Speed Recap Unix pipes

- mechanism to easy daisy chain several commands via output/input connections
- typical linux commands:
  - read from stdin (keyboard)
  - writes to stdout (screen)
  - redirection of output to another command: ‘|’
  - redirection of output to a file: ‘>’
Data Shape Considerations

- Good for regular structured data, i.e. text in csv or tsv
- To develop a strategy consider your data to have a ‘matrix’ like shape
- by a clever combination of the mentioned tools your can filter out the desired fields
Commands

- `cat`/ `paste` for vertical/horizontal combination of files
- `join` to combine line based on the values of specified join attribute
- `head/tail` row selection based on indices
- `grep` row selection based on an occurring value
- `cut` index based column selection
- `wc` aggregation to counts
- `uniq` to filter duplicates
Bioinformatics Resources
XML / Web Access

Lecture & Exercises
Prof. B. Rost, Dr. L. Richter, J. Reeb
Institut für Informatik I12
XML Infusion (in 10 sec)

- compilation from http://www.w3schools.com/xml/default.asp
- XML is a software- and hardware-independent tool to store and to transport data
- XML stands for eXtensible Markup Language
- designed to store and transport data
- designed to be self-descriptive
- W3C recommendation
- it does NOT DO anything
About Tags

- XML tags are not predefined like HTML tags
- everybody can/has to invent his own tags
- new tags can be added any time
- the author has to define content and structure of the document
- everything is plain text
Document Structure

```xml
<?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>

  ....

</bookstore>

taken from http://www.w3schools.com/xml/xml_usedfor.asp
```
Syntax Rules

- elements are defined using tags:
  \[
  <\text{tagName}> \ldots </\text{tagName}>
  \]
  or
  \[
  <\text{tagName}/>
  \]

- elements can be nested (contain other elements
  - parent and child nodes, sibling nodes)

- elements can have text content

- each document must contain ONE root element
  that is the parent of all other elements
Syntax Refined

- prolog line `<?xml ...>` is optional
- tags must be (self-) closed
- tags are case sensitive
- tags must be properly nested:
  `<a><b>....</a></b>`  Wrong!
  `<a><b>....</b></a>`  Right!
Syntax Refined

- tags may have attributes
- attribute values must always be quoted
- some special characters cannot be used directly
  - -> coded by entity references:
    - &lt; < less than
    - &gt; > greater than
    - &amp; & ampersand
    - &apos; ‘ apostrophe
    - &quot; “ quotation mark
- comments: <!-- . . . -->

BioinfRes SoSe 19 / 10
Tag Names

- case sensitive
- must start with a letter or underscore
- must not start with the letters xml in any case
- can contain: letters, digits, hyphens, underscores and periods
- cannot contain spaces
- apply common sense and a consistent style
- avoid: minus (-), period (.), colon (:), non-english characters for compatibility reasons
XML Element

- everything between the start and the end tag
- tags are included
- can contain:
  - text
  - attributes
  - other elements
  - a mix of all
- are extensible
XML Attributes

- values must be quoted: single or double quotes
- the unused quoting character can be used inside the value
- decision for attribute or element undecided, but:
  - attributes cannot contain multiple values
  - attributes cannot contain tree structures
  - attributes are not easily expandable
- useful to store meta data, like element id, etc.
A Glimpse of Namespaces

- allows to prevent tag name collisions between different authors/applications/domains
- implemented by the introduction of prefixes
- defined as an attribute:
  \[ \text{xmlns:prefix=“URI”} \]
- usage: \[<prefix:tagName>\]
- the URI is only needed to be unique
- used to integrate other specifications, e.g. XSLT
Levels of Correctness

- **well formed**: a document obey the syntax rules:
  - root element
  - closing tag
  - case sensitive
  - properly nested
  - attribute values quoted

- **valid documents**: in addition to being valid the also conform to a document type definition (format specification)
Document Type Definitions

- two ways to specify a document structure:
  - DTD: Document Typ Definition
  - XML Schema: XML based alternative to DTD
Example

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!
    &copyright;
  </body>
</note>
Example

<!DOCTYPE note [ <!ELEMENT note (to, from, heading, body)> <!ELEMENT to (#PCDATA)> <!ELEMENT from (#PCDATA)> <!ELEMENT heading (#PCDATA)> <!ELEMENT body (#PCDATA)> <!ENTITY copyright "Copyright by .."> ]>
XML DTD

- referenced from a document with:
  ```xml
  <!DOCTYPE note SYSTEM "Note.dtd">
  ```
- `!DOCTYPE` defines the root element
- `!ELEMENT` defines the structure of the elements
- `#PCDATA` means parse-able text data
- `!ENTITY` defines special characters or strings
XML Schema

- alternative to DTD

```xml
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- support of data types and namespaces
- written in XML and extensible

BioinfRes SoSe 19 / 20
XML with Python

- **basic**: `xml.etree.ElementTree`
  
  ```python
  import xml.etree.ElementTree as ET
  ```

- if you parse XML from untrusted sources use safeguarded libraries like `defusedxml`

- read XML input from file into a tree element:
  ```python
  parse("fileName")
  ```

- read XML input from string into a tree element:
  ```python
  fromString(data_as_string)
  ```

- retrieve root element from tree element:
  ```python
  getroot()
  ```
XML with Python

- element type: `tag` (string)
- element attributes: `attrib` (dictionary). Prefer access via `e.get(key, default=None)`, `e.items()`, `e.keys()`, `e.set(key, value)`
- element (text) content: `text` (string)
- child elements: simply iterate
- if you know the structure you can access children via index notation: `root[0][1]`
- select child nodes of a certain type:
  `root.iter('typeName')`
XML with Python

- insertion of new elements:
  - create new element with: `Element('typeName')`
  - add content via `text` and attributes via `attrib`
  - add existing sub-elements with `append()`
  - append new element to the parent element

- simple output:
  ```python
treeElement.write('fileName')
  ```

- for validation and full XPath support use: lxml
Web Access

- URI (Uniform Resource Identifier): is a string
- URL is a specific type of URI including the location of a resource
  - scheme://location/path?query#fragment
- scheme: protocol in use
- location: the hosting server
- path: path to the resource
- query + fragment: optional specifications
Web Access

- **URL manipulation:**
  - `urljoin(base_url_string,relative_url_string)`

```python
from urllib import parse as urlparse

urlparse.urljoin('http://somehost.com/some/path/here','../other/path')
# Result is: 'http://somehost.com/some/other/path'
```

- `urlsplit(url_string,default_scheme='',allow Fragments=True)`

```python
# Result is: ('http','www.python.org:80','/faq.cgi','src=fie','')
```
Web Access

- easy to use third party package requests
  
  [http://docs.python-requests.org/en/master/api/#requests.request](http://docs.python-requests.org/en/master/api/#requests.request)

- support Python v2 and v3

- three main classes: Request, Response, Session
  
  - Request: models a HTTP request send to the server
  
  - Response: models the HTTP response
  
  - Session: if you need continuity (ignore by now)
Requests

- requests.request(method, url, **kwargs)
  - mandatory: method (delete, get, head, options, patch, post, put)
  - mandatory: URL
  - optional parameters: too many to list here

- convenience functions:
  - requests.get/post/...(URL,[data=None],**kwargs)

- import requests
  req = requests.request('GET', 'http://www.example.com/')
Response

- useful attributes: `status_code` tells us about success or failure
- `r.content`: holds the response’s content
- `r.iter_content` and `r.iter_lines` allow chunkwise reading of the response data, especially in a stream context or when handling huge responses