Julia File Formats

https://github.com/jamescormack/Julia-IO-Workshop

Contents:

- Working with CSV Files
- Working with Excel Files
- Working with JSON
- Working with HTML/XML
- Databases
- Cloud API's
- Images
- Finding Packages

Working with CSV Files

CSV files look like this:

```
Animal,Colour,Cover
bunny,white,fur
dragon,green,scales
cow,brown,fur
pigeon,grey,feathers
```

```julia
# Create a CSV file

#import Pkg; Pkg.add("DataFrames"); Pkg.add("CSV"); Pkg.add("StringEncodings")
using CSV, DataFrames, StringEncodings

# specify CSV content for demo
my_content ="""Animal,Colour,Cover
bunny,white,fur
dragon,green,scales
cow,brown,fur
pigeon,grey,feathers"""

# Write this content to animals.csv
open("animals.csv", "w") do out_file
    # write will return the number of bytes written to the file
    write(out_file, my_content)
end
```

```
90
```

```
# Reading a CSV

animals = CSV.read("animals.csv", DataFrame)

@show typeof(animals)
animals
```

```
typeof(animals) = DataFrame
```

4 rows × 3 columns

	Animal	Colour	Cover
	String	**String**	**String**
1	bunny	white	fur
2	dragon	green	scales
3	cow	brown	fur
4	pigeon	grey	feathers

```
# Writing CSV from DataFrames

var = DataFrame(a = ["Aval", "Bval"], b =[1, 2], c=[3, 4])

open("test.csv", "w") do io
    CSV.write(io, var)
end
```

```
IOStream(<file test.csv>)
```

```
# Reading from CSV using iterator

reader = CSV.File("test.csv")
for row in reader
    println("Values: $(row.a), $(row.c)")
end
```

```
Values: Aval, 3
Values: Bval, 4
```

```
# Multiple ways of reading CSV into DataFrames (inc. Angus' examples from
previous weeks)

df = DataFrame(CSV.File("test.csv"))
#OR
df = CSV.read("test.csv", DataFrame)
#OR
df = CSV.File("test.csv", delim=",", quotechar='"', header=1) |> DataFrame

# Note: CSV read will also guess parameters for you if you don't specify the
parameters
```

2 rows × 3 columns

	a	b	c
	String	**Int64**	**Int64**
1	Aval	1	3
2	Bval	2	4

```
# No headers in CSV
CSV.read("test.csv", DataFrame, header=false)
```

3 rows × 3 columns

	Column1	Column2	Column3
	String	**String**	**String**
1	a	b	c
2	Aval	1	3
3	Bval	2	4

```
# Manually specified header names

CSV.read("test.csv", DataFrame, header=["Animal", "Colour", "Cover"])
```

3 rows × 3 columns

	Animal	Colour	Cover
	String	String	String
1	a	b	c
2	Aval	1	3
3	Bval	2	4

```
# My Convenience function, look away, nothing to see here, whats that next item? Reading booleans you say? Wow, looks interesting! Maybe you should look over there?

function write_string(path, x)
    open(path, "w") do out_file
        write(out_file, x)
    end
end

# Create my animals.csv for next demo
my_animals = """Animal,Colour,Cover,Liked
bunny,white,fur,Y
dragon,green,scales,Y
cow,brown,fur,N
pigeon,grey,feathers,N
pegasus,white,"feathers,fur",Y"""

write_string("animals_like.csv", my_animals);
```

```
# Reading booleans (not booleans in resultant DataFrame)

using CSV; using DataFrames;
animals_table = CSV.read("animals_like.csv", DataFrame)

# Notice that the Liked column is still a string.
```

5 rows × 4 columns

	Animal	Colour	Cover	Liked
	String	String	String	String
1	bunny	white	fur	Y
2	dragon	green	scales	Y
3	cow	brown	fur	N
4	pigeon	grey	feathers	N
5	pegasus	white	feathers,fur	Y

```
# Now read while setting truestrings and falsestrings (booleans in resultant
DataFrame)

animals_table2 = CSV.read(
  "animals_like.csv", DataFrame,
  truestrings=["Y"],
  falsestrings=["N"])

# Much better; Liked column is now a bool
```

5 rows × 4 columns

	Animal	Colour	Cover	Liked
	String	String	String	Bool
1	bunny	white	fur	1
2	dragon	green	scales	1
3	cow	brown	fur	0
4	pigeon	grey	feathers	0
5	pegasus	white	feathers,fur	1

```
# Reading floats and Ints

# Create another demo file...
my_animals_price = """Animal,Colour,Price,Liked
bunny,white,10.5,Y
dragon,green,9,Y
cow,brown,23.55,N
```

```
pigeon,grey,0,N
pegasus,white,999,Y"""
write_string("animals_price.csv", my_animals_price);


using CSV; using DataFrames;
animals_table = CSV.read("animals_price.csv", DataFrame)

# Notice it is clever enough to figure out that these values are floats
# (or Ints if they are all integers)

# Can specifiy decimal delimiter if you are pulling European formats for
instance
#something = CSV.read("something.csv", DataFrame, delim=';', decimal=',')
```

5 rows × 4 columns

	Animal	Colour	Price	Liked
	String	**String**	**Float64**	**String**
1	bunny	white	10.5	Y
2	dragon	green	9.0	Y
3	cow	brown	23.55	N
4	pigeon	grey	0.0	N
5	pegasus	white	999.0	Y

CSVFiles Package

CSV equivalent of FileIO. Provides load() and save() support for CSV files under FileIO.

```
#using Pkg; Pkg.add("CSVFiles")
using CSVFiles, DataFrames

save("data.csv", df)
df2 = DataFrame(load("data.csv"))

# Can also do it similarly using pipes
df = load("data.csv") |> DataFrame
df2 |> save("data.csv")
```

2 rows × 3 columns

	a	b	c
	String	**Int64**	**Int64**
1	Aval	1	3
2	Bval	2	4

```
# Can also work directly on gzipped files to save space.

# save as a gzipped csv (note the format"CSV" specifies that it is CSV file
regardless of extension)
save(File(format"CSV", "data.csv.gz"), df)

# Load gzipped csv directly into dataframe
df2 = DataFrame(load(File(format"CSV", "data.csv.gz")))
```

2 rows × 3 columns

	a	b	c
	String	**Int64**	**Int64**
1	Aval	1	3
2	Bval	2	4

```
# Can load directly from a URL
df3 =
DataFrame(load("https://people.sc.fsu.edu/~jburkardt/data/csv/addresses.csv"))
```

5 rows × 6 columns (omitted printing of 2 columns)

	John	Doe	120 jefferson st.	Riverside
	String	**String**	**String**	**String**
1	Jack	McGinnis	220 hobo Av.	Phila
2	John "Da Man"	Repici	120 Jefferson St.	Riverside
3	Stephen	Tyler	7452 Terrace "At the Plaza" road	SomeTown
4		Blankman		SomeTown
5	Joan "the bone", Anne	Jet	9th, at Terrace plc	Desert City

Working with Excel

Reading from Excel

```
#using Pkg; Pkg.add("XLSX")
import XLSX

xf = XLSX.readxlsx("ExcelFile.xlsx")
```

```
XLSXFile("ExcelFile.xlsx") containing 4 Worksheets
            sheetname size          range
-------------------------------------------------
            TestSheet 5x6           A1:F5
            MainSheet 1x1           A1:A1
               Sheet2 5x2           A1:B5
               Sheet3 1x1           A1:A1
```

```
@show xf["Dog"] # get cell or range by name

@show xf["Sheet2!A2:B4"] # get range explicitly

@show xf["Sheet2!A:B"] # Column ranges are also supported
```

```
xf["Dog"] = "Dog"
xf["Sheet2!A2:B4"] = Any["Rabbit" 4; "Dog" 4; "Fish" 0]
xf["Sheet2!A:B"] = Any["Animal" "Legs"; "Rabbit" 4; "Dog" 4; "Fish" 0; "Human" 2]
```

```
5×2 Matrix{Any}:
 "Animal"   "Legs"
 "Rabbit"  4
 "Dog"     4
 "Fish"    0
 "Human"   2
```

```
@show XLSX.sheetnames(xf) # list all sheets

sh = xf["Sheet2"] # get a reference to a Worksheet

@show sh[2, 1] # access element "B2" (2nd row, 2nd column)

@show sh["A2"] # you can also use the cell name

@show sh["A2:B4"] # or a cell range

@show sh[:] # all data inside a worksheet's dimension
```

```
XLSX.sheetnames(xf) = ["TestSheet", "MainSheet", "Sheet2", "Sheet3"]
sh[2, 1] = "Rabbit"
sh["A2"] = "Rabbit"
sh["A2:B4"] = Any["Rabbit" 4; "Dog" 4; "Fish" 0]
sh[:] = Any["Animal" "Legs"; "Rabbit" 4; "Dog" 4; "Fish" 0; "Human" 2]
```

```
5×2 Matrix{Any}:
 "Animal"   "Legs"
 "Rabbit"  4
 "Dog"     4
 "Fish"    0
 "Human"   2
```

```
XLSX.readdata("ExcelFile.xlsx", "Sheet2", "A2:B4") # shorthand for all above
```

```
3×2 Matrix{Any}:
 "Rabbit"   4
 "Dog"      4
 "Fish"     0
```

```julia
# To see the structure of the excel file

columns, labels = XLSX.readtable("ExcelFile.xlsx", "Sheet2")

@show labels
@show columns
```

```
labels = [:Animal, :Legs]
columns = Any[Any["Rabbit", "Dog", "Fish", "Human"], Any[4, 4, 0, 2]]
```

```
2-element Vector{Any}:
 Any["Rabbit", "Dog", "Fish", "Human"]
 Any[4, 4, 0, 2]
```

```julia
# Excel and DataFrames

using DataFrames, XLSX

df = DataFrame(XLSX.readtable("ExcelFile.xlsx", "Sheet2")...)
```

4 rows × 2 columns

	Animal	Legs
	Any	Any
1	Rabbit	4
2	Dog	4
3	Fish	0
4	Human	2

```julia
# Cache disabled => Always read from disk
# enable_cache=false is good for spreadsheets that are too big for memory
```

```
XLSX.openxlsx("ExcelFile.xlsx", enable_cache=false) do f
  sheet = f["Sheet2"]
  for r in XLSX.eachrow(sheet)

    # r is a `SheetRow`, values are read
    # using column references
    rn = XLSX.row_number(r) # `SheetRow` row number
    v1 = r[1]   # will read value at column 1
    v2 = r[2]   # will read value at column 2
    v3 = r["B"]
    v4 = r[3]
    println("v1=$v1, v2=$v2, v3=$v3, v4=$v4")
  end
end
```

```
v1=Animal, v2=Legs, v3=Legs, v4=missing
v1=Rabbit, v2=4, v3=4, v4=missing
v1=Dog, v2=4, v3=4, v4=missing
v1=Fish, v2=0, v3=0, v4=missing
v1=Human, v2=2, v3=2, v4=missing
```

Writing to Excel

```
XLSX.openxlsx("ExcelFile.xlsx", mode="rw") do xf  # mode="w" for brand new blank file

    sheet = xf["Sheet3"]

    XLSX.rename!(sheet, "new_sheet")

    sheet["A1"] = "this"
    sheet["A2"] = "is"
    sheet["A3"] = "new data"
    sheet["A4"] = 100

    # will add a row from "A5" to "E5"
    sheet["A5"] = collect(1:5) # equivalent to `sheet["A5", dim=2] = collect(1:5)`

    # will add a column from "B1" to "B4"
    sheet["B1", dim=1] = collect(1:4)

    # will add a matrix from "A7" to "C9"
    sheet["A7:C9"] = [ 1 2 3 ; 4 5 6 ; 7 8 9 ]

    XLSX.rename!(sheet, "Sheet3")
end
```

```
# Writing from dataframes

using Dates
import DataFrames, XLSX
df = DataFrames.DataFrame(integers=[1, 2, 3, 4], strings=["Hey", "You", "Out",
"There"], floats=[10.2, 20.3, 30.4, 40.5], dates=[Date(2018,2,20),
Date(2018,2,21), Date(2018,2,22), Date(2018,2,23)], times=[Dates.Time(19,10),
Dates.Time(19,20), Dates.Time(19,30), Dates.Time(19,40)], datetimes=
[Dates.DateTime(2018,5,20,19,10), Dates.DateTime(2018,5,20,19,20),
Dates.DateTime(2018,5,20,19,30), Dates.DateTime(2018,5,20,19,40)])

# To write to a new spreadsheet
## Writetable(filename, vector of columns, vector of names,
overwrite(optional), sheetname(optional))
#XLSX.writetable("ExcelFile.xlsx",
#    collect(DataFrames.eachcol(df)),
#    DataFrames.names(df),
#    overwrite=true,
#    sheetname="TestSheet")

# To modify existing spreadsheet
XLSX.openxlsx("ExcelFile.xlsx", mode="rw") do xf
    sheet = xf["NewSheet"]
    XLSX.writetable!(sheet,
            DataFrames.eachcol(df),
            DataFrames.names(df))
end
```

```
# Writing multiple structures into two sheets

df1 = DataFrames.DataFrame(COL1=[10,20,30], COL2=["Fist", "Sec", "Third"])
df2 = DataFrames.DataFrame(AA=["aa", "bb"], AB=[10.1, 10.2])
XLSX.writetable("ExcelFile2.xlsx", REPORT_A=( collect(DataFrames.eachcol(df1)),
DataFrames.names(df1) ), REPORT_B=( collect(DataFrames.eachcol(df2)),
DataFrames.names(df2) ))
```

Working with JSON

This is what JSON looks like:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

```julia
#using Pkg; Pkg.add("JSON3")
using JSON3

# Create a JSON string
json_string = """{"a": 1, "b": "hello, world"}"""

json_object = JSON3.read(json_string)

# can access the fields with dot or bracket notation
println(json_object.b)
println(json_object["a"])
```

```
hello, world
1
```

```
"{\"a\":1,\"b\":\"hello, world\"}"
```

```julia
# Write JSON out
JSON3.write(json_object)
```

```
"{\"a\":1,\"b\":\"hello, world\"}"
```

```julia
# Pretty print
JSON3.pretty(JSON3.write(json_object))
```

```
{
   "a": 1,
   "b": "hello, world"
}
```

```
# Read and write from/to a file

open("file.json", "w+") do io
    JSON3.pretty(io, json_object)  # pretty print rather than just write
end

json_string = read("file.json", String)

json_object = JSON3.read(json_string)
```

```
JSON3.Object{Base.CodeUnits{UInt8, String}, Vector{UInt64}} with 2 entries:
  :a => 1
  :b => "hello, world"
```

Working with HTML/XML

There are a number of XML packages. The most recommended one seems to be EzXML. LightXML seems to be another popular package.

This is what XML looks like:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

```
#using Pkg; Pkg.add("EzXML")
using EzXML

# Parse an XML string
# (use `readxml(<filename>)` to read a document from a file).
doc = parsexml("""
<primates>
    <genus name="Homo">
        <species name="sapiens">Human</species>
    </genus>
    <genus name="Pan">
        <species name="paniscus">Bonobo</species>
        <species name="troglodytes">Chimpanzee</species>
    </genus>
</primates>
""")

# Get the root element from `doc`.
primates = root(doc)  # or `doc.root`

# Iterate over child elements.
for genus in eachelement(primates)
    # Get an attribute value by name.
    genus_name = genus["name"]
    println("- ", genus_name)
    for species in eachelement(genus)
        # Get the content within an element.
        species_name = nodecontent(species)  # or `species.content`
        println("  └ ", species["name"], " (", species_name, ")")
    end
end
```

```
- Homo
  └ sapiens (Human)
- Pan
  └ paniscus (Bonobo)
  └ troglodytes (Chimpanzee)
```

```
# Find texts using XPath query.
for species_name in nodecontent.(findall("//primates/genus/species/text()",
primates))
    println("- ", species_name)
end
```

```
- Human
- Bonobo
- Chimpanzee
```

Other Formats

- YAML
- TOML

https://gohugohq.com/howto/toml-json-yaml-comparison/

JSON (JavaScript Object Notation):

```json
{
    "date" : "2016-12-14T21:27:05.454Z",
    "publishdate" : "2016-12-14T21:27:05.454Z",
    "title" : "Deep dive into TOML, JSON and YAML",
    "tags" : ["toml","yaml","json", "front matter"],
    "type" : "article",
    "amp" : {
        "elements" : []
    },
    "article" : {
        "lead" : "Lorem ipsum.",
        "category" : "frontmatter",
        "related" : []
    },
    "sitemap" : {
      "changefreq" : "monthly",
      "priority" : 0.5,
      "filename" : "sitemap.xml"
    }
}
```

YAML (YAML Ain't Markup Language):

```yaml
---
date: '2016-12-14T21:27:05.454Z'
publishdate: '2016-12-14T21:27:05.454Z'
title: Deep dive into TOML, JSON and YAML
tags:
- toml
- yaml
- json
- front matter
type: article
amp:
  elements: []
article:
  lead: Lorem ipsum.
  category: frontmatter
  related: []
sitemap:
  changefreq: monthly
  priority: 0.5
  filename: sitemap.xml
```

```
---
```

TOML (Tom's Obvious Markup Language):

```
+++
date = "2016-12-14T21:27:05.454Z"
publishdate = "2016-12-14T21:27:05.454Z"

title = "Deep dive into TOML, JSON and YAML"
tags = ["toml","yaml","json", "front matter"]

type = "article"

[amp]
    elements = []

[article]
    lead = "Lorem ipsum."
    category = "frontmatter"
    related = []

[sitemap]
  changefreq = "monthly"
  priority = 0.5
  filename = "sitemap.xml"
+++
```

Databases

- SQLite
- MySQL (MariaSQL)
- Postgres
- ODBC, JDBC, Mongo,... (the list goes on)

```
# SQLite (simple file based DB)
#import Pkg; Pkg.add("SQLite")

using SQLite

# Open demo database
db = SQLite.DB("Chinook_Sqlite.sqlite")

SQLite.tables(db)

# Reading into dataframe
df = DBInterface.execute(db,
    "SELECT FirstName, LastName FROM Employee") |> DataFrame
```

8 rows × 2 columns

	FirstName	LastName
	String	**String**
1	Andrew	Adams
2	Nancy	Edwards
3	Jane	Peacock
4	Margaret	Park
5	Steve	Johnson
6	Michael	Mitchell
7	Robert	King
8	Laura	Callahan

```
# Writing from dataframe
df |> SQLite.load!(db, "NewEmployee")
```

```
"NewEmployee"
```

```
# Check that write happened
DBInterface.execute(db,
    "SELECT FirstName, LastName FROM NewEmployee") |> DataFrame
```

8 rows × 2 columns

	FirstName	**LastName**
	String	**String**
1	Andrew	Adams
2	Nancy	Edwards
3	Jane	Peacock
4	Margaret	Park
5	Steve	Johnson
6	Michael	Mitchell
7	Robert	King
8	Laura	Callahan

Cloud API's

AWS, Azure, Google Cloud etc.

Example code:

```
# Access an Amazon S3 bucket using AWS.jl
(https://github.com/JuliaCloud/AWS.jl)
import Pkg; Pkg.add("AWS")

using AWS.AWSServices: s3

df = s3("GET", "https://my-bucket.s3.us-west-2.amazonaws.com/datafile") |>
DataFrame
```

Working with Images

```
#import Pkg; Pkg.add("Images"); Pkg.add("ImageView")
using Images, ImageView

img_path = "testimage.png"

img = load(img_path)

imshow(img)
```

```
Dict{String, Any} with 4 entries:
  "gui"         => Dict{String, Any}("window"=>GtkWindowLeaf(name="", parent,
w…
  "roi"         => Dict{String, Any}("redraw"=>64: "map(f-mapped image, input-
1…
  "annotations" => 34: "input-14" = Dict{UInt64, Any}() Dict{UInt64, Any}
  "clim"        => nothing
```

Finding Packages

- JuliaHub (https://juliahub.com)
- JuliaObserver (https://juliaobserver.com)
- Julia.jl (https://github.com/svaksha/Julia.jl)
- Awesome Julia (https://github.com/greister/Awesome-Julia)