In [1]:
```
print("Week 4:
Working with Data frames and it's functions.
Data Frames:
A Data Frame is (simplistically) just a dictionary whose keys are the var
iables, and whose columns are the values associated with each. But Data F
rames are a little bit more than this: the columns are all the same lengt
h, so they need to incorporate the idea of missing data, and there are ma
ny operators defined on top of the standard dictionary operators.
The upshot is that in one sense Data Frames are really just advanced tabl
es, but Data Frames are a little cleverer than a simple 2D array of data.
    a. They can contain different types of data in different columns.
    b. Unlike a simple 2D array, they have column headings (the keys), an
d data can be indexed by these, which means adding a column to a dataset
is easy, and often doesn't require changes to code.
    c. They naturally map to the way we often store data (e.g., in CSV or
spreadsheet files), so input and output are easy.
    d. They allow for missing data, which is endemic in real datasets, in
a natural and (hopefully) efficient manner.
    e. We can perform joins across them.
    f. Data Frames match naturally to the philosophy of Tidy Data
With the CSV package it is very easy to read in a CSV file into a Data Fr
ame. For instance, download the file movie_sequence.csv,")
```

```
Week 4:
Working with Data frames and it's functions.
Data Frames:
A Data Frame is (simplistically) just a dictionary whose keys are the va
riables, and whose columns are the values associated with each. But Data
Frames are a little bit more than this: the columns are all the same len
gth, so they need to incorporate the idea of missing data, and there are
many operators defined on top of the standard dictionary operators.
The upshot is that in one sense Data Frames are really just advanced tab
les, but Data Frames are a little cleverer than a simple 2D array of dat
a.
    a. They can contain different types of data in different columns.
    b. Unlike a simple 2D array, they have column headings (the keys), a
nd data can be indexed by these, which means adding a column to a datase
t is easy, and often doesn't require changes to code.
    c. They naturally map to the way we often store data (e.g., in CSV o
r spreadsheet files), so input and output are easy.
    d. They allow for missing data, which is endemic in real datasets, i
n a natural and (hopefully) efficient manner.
    e. We can perform joins across them.
    f. Data Frames match naturally to the philosophy of Tidy Data
With the CSV package it is very easy to read in a CSV file into a Data F
rame. For instance, download the file movie_sequence.csv,
```

In [2]:
```
# link to download datase:https://github.com/mroughan/AlephZeroHeroesData
/blob/master/MarvelCinematicUniverse/movie_sequence.csv
```

In [ ]:
```
import Pkg; Pkg.add("DataFrames")
```

```
    Updating registry at `C:\Users\sony\.julia\registries\General.toml`
```

In [ ]:
```
using Pkg
```

In [ ]:
```
Pkg.add("CSV")
```

In [3]:
```
using DataFrames
using CSV
```

In [4]:
```
movies = CSV.read("C:/Users/sony/Downloads/movie_sequence.csv", DataFram
e)
```

Out[4]: 28×7 DataFrame *3 rows omitted*

Row	Code	Phase	ReleaseYear	SettingYear	SettingMonth	Title	Notes
	String1	Int64	Int64?	Int64	String15?	String	String?
1	a	1	2008	2010	October	Iron Man	*missing*
2	b	1	2010	2011	May	Iron Man 2	Nick Fury's big week
3	c	1	2008	2011	May	The Incredible Hulk	Nick Fury's big week
4	d	1	2011	2011	May	Thor	Nick Fury's big week
5	e	1	2011	2012	April	Captain America: The First Avenger	note that this is the modern part of the movie
6	f	1	2012	2012	May	The Avengers	*missing*
7	g	2	2013	2012	December	Iron Man 3	flash backs aren't included in the time line
8	h	2	2013	2013	November	Thor: The Dark World	*missing*
9	i	2	2014	2014	April	Captain America: The Winter Soldier	*missing*
10	j	2	2014	2014	August	Guardians of the Galaxy	*missing*
11	k	3	2017	2014	October	Guardians of the Galaxy Vol. 2	not in release sequence
12	l	2	2015	2015	May	Avengers: Age of Ultron	*missing*
13	m	2	2015	2015	July	Ant-Man	*missing*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
17	q	3	2016	2016	May	Doctor Strange	*happens over 2016-2017, so month is arbitrary*
18	r	3	2017	2017	June	Thor: Ragnarok	*missing*
19	s	3	2018	2017	July	Avengers: Infinity War	*missing*
20	t	3	2018	2017	July	Ant-Man and the Wasp	in parallel with Infinity War

Row	Code	Phase	ReleaseYear	SettingYear	SettingMonth	Title	Notes
	String1	Int64	Int64?	Int64	String15?	String	String?
21	u	4	2019	2018	missing	Captain Marvel	*release date March 8th, 2019*
22	v	4	2019	2018	missing	Avengers: Endgame	*release date April 26th, 2019*
23	w	4	2019	2018	missing	Spider-Man: Far From Home	release date July 5th, 2019
24	A	4	missing	2019	*missing*	Guardians of the Galaxy Vol. 3	hypothetical
25	B	4	missing	2019	*missing*	Black	hypothetical

In [ ]:
```
#or you can access csv file as below
file= "C:/Users/sony/Downloads/movie_sequence.csv"
movies = CSV.read(file, DataFrame)
```

In [10]:
```
#It should display a table of showing the contents of the CSV file. For i
nstance, we can extract a set of columns using
```

In [9]:
```
movies.Code
```

Out[9]:
```
28-element Vector{String1}:
 "a"
 "b"
 "c"
 "d"
 "e"
 "f"
 "g"
 "h"
 "i"
 "j"
 "k"
 "l"
 "m"
 ⋮
 "q"
 "r"
 "s"
 "t"
 "u"
 "v"
 "w"
 "A"
 "B"
 "C"
 "D"
 "E"
```

In [15]:
```
function replace_missing!( DF::DataFrame)
    # replace all of the missing strings with blanks, and get rid of lead
ing or trailing spaces
    for (k,s) in eachcol(DF, true)
        if eltype(DF[k]) == Union{Missing, String}
            DF[k] = Array{Union{Missing, String},1}(String.(strip.(coales
ce.(s, ""))))
        end
    end
end
```

Out[15]:
```
replace_missing! (generic function with 1 method)
```

In [18]:
```
print("Arrays")
```

```
Arrays
```

In [19]:
```
arr = [1,2,3]
```

Out[19]:
```
3-element Vector{Int64}:
 1
 2
 3
```

In [20]:
```
array = Array{Int64}(undef, 3)
```

Out[20]:
```
3-element Vector{Int64}:
 1643202293288
 1643202451944
 1643202322072
```

In [21]:
```
array = Array{Int64}(undef, 3, 3, 3)
```

Out[21]:
```
3×3×3 Array{Int64, 3}:
[:, :, 1] =
 1643241712144  1643241712192  1643202155248
 1643241712160  1643241712208  1643202153712
 1643241712176  1643241712224  1643241712640

[:, :, 2] =
 140704102297936  140704102297936  1643202156016
 140704102297936    1643202156208  1643202154224
 140704102297936    1643202155952  1643202151280

[:, :, 3] =
   1643202156528  140704102297936  140704102297936
   1643202156336  140704102297936  140704102297936
 140704102297936    1643241712416  140704102297936
```

In [24]:
```
#-Arrays of anything Julia gives us the freedom to create arrays with ele
ments of different types. Let us see theexample below in which we are goi
ng to create array of an odd mixture numbers, strings,functions, constant
s-#
[1, "TutorialsPoint.com", 5.5, tan, pi]
```

Out[24]:
```
5-element Vector{Any}:
 1
  "TutorialsPoint.com"
 5.5
  tan (generic function with 13 methods)
 π = 3.1415926535897...
```

In [25]:
```
#Empty Arrays
A = Int64[]
```

Out[25]:
```
Int64[]
```

In [26]:
```
A = String[]
```

Out[26]:
```
String[]
```

In [27]:
```
#2D arrays & matrices
[1 2 3 4 5 6 7 8 9 10]
```

Out[27]:
```
1×10 Matrix{Int64}:
 1  2  3  4  5  6  7  8  9  10
```

In [28]:
```
[1 2 3 4 5 ; 6 7 8 9 10]
```

Out[28]:
```
2×5 Matrix{Int64}:
 1  2  3  4   5
 6  7  8  9  10
```

In [29]:
```
#Creating arrays using range objects
#Collect() function
collect(1:5)
```

Out[29]:
```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

In [30]:
```
#floating
collect(1.5:5.5)
```

Out[30]:
```
5-element Vector{Float64}:
 1.5
 2.5
 3.5
 4.5
 5.5
```

In [31]:
```
#Array Constructor
Array{Int64}(undef, 5)
```

Out[31]:
```
5-element Vector{Int64}:
 1
 1
 0
 0
 0
```

In [32]:
```
#Arrays of arrays
Array[1:5,6:10]
```

Out[32]:
```
2-element Vector{Array}:
 [1, 2, 3, 4, 5]
 [6, 7, 8, 9, 10]
```

In [33]:
```
#Copying arrays
#Suppose you have an array and want to create another array with similar
dimensions, then you
 A = collect(1:5);
#Here we have hide the values with the help of semicolon(;)
 B = similar(A)
```

Out[33]:
```
5-element Vector{Int64}:
 140704116898320
 140704155291216
               0
               0
               0
```

In [34]:
```
#Matrix Operations
Array[[3,4],[5,6]]
```

Out[34]:
```
2-element Vector{Array}:
 [3, 4]
 [5, 6]
```

In [35]:
```
#Adding Elements
#We can add elements to an array in Julia at the end, at the front and at
the given index using
#push!(), pushfirst!() and splice!() functions respectively.
#At the end
push!(arr,55)
```

Out[35]:
```
4-element Vector{Int64}:
  1
  2
  3
 55
```

In [36]:
```
#At the front
pushfirst!(arr,0)
```

Out[36]:
```
5-element Vector{Int64}:
  0
  1
  2
  3
 55
```

In [37]:
```
#At a given index
#We can use splice!() function to add an element into an array at a given
index. For example,
splice!(arr,2:5,2:6)
```

Out[37]:
```
4-element Vector{Int64}:
  1
  2
  3
 55
```

In [38]:
```
#Removing Elements
#Remove the last element
#We can use pop!() function to remove the last element of an array. For e
xample,
pop!(arr)
```

Out[38]:
```
6
```

In [39]:
```
#Removing the first element
#We can use popfirst!() function to remove the first element of an array.
For example,
 popfirst!(arr)
```

Out[39]:
```
0
```

In [41]:
```
#Removing element at given position
#We can use splice!() function to remove the element from a given positio
n of an array. For example,
splice!(arr,3)
```

Out[41]:
```
4
```

In [ ]:
```

```