:
 Symbol("000")
 Symbol("000")
 Symbol("000")
 Symbol("000")
```

```
 Symbol("000")
 Symbol("000")
 Symbol("001")
 Symbol("001")
 Symbol("001")
 Symbol("002")
 Symbol("002")
 Symbol("002")
 Symbol("003")
 ⋮
 :zzx
 :zzx
 :zzx
 :zzy
 :zzy
 :zzy
 :zzy
 :zzz
 :zzz
 :zzz
 :zzz
 :zzz
```

[339]:
```
@btime unique(string_basic)
```

```
  40.564 ms (49 allocations: 10.46 MiB)
```

[339]:
```
234722-element Vector{String}:
 "HFC"
 "WV7"
 "PCf"
 "H2X"
 "h6Q"
 "gwL"
 "ZhG"
 "KRK"
 "opj"
 "OzO"
 "25v"
 "zbH"
 "647"
 ⋮
 "ZSo"
 "x45"
 "uqK"
 "x6i"
 "qTL"
 "YvQ"
```

```
"CLx"
"zK1"
"Anp"
"LBh"
"DdD"
"s8g"
```

[341]:
```julia
@btime unique(string_three)
```

```
22.712 ms (48 allocations: 6.16 MiB)
```

[341]:
```
234722-element Vector{String3}:
 "HFC"
 "WV7"
 "PCf"
 "H2X"
 "h6Q"
 "gwL"
 "ZhG"
 "KRK"
 "opj"
 "OzO"
 "25v"
 "zbH"
 "647"

 "ZSo"
 "x45"
 "uqK"
 "x6i"
 "qTL"
 "YvQ"
 "CLx"
 "zK1"
 "Anp"
 "LBh"
 "DdD"
 "s8g"
```

[340]:
```julia
@btime unique(s3)
```

```
16.843 ms (49 allocations: 10.46 MiB)
```

[340]:
```
234722-element Vector{Symbol}:
 :HFC
 :WV7
 :PCf
 :H2X
```

```
:h6Q
:gwL
:ZhG
:KRK
:opj
:OzO
Symbol("25v")
:zbH
Symbol("647")

:ZSo
:x45
:uqK
:x6i
:qTL
:YvQ
:CLx
:zK1
:Anp
:LBh
:DdD
:s8g
```

2.1 Compressing vectors of Strings with PooledArrays.jl

We may be working with large datasets consisting of millions of rows of data, typically stored in a text file as a vector - vectors of strings. When in the millions of rows, with potentially multiple files, storage-memory considerations need to be accounted for under the banner of efficiency. Storing our vector of strings as pooled arrays is a good option

Create a large text file with a million rows

[342]:
```
open("iris.txt", "w") do io
    for i in 1:10^6
        println(io, "Iris setosa")
        println(io, "Iris virginica")
        println(io, "Iris versicolor")
    end
end
```

Now let's open the file

[354]:
```
isfile("iris.txt")
```

[354]:
```
true
```

[355]:
```
uncompressed = readlines("iris.txt")
```

[355]:
```
3000000-element Vector{String}:
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"

 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
```

Load up PooledArrays

[356]:
```
Pkg.add("PooledArrays") ; using PooledArrays
```

```
   Resolving package versions…
    Updating `~/.julia/environments/v1.10/Project.toml`
  [2dfb63ee] + PooledArrays v1.4.3
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Load in the uncompressed vector as a PooledArray

[357]:
```
compressed = PooledArray(uncompressed)
```

[357]:
```
3000000-element PooledVector{String, UInt32, Vector{UInt32}}:
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
```

```
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"

 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
 "Iris setosa"
 "Iris virginica"
 "Iris versicolor"
```

What are there respective memory footprints?

[358]:
```
Base.summarysize(uncompressed)
```

[358]: 88000040

[359]:
```
Base.summarysize(compressed)
```

[359]: 12000600

Now that's a lot smaller

[360]:
```
12000600 / 88000040
```

[360]: 0.13637039255891248

13% of the uncompressed size - impressive

But how is this magic functioning - how's it doing it's job?

[ ]:
```

```

[ ]:
```

```