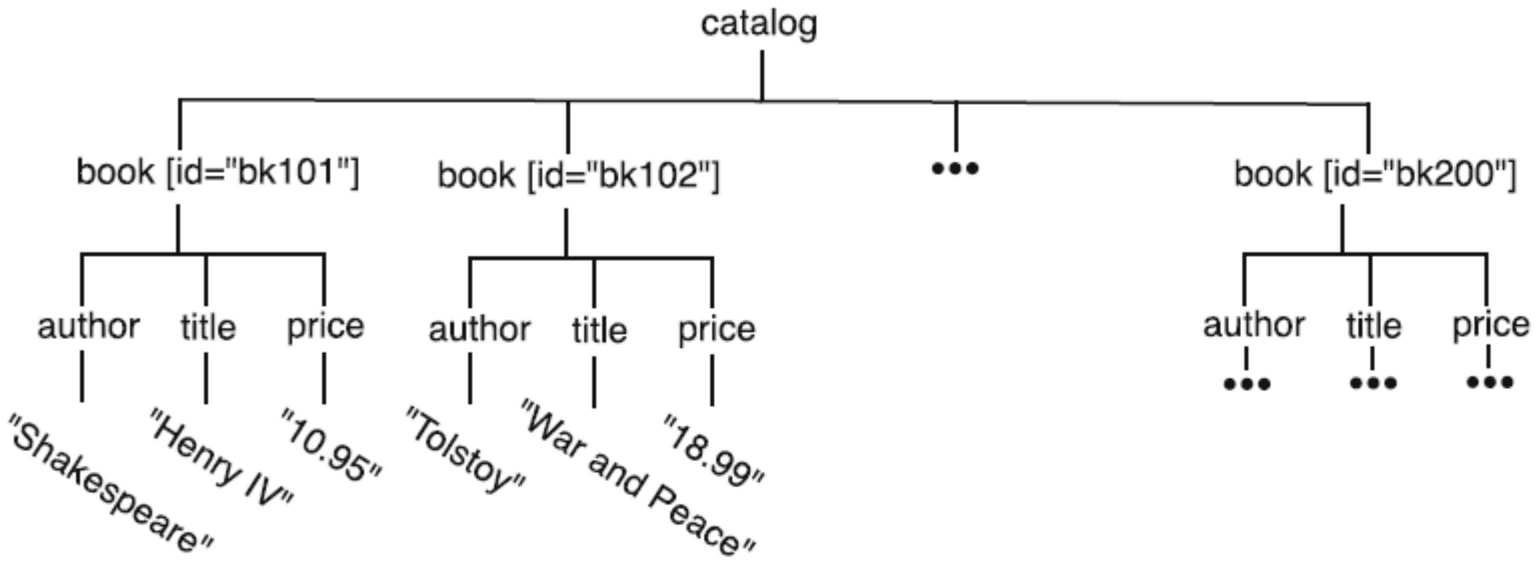
Chapter 15

Hierarchical Model: Structure and Formats

Hierarchical Database?

- A ***hierarchical database*** is a data model in which data is stored in the form of records and organized into a *tree-like structure*, or parent-child structure, in which one parent node can have many child nodes connected through links.



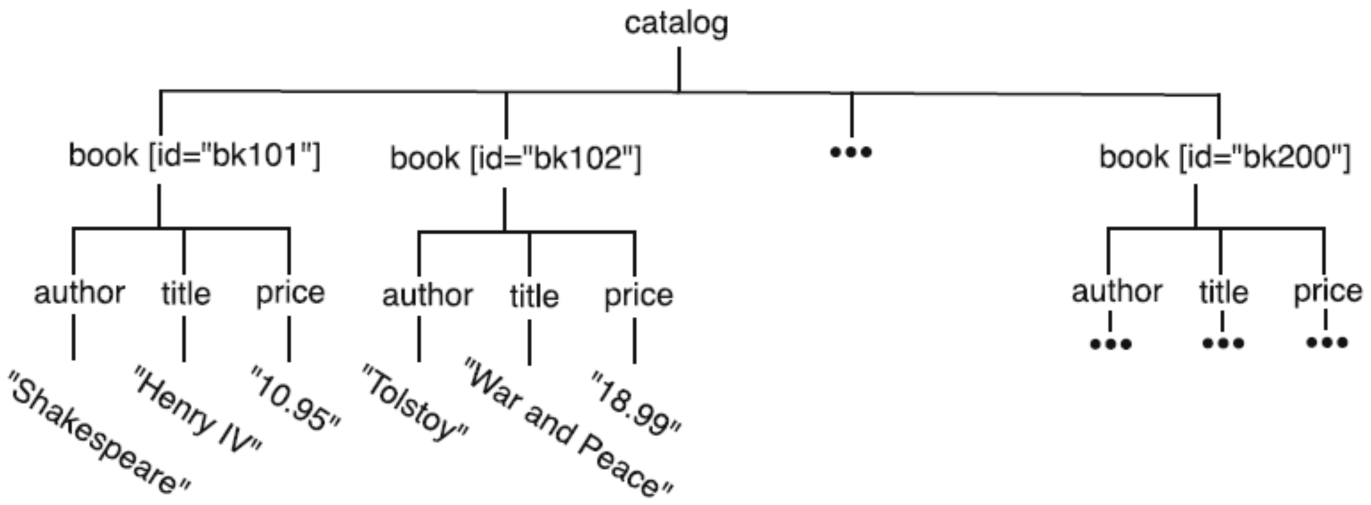
Difference?

- Structure: Hierarchical database architecture is tree-like, data in a relational database is exists in tables **with a unique identifier** for each record.
- Hierarchical database models are simpler, and are more suitable for OS or websites:
  - The one-to-many organization of data makes traversing the database simple and fast, which is ideal for use cases such as website drop-down menus or folder structures of an operating system.
  - Information can easily be added or deleted without affecting the entirety of the database.

Motivation

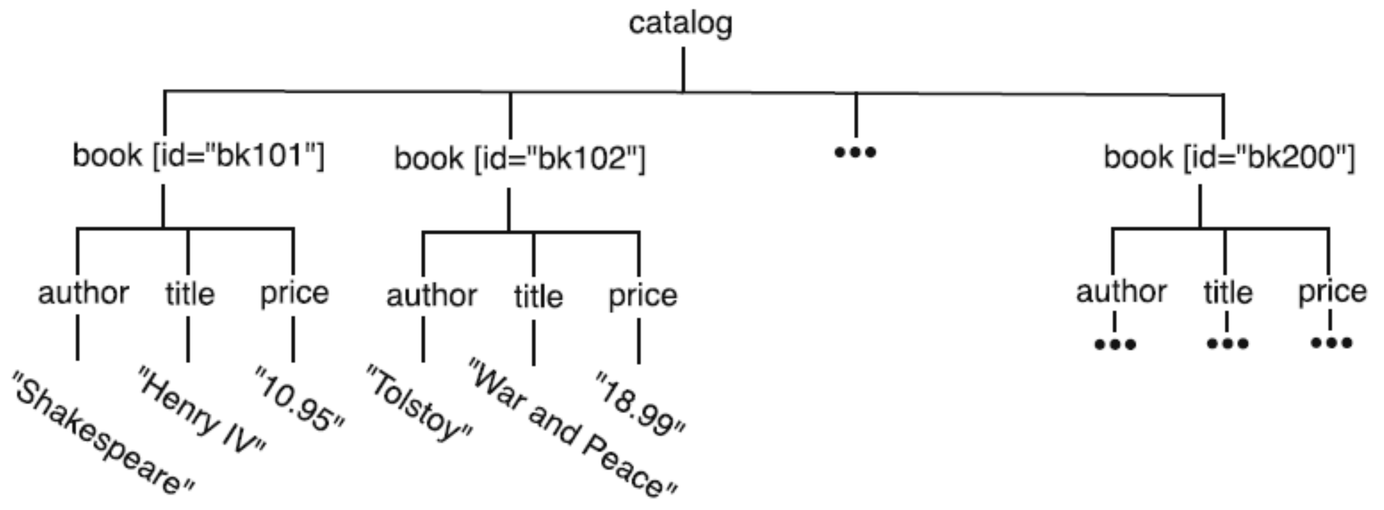
- Data obtained from web scraping and from APIs is often naturally hierarchical.
- We will cover the most widely used formats: JSON, XML, and HTML

More Terminology

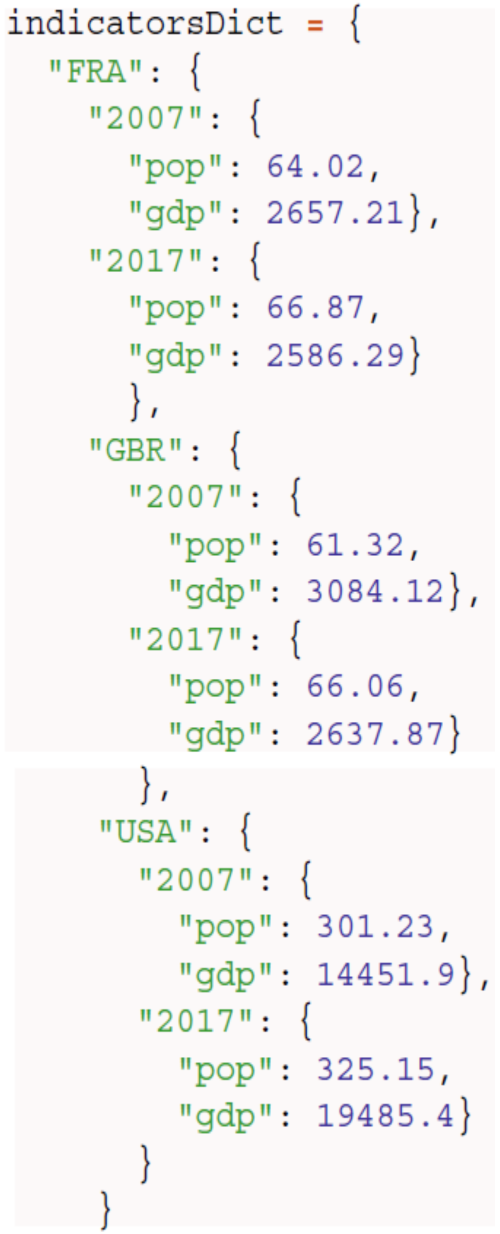


- The node at the top is called the ***root***.
- For a given node, the nodes directly below it are called its ***children***.
- For a given node, the node directly above it is called its ***parent***.
- For a given node *X*, the nodes along the path from the root to *X* are the ***ancestors*** of *X*. Nodes below *X*, whose paths from the root contain *X*, are the ***descendants*** of *X*.

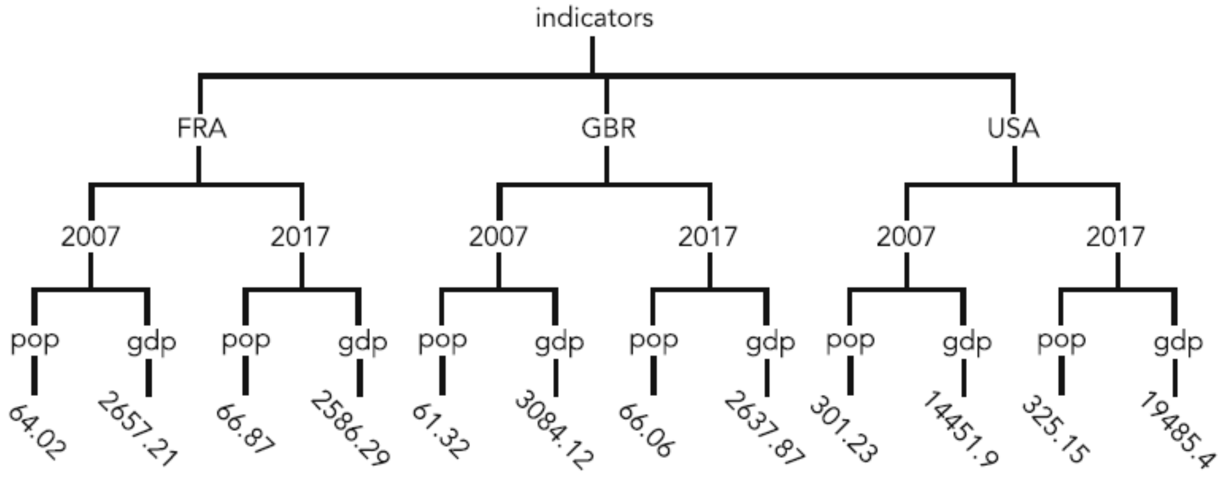
More Terminology



- If two nodes have the same parent, we call them ***siblings***.
- We allow nodes to have a symbolic name, sometimes called a *tag* or *label*.
  - We allow two different nodes to have the same label (e.g., there are 100 nodes labeled author), and we use their location in the tree to distinguish them.
- A node with no children is called a ***leaf*** node.



```
indicatorsDict = {
  "FRA": {
    "2007": {
      "pop": 64.02,
      "gdp": 2657.21},
    "2017": {
      "pop": 66.87,
      "gdp": 2586.29}
      },
    "GBR": {
      "2007": {
        "pop": 61.32,
        "gdp": 3084.12},
      "2017": {
        "pop": 66.06,
        "gdp": 2637.87}
      },
     "USA": {
       "2007": {
         "pop": 301.23,
         "gdp": 14451.9},
       "2017": {
         "pop": 325.15,
         "gdp": 19485.4}
      }
    }
```



- We can represent `ind0` as dictionary whose name corresponds to the root note (`indicators`) and whose keys correspond to the children of the root node (here, “FRA,” “GBR,” and “USA”).

- When we use dictionaries as the primary structure used in the nesting to achieve a tree, we use the ***Dictionary of Dictionaries*** (DoD) term introduced earlier in the course.

Trees as List of Lists

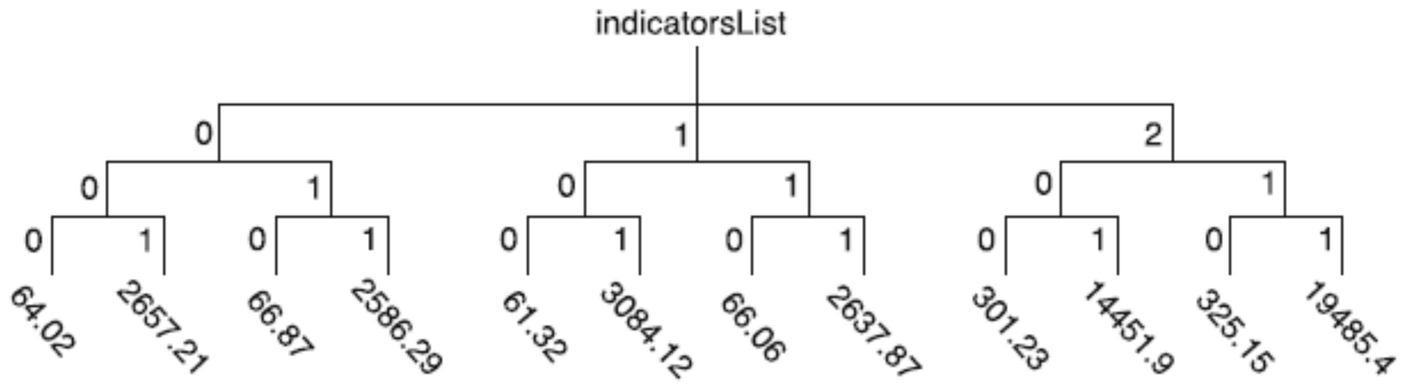
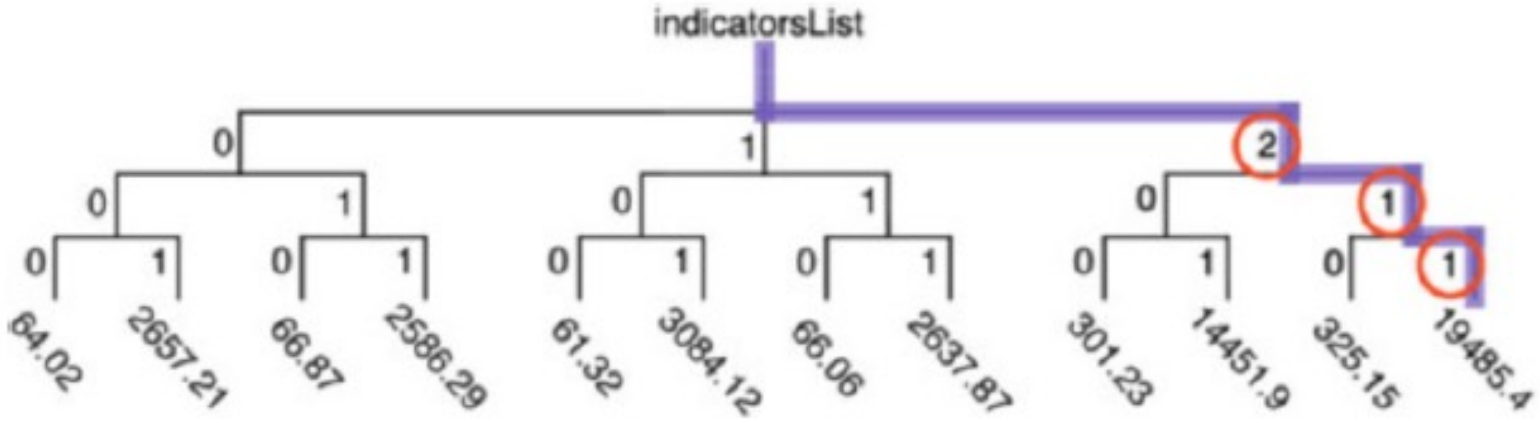


Fig. 15.5 Nested lists as a tree

```
indicatorsList = [[[64.02,2657.21],[66.87,2586.29]],
                  [[61.32,3084.12],[6606,2637.87]],
                  [[301.23,14451.9],[325.15,19485.4]]]
```

Traversals and Paths

- With either of these representations of trees through nested Python data structures, it is easy to build *paths* (*traversals*) in the tree.



JSON

- JSON is a very convenient format for hierarchical data.
- It is important to note that the only scalars allowed for JSON are Booleans, numbers, strings, and a special data type called *null*.
- Thus, a JSON file cannot store a tree of file objects, but can store a tree of strings (including file names).
- Furthermore, strings must be represented by double quotes, instead of single quotes, e.g., “Chris” is an acceptable string, but ‘Lane’ is not.

JSON Specifications

- In JSON, when data is stored inside square brackets, in an ordered list, such as `[1,"one",True]`, the data type is known as an ***array***.

- When data is stored inside curly brackets, with `key:value` pairs, like `{"Noah":2, "Becky":1, "Evan":1}`, the data type is known as an ***object***.

XML

- More powerful and widespread than the JSON format, the ***eXtensible Markup Language*** (XML) is a general structure for modeling hierarchical data.

- XML is a markup language much like HTML.

- XML was designed to store and transport data.

- XML was designed to be self-descriptive.

XML Continued

- XML data is stored as plain text, so an XML file can be opened with any text editor (ideally, one that is “syntax aware”).
- Like JSON, XML is designed to be programming language agnostic, that is, to work with any application that seeks to use it.
- Because XML data is simple text data, it is easy to communicate between operating systems, data systems, applications, etc.
- In web scraping, one often comes across XML pages as well as HTML.
- XML files are also commonly used as a format for data hosted by the US Government.

Difference Between XML and HTML

- XML and HTML were designed with different goals:
  - XML was designed to carry data—with focus on what data is.
  - HTML was designed to display data—with focus on how data looks.
  - XML tags are not predefined like HTML tags are.

XML Does Not Use Predefined Tags

- HTML works with predefined tags like `<p>`, `<h1>`, `<table>`, etc.

- The XML language has no predefined tags.

- These tags are “invented” by the author of the XML document and are analogous to column headings in the tabular model.

- With XML, the author must define both the tags and the document structure.

XML Simplifies Things

- Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.

- XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.

- XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Separates Data from Presentation

- XML does not carry any information about how to be displayed.

- The same XML data can be used in many different presentation scenarios.

- In many HTML applications, XML is used to store or transport data, while HTML is used to format and display the same data.

Transaction Data

- Thousands of XML formats exist, in many different industries, to describe day-to-day data transactions:
  - Stocks and Shares
  - Financial transactions
  - Medical data
  - Mathematical data
  - Scientific measurements
  - News information
  - Weather services

Section 16.4

XPath

Introduction to XPath

- The ability to specify paths for locating and updating resources is fundamental to the hierarchical data model.
- XPath is a language that facilitates path-based operations in XML documents.
- The name “XPath” comes from “XML Path Language.” XPath has a number of important applications:
  - Extracting text or attribute data in XML documents.
  - Retrieving data stored within HTML files.
  - Iterating over web pages or documents in a file structure.

Introduction

- XPath uses a compact, string-based, rather than XML element-based syntax.
- Operates on the abstract, logical structure of an XML document (tree of nodes) rather than its surface syntax.
- Uses a path notation (like URLs) to navigate through this hierarchical tree structure.

Introduction Continued

- XPath defines a way to compute a string-value for each type of node: element, attribute, text.

- Evaluated to yield an object of 4 basic types.
  - node-set (unordered collection of nodes w/o duplicates).
  - boolean (true/false)
  - number (float)
  - string

Introduction Continued

- A location path describes a path from 1 point to another.

- Analogy: Set of street directions.

  “Second store on the left after the third light”

- The location path provides the mechanism for ‘addressing’ items in an XML document.

Expressions in XPath

- An XPath expression is a string, representing a sequence of steps in a tree, making up a path from some “initial” node to some “target” node.

- The initial node is often the root but does not have to be.

- XPath has the same notions of *absolute* and *relative* path to specify a traversal.

ind2.xml

```
<ind2>
  <FRA>
    <y2007>
      <pop>64.02</pop>
      <gdp>2657.21</gdp>
    </y2007>
    <y2017>
      <pop>66.87</pop>
      <gdp>2586.29</gdp>
     </y2017>
  </FRA>
  <GBR>
    <y2007>
      <pop>61.32</pop>
      <gdp>3084.12</gdp>
    </y2007>
    <y2017>
      <pop>66.06</pop>
      <gdp>2637.87</gdp>
    </y2017>
  </GBR>
</ind2>
```

Example XPath String

Path: `/ind2/FRA/y2017`

- This defines an individual internal node (Element):

Return:
```
Element='<y2017>
            <pop>66.87</pop>
            <gdp>2586.29</gdp>
            </y2017>'
```

Example XPath String

Path: `/ind2/FRA/y2017/pop`

Return: `Element='<pop>66.87</pop>'`

Path: `/ind2/FRA/y2017/pop/text()`

Return: `Text='66.87'`

`text()` extracts the text of the current node

Typical Goal

- The goal of XPath is often like `findall()` to get a nodeset, the set of nodes (over multiple path-traversals) that satisfy a given path pattern.
- In Python, once we have an XPath expression, we give it as a parameter to the `xpath()` method of an `ElementTree` or an `Element`, which returns the nodeset.

Path Pattern

- A string argument for a *path pattern* defines location-steps with parts that have to “match” as specified, and other parts that are wildcards for one or more nodes or traversals that can match.

- The logic of wildcard characters in XPath works similar to regular expressions.

Path Pattern Example

Path: `/ind2/FRA/*/pop`

Return: `Element='<pop>64.02</pop>'`
`Element='<pop>66.87</pop>'`

Matches all grandchildren tagged `'pop'` under the `Element` tagged `'FRA'` in `ind2`, regardless of the parent of `'pop'`.

There are two elements in the returned nodeset.

Note: A wildcard matches any node at the given level, but not a path worth of nodes. The example above would not match nodes where `'pop'` is a greatgrandchild of `'FRA'`.

indicators.xml

```
<indicators>
  <country code="USA" name="United States">
    <timedata year="1960">
      <pop>180.67</pop>
      <gdp>543.3</gdp>
      <life>69.8</life>      ⋮
      <cell>0.0</cell>
      <imports>16170.7</imports>
      <exports>20535.0</exports>
    </timedata>
    <timedata year="1961">
      <pop>183.69</pop>
      <gdp>563.3</gdp>
      <life>70.3</life>
       …
```

Expressions

- Certain characters/sequences of characters are *expressions* that have *meaning* that is not interpreted as tagged element names.
- **The following two paths are equivalent**, but the second one uses `[@code = 'FRA']` to specify a *predicate*, which will be satisfied by the country node whose code is `'FRA'`, but not the other country nodes.

Path: `/ind2/FRA/y2007/pop`

Path: `/indicators/country[@code='FRA']/timedata[@year`
`='2007']/pop`

This should be interpreted as a single string without a line break.

Return: `Element='<pop>64.02</pop>'`

Logical Or

- If we wanted the population of France and the USA in *all* years, we would use a logical *or*, yielding the expression

Path: `/indicators/country[@code ='FRA' or @code ='USA']/*/pop`

Return:

```
Element= '<pop>46.62</pop>'
Element= '<pop>47.24</pop>'
Element= '<pop>47.9</pop>'
Element= '<pop>48.58</pop>'
…
```

Alternate Method

- An alternate method for finding the population of France and the USA in all years makes use of the | operator:

Path: `/indicators/country[@code='FRA']/*/pop |`

```
/indicators/country[@code = 'USA']/*/pop
```

- The vertical bar (|) is used between XPaths in the same string to combine the nodeset results from the first XPath with the nodeset results from the second XPath.

school.xml

```
<school>
  <departments>
    <department id="ANSO">
      <name>Anthropology and Sociology</name>
      <division>Social Sciences</division>
    </department>
    …
  </departments>
  <courses>
    <course subject="ARAB" num="111">
      <title>Beginning Arabic I</title>
      <hours>4.0</hours>
      <class id="40184">
        <term>FALL</term>
        <section>01</section>
        <meeting>09:30-10:20 MWRF</meeting>
        <instructorid>9216</instructorid>
      </class>
    </course>
    …
```

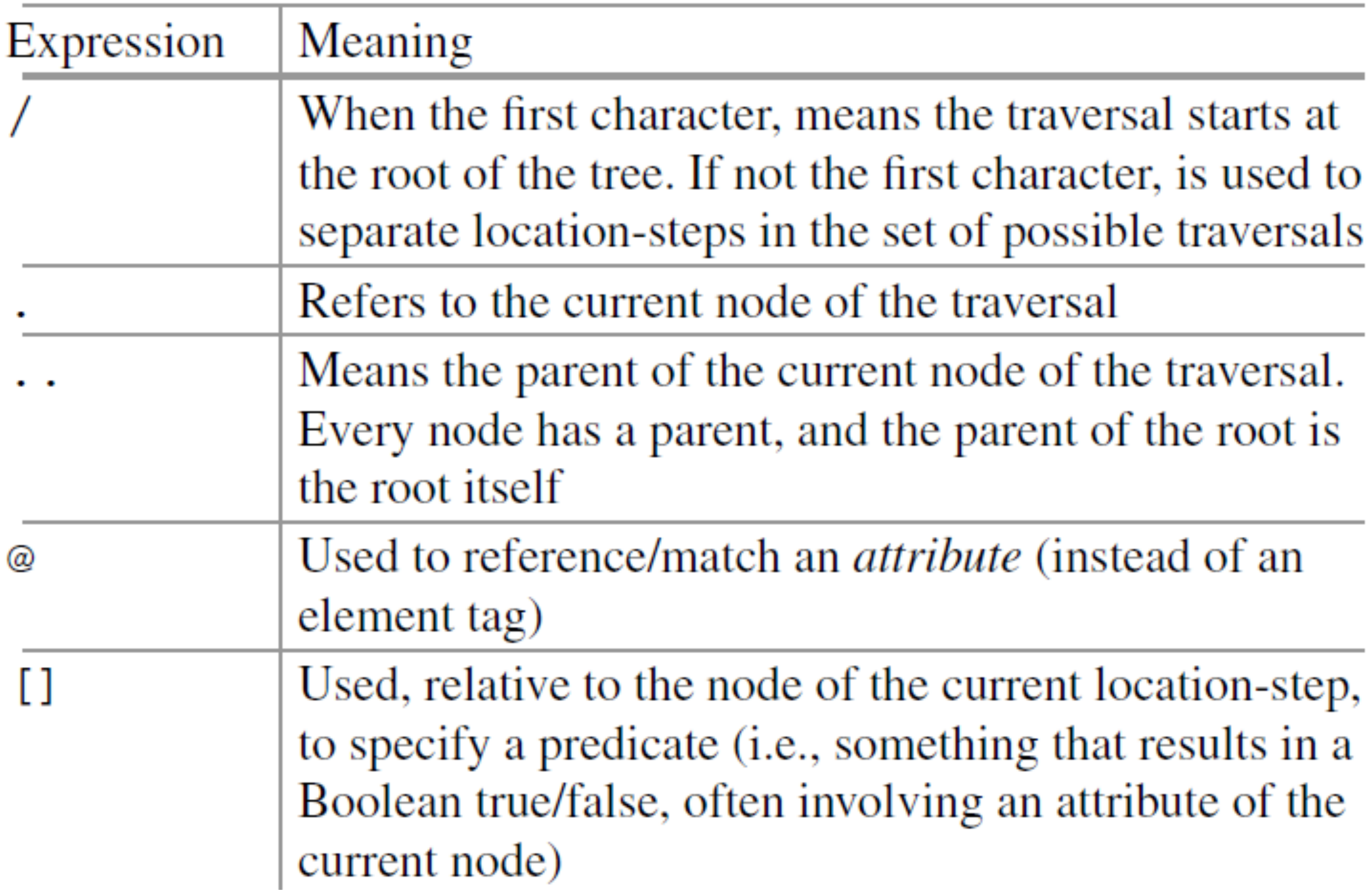
Logical And

- We could use a logical *and* to find the title "Elementary Cinema Prod" with course number 219 in the `school.xml` file as follows:

Path:
```
/school/courses/course[@subject='CINE' and
             @num='219']/title
```

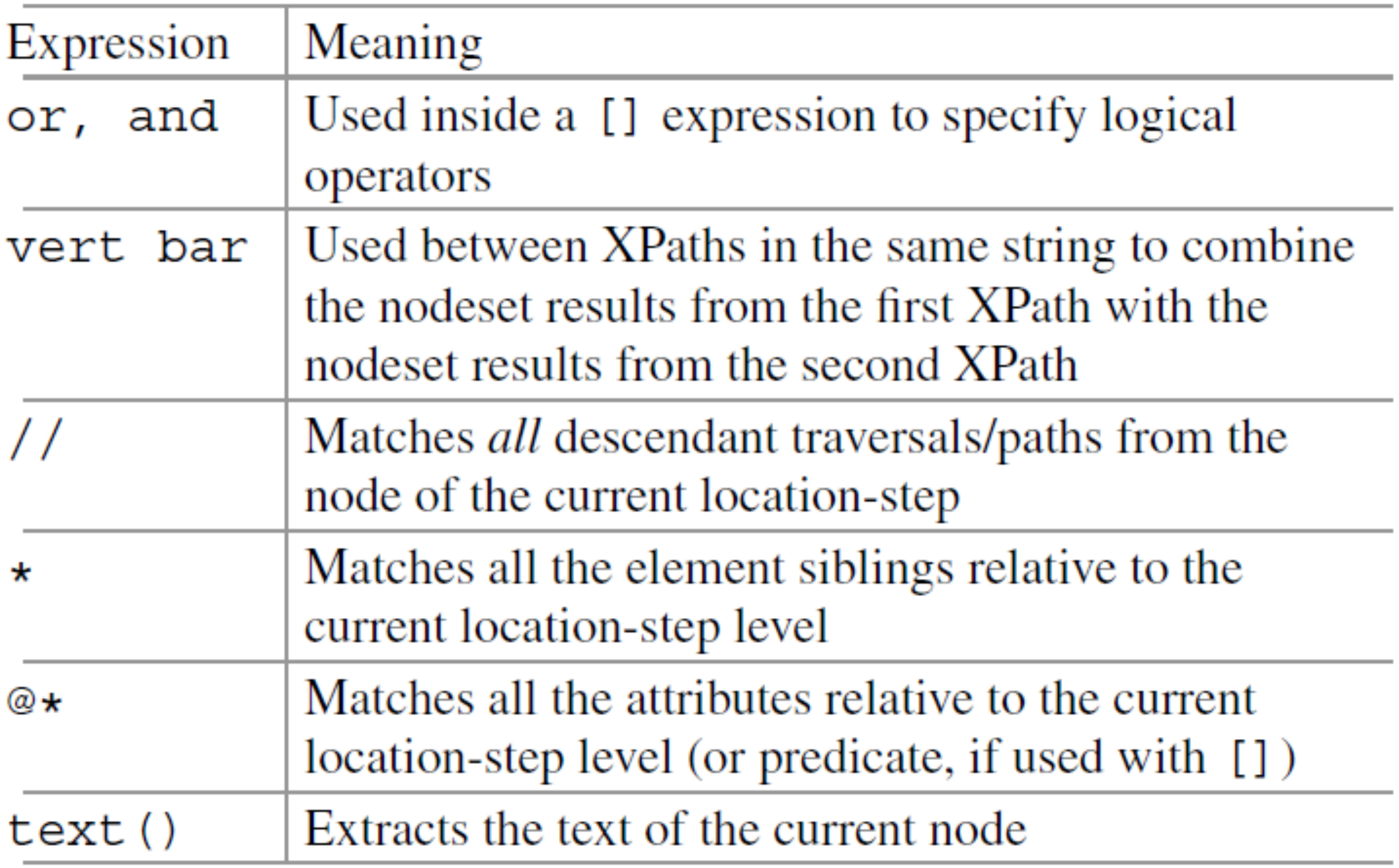
Return:
```
Element= '<title>Elementary Cinema Prod</title>'
```

Expressions



Expression	Meaning
/	When the first character, means the traversal starts at the root of the tree. If not the first character, is used to separate location-steps in the set of possible traversals
.	Refers to the current node of the traversal
. .	Means the parent of the current node of the traversal. Every node has a parent, and the parent of the root is the root itself
@	Used to reference/match an *attribute* (instead of an element tag)
[ ]	Used, relative to the node of the current location-step, to specify a predicate (i.e., something that results in a Boolean true/false, often involving an attribute of the current node)

Expressions



Expression	Meaning
`or, and`	Used inside a `[]` expression to specify logical operators
`vert bar`	Used between XPaths in the same string to combine the nodeset results from the first XPath with the nodeset results from the second XPath
`//`	Matches *all* descendant traversals/paths from the node of the current location-step
`*`	Matches all the element siblings relative to the current location-step level
`@*`	Matches all the attributes relative to the current location-step level (or predicate, if used with `[]`)
`text()`	Extracts the text of the current node

Match All Descendants

- The expression `//` is often a useful start of an XPath.
- We could use this to find all nodes with a specified text. For example, in `indicators` we could match all nodes with population larger than 300 million:

Path: `//pop[text() > 300]`

Backing Up a Level

- Suppose we wanted to find what department offers the course with the title "Elementary Cinema Prod“ in the `school.xml` data.
- We could locate that course then **back up one level** and extract the subject attribute:

Path:
```
//title[text() =
        "Elementary Cinema Prod"]/../@subject
```

Return: `Attribute='subject=CINE'`

breakfast.xml

```xml
<menu>
    <food price="5.95" calories="650">
        <name>Belgian Waffles</name>
        <description>Two of our famous Belgian Waffles with maple syrup</description>
    </food>
    <food price="7.95" calories="900">
        <name>Strawberry Belgian Waffles</name>
        <description>Light Belgian waffles covered with strawberries</description>
    </food>
    <food price="8.95" calories="900">
        <name>Berry-Berry Belgian Waffles</name>
        <description>Light Belgian waffles covered with fresh berries</description>
    </food>
    <food price="4.5" calories="600">
        <name>French Toast</name>
        <description>Thick slices made from our homemade sourdough bread</description>
    </food>
    <food price="6.95" calories="950">
        <name>Homestyle Breakfast</name>
        <description>Two eggs, bacon, toast, and hash browns</description>
    </food>
</menu>
```

Another Example

- The following (**equivalent**) paths extracts all the prices of the food in the `breakfast.xml` data set.

Input: `//@price`

Input: `/menu/food/@price`

Keep in mind that using `\\` can be very slow!

```
Attribute='price=5.95’
Attribute='price=7.95’
Attribute='price=8.95’
Attribute='price=4.5’
Attribute='price=6.95'
```

Searching Text Nodes

- XPath can also help us search text nodes, in a way similar to what we saw with regular expressions and in SQL `LIKE` queries.

- For instance, the `contains()` function allows us to match nodes where the text contains some specified string.

- For example, in the `topnames.xml` data, the expression

  ```
  //name[contains(text(),’Jo’)]
  ```

  matches all name nodes where the text contains `'Jo'`. This includes `'John’` and would also contain `'Joanna'` or `'MoJo’` if those were ever top names.

Searching Text Nodes Continued

- A related function, `starts-with()`, determines whether the text starts with a given string.

- For example,

```
    //name[starts-with(text(),’Jo’)]/text()
```

matches `'John'` but would not match `'MoJo'`.

Python Programming with XPath

- We now show how to use XPath expressions in Python programming.

- We assume that we have already imported `etree` and parsed our set of input files (we will use the `getLocalXML(filename, datadir)` function provided in the next slide.

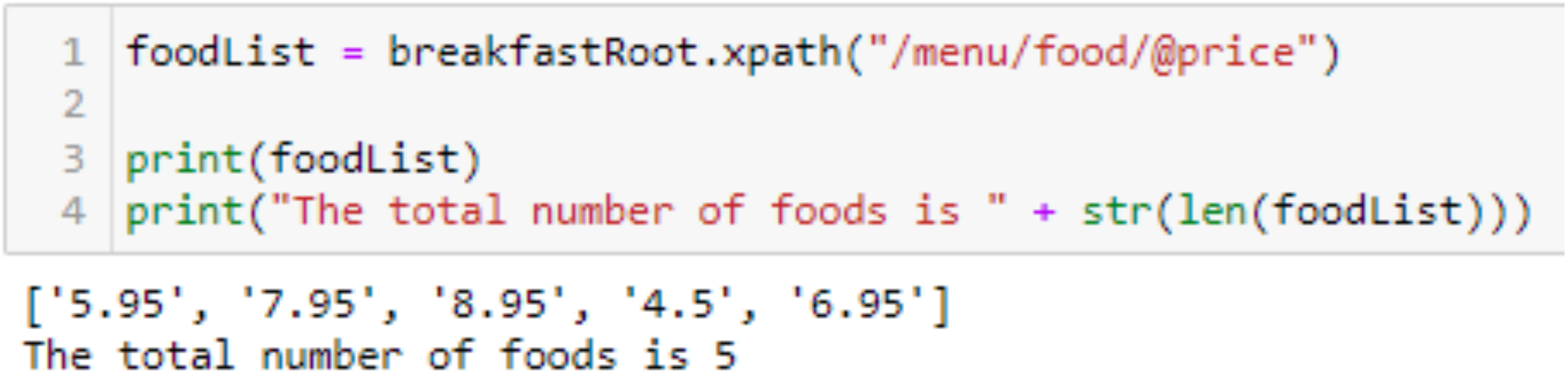
getLocalXML()

```
def getLocalXML(filename, datadir="."):
    # Set the data directory
    xmlPath = os.path.join(datadir, filename)

    try:
        tree = etree.parse(xmlPath)
        root = tree.getroot()
        return root
    except:
        print("Exception in parsing XML")
        return None
```

Obtain Root of `breakfast.xml`

```
breakfastRoot =
     getLocalXML("breakfast.xml", datadir)
```

Python Example 1



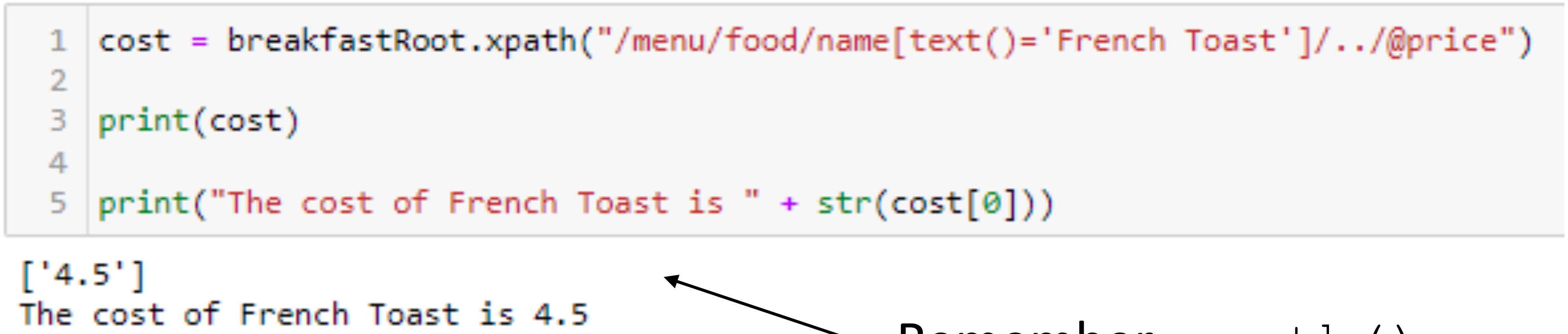
```python
foodList = breakfastRoot.xpath("/menu/food/@price")

print(foodList)
print("The total number of foods is " + str(len(foodList)))
```

```
['5.95', '7.95', '8.95', '4.5', '6.95']
The total number of foods is 5
```

- Note that `xpath()` takes as input an XPath expression as a Python string and **returns a list** of nodes that match.

Python Example 2

- Below we find the cost of French Toast:



```python
cost = breakfastRoot.xpath("/menu/food/name[text()='French Toast']/../@price")

print(cost)

print("The cost of French Toast is " + str(cost[0]))
```

```
['4.5']
The cost of French Toast is 4.5
```

Remember: `xpath()` always returns a list!

Python Example 3

- Below we use functional abstraction to find the cost of a provided food.
- Note the use of the triple double quotes (""").

```python
def findCost(root, food):
    xpath_string = """/menu/food/name[text()='{}']/../@price""".format(food)
    cost = root.xpath(xpath_string)

    return float(cost[0])
```

```python
findCost(breakfastRoot, "French Toast")
```

```
4.5
```

```python
findCost(breakfastRoot, "Belgian Waffles")
```

```
5.95
```

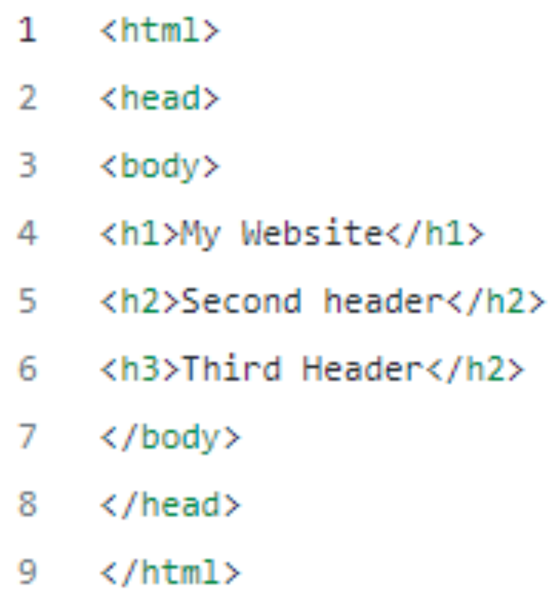
```python
findCost(breakfastRoot, "Egg McMuffin")
```

- This throws an error because the item is not in the data, so `xpath()` returns an empty list.
- The error occurs when we try to access `cost[0]`

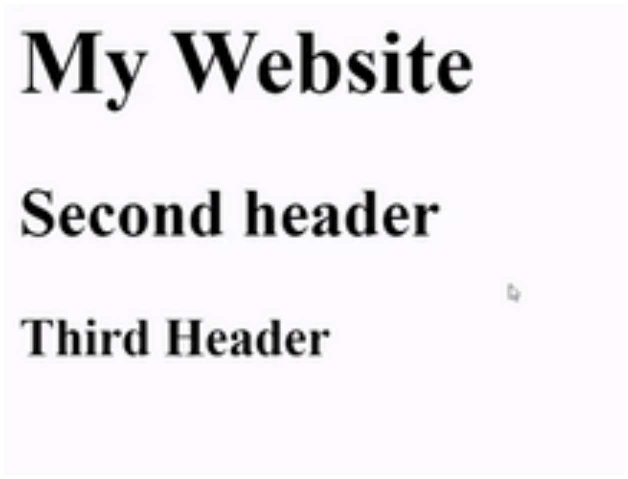
HTML Basics

- HTML stands for HyperText Markup Language.
- It’s not a programming language like Python or Java, but it’s a *markup* language.
- It describes the elements of a page through tags characterized by angle brackets.

Simple HTML Example



```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<h3>Third Header</h2>
</body>
</head>
</html>
```



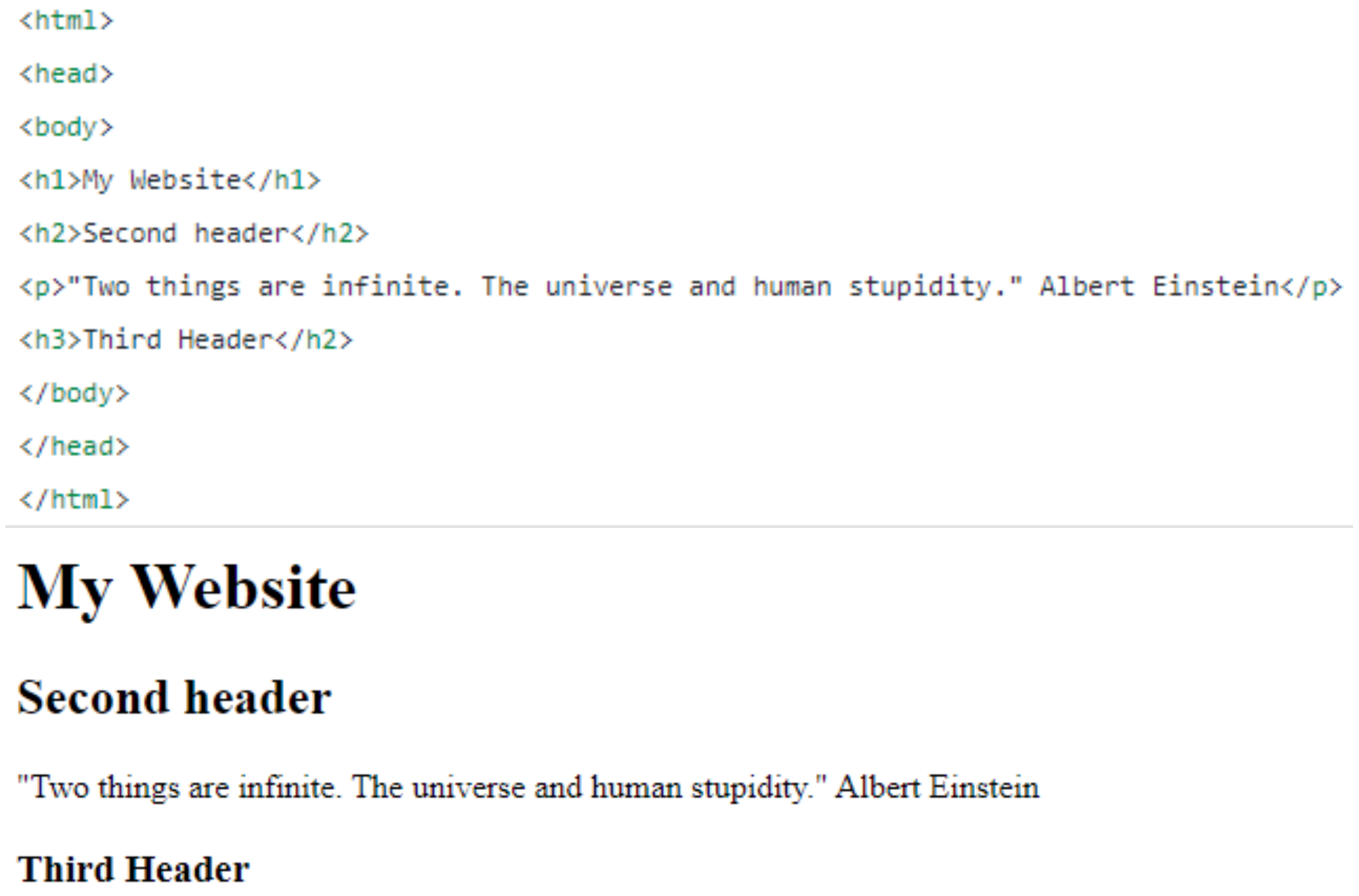
My Website

Second header

Third Header

- The document always begins and ends using `<html>` and `</html>`.
- `<body></body>` constitutes the visible part of HTML document.
- `<h1>` to `<h3>` tags are defined for the headings.

- We can also add a brief paragraph under the second header using the `<p>` tag:



```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." Albert Einstein</p>
<h3>Third Header</h2>
</body>
</head>
</html>
```

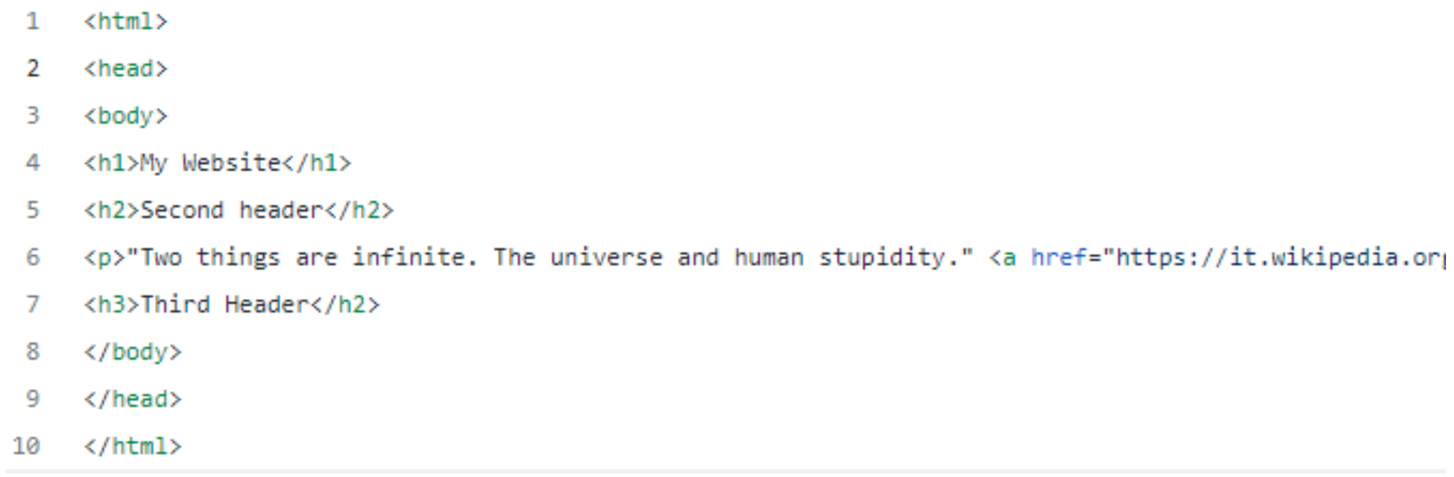
My Website

Second header

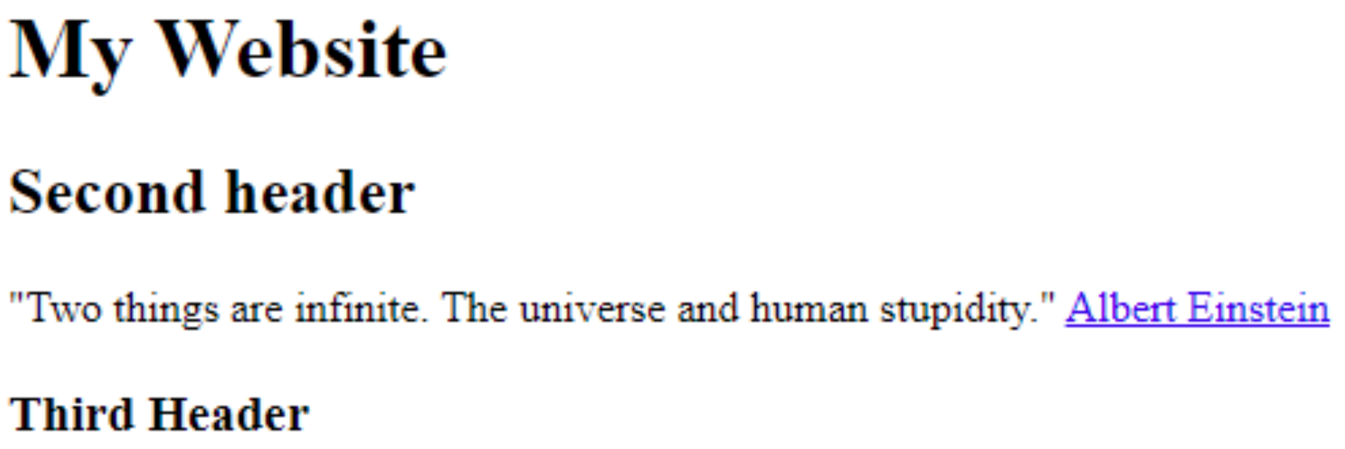
"Two things are infinite. The universe and human stupidity." Albert Einstein

Third Header

- To be more informative, we can add a link on the word “Albert Einstein” to send visitors to the Wikipedia page directly. `<a>` is the tag specialized for HTML links. It has the `href` attribute to specify the link:



```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." <a href="https://it.wikipedia.or
<h3>Third Header</h2>
</body>
</head>
</html>
```



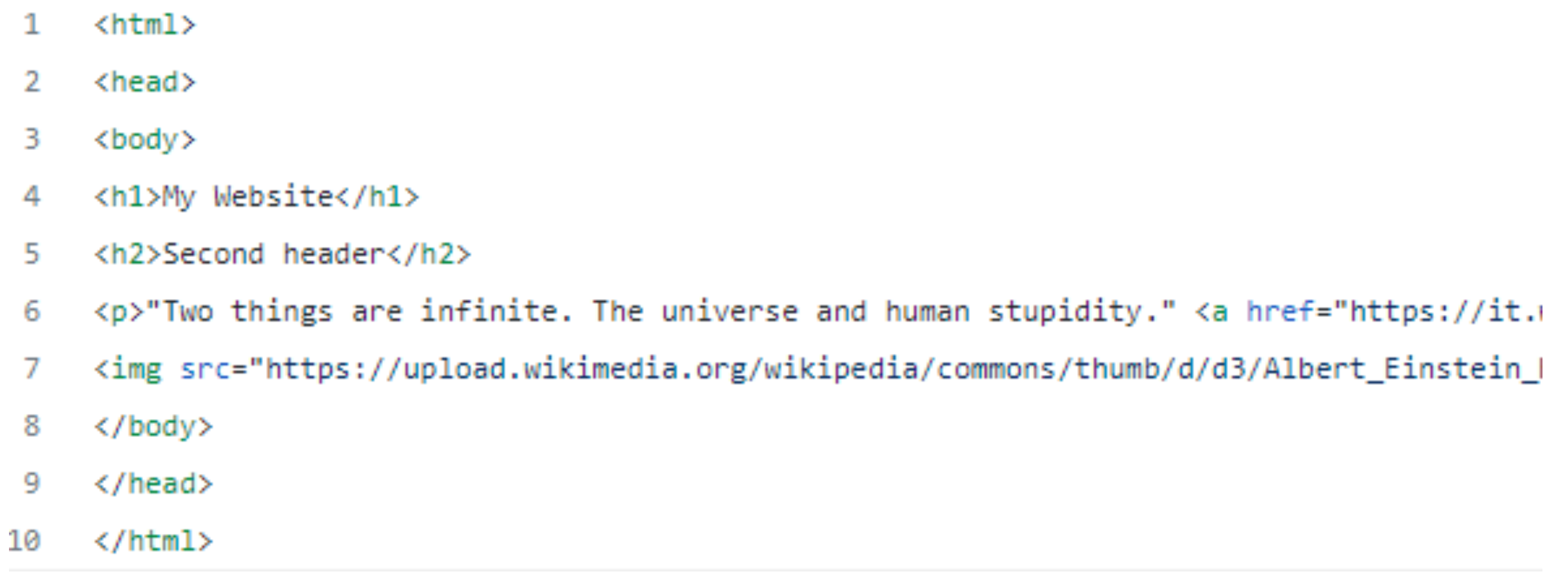
My Website

Second header

"Two things are infinite. The universe and human stupidity." Albert Einstein

Third Header

- We can insert an image of Albert Einstein. The tag for this task is `<img>`, which has the attribute `src` to specify the URL of the image.

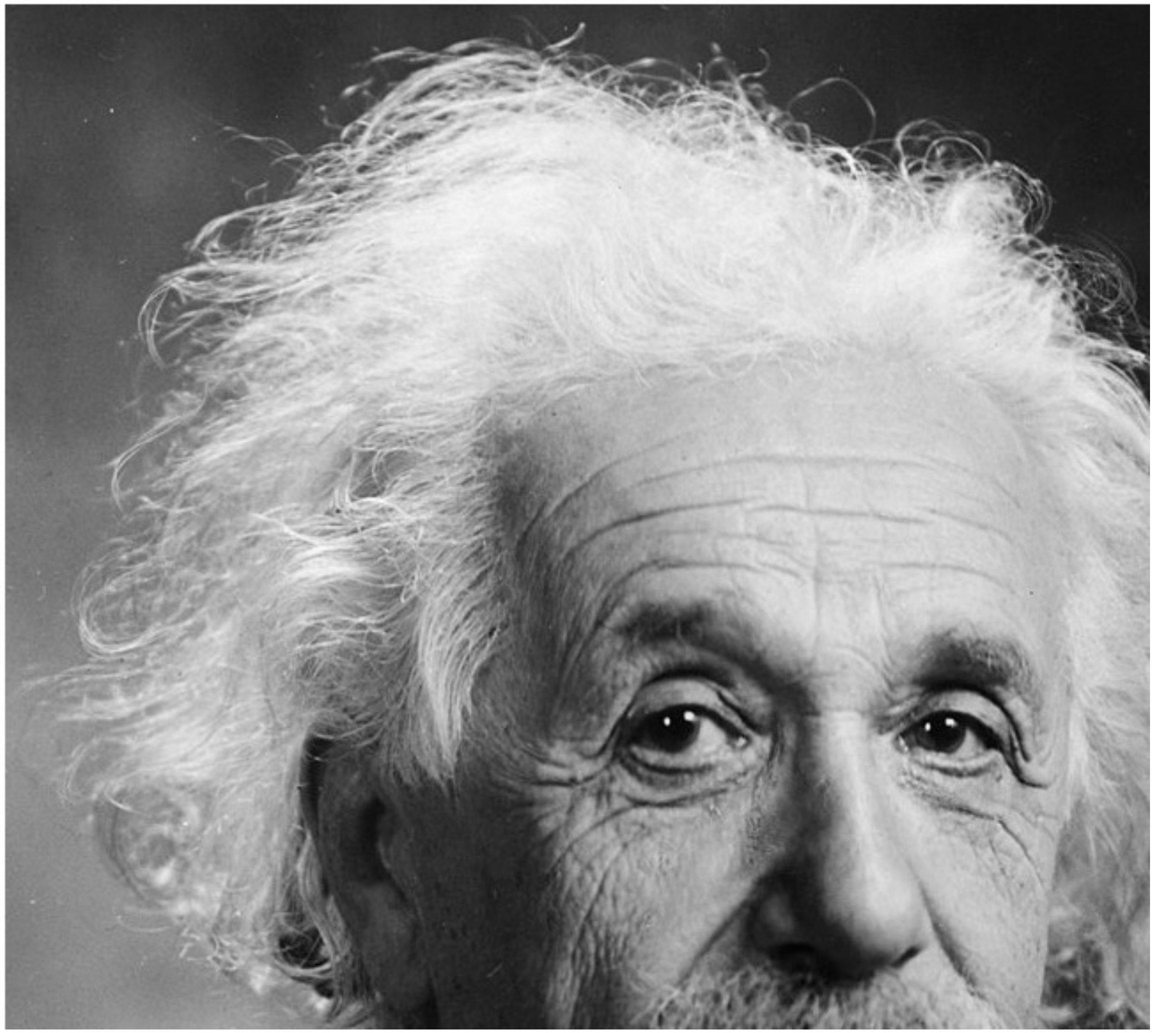


```
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"Two things are infinite. The universe and human stupidity." <a href="https://it.
<img src="https://upload.wikimedia.org/wikipedia/commons/thumb/d/d3/Albert_Einstein_
</body>
</head>
</html>
```

My Website

Second header

"Two things are infinite. The universe and human stupidity." Albert Einstein



Classes and ID

- The `id` is an attribute to specify a unique ID for an element. For example, you want a particular color and size for the title in the first header.

- The `class` is an attribute to define different elements with the same class name. Why do you need the same class in some elements? Because you would probably want to write some phrases with the same font, color, and size.

Classes and ID

- Both IDs and classes are defined in the `<style>` tags and the properties are defined in the curly braces (`{}`).
- The syntax for the class needs the period (`.`) followed by the name of the class, while the ID needs the hashtag (`#`) followed by the name of the ID.
- Once you create the class and the ID in the `<style>` tags, you need to pass them in the elements you want.

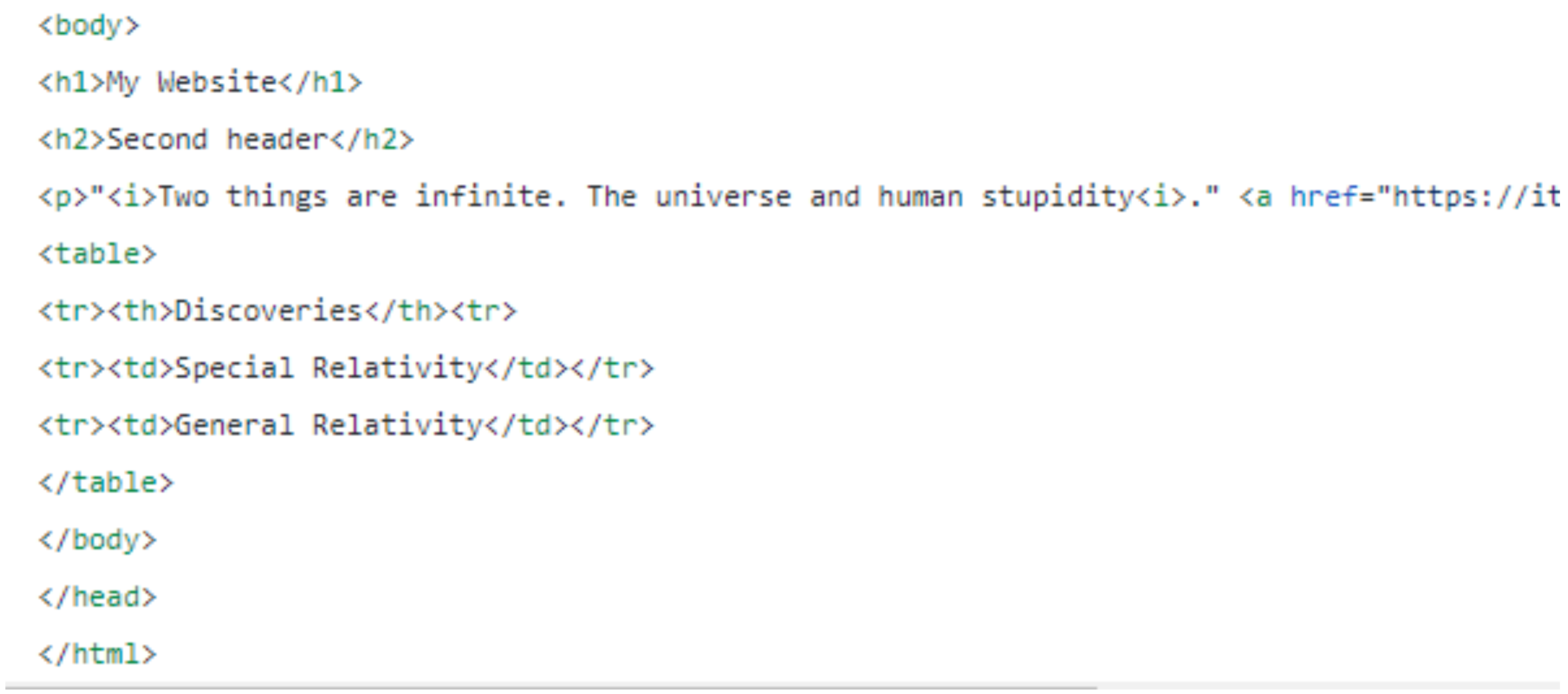
```html
<html>
<head>
<style>
#mytitle {
  font-size: 60;
  color: blue;
}
.cit {
  background-color: pink;
  color: white;
  font-family: Arial, Helvetica, sans-serif;
}
</style>
<body>
<h1 id="mytitle">My Website</h1>
<h2>Second header</h2>
<p class='cit'>"<i>Two things are infinite. The universe and human stupidity<i>.'
<p class='cit'>"<i>Immagination is more important than knowledge<i>." <b>Albert E
</body>
</head>
</html>
```

Tables

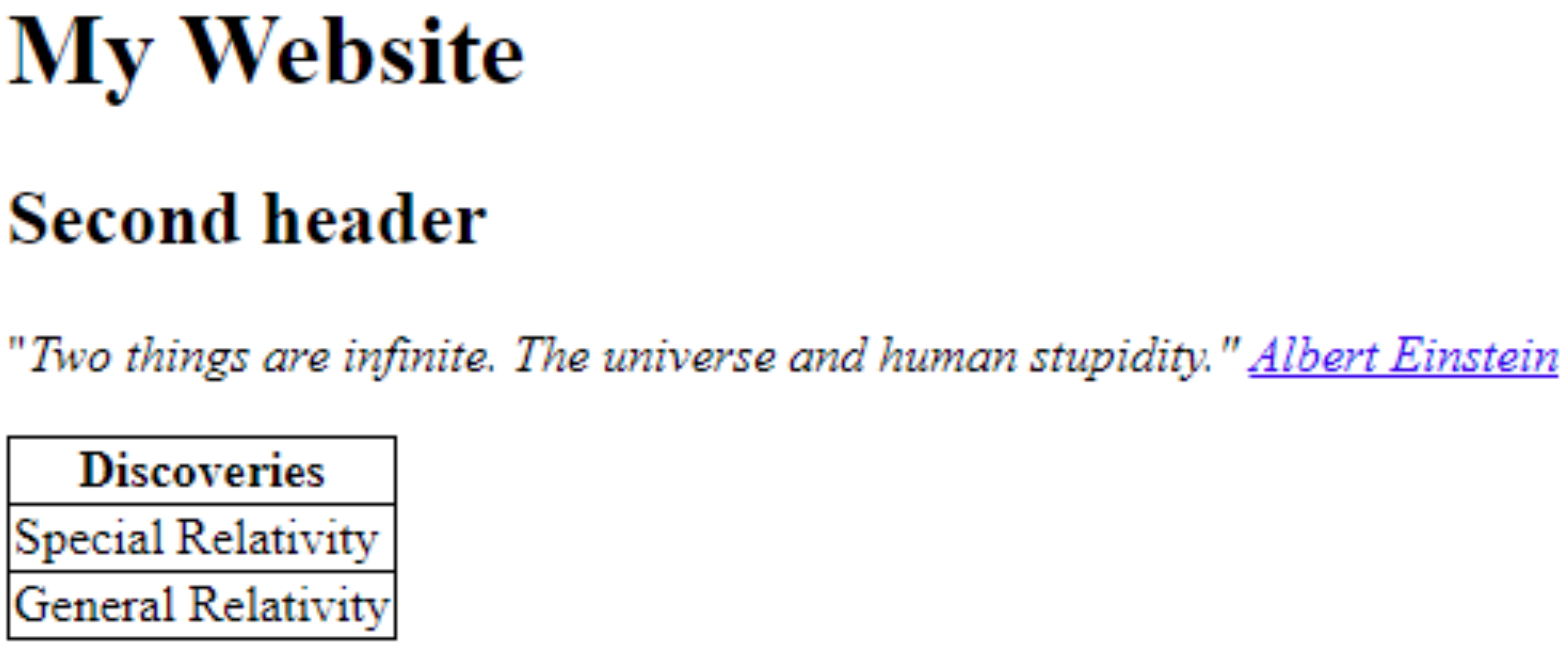
- Another important feature of HTML is the table, which is defined by the `<table>` tag.

- Within the `<table>` tag, there are three principal tags to remember:
  - The `<tr>` tag is used to build each row of the table.
  - The `<th>` tag is used to define the header.
  - The `<td>` tag is used to define the cell within the row.

```html
<html>
<head>
<style>
table, th, td {
  border: 1px solid black;
  border-collapse: collapse;
}
</style>
```



```
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"<i>Two things are infinite. The universe and human stupidity<i>." <a href="https://it
<table>
<tr><th>Discoveries</th><tr>
<tr><td>Special Relativity</td></tr>
<tr><td>General Relativity</td></tr>
</table>
</body>
</head>
</html>
```



My Website

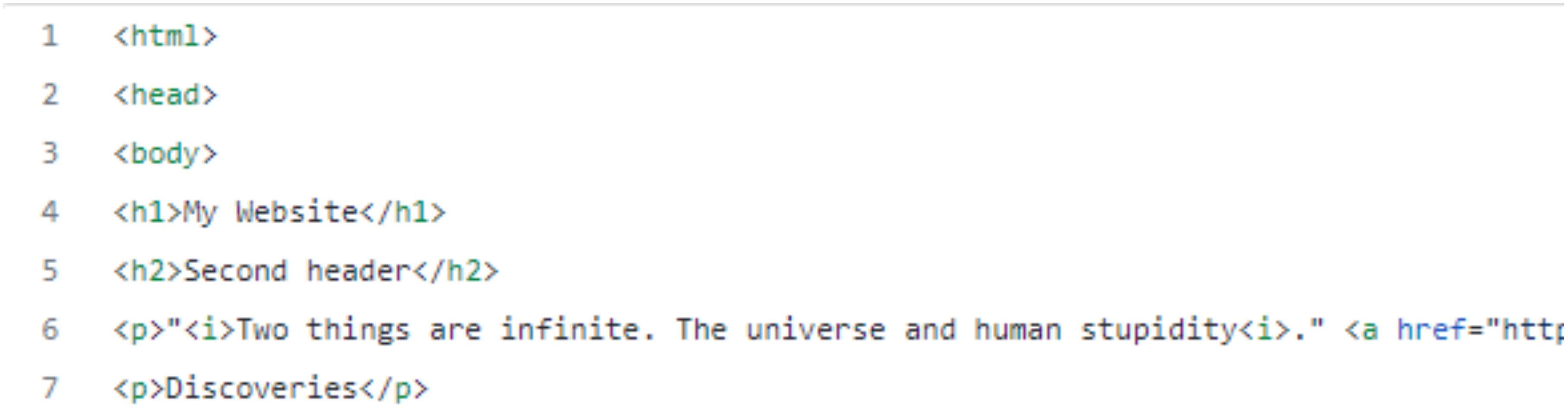
Second header

"Two things are infinite. The universe and human stupidity." *Albert Einstein*

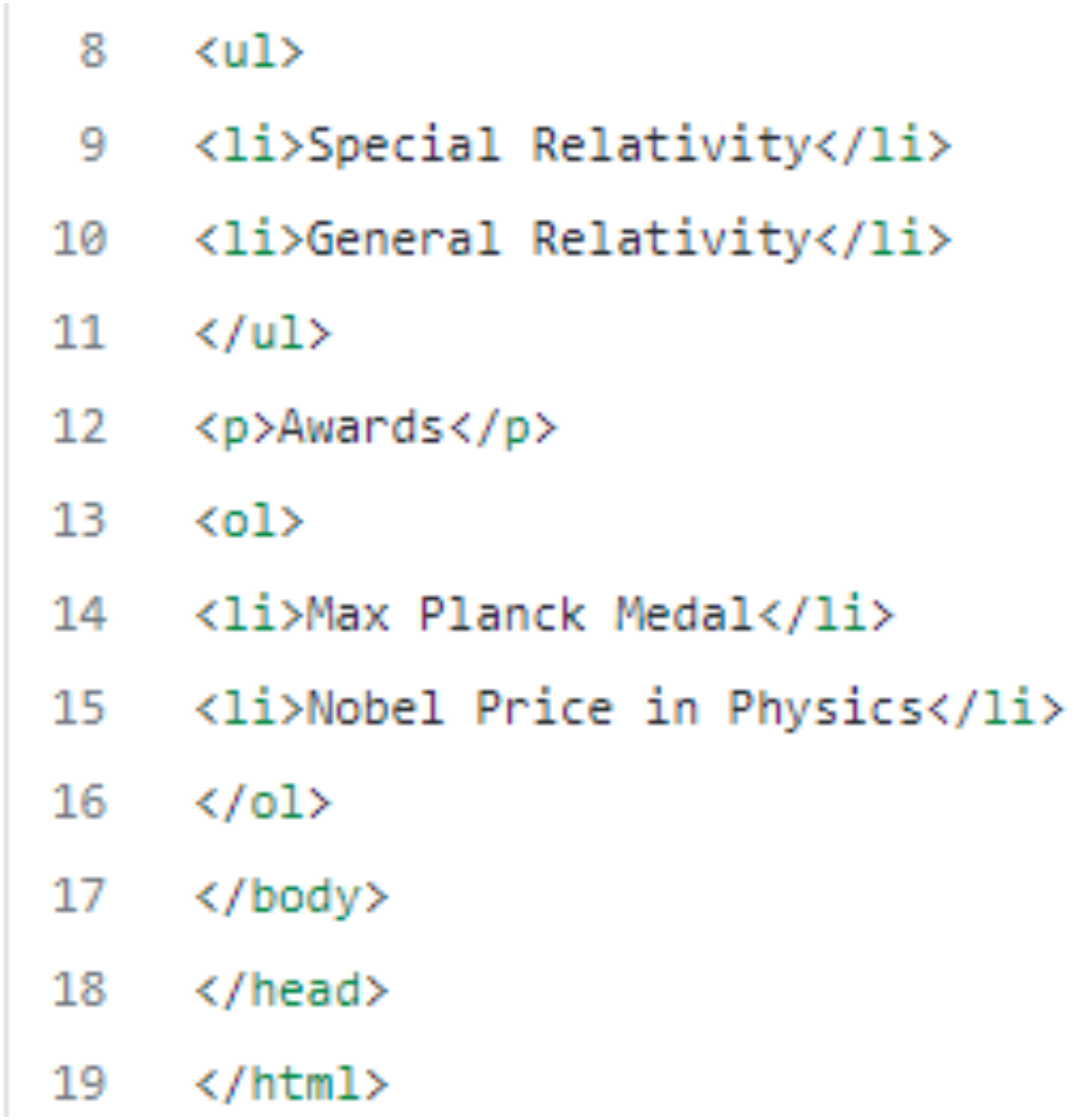
Discoveries
Special Relativity
General Relativity

Lists

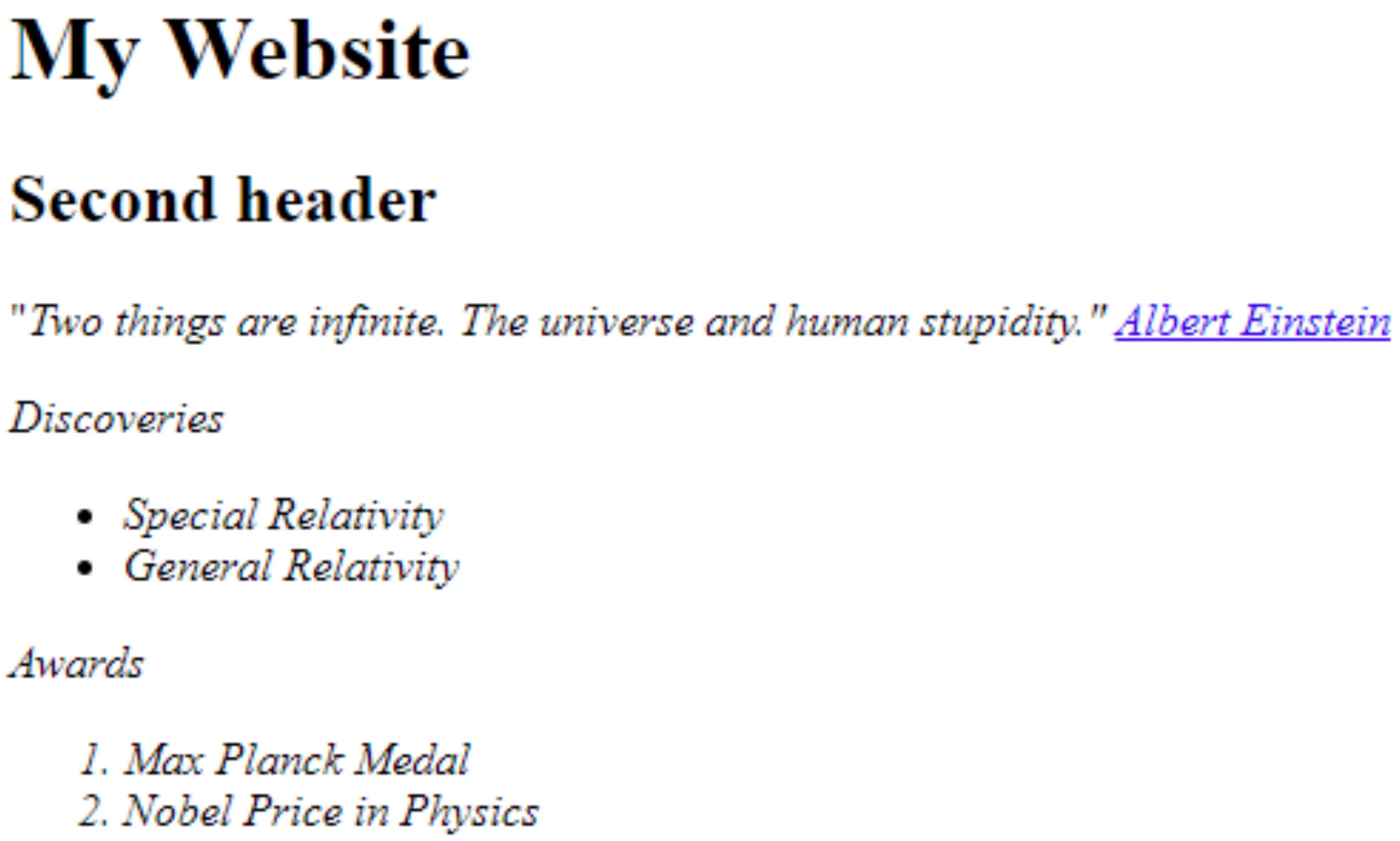
- There are two types of lists that can be defined in HTML.
- The first one is an unordered list that starts with the `<ul>` tag, while the other type is an ordered list specified by the `<ol>` tag.
- Each item of both types of the list is specified by the `<li>` tag.



```html
<html>
<head>
<body>
<h1>My Website</h1>
<h2>Second header</h2>
<p>"<i>Two things are infinite. The universe and human stupidity<i>." <a href="http
<p>Discoveries</p>
```



```html
<ul>
<li>Special Relativity</li>
<li>General Relativity</li>
</ul>
<p>Awards</p>
<ol>
<li>Max Planck Medal</li>
<li>Nobel Price in Physics</li>
</ol>
</body>
</head>
</html>
```



My Website

Second header

"Two things are infinite. The universe and human stupidity." *Albert Einstein*

Discoveries

- *Special Relativity*
- *General Relativity*

Awards

1. *Max Planck Medal*
2. *Nobel Price in Physics*

Blocks

- The most common elements of a website are usually called Blocks or Containers.
- They are useful to group together different elements and apply the same properties. So, the elements we did until now, `<h1>` to `<h3>`, `<p>`, `<ul>`, `<ol>`, can form one block together.
- For example, we want to divide the page into two parts. To create these two different blocks we can specify the `<div>` tag.

```
<html>
<head>
<style>
* {
  box-sizing: border-box;
}

/* Create two equal columns that floats next to each other */
.column {
  float: left;
  width: 50%;
  padding: 10px;
  height: 300px;
}

.row:after {
  content: "";
  display: table;
  clear: both;
}
</style>
```

```html
<body>

<h2>Glossary</h2>

<div class="row">
  <div class="column" style="background-color:#e6e6ff;">
    <h2>General Relativity</h2>
    <p>General relativity, also known as the general theory of relativity, is the geometric the
  </div>
  <div class="column" style="background-color:#9999ff;">
    <h2>Special relativity</h2>
    <p>In physics, the special theory of relativity, or special relativity for short, is a scie
  </div>
</div>

</body>
</head>
</html>
```

Glossary

General Relativity

General relativity, also known as the general theory of relativity, is the geometric theory of gravitation published by Albert Einstein in 1915 and is the current description of gravitation in modern physics.

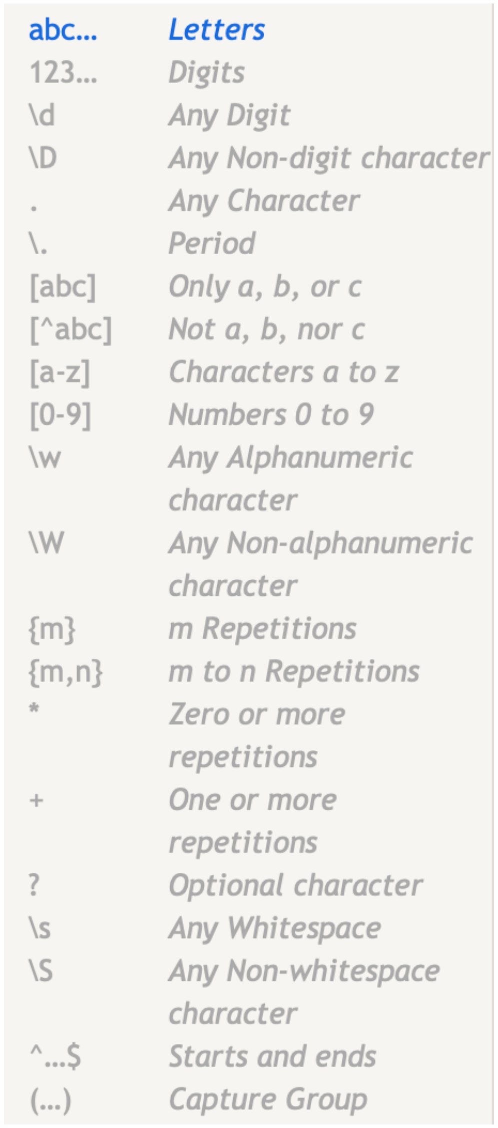
Special relativity

In physics, the special theory of relativity, or special relativity for short, is a scientific theory regarding the relationship between space and time.

Regular Expressions (RegEx)

- Chapter 4.3 in Bressoud and White (not in course pack)
- Used to extract specific groups of texts from repetitive patterns that occur in a larger body of text.
- Very powerful, but can get very complicated quickly!
  - (?<!\d)\d{4}(?!\d)
- Test first before implementing (it’s fun):

https://regex101.com



abc…	Letters
123…	Digits
\d	Any Digit
\D	Any Non-digit character
.	Any Character
\.	Period
[abc]	Only a, b, or c
[^abc]	Not a, b, nor c
[a-z]	Characters a to z
[0-9]	Numbers 0 to 9
\w	Any Alphanumeric character
\W	Any Non-alphanumeric character
{m}	m Repetitions
{m,n}	m to n Repetitions
*	Zero or more repetitions
+	One or more repetitions
?	Optional character
\s	Any Whitespace
\S	Any Non-whitespace character
^…$	Starts and ends
(…)	Capture Group

Regular Expressions (RegEx)

- Can use regular expressions in Python or R.
  - import re in Python

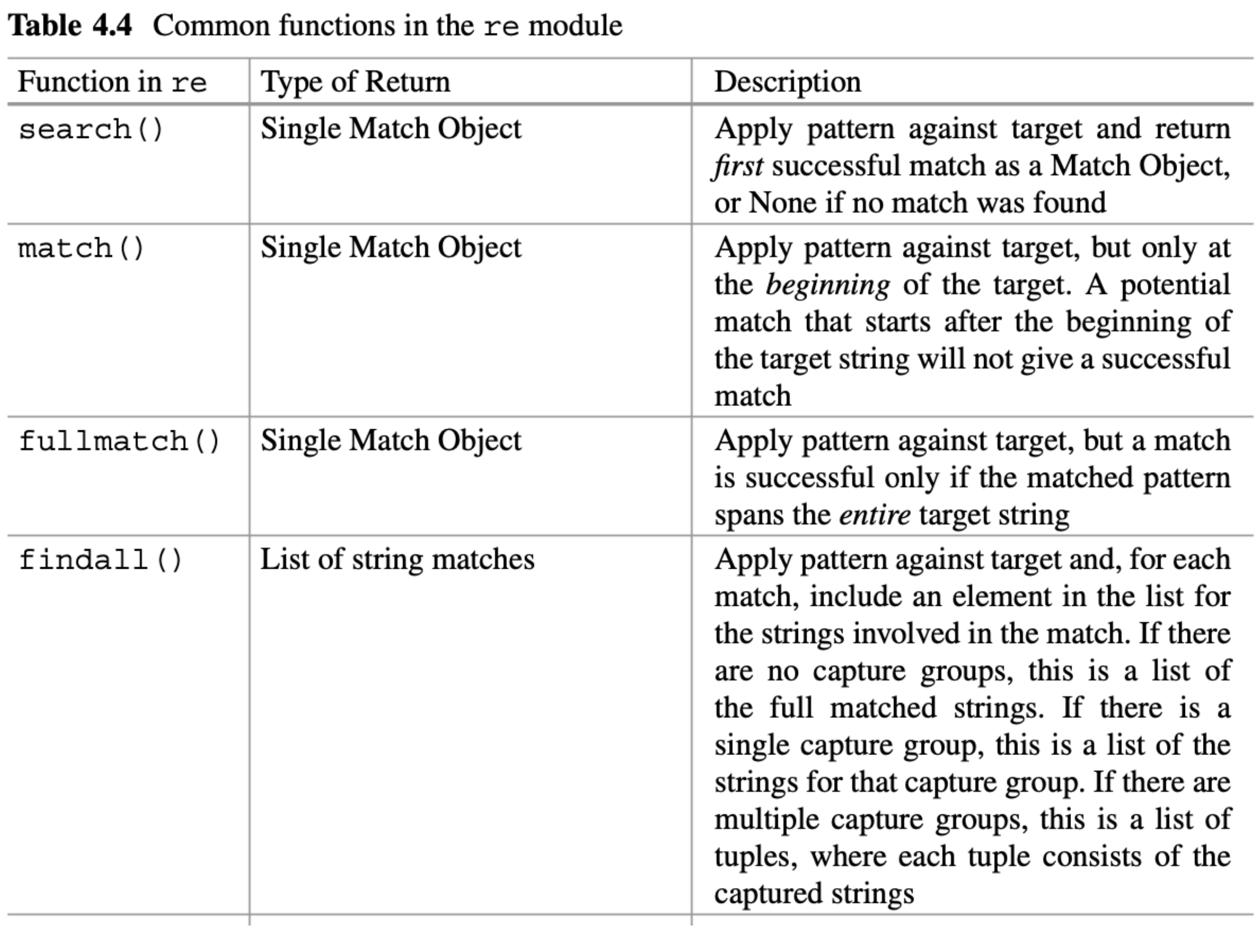


Table 4.4 Common functions in the `re` module

Function in `re`	Type of Return	Description
`search()`	Single Match Object	Apply pattern against target and return *first* successful match as a Match Object, or None if no match was found
`match()`	Single Match Object	Apply pattern against target, but only at the *beginning* of the target. A potential match that starts after the beginning of the target string will not give a successful match
`fullmatch()`	Single Match Object	Apply pattern against target, but a match is successful only if the matched pattern spans the *entire* target string
`findall()`	List of string matches	Apply pattern against target and, for each match, include an element in the list for the strings involved in the match. If there are no capture groups, this is a list of the full matched strings. If there is a single capture group, this is a list of the strings for that capture group. If there are multiple capture groups, this is a list of tuples, where each tuple consists of the captured strings

Regular Expressions (RegEx)

- Examples:

```
import re

text = "It is true that string cleaning is a topic in
this chapter. string editing is another topic."
re.findall("string \w+\s", text)
['string cleaning ', 'string editing ']



new_text = "new text here! "
re.sub("string \w+\s", new_text, text)
It is true that new text here! is a topic in this
chapter. new text here! is another topic.
```

Regular Expressions (RegEx)

- Lazy vs greedy matching:

```
test_string = "stackoverflow"
greedy_regex = "s.*o"
lazy_regex = "s.*?o"

print(f"The greedy match is {re.findall(greedy_regex,
test_string)[0]}")
print(f"The lazy match is {re.findall(lazy_regex,
test_string)[0]}")
The greedy match is stackoverflo
The lazy match is stacko
```

```
# get words between "string" and "topic"

text = "It is true that string cleaning is a topic in
this chapter. string editing is another topic."
re.findall("string (.*?) topic", text)
['cleaning is a', 'editing is another']
```

Chapter 22

Introduction to Web Scraping

What is Web Scraping?

- ***Web scraping*** is a technique for gathering data or information on web pages.
- That is, it is a method to extract data from a website that does not have an API, or we want to extract a LOT of data which we can not do through an API due to rate limiting.
- **Through web scraping we can extract any data which we can see while browsing the web.**

But What is an API?

- The term API stands for “Application Programming Interface.”
- APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.
- For example, the weather bureau’s software system contains daily weather data. The weather app on your phone “talks” to this system via APIs and shows you daily weather updates on your phone.

API Continued

- API architecture is usually explained in terms of client and server.

- The application sending the request is called the ***client***, and the application sending the response is called the ***server***.

- So in the weather example, the bureau’s weather database is the server, and the mobile app is the client.

REST APIs

- REST stands for “Representational State Transfer.”
- REST APIs are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client.
- Many important websites provide APIs, such as:
  - GitHub
  - IMDb (movie database)
  - Wikipedia
  - Twitter

Web Scraping in Real Life

- Extract product information
- Extract job postings and internships
- Extract offers and discounts from deal-of-the-day websites
- Crawl forums and social websites
- Extract data to make a search engine
- Gathering weather data

Web Scraping vs. Using an API

- Web Scraping is not rate limited
- Anonymously access the website and gather data
- Some websites do not have an API
- Some data is not accessible through an API
- and many more !

Essential Parts of Web Scraping

- Web Scraping follows this workflow:
  - Get the website - using HTTP library
  - Parse the HTML document - using any parsing library
  - Store the results - either a db, csv, text file, etc
- We will focus more on parsing.

Useful Libraries for Web Scraping

- Beautiful Soup
- lxml
- Selenium
- Scrapy

When to Use Web Scraping

- Make sure to check APIs or FTP servers before scraping.
- If no other way of getting the data exists, consider web scraping.
- Follow terms of use in all scraping projects.