**DATA MANAGEMENT**

**Three key-features of an OS:**

1. Multi-user: A multi-user OS allows for multiple users to use the same computer at the same time and/or different times (eg. ECS Linux server)

2. Multi-processing: An OS capable of supporting and utilizing more than one computer processor. Processors can be tightly (high-end servers) or loosely coupled (clusters of computing nodes).

3. Multi-tasking: An OS that is capable of allowing multiple software processes to run at the same time.

**Some commands:**

1. “whoami” – prints your login name

2. “passwd” – can be used to change your password

**Quick Tips:**

Shortcut 1: use *up* and *down* arrow keys to bring up previously typed commands.

Shortcut 2: use *tab* for autocomplete to type commands and paths faster.

Shortcut 3: hit *tabtab* to see all possible options.

**Good to know:**

The *root* is the root from which you can access any file in the system. You can refer it with “/”.

eg. The path to the student folder is /home/student.

**/etc** stores config files for the system.

**/var/log** stores log files for various system programs.

**/bin** and **/usr/bin** stores several commonly used programs.

**Meta-information on UNIX files:**

A UNIX directory contains a table of the names of files it contains and their respective Inode numbers. Keeping the meta-data separate from the contents of the files has its advantages.

Moving files from one directory to another is easy and fast.

A user can rename or delete a file even if another application has opened it and is working on it. So no application can “hijack” a file.

You can get the Inode number via “stat NameOfFile

**More commands:**

**pwd** -displays the current absolute path.

**ls** – lists the files in the current directory. It has many options: **ls [options] [location]**

**options:**

**-l** – long list

**-t** – sort by modification time

**-S** – sort by size

**-h** – list file sizes in human readable format (bytes instead of blocks)

**-r** – reverse the order

**location:** the path to the file.

You can also combine some options: **ls –ltrh** -> lists files by time (t) in reverse order (r) with long listng (l) and human readable format (h).

**Paths:**

**Absolute path** -> eg. “/home/faculty/sarwar”

**Relative path** -> eg. You are in faculty directory. “cd sarwar”

**~ (tilde):** Shortcut to your **home** directory: eg. “/home/students/downloads is the same as ~/downloads”

**. (dot)** -> reference to current directory.

**.. (dotdot)** -> a reference to the parent directory. You can use this several times in a path to keep going up the hierarchy. (eg. “../..”)

UNIX filenames are **case sensitive**!

**Man command**:

**man “COMMAND”** -> displays information about the command

**man –k “SEARCH ITEM”** -> if you are not sure what command you need

**More commands:**

**mkdir [options] <directory name>** -> makes directory

**rmdir [options] <directory name>** -> removes the directory. The directory must be empty before it is removed.

**Managing files:**

**touch [options] <filename>** - creates a file in the directory

**rm [options] <filename>** - removes a file

**rm –r <directory name>** - to remove **non-empty** directory

**cp [options] <source> <destination>**- copy file to a directory

**mv [options] <source> <destination>** - moving files or directories

**File editor:**

To open it, type in **vim**;

To display the contents of the file:

**cat** -> displays contents of text file

**less** -> displays a file, but allows forward and backward movement within it:

**return** (Enter key) scrolls forward one line, **space** to go ahead one page;

**y** key scrolls back one line, and **b** one page.

The command does not read entire file before starting (good to view very large files)

Use “**/**” to search for a string (

Hit **q** to quit.

**head** -> prints out the first 10 lines of the file

**tail** -> prints out the last 10 lines of the file

Both head and tail can be used to print out a custom amount of lines. (eg. **head –n3 textfile.txt** -> displays the first 3 lines)

**Processes:**

A process is simply a *program* in *execution.*

Every process is assigned a unique process identifier or PID: eg. 6720, 1242, etc.

To run a program you must first make sure it has executable permissions. (Use **chmod u+x <program-name>**)

The the prefix **./** to run a program located in your current directory.

To end a program – kill the process – if it does not end on its own, use “**CTRL+C**”;

**ps [options]** -> view the processes that you are running. (and see the PID of each process)

**top** -> view the CPU usage of all processes

**kill [options] <PID>** -> to terminate a process

EXAMPLE: commonly used options ( **kill –SIGTERM 1710** )

**SIGTERM**: Request a process to stop running; signal **can be ignored**. Process is given time to gracefully shutdown. When a program gracefully shuts down, that means it is given time to save its progress and release resources.

**SIGKILL**: Forces process to **stop executing immediately**. The program cannot ignore this signal. This process does not get to clean-up either.

**CTRL Z** -> pauses the current foreground process.

**bg** -> moves the process to the background.

**fg** -> brings process to the foreground.

**Running a process when logged off:** (haven’t tested it properly, should work)

**screen** -> create a new screen.

**screen –d** -> detach from an existing screen.

**screen –list** -> to list available screens.

**screen –r** **screen-id** -> to resume an existing screen.

**CTRL + D** -> to kill the screen.

**Piping:**

Bash shell allocates 3 file descriptors for each process:

**STDIN** is opened for keyboard input

**STDOUT**, **STDERR** to screen output

Definition of piping:

*program\_1 | program\_2*

\*program\_1’s outcome becomes program\_2’s input

*program\_1 > file.txt*

\*program\_1’s output and error logs are written to a file called “file.txt”

\*We can use >> instead of > to append to end of file.txt

*program\_1 < input.txt*

\*program\_1 gets its input from a file called “input.txt”

**Filters:**

A filter is a program which accepts textual input and transforms it in some way.

Filets can be connected together by pipes.

Filters can be thought of as building blocks to be easily put together to do what you want.

Eg: cat student\_attendance.txt | grep –e August | sort

Example filters:

**head**

**tail**

**sort** -> organize the data into order

**wc** -> print a count of number of lines, words and characters

**uniq** -> removes duplicate lines

**du** -> estimates file space usage

**xargs**  -> builds and executes command lines from standard inputs

Example: **cat** tomslee\*.csv | **cut** **–f4 –d,** | **sort** | **uniq** > processedlist.txt

**Version control** – easy

**UNIX Wildcards:**

Wildcards allow you to operate on multiple files at a time.

If the command-line argument has a wildcard, your shell – the command line interpreter – will replace it with a list of matching filenames.

**Wildcards:**

**\* -** any character *zero or more times* (eg. **rm \*.txt)**

**?** – any *single* character (eg. **rm a?.txt)**

**[]** – a range of characters (eg. **rm example\_[abc].txt** OR **rm example\_[a-c].txt)**

**Wildcards** can be combined.

(eg. **rm\*.???**)

**\* matches zero or more characters**

**? exactly one character**

**[abcde] exactly one character listed**

**[a-e] exactly one character in the given range**

**[!abcde] any character that is not listed**

**[!a-e] any character that is not in the given range**

**{debian,linux} exactly one entire word in the options given**

**Permissions on files**

The Unix file system provides permissions (also known as modes) to restrict access to your files and directories.

3 types of permissions – **user (u), group (g), others (o).**

**chmod** changes modes; Command to manipulate permissions of *owner, group* and *others.*

**ls –l** gives a long listing of files and directories. The first column is a 10-character string indicating

* The type of file (regular file, directory, symbolic link, etc.), and
* The access permissions for the
  + File’s owner,
  + Group, and
  + Other users (outside of group)

|  |  |  |  |
| --- | --- | --- | --- |
| **File Type (character 1)** | **Owner Access (characters 2-4)** | **Group Access (characters 5-7)** | **Other Access (character 8-10)** |
| - = regular file d = directory | r = read w = write x = execute | r = read w = write x = execute | r = read w = write x = execute |

Remember: Only two people may usually make changes to the permissions.

The owner of the file or directory and the root user.

The root user is a **superuser** who is allowed to do anything and everything on the system,

**UNIX Shells:**

The shell is a command line interface, which allows you to give commands to the UNIX OS.

Many types of shells are available (csh, korn), but we will we using **bash**.

**Environment variables in a shell:**

(<https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps>)

Every process (executed program) run in a shell has access to a set of variables called **environment variables.**

These variables are part of the environment in which the process runs.

Example: **PATH:** List of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.

**Setting up environment variables in a shell:** (should re-read)

(<https://www.geeksforgeeks.org/export-command-in-linux-with-examples/>)

Use the **export** command

**export [name of variable=value]**

Example: Use the variable to control how your shell prompt is displayed

**export PS1=’$(whoami)@$(hostname):$(pwd)>’**

**export** with no argument given will list the values of all environment variables in the shell.

**(Basically it just creates a system variable which can be accessed later) ☺**

**Recap on writing a Bash shell script:**

Shell programming is accomplished by directly executing shell commands at the shell prompt.

An alternative is to store commands in the order of execution, in a text file, called a shell script, and then execute the shell script.

Shell script files usually have the extension **.sh**.

The first line of the shell script file begins with **#!** Followed by the full path where the shell interpreted is located.

For a bash script, **#!/bin/bash**.

Remember to use the command **which bash** to make sure this is the correct path!

**Variables in a bash shell script:**

Standard variable assignment syntax:

sum=0

average=0

To display them, simply: **echo $sum**

**Variables** can also be set **directly** from the output of a **UNIX command**.

Eg: **varName=$(echo “hello world”)**

OR

You can use the name of the script file: **varName=$(./hello.sh)**.

**Loops in a Bash shell script:**

For loop: **for i in /home/dst1m17/\*; do echo $i; du -sh $i; done** -> displays all files in the directory and their sizes.

When writing this script in a file, you can choose to not use semicolons. (Which is what I prefer)

**It looks something like this:**

**for I in /home/dst1m17/\***

**do echo $i**

**du –sh**

**done**

Another example: You can use the for loop to create many new files or directories – e.g., with nested loops:

**for i in {A..Z}; do for j in {1..5}; do touch $i$j.dat; done; done**

**Loop through all the lines in a text file:**

**cat file-name | while read line; do <something with $line>; done**

**eg: cat tomslee\* | while read p; do echo $p; sleep 2s; done**

**GREP:**

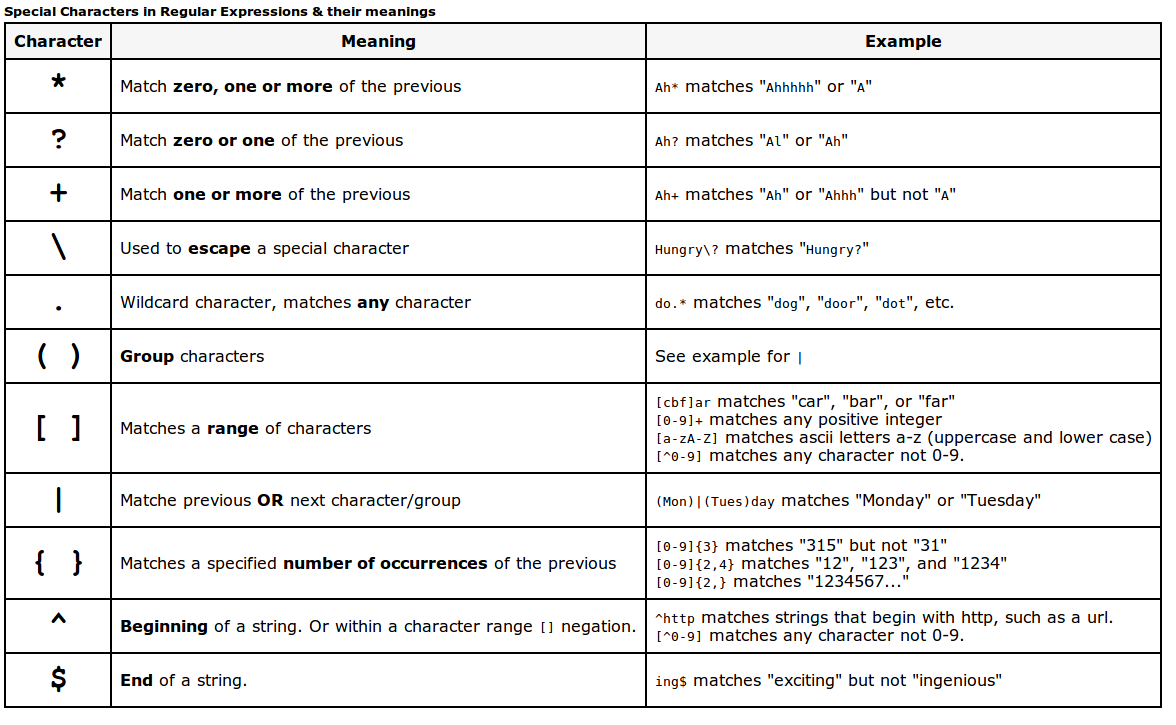
**grep** is a command to search input given to it.

It looks for lines in the input that match a particular pattern or regular expression.

Usage: **grep pattern input**

The pattern can be constructed from regular expressions.

Using regular expressions



**-**

**-**

**-**

**-**

**-**

Example grep: **grep keyword file**

**SED**

sed is a text stream editor

It reads input and modifies it as specified by a list of commands. The modified input is then written to the standard output.

Usage: **sed [options] command [file …]**

Most commonly used command is to substitute text with something else.

Example: **echo Hello World | sed "s/hello/Ii/I"**– substitutes Hello with Hi, case insensitive

Another example: **cat smallfile.csv | sed 's/,/\t/g'** – changes the delimiter character from comma to tab.

Another example: **cat data-plots.dat | sed ‘/Run/d’** – deletes all lines that contain run?

Besides substituting text, sed can also be used to:

Print or delete specified lines in the input.

Append a line following pattern detected in the input (using /a command)

Example: inserts line following detection of pattern.

**sed '/pattern/a\line-to-append' input**

Example: deletes from line number n to end of file.

**sed 'n,$d' input**

**AWK**

What can we do with awk:

Scans a file line by line

Splits each input line into fields

Compares input line/fields to pattern

Performs action(s) on matched line

Usage: **pattern {action}**

Example using the bash shell. Create file **myscript.sh**

(need to have file which has File and Owner)

**#!/bin/sh awk '**

**BEGIN { print "File\tOwner" } { print $9, "\t", $3} END { print "done"} '**

Then calling awk: **ls –l | myscript.sh**

**STRUCTURED DATA**

We need structured data because that way we can easily manipulate and process the data!

**CSV (Comma Separated)** & **TSV (Tab Separated)** are two popular formats of structured data for representing rows and cols. (cut, sort, grep, paste, join etc. use them). But they don’t work well for flexible formats (conversations, annotations, doc representation…)

We need the stored data to also be understandable by machines, because it makes it easier for ->

* Searching – Easier to find more appropriate results for our search
* Summarization
* Prediction – Last time A led to B. This situation is 80% like A, will B happen?
* Linking – Information from different sources can be linked together

**Human-Readable** -> text, images, forms;

Mostly unstructured;

Focuses on usability;

**Machine readable** -> Needs to be structured;

Focuses on efficiency;

Generally, not human-readable;

**Going machine readable ->**

Key Concepts ->

* Entities
* Attributes
* Relationships

Types of Relationship modelling ->

* Hierarchical (single parent)
* Network (multiple parents)
* Object-orientated (pointers)
* Object-relational (object as tables)
* Entity-relational (conceptual entities as tables)

We represent entities and attributes and hierarchical relationships using a **markup language**.

Markup: annotation a document to add metadata.

In addition to the real-world data, we often need to add more data. Data that is useful for the machine, but not for the human. This metadata is typically stored alongside the main content (often as annotations).

**SGML – Standard Generalized Markup Language**.

* An ISO (International Organization for Standardization) for defining Markup Languages.
* A Super set of all markup languages.
* **Separates structure from content**.
* Metalanguage: Describing formatting and markup languages.
* Tags: <tag> … </tag>
* Attributes: <tag attribute=”value”>
* Content: <tag>Content Goes Here</tag>

Common subsets of **SGML**:

* **XML** (including **XHTML**)
* **HTML**
* **OED** (The Oxford English Dictionary)

These are all SGML, but not all SGML are XML/HTML…

From SGML to XML ->

**SGML** is very flexible

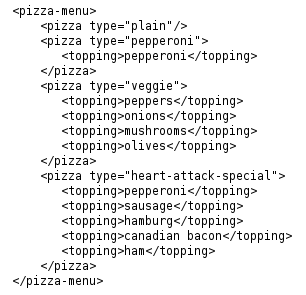
* You can do a lot, but it is very complex.
* There is NO strict structure.
* A lot of things had to be inferred.
* Requires a definition of the structure as well as the SGML itself.

**XML** was designed to be less flexible

* Easier and more efficient to parse.
* Simplifies SGML, while keeping the bits that were needed.
* Data can be delivered without a definition of structure.

**XML** – Extensible Markup Language.

* **Hierarchical**
* Designed to carry data, not display it
* No defined tags (you define your own XML tags)
* Meta-tools to define purpose of specific tags. (DTD: Doc Type Definition, Schema)
* Unicode
* W3C recommendation (World wide web consortium)



**XML** syntax:

* All XML elements must have a **closing tag**
* XML tags are **case sensitive**
* XML elements must be properly **nested**
* XML Docs must have a **root element**
* XML attributes must be quoted e.g. <tag attr=”…”>

**HTML**

HTML is an application of SGML. **HTML** tells you how to display tags, whereas **XML** tells you what the data means. **HTML** has rules based on tags (e.g. b = bold).

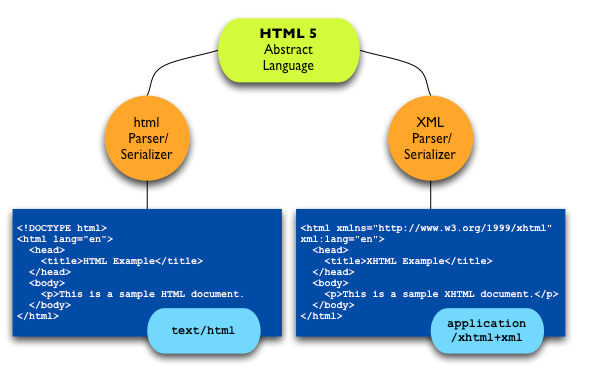
**XHTML** is an application of **XML** (more restrictive than **SGML**)

Modern **HTML5** while based on the ideas in **SGML** in not **SGML**.



This piece of code in **HTML5**. It is an example of why **HTML5** is not **XML**.

1. Not all tags need closing tags (e.g. img).
2. Special tags (e.g. !doctype)
3. Automatic escaping (e.g. inside script)
4. Attributes without values
5. Multiple top-level elements

There are two serializations of HTML5: html and xml (xhtml5). 

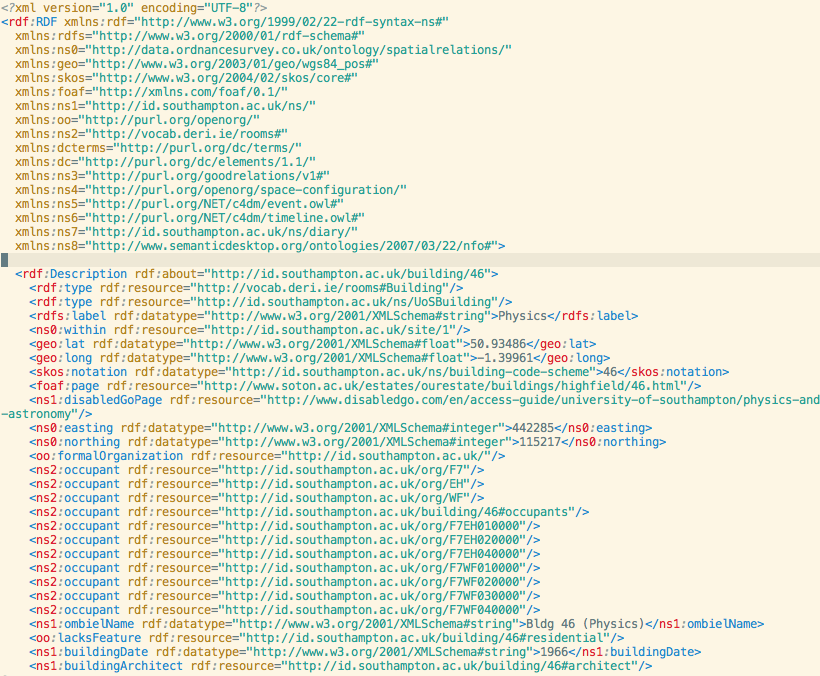
The **XML** suite:

* **Syntax** -> Defines how XML should be written (write it correctly)
* **Namespaces** -> global semantic partitions (avoid name conflicts)
* **Schema** -> Semantic definitions or validity (write it properly)
* **XSLT** -> Language to transform XML documents (format it)
  + E.g., stylesheets (much more powerful than CSS)
* **XPATH** -> individual items in XML documents (locate data)
* **XQUERY** -> Generalized query language for XML-based databases (query it)

**Schema:**

* We want to define a correctly formatted XML file with our own tags, attributes and structure.
* Can be used to ensure a valid file.
* Can be used as a template.
* Can be used to identify errors.
* Eases parsing (can build rules based on tags and attributes).
* Greater efficiency.
* Various elements (elements, sequences, types, attributes).

**Namespaces:**

* XML files can reference and include other files.
* Take the university for example:
  + There are XML files which contain:
    - People
    - Buildings
    - Rooms
  + There are external XML files which link in:
    - Bus times
    - Roads
* You define your own tags for your things, but other people may also define tags with the same name, which are different to yours.
* Solution: **Namespaces**.
  + You define your tags in your namespace.
  + They define their tags in their namespace.
* Define using **xmlns**.
* ****

Looks messy.

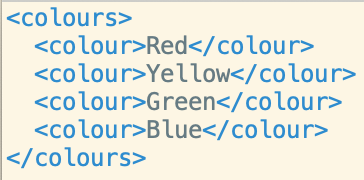
**Schema basics: Defining a Schema.**

Brin in the XML Schema namespace (xs).

Start with the <xs:schema> Element

Define elements with <xs:element>

We can then define **simple** and **complex** elements.

* Simple element:
  + Contains text only
  + Cannot contain further elements
  + Cannot have attributes
  + Cannot be empty
  + 
* Complex element
  + Can have attributes
  + Can contain
    - Nothing
    - Other elements
    - Text
    - Combination of text and other elements
  + 

Building up our schema:

* Defining sub-elements
  + <xs:sequence> Specified sequence of inner elements.
  + <xs:all> Allows elements to appear in any order.
* Maximum and minimum occurences
  + minOccurs=”0” makes an element optional.
  + maxOccurs=”unbounded” allows repetition.
* Defining types
  + <xs:elements … type=”type”>
  + <xs:attribute … type=”type”>
* Defining attributes
  + <xs:attribute name=”name” type=”type”>
    - use=”required” allows required attributes
* Mixed elements
  + Can contain both text and elements
  + Xs:complexType … mixed=”true”

Built in data types

* xs:string
* xs:boolean
* xs:float
* xs:double
* xs:decimal
* xs:dateTime
* xs:duration
* xs:hexBinary
* xs:base64Binary
* xs:anyURI
* xs:integer
* xs:nonNegativeInteger
* xs:positiveInteger
* xs:nonPositiveInteger
* xs:negativeInteger
* xs:byte
* xs:int
* xs:long
* xs:short
* xs:unsignedByte
* xs:unsignedInt
* xs:unsignedLong
* xs:unsignedShort
* xs:date
* xs:time

Example schema:





**XML + CSS -> can apply stylesheet over and xml file to add presentation to structure**

**XPATH** -> used to navigate through elements and attributes in XML docs.

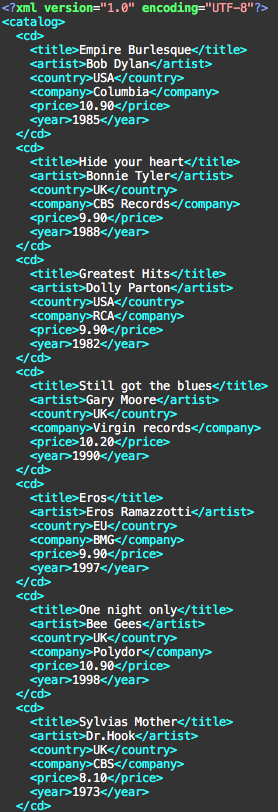
* **Getting a node:** /path/to/element
* **Getting all nodes:** //node, //\*, /node/\*/node2
* **Indirect nodes:** /node1//node3
* **Getting attributes:** /node/@Attribute
* **Getting text:** /node/text()
* **Getting everything:** //text()
* **Numerical referencing:** /node[n]
* **Filtering by node:** //Node1[Node2]
* **Filtering by attribute:** //Node1[@Attribute=“Value”]
* **Boolean logic:** //Node|//Node2, //Node[@id=1 or @id=2]
* **Navigation:** . And ..
* **String functions:** contains, string-length, starts-with etc.

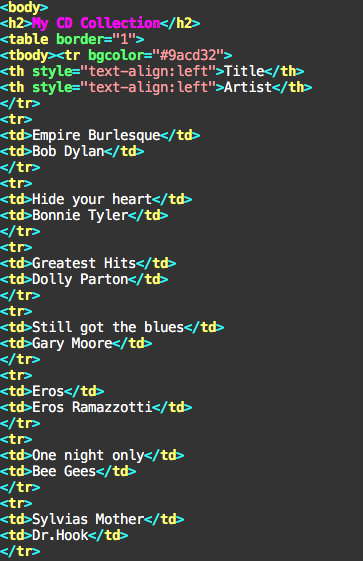
**XSLT**

* A step beyond CSS
  + Just decides how to display certain elements.
* Extensible Stylesheet Language
  + It transforms the input -> output
* Use to add/remove elements to/from output file
* Re-arrange/sort elements
* Perform tests, make decisions.
* Uses XPATH

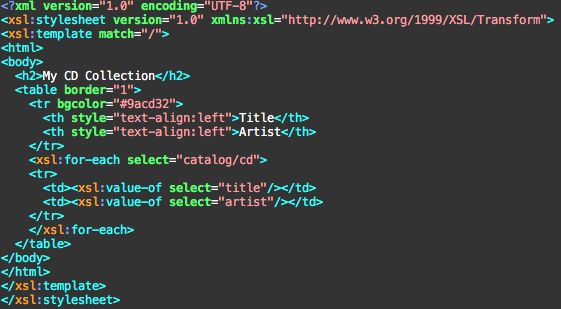
Using XPath to build a new document

* <xsl:stylesheet> or <xsl:transform> (the same)
  + Define the root element
* <xsl:template match=”XPath">
  + Contains the rules, applied to the match
* <xsl:for-each select=”*XPath*">
  + Loop over the given path
* <xsl:value-of select=”*XPath*”>
  + Display the value of the given path
* <xsl:if test="*expression*">
  + Allows conditional branches





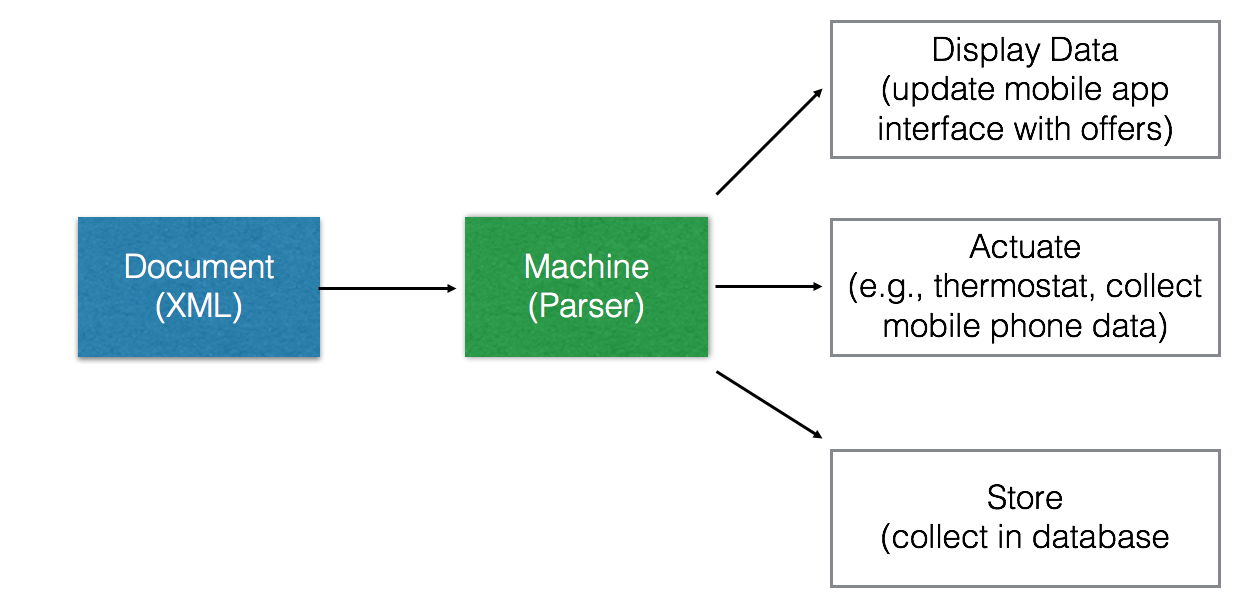
=



+

Parsin/Manipulating

* Java: Saxparser (and many others)
* Python: lxml (html and XML) (and many others)
* Allow you to:
  + navigate the document tree
* <root> <child></child> <child></child> </root>
* extract specific elements with specific attributes
  + e.g., <div class=“ab’>
* extract elements with specific contents
* Add children elements to the tree!



XML-based languages:

* VML = Vector (graphics) Markup Language
* SSML = Speech Synthesis Markup Language
* CPML = Call Policy Markup Language
* DSML = Directory Services Markup Language
* MathML = Mathematical Markup Language
* CML = Chemical Markup Language
* AML = Astronomical Markup Language
* BSML = Bioinformatic Sequence Markup Language
* GedML = Genealogical Data Markup Language
* FinXML = Financial market Markup Language
* ChessML
* SDML = Signed Document Markup Language
* RELML = Real Estate Listing Markup Language
* ECSML = Annotating ECS students