

# XML and JSON

Department of Biostatistics and Bioinformatics

Steve Pittard [wsp@emory.edu](mailto:wsp@emory.edu)

October 21, 2015

We'll talk about XML and JSON which are two important formats used in/on the Web for exchange of information. Parsing web information will usually involve processing one or both of these document types

- XML is EXtensible Markup Language
- It organizes documents in a way that can be read by both humans and computers
- It is commonly used for exchange of data over the Internet
- It is also the default format for popular word processing programs such as MS Word and Open Office.
- XML is designed to be self-descriptive
- XML separates data from presentation

# XML - Google

Services like Google's GeoCoding service can return XML.

<https://developers.google.com/maps/documentation/geocoding/intro>

```
- <GeocodeResponse>
  <status>OK</status>
  - <result>
    <type>street_address</type>
    - <formatted_address>
      1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
    </formatted_address>
    - <address_component>
      <long_name>1600</long_name>
      <short_name>1600</short_name>
      <type>street_number</type>
    </address_component>
    - <address_component>
      <long_name>Amphitheatre Parkway</long_name>
      <short_name>Amphitheatre Pkwy</short_name>
      <type>route</type>
    </address_component>
    - <address_component>
```

# XML - NCBI

NCBI's PubMed service also supports XML formatted output

The screenshot shows the NCBI PubMed website interface. The search bar contains the text "Pittard WS". Below the search bar, there are links for "Create RSS", "Create alert", and "Advanced". On the left side, there are navigation links for "Article types", "Text availability", "PubMed Commons", and "Trending articles". A dropdown menu is open, showing the "Format" options: "Summary" (selected), "Summary (text)", "Abstract", "Abstract (text)", "MEDLINE", "XML", and "PMID List". The search results display a list of articles, with the first article being "automated pipeline for improved feature detection and downstream ge-scale, non-targeted metabolomics data." by Uppal K, Soltow QA, Strobel FH, Pittard WS, Gernert KM, Yu T, Jones DP. The article is from BMC Bioinformatics, 2013 Jan 16;14:15. doi: 10.1186/1471-2105-14-15. The PMID is 23323971. There are links for "Free PMC Article" and "Similar articles".

www.ncbi.nlm.nih.gov/pubmed/?term=Pittard+WS

pubmed

NCBI Resources How To

PubMed.gov

US National Library of Medicine  
National Institutes of Health

PubMed

Pittard WS

Create RSS Create alert Advanced

Article types

Clinical Trial

Review

Customize ...

Text availability

Abstract

Free full text

Full text

PubMed Commons

Reader comments

Trending articles

Format

- ☒ Summary
- ☐ Summary (text)
- ☐ Abstract
- ☐ Abstract (text)
- ☐ MEDLINE
- ☐ XML
- ☐ PMID List

Summary 20 per page Sort by Most Recent Send to:

[automated pipeline for improved feature detection and downstream ge-scale, non-targeted metabolomics data.](#)

Uppal K, Soltow QA, Strobel FH, **Pittard WS**, Gernert KM, Yu T, Jones DP.  
BMC Bioinformatics. 2013 Jan 16;14:15. doi: 10.1186/1471-2105-14-15.  
PMID: 23323971 **Free PMC Article**  
[Similar articles](#)

# XML - NCBI

NCBI's PubMed service also supports XML formatted output

<http://www.ncbi.nlm.nih.gov/pubmed/23323971?report=xml&format=text>

```
<PubmedArticle>
```

```
  <MedlineCitation Owner="NLM" Status="MEDLINE">
```

```
    <PMID Version="1">23323971</PMID>
```

```
    <DateCreated>
```

```
      <Year>2013</Year>
```

```
      <Month>02</Month>
```

```
      <Day>04</Day>
```

```
    </DateCreated>
```

```
    <DateCompleted>
```

```
      <Year>2013</Year>
```

```
      <Month>09</Month>
```

```
      <Day>18</Day>
```

```
    </DateCompleted>
```

```
    <DateRevised>
```

```
      <Year>2014</Year>
```

```
      <Month>11</Month>
```

```
      <Day>04</Day>
```

```
    </DateRevised>
```

```
    <Article PubModel="Electronic">
```

```
      <Journal>
```

```
        <ISSN IssnType="Electronic">1471-2105</ISSN>
```

```
        <JournalIssue CitedMedium="Internet">
```

```
          <Volume>14</Volume>
```

# XML - Wunderground

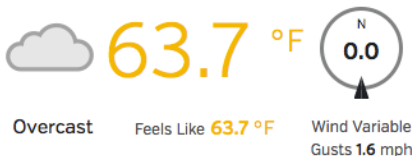
The Wunderground service pulls weather information from Personal Weather Stations located all over the country and world

[www.wunderground.com](http://www.wunderground.com)

## Atlanta, GA [30322] ★ 🏠

☎ Emory/Druid Hills | [Report](#) | [Change Station](#) ▼

Elev 925 ft 33.79 °N, 84.32 °W | Updated 4 min ago



[Rain possible at 5:30pm.](#)

Today is forecast to be **MUCH COOLER** than yesterday.

Today

High 65 | Low 61 °F

Yesterday

High 76.3 | Low 59.5 °F

Pressure	30.05 in
Visibility	1.8 miles
Clouds	Overcast 900 ft
Dew Point	62 °F
Humidity	93%
Rainfall	0.43 in
Snow Depth	Not available.
UV	2 out of 12
Pollen	.90 out of 12
Air Quality	Not available.
Flu Activity	Not available.

SPECI KPDK 251515Z VRB06KT 1  
3/4SM -RA BR OVC009 17/17 A3005  
RMK A02 RAB05 CIG 006V012 P0001  
T01720167

# XML - Wunderground

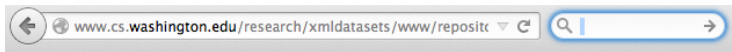
The information from the previous graphic can be obtained in an XML document that could later be parsed for information

```
<current_observation>
  <credit>Weather Underground Personal Weather Station</credit>
  <credit_URL>http://wunderground.com/weatherstation/</credit_URL>
  <image></image>
  <location>
    <full>Emory/Druid Hills, Atlanta, GA</full>
    <neighborhood>Emory/Druid Hills</neighborhood>
    <city>Atlanta</city>
    <state>GA</state>
    <zip/>
    <latitude>33.788582</latitude>
    <longitude>-84.320755</longitude>
    <elevation>925 ft</elevation>
  </location>
  <station_id>KGAATLAN80</station_id>
  <station_type>Ambient Weather WS-2090</station_type>
```

<http://api.wunderground.com/weatherstation/WXCurrentObXML.asp?ID=KGAATLAN80>

# XML - Repositories

There are repositories that provide access to large collections of research information in XML format. We would usually process these using SAX or some other stream-based approach



## XML Data Repository

[Repository Home](#)

---


### Datasets, Details, and Download

#### Protein Sequence Database

Integrated collection of functionally annotated protein sequences.

from [Georgetown Protein Information Resource](#)

Nov 9 2001

filename	DTD	Description	Download	elements
psd7003.xml	 <a href="#">dtd</a>	Protein Sequence Database	<a href="#">.xml</a> (683 MB) <a href="#">.gz</a> (103 MB) <a href="#">.xmi</a> (70 MB)	21305818



# XML - Characteristics - Schemas

Various industries have developed “schemas” for their XML work. The following Wikipedia page lists some of these domains.

[https://en.wikipedia.org/wiki/List\\_of\\_types\\_of\\_XML\\_schemas](https://en.wikipedia.org/wiki/List_of_types_of_XML_schemas)

- 1 Bookmarks
- 2 Brewing
- 3 Business
- 4 Elections
- 5 Engineering
- 6 Financial
- 7 Geographical Information Systems and Geotagging
- 8 Graphical user interfaces
- 9 Humanities texts
- 10 Industrial property
- 11 Libraries
- 12 Math and science
- 13 Metadata
- 14 Music playlists
- 15 News syndication
- 16 Paper and forest products
- 17 Publishing
- 18 Statistics
- 19 Vector images

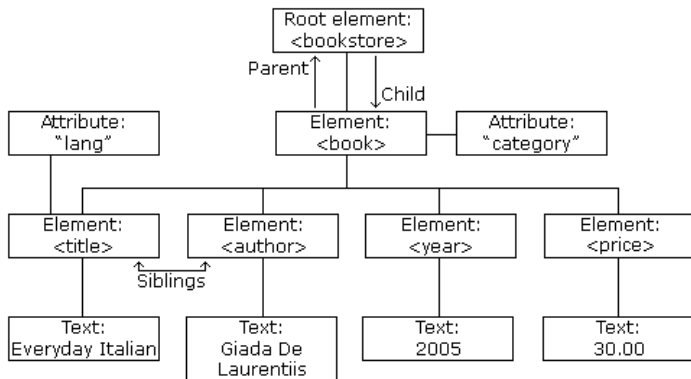
# XML

Here is a simpler XML document that represents books contained in a bookstore inventory. This describes three books

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

# XML

XML is readable by humans (though it's not always easy on the eyes). The indentions show you what elements are subordinate to other elements. We can see that, for example, the “title” tags are subordinate (or are “children”) of the “book” tag.



# XML

- Even though XML is readable by humans it is usually intended for display on the web.
- We use something called XSLT (eXTensible Stylesheet Language Transformation) to “stylize” XML into something that can be nicely displayed on the Web.
- As an example see [http://www.w3schools.com/xml/xml\\_xsl.asp](http://www.w3schools.com/xml/xml_xsl.asp)
- A second use for XML is “consumption” by software packages that know how to parse XML to enable the extraction of only those tags of interest.
- For example we might write code to extract just the names and prices of the books. Or we might just want to calculate how many books there are in the document.

- Since we aren't web designers we won't delve too much into XSLT although we will examine a subset of it to help us “parse” documents.
- Most major programming languages have addons that exist specifically for parsing XML documents.
- For very large XML documents we use something called SAX (Simple API Parser for XML) which is API that processes each part of an XML document sequentially and can respond to “events” using predefined “handlers”.
- This works very well for large documents and is very efficient.
- For smaller XML documents we use an approach involving XPath which processes the document as a whole - that is it loads the entire document into memory first and then we can process it using XPATH expressions.

# XML - Characteristics - DTD

- XML contains tags of our own definition. This is the power of XML. We create our own structure and as long as we specify the expected format other users can use the resulting document reliably.
- They can validate our XML document against our standard using something called a DTD. This is a “document type definition” that establishes the accepted components of an XML document.
- Using DTD documents means a work group can adopt a common standard for information exchange.
- DTDs allow us to assure that an XML document is “well formed”
- Use of a DTD is optional but encouraged

# XML - Characteristics

Let's look at the first part of previous XML doc and introduce some formal terminology

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

The first line is the declaration. Anything that begins with a “<” and ends with a “>” is called a tag. Things enclosed within the tags are usually the content also known as “elements”. Tags can also have “attributes”. In the above example the “title” tag has an attribute called “lang” and the “book” tag has an attribute called “category”.

# XML - Characteristics

Let's look at the first part of previous XML doc and introduce some formal terminology

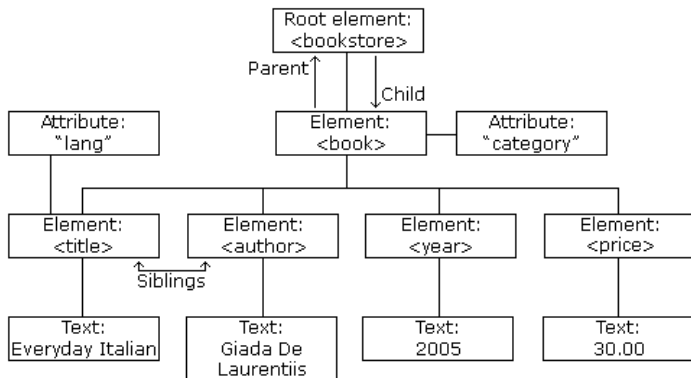
```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

The first line is the declaration. Anything that begins with a “<” and ends with a “>” is called a tag. Things enclosed within the tags are usually the content also known as “elements”. Tags can also have “attributes”. In the above example the “title” tag has an attribute called “lang” and the “book” tag has an attribute called “category”.



# XML - Characteristics

Check out the graphic we saw earlier that shows us explicitly how the document is organized. Note that the indentation tells us what tags are “children” of other tags. This becomes important when we want to extract say just the element values associated with the “book” tags.



# XML - Parsing

- Visually we can easily parse this document and simply read off any information we want.
- But what if the XML document was much larger and had thousands or hundreds of thousands of books in the inventory.
- We would need a way to programmatically address this. As we mentioned previously we will use XPath to help us inspect a document to extract the information we want.
- In effect we can compute values from the document
- XPath is a language that is implemented by packages such as the R XML package <https://cran.r-project.org/web/packages/XML/>

# XML - XPath

The Firefox browser has an addon called XPath Checker that will help us come up to speed on using XPath expressions.

This will help later when we use the XML package with R to parse “real world” XML documents.

<https://addons.mozilla.org/en-US/firefox/addon/xpath-checker/>



***XPath Checker*** 0.4.4.1-signed

by [Brian Slesinsky](#)

An interactive editor for XPath expressions.

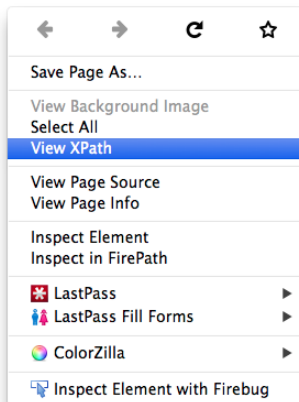
**+ Add to Firefox**

# XML - XPath

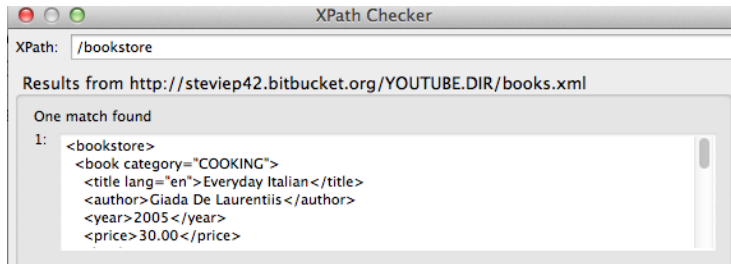
- Once you have installed the plugin and restarted Firefox we can load up an XML file. Let's take the simple book example from above.
- Use the following URL to load your own copy.  
`http://stevie42.bitbucket.org/YOUTUBE.DIR/books.xml`
- Once you have this file loaded in your browser then right-click anywhere in the browser and select "View XPath" from the resulting menu.

# XML - XPath

```
- <bookstore>
- <book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
- <book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
- <book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```



# XML - XPath



# XML - XPath

We can use path expressions to access or select nodes from with an XML document.

Expression	Description
<i>nodename</i>	Selects all nodes with the name " <i>nodename</i> "
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

[http://www.w3schools.com/xsl/xpath\\_syntax.asp](http://www.w3schools.com/xsl/xpath_syntax.asp)

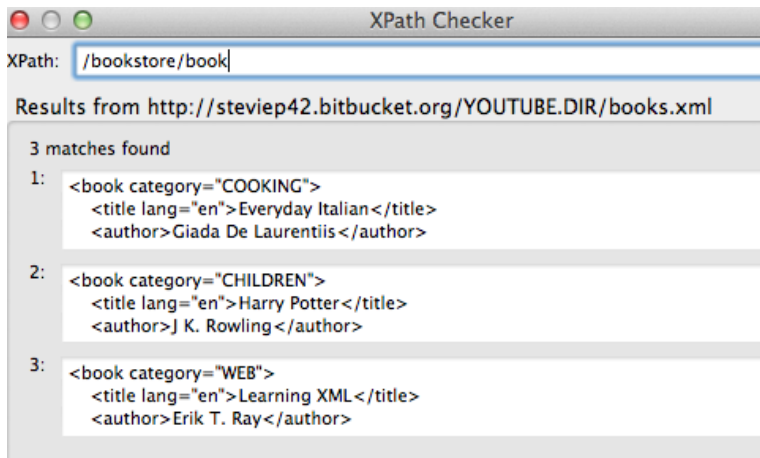
# XML - XPath

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore  <b>Note:</b> If the path starts with a slash ( / ) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

[http://www.w3schools.com/xsl/xpath\\_syntax.asp](http://www.w3schools.com/xsl/xpath_syntax.asp)



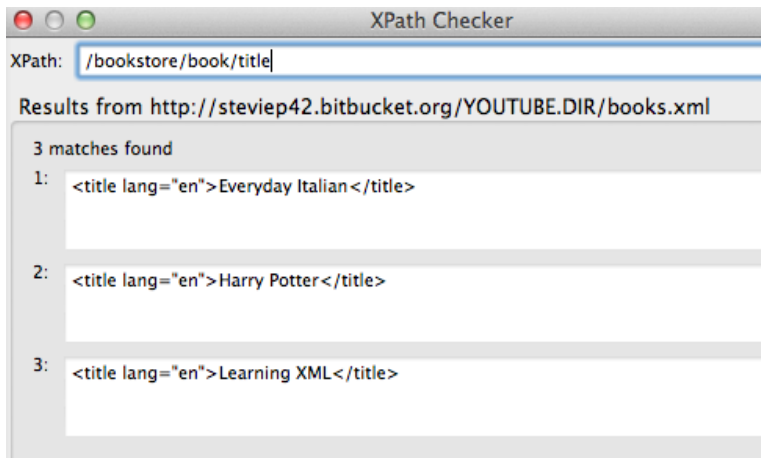
# XML - XPath



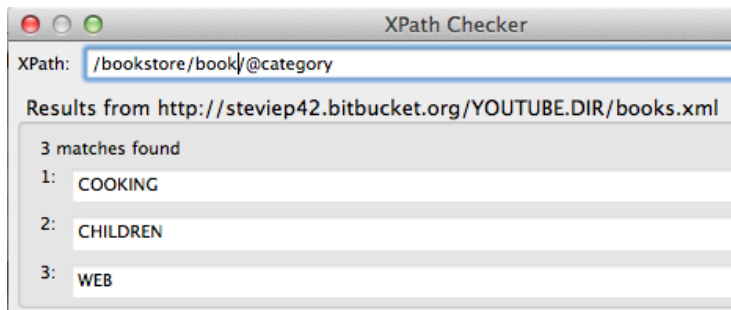
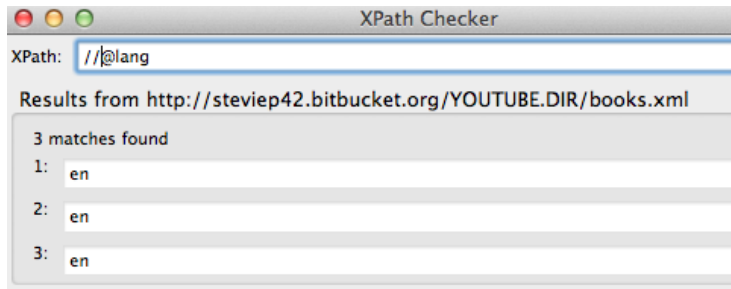
The screenshot shows a window titled "XPath Checker". Inside, there is a text input field labeled "XPath:" containing the expression `/bookstore/book`. Below the input field, it says "Results from <http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml>". Underneath, a section titled "3 matches found" lists three XML elements:

- 1: `<book category="COOKING">`  
    `<title lang="en">Everyday Italian</title>`  
    `<author>Giada De Laurentiis</author>`
- 2: `<book category="CHILDREN">`  
    `<title lang="en">Harry Potter</title>`  
    `<author>J K. Rowling</author>`
- 3: `<book category="WEB">`  
    `<title lang="en">Learning XML</title>`  
    `<author>Erik T. Ray</author>`

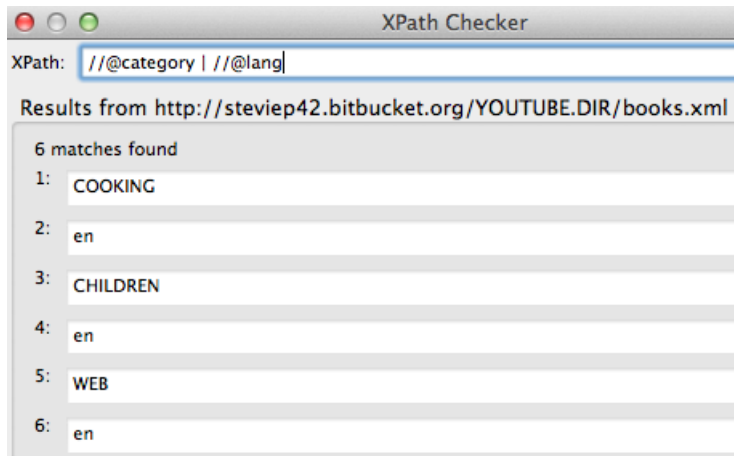
# XML - XPath



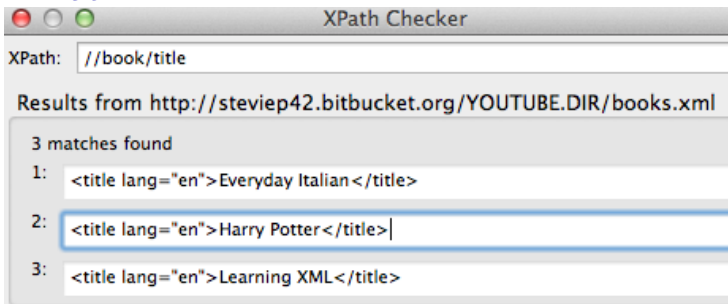
# XML - XPath



# XML - XPath



# XML - XPath

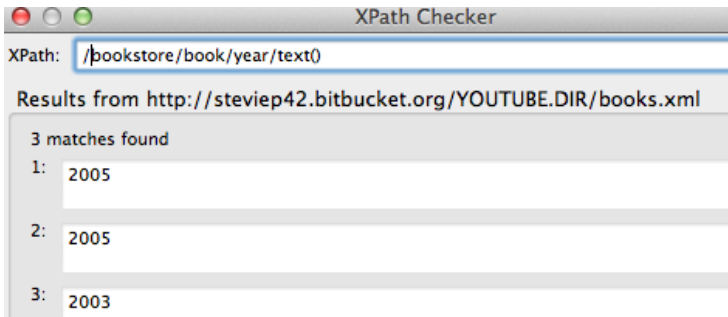


XPath: `//book/title`

Results from <http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml>

3 matches found

- 1: `<title lang="en">Everyday Italian</title>`
- 2: `<title lang="en">Harry Potter</title>`
- 3: `<title lang="en">Learning XML</title>`



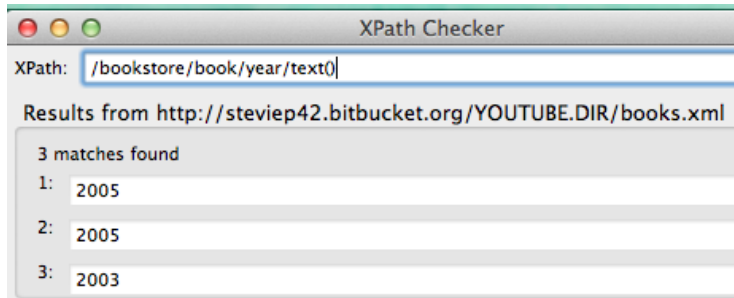
XPath: `/bookstore/book/year/text()`

Results from <http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml>

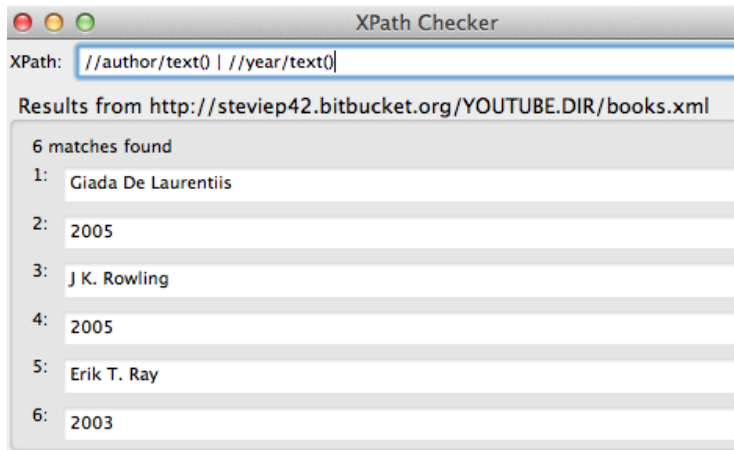
3 matches found

- 1: `2005`
- 2: `2005`
- 3: `2003`

# XML - XPath



# XML - XPath



# XML - XPath

XPath Checker

XPath: `/bookstore/book/title[@lang="en"]`

Results from <http://stevie42.bitbucket.org/YOUTUBE.DIR/books.xml>

3 matches found

- 1: `<title lang="en">Everyday Italian</title>`
- 2: `<title lang="en">Harry Potter</title>`
- 3: `<title lang="en">Learning XML</title>`

XPath Checker

XPath: `/bookstore/book/title/@lang`

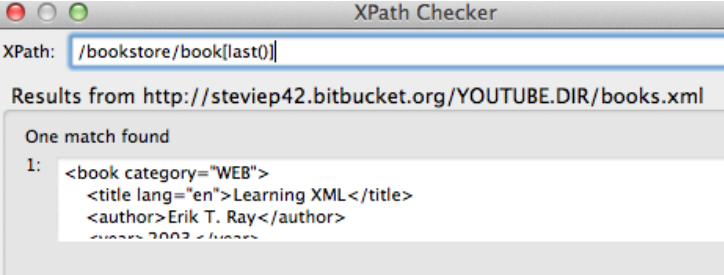
Results from <http://stevie42.bitbucket.org/YOUTUBE.DIR/books.xml>

3 matches found

- 1: `en`
- 2: `en`
- 3: `en`



# XML - XPath



XPath Checker

XPath: `/bookstore/book[last()]`

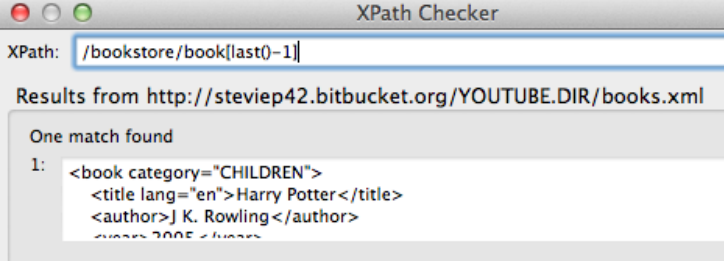
Results from <http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml>

One match found

1: 

```
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
```

This screenshot shows the XPath Checker application with the query `/bookstore/book[last()]` entered. The results show a single match for a book titled "Learning XML" by Erik T. Ray, published in 2003, with the category "WEB".



XPath Checker

XPath: `/bookstore/book[last()-1]`

Results from <http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml>

One match found

1: 

```
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
```

This screenshot shows the XPath Checker application with the query `/bookstore/book[last()-1]` entered. The results show a single match for a book titled "Harry Potter" by J K. Rowling, published in 2005, with the category "CHILDREN".

# XML - XPath

- Let's review. R has a package called XML (not so imaginatively named I know) that can help us parse documents from R code
- This we could write functions that could take, for example, a city and state name, pass it on to Google's GeoCode service, get back an XML document and then parse it into a latitude and longitude pair.
- Basically we get back the XML document and use XPath expressions to pick out only what we want from the document

The XML package has lots of functions and it can be confusing at first to get used to.

# XML - R Package

We have to learn how to use the XML package to process documents.

- Read the XML file directly from the Web or from your local hard drive
- Use an R function to turn or parse the XML file into a format that R can understand
- Use an XPath expression in conjunction with an R function to extract the nodes and elements of interest.
- The R XML package supports a number of functions to help do this

# XML - R Package

The primary function to parse an XML document is the **xmlParse()** function. It has several arguments that are useful when attempting to handle HTML and XML documents. Using the book.xml file let's get started.

```
library(XML)
library(RCurl)
```

```
url <- "http://steviep42.bitbucket.org/YOUTUBE.DIR/books.xml"
xml.report <- xmlParse(url,useInternalNodes=TRUE)
```

```
names(xml.report)
[1] "doc" "dtd"
```

# XML - R Package

Function name	Result
<code>xmlName (node)</code>	Gets name of the element, i.e. tag name
<code>xmlAttrs (node, ...)</code>	Returns (char.) vector of name value pairs
<code>xmlGetAttrs (node, name)</code>	Returns value of named attribute
<code>xmlChildren (node)</code>	Returns the children of the node
<code>getSibling</code>	Returns the sibling of the node
<code>xmlValue</code>	Returns the value of a node
<code>xmlApply</code>	Applies a function to each node
<code>newXMLNode</code>	Creates a new XML node

## XML - R Package

There are functions that let us examine this document like we might with the XPath Firefox plugin. Once we've read in the document we find the root name and then the children of the root. This is analogous to our XPath expressions.

```
root <- xmlRoot(xml.report)
books <- xmlChildren(root)
```

```
xmlName(root)
[1] "bookstore"
```

```
xmlSize(root)
[1] 3
```

```
italian <- books[[1]]    # First book entry
potter <- books[[2]]     # Second book entry
```

# XML - R Package

There are functions that let us examine this document like we might with the XPath Firefox plugin. Once we've read in the document we find the root name and then the children of the root. We can examine the children (the books) and see what info they contain

```
xmlChildren(italian)
```

```
$title
```

```
<title lang="en">Everyday Italian</title>
```

```
$author
```

```
<author>Giada De Laurentiis</author>
```

```
$year
```

```
<year>2005</year>
```

```
$price
```

```
<price>30.00</price>
```

# XML - R Package

```
xmlAttrs(italian)
```

```
  category  
"COOKING"
```

```
xmlSize(italian)
```

```
[1] 4
```

```
getSibling(italian)
```

```
<book category="CHILDREN">  
  <title lang="en">Harry Potter</title>  
  <author>J K. Rowling</author>  
  <year>2005</year>  
  <price>29.99</price>  
</book>
```



# XML - R Package

So there are other functions that help us loop over these structure so we don't have to do everything manually. We can use **lapply** and **sapply** R functions though XML gives us some analogous functions called **xmlApply()** and **xmlSapply()**

```
xmlSapply(root, names)
```

	book	book	book
title	"title"	"title"	"title"
author	"author"	"author"	"author"
year	"year"	"year"	"year"
price	"price"	"price"	"price"

```
xmlSapply(root, xmlAttrs)
```

book.category	book.category	book.category
"COOKING"	"CHILDREN"	"WEB"

```
xmlSapply(italian, xmlValue)
```

title	author	year	price
"Everyday Italian"	"Giada De Laurentiis"	"2005"	"30.00"

```
xmlSapply(italian, xmlSize)
```

title	author	year	price
1	1	1	1

## XML - R Package

There is a function called **getNodeSet()** that allows us to apply XPath expressions to the parsed XML file within R.

```
# Get the third book in the Book inventory
```

```
getNodeSet(xml.report, "/bookstore/book[3]")  
[[1]]  
<book category="WEB">  
  <title lang="en">Learning XML</title>  
  <author>Erik T. Ray</author>  
  <year>2003</year>  
  <price>39.95</price>  
</book>
```

```
# Get all the title names as text
```

```
getNodeSet(xml.report, "//title/text()")  
[[1]]  
Everyday Italian  
  
[[2]]  
Harry Potter  
  
[[3]]  
Learning XML
```

# XML - R Package

Write a function that returns the average price of all books

```
myfunc <- function(x) {  
  library(XML)  
  # Apply XPath expression to get text associated with all prices  
  
  tmp <- getNodeSet(xml.report,"//price/text()")  
  
  # Get actual values  
  nums <- xmlSApply(tmp,xmlValue)  
  
  # Turn into numeric values  
  avg <- mean(as.numeric(nums))  
  
  return(round(avg,2))  
}  
  
myfunc(xml.report)  
[1] 33.31
```

# XML - R Package

Let's use the Google Geocoding capability and use R and the XML package to process the results.

```
library(XML)
library(RCurl)

# What is the Lat Lon for Atlanta, GA ?

geourl <- "http://maps.googleapis.com/maps/api/geocode/xml?address=Atlanta,GA&sensor=false"
locale <- getURL(geourl)
plocal <- xmlParse(locale,useInternalNodes=TRUE)

# Okay let's extract Formatted address, the County name and the lat and lon

format <- getNodeSet(plocal,"/GeocodeResponse/result/formatted_address")
county <- getNodeSet(plocal,"/GeocodeResponse/result/address_component[2]/long_name")
latlon <- getNodeSet(plocal,"/GeocodeResponse/result/geometry/location/descendant::*")

format <- xmlSApply(format,xmlValue)
lat <- as.numeric(xmlSApply(latlon,xmlValue))[1]
lon <- as.numeric(xmlSApply(latlon,xmlValue))[2]

holdf <- data.frame(format,lat,lon)

      format      lat      lon
1 Atlanta, GA, USA 33.749 -84.38798
```

## XML - Google

Put this into a function to use on any number of locales. Because we will GeoCode several cities we have to add rows to our return data frame. We use `rbind` for this. We also set up an empty data frame at the begining to hold the results

```
google <- function(citystate) {  
  library(XML)  
  library(RCurl)  
  holdf <- data.frame()  
  
  # What is the Lat Lon for the given city and state ?  
  
  for (ii in 1:length(citystate)) {  
    geourl <- paste("http://maps.googleapis.com/maps/api/geocode/xml?address=",citystate[ii],sep="")  
    geourl <- paste(geourl,"&sensor=false",sep="")  
    locale <- getURL(geourl)  
    plocal <- xmlParse(locale,useInternalNodes=TRUE)  
  
    # Okay let's extract Formatted address, the County name andthe lat and lon  
  
    format <- getNodeSet(plocal,"/GeocodeResponse/result/formatted_address")  
    county <- getNodeSet(plocal,"/GeocodeResponse/result/address_component[2]/long_name")  
    latlon <- getNodeSet(plocal,"/GeocodeResponse/result/geometry/location/descendant::*")  
  
    format <- xmlSApply(format,xmlValue)  
    lat <- as.numeric(xmlSApply(latlon,xmlValue))[1]  
    lon <- as.numeric(xmlSApply(latlon,xmlValue))[2]  
  
    holdf <- rbind(holdf,data.frame(format,lat,lon))  
  }  
  return(holdf)  
}
```

## XML - Google

Here we create an empty list to contain a list of lists with each sublist containing the format, lat, and lon fields for the given city.

```
google <- function(citystate) {  
  library(XML)  
  library(RCurl)  
  masterlist <- list() # Blank master list  
  jj = 1 # Counter for incrementing the masterlist  
  
  # What is the Lat Lon for the given city and state ?  
  
  for (ii in 1:length(citystate)) {  
    geourl <- paste("http://maps.googleapis.com/maps/api/geocode/xml?address=", citystate[ii], sep="")  
    geourl <- paste(geourl, "&sensor=false", sep="")  
    locale <- getURL(geourl)  
    plocal <- xmlParse(locale, useInternalNodes=TRUE)  
  
    # Okay let's extract Formatted address, the County name and the lat and lon  
  
    format <- getNodeSet(plocal, "/GeocodeResponse/result/formatted_address")  
    county <- getNodeSet(plocal, "/GeocodeResponse/result/address_component[2]/long_name")  
    latlon <- getNodeSet(plocal, "/GeocodeResponse/result/geometry/location/descendant::*")  
  
    format <- xmlSApply(format, xmlValue)  
    lat <- as.numeric(xmlSApply(latlon, xmlValue))[1]  
    lon <- as.numeric(xmlSApply(latlon, xmlValue))[2]  
  
    masterlist[[jj]] <- list(format=format, lat=lat, lon=lon)  
    jj = jj + 1  
  }  
  holdf <- do.call(rbind, masterlist)  
  return(holdf)  
}
```

We can put this into a function and then use it on any number of city and state pairs

```
namevec <- c("Atlanta,GA", "Birmingham,AL", "Seattle,WA", "Sacramento,CA",  
"Denver,CO", "LosAngeles,CA", "Rochester,NY")
```

```
google(namevec)
```

	format	lat	lon
1	Atlanta, GA, USA	33.74900	-84.38798
2	Birmingham, AL, USA	33.52066	-86.80249
3	Seattle, WA, USA	47.60621	-122.33207
4	Sacramento, CA, USA	38.58157	-121.49440
5	Denver, CO, USA	39.73924	-104.99025
6	Los Angeles, CA, USA	34.05223	-118.24368
7	Rochester, NY, USA	43.16103	-77.61092

# XML - Google

```
library(googleVis)
geocode <- google(namevec)
geocode$latlon <- paste(geocode$lat,geocode$lon,sep=":")
state.plot <- gvisMap(geocode,locationvar="latlon",tip="format")
plot(state.plot)
```





# XML - XPATH

See Google GeoCoding Example at <https://rollingyours.wordpress.com/2013/03/20/geocoding-r-and-the-rolling-stones-part-1/>

# JSON

JSON, (JavaScript Object Notation), is an open standard format that uses human readable text to transmit data objects consisting of attribute value pairs. It is the primary data format used for asynchronous browser/server communication (AJAJ), largely replacing XML (used by AJAX).

Although originally derived from the JavaScript scripting language, JSON is a language-independent data format. Code for parsing and generating JSON data is readily available in many programming languages.

— Wikipedia

# JSON

There are rules and regulations about how JSON is formed and we will learn them by example but you can look at the numerous tutorials on the web to locate definitive references. See

<http://www.w3schools.com/json/>

Here is an XML file that describes some employees:

```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName>
    <lastName>Jones</lastName>
  </employee>
</employees>
```

# JSON

Here is the equivalent JSON file. Is it easier to read ? I think so. It can also be read by a variety of languages.

These types of files can be obtained from the services we used in the XML examples.

That is Google GeoCoding and Wunderground (as well as many other services) will return JSON documents

```
{
  "employees": [
    {"firstName": "John", "lastName": "Doe"},
    {"firstName": "Anna", "lastName": "Smith"},
    {"firstName": "Peter", "lastName": "Jones"}
  ]
}
```

# JSON

Here is what a JSON Google GeoCode result might look like.

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "Atlanta",
          "short_name" : "Atlanta",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Fulton County",
          "short_name" : "Fulton County",
          "types" : [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name" : "Georgia",
          "short_name" : "GA",
          "types" : [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name" : "United States",
          "short_name" : "US",
          "types" : [ "country", "political" ]
        }
      ]
    }
  ]
}
```

# JSON

- It is important to note that the actual information in the document, things like city name, county name, latitude, and longitude are the same as they would be in the comparable XML document.
- JSON documents are at the heart of the NoSQL “database” called MongoDB
- JSON can be found within many webpages since it is closely related to JavaScript which is a language strongly related to web pages.
- But why is it important when XML is around ?

# JSON

- Many sites use a technology called AJAX (Asynchronous JavaScript and XML) to produce web applications that can do several things at once while a page loads.
- This is called asynchronous behavior. So we can, for example, load parts of a page without causing a reload of that page.
- Another example is that many web pages rely upon Twitter feeds or News/RSS feeds to supply content. This information can be fetched as the page is loading without impacting the user experience
- JSON is very compact and lightweight which has made it a natural followon to XML so much so that it appears to be replacing XML

# JSON

See <http://www.json.org/>

- An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).
- An array is an ordered collection of values. An array begins with [ (left bracket) and ends with ] (right bracket). Values are separated by , (comma).
- A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.
- A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string.



# JSON

See <http://www.json.org/>

```
object
    {}
    { members }
members
    pair
    pair , members
pair
    string : value
array
    []
    [ elements ]
elements
    value
    value , elements
value
    string
    number
    object
    array
    true
    false
    null
```

# JSON

Check out what we get back from the Google Geocoding service if we specify JSON output. Paste the following into any browser.

```
https://maps.googleapis.com/maps/api/geocode/json?address=1600+Amphitheatre+Parkway,+Mountain+View,+CA
```

# JSON

To read/parse this in R we use a package called RJSONIO. There are other packages but this is the one we will be using. Download and install it.

There is a function called **fromJSON** which will parse the JSON file and return a list to contain the data.

So we parse lists instead of using XPath. Many people feel this to be easier than trying to construct XPath statements. You will have to decide for yourself.

```
geo <- fromJSON("geocode.json")
str(geo,3)
List of 2
 $ results:List of 1
  ..$ :List of 5
  .. ..$ address_components:List of 7
  .. ..$ formatted_address : chr "1600 Amphitheatre Pkwy, Mountain View, CA
  .. ..$ geometry          :List of 3
  .. ..$ place_id          : chr "ChIJ2eUgeAK6j4ARbn5u_wAGqWA"
  .. ..$ types              : chr "street_address"
 $ status : chr "OK"
```

# JSON

Okay so if we wanted to get the information for the geometry it is the third list contained in the geo\$results list

```
geo$results[[1]][3]
```

```
$geometry
```

```
$geometry$location
```

```
lat lng
```

```
37.42224 -122.08401
```

```
$geometry$location_type
```

```
[1] "ROOFTOP"
```

```
$geometry$viewport
```

```
$geometry$viewport$northeast
```

```
lat lng
```

```
37.42359 -122.08266
```

```
$geometry$viewport$southwest
```

```
lat lng
```

```
37.4209 -122.0854
```

# JSON

Note that we could have used the list name to make it more intuitive.

```
geo$results[[1]]$geometry
```

```
$location
```

```
lat lng
```

```
37.42224 -122.08401
```

```
$location_type
```

```
[1] "ROOFTOP"
```

```
$viewport
```

```
$viewport$northeast
```

```
lat lng
```

```
37.42359 -122.08266
```

```
$viewport$southwest
```

```
lat lng
```

```
37.4209 -122.0854
```

# JSON

```
geo$results[[1]]$geometry$location  
  lat      lng  
37.42224 -122.08401
```

# To get this into a data frame we could do:

```
mydf <- do.call(rbind,geo$results[[1]]$geometry[1])  
  lat      lng  
location 37.42224 -122.084
```

# Or we could do the following. Both approaches are right. Pick the one  
# that makes the most sense to you. Processing lists into data frames can  
# sometimes be confusing

```
lat <- geo$results[[1]]$geometry$location[1]  
lon <- geo$results[[1]]$geometry$location[2]  
  
mydf <- data.frame(lat=lat, lon=lon)
```

To get JSON output from Wunderground requires you to sign up for their free service to get an API key. Go to

<http://www.wunderground.com/weather/api/d/login.html>

[API Home](#)

[Pricing](#)

[Featured Applications](#)

[Documentation](#)

[Forums](#)

## Create Your Free Account!

\*All fields are required



[What's a Handle?](#)

☐ I agree to the [Terms of Service](#).

**Sign Up >>**

<http://www.wunderground.com/weather/api/d/docs?d=data/geolookup&MR=1>

## Examples

Try with **api**gee

### City within the USA

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/CA/San\\_Francisco.json](http://api.wunderground.com/api/Your_Key/geolookup/q/CA/San_Francisco.json)

Show Response

### City outside of the USA

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/France/Paris.json](http://api.wunderground.com/api/Your_Key/geolookup/q/France/Paris.json)

Show Response

### AutoIP Address Location

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/autoip.json](http://api.wunderground.com/api/Your_Key/geolookup/q/autoip.json)

Show Response

### Zip or Postal Code

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/94107.json](http://api.wunderground.com/api/Your_Key/geolookup/q/94107.json)

Show Response

### Airport Code

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/SFO.json](http://api.wunderground.com/api/Your_Key/geolookup/q/SFO.json)

Show Response

### Latitude, Longitude Coordinates

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/37.776289,-122.395234.json](http://api.wunderground.com/api/Your_Key/geolookup/q/37.776289,-122.395234.json)

Show Response

### Personal Weather Station (PWS)

[http://api.wunderground.com/api/Your\\_Key/geolookup/q/pws:KMNCHASK10.json](http://api.wunderground.com/api/Your_Key/geolookup/q/pws:KMNCHASK10.json)

Show Response



Here is how we access information for a specific weather station. Note that in the "yourkey" field you need to supply the key you received when you signed up for the free Wunderground API service.

```
library(RCurl)
atl <- getURL(http://api.wunderground.com/api/Your_Key/geolookup/q/GA/Atlanta.json)

atl.json <- fromJSON(atl)

# Note that the second argument to str let's us limit or expand the
# amount of detail present in the data structure. The higher the number
# the more detail.

str(atl.json,1)
List of 2
 $ response:List of 3
 $ location:List of 17
```

So atl.json is a two element list. The first element is called **response** which contains three elements. The second element in atl.json is also a list called **location** that has 17 elements.

```

str(atl.json,2)
List of 2
 $ response:List of 3
  ..$ version      : chr "0.1"
  ..$ termsOfService: chr "http://www.wunderground.com/weather/api/d/terms.html"
  ..$ features      : Named num 1
  .. ..- attr(*, "names")= chr "geolookup"
 $ location:List of 17
  ..$ type          : chr "CITY"
  ..$ country       : chr "US"
  ..$ country_iso3166 : chr "US"
  ..$ country_name   : chr "USA"
  ..$ state         : chr "GA"
  ..$ city          : chr "Atlanta"
  ..$ tz_short      : chr "EDT"
  ..$ tz_long       : chr "America/New_York"
  ..$ lat           : chr "33.85500717"
  ..$ lon           : chr "-84.39598846"
  ..$ zip           : chr "30301"
  ..$ magic         : chr "1"
  ..$ wmo           : chr "99999"
  ..$ l             : chr "/q/zmw:30301.1.99999"
  ..$ requesturl    : chr "US/GA/Atlanta.html"
  ..$ wuiurl        : chr "http://www.wunderground.com/US/GA/Atlanta.html"
  ..$ nearby_weather_stations:List of 2

```

# JSON

So in this case it winds up being pretty easy to get things like that latitude and longitude.

```
atl.json$location[9:10]
```

```
$lat
```

```
[1] "33.85500717"
```

```
$lon
```

```
[1] "-84.39598846"
```

```
# We can unlist a list
```

```
unlist(atl.json$location[9:10])
```

```
lat
```

```
lon
```

```
"33.85500717" "-84.39598846"
```

```
# Let's turn them into numbers
```

```
as.numeric(unlist(atl.json$location[9:10]))
```

```
[1] 33.85501 -84.39599
```

# JSON

At the end of the structure is a list containing information about nearby weather stations. Let's see what they are and get some basic information from them.

```
str(atl.json$location$nearby_weather_stations,2)
```

```
List of 2
```

```
$ airport:List of 1
```

```
..$ station:List of 4
```

```
$ pws      :List of 1
```

```
..$ station:List of 33
```

```
# We get names of the lists contained within the nearby_weather_stations
# There is always a station at the Airport but we are interested in the
# PWS (Personal Weather Stations) in the neighborhoods
```

```
sapply(atl.json$location$nearby_weather_stations,names)
```

```
airport      pws
"station" "station"
```

```
# How many nearby weather stations are there ?
```

```
str(atl.json$location$nearby_weather_stations$pws,1)
```

```
List of 1
```

```
$ station:List of 33
```

# JSON

At the end of the structure is a list containing information about nearby weather stations. Let's see what they are and get some basic information from them.

```
str(atl.json$location$nearby_weather_stations$pws,1)
```

```
List of 1
```

```
$ station:List of 33
```

```
# We need to look into this - We have a trick at our disposal. Anytime  
# we have a list which contains one level of lists we can assemble that  
# into a data frame using the do.call function
```

```
pwsdf <- do.call(rbind,atl.json$location$nearby_weather_stations$pws[[1]])
```

```
head(pwsdf)
```

```
head(pwsdf)
```

	neighborhood	city	state	country	id	lat	lon	dis
[1,]	"Argonne Forest"	"Atlanta"	"GA"	"US"	"KGAATLAN134"	33.84309	-84.39918	1
[2,]	"Buckhead"	"Atlanta"	"GA"	"US"	"KGAATLAN203"	33.85439	-84.41425	1
[3,]	"Chastain"	"Atlanta"	"GA"	"US"	"KGAATLAN195"	33.87098	-84.4015	1
[4,]	"Argonne Forest"	"Atlanta"	"GA"	"US"	"KGAATLAN111"	33.83882	-84.4078	2
[5,]	"North Buckhead"	"Atlanta"	"GA"	"US"	"KGAATLAN119"	33.85077	-84.37355	2
[6,]	"Buckhead Forest"	"Atlanta"	"GA"	"US"	"KGAATLAN87"	33.84486	-84.3762	2

## Eplilogue - Lists in R

Let's look at processing a list for purposes of turning it into a data frame. In our Geocoding example we had something like this:

```
geo$results[[1]]$geometry
$location
      lat      lng
37.42224 -122.08401
```

```
$location_type
[1] "ROOFTOP"
```

```
$viewport
$viewport$northeast
      lat      lng
37.42359 -122.08266
```

```
$viewport$southwest
      lat      lng
37.4209  -122.0854
```

```
# Location is a list with a vector as a value as is location_type. viewport is
# a list of two elements each of which are vectors. How could we get all the lat and lons
# into a single data frame ? There are a number of ways.
```

```
str(geo$results[[1]]$geometry,1)
List of 3
 $ location      : Named num [1:2] 37.4 -122.1
  ..- attr(*, "names")= chr [1:2] "lat" "lng"
 $ location_type: chr "ROOFTOP"
 $ viewport      :List of 2
```

## Eplilogue - Lists in R

Let's look at processing a list for purposes of turning it into a data frame. In our Geocoding example we had something like this:

```
len <- length(geo$results[[1]]$geometry)

for (ii in 1:len) {

  # First process the lat and lon in location which is the first element of the list

  if (ii == 1) {

    lat <- geo$results[[1]]$geometry$location[1]
    lon <- geo$results[[1]]$geometry$location[2]
    tmpdf <- data.frame(lat=lat,lon=lon)

  } else if (ii == 3) {

    nelat <- geo$results[[1]]$geometry$viewport$northeast[1]
    nelon <- geo$results[[1]]$geometry$viewport$northeast[2]
    swlat <- geo$results[[1]]$geometry$viewport$southwest[1]
    swlon <- geo$results[[1]]$geometry$viewport$southwest[2]
    tmpdf <- cbind(tmpdf,nelat=nelat,nelon=nelon,swlat=swlat,swlon=swlon)

  }

}

tmpdf
  lat      lon    nelat    nelon   swlat   swlon
lat 37.42224 -122.084 37.42359 -122.0827 37.4209 -122.0854
```