

# 11 - Markdown, Jupyter, HTML

Today's lecture will have two to three objectives.

- The main one is to learn how to harvest data from web pages that are not primarily designed to harvest data from web pages. This may be relatively easy or relatively impossible, as it may require us to run JavaScript inside the web page to try to convince it that the requests are coming from normal usage and not from the application.
- The second is a side issue: we read XML last time. HTML is a bit similar, so we'll also be doing a bit of reinforcement of the approaches we need for XML as we read HTML.
- The third is more general: we will learn, by the way, about regular expressions that can be used to read arbitrary text. Fortunately, in HTML we are mostly spared this, because there are better libraries for it. However, it is useful to know regular expressions, because we will come across them in many places - even in some form in Word.

## Markdown

Markdown is a simple language for basic text formatting: it allows you to enumerate, bold and italicise, quote, insert code and image titles and web links. We use it everywhere on the web (and elsewhere) where we want some formatted text. Markdown is used for these notes, for everything I post on the Classroom, and for writing comments on various websites.

The language is trivial to learn: what you enclose between the asterisks is written out `*slash*` (because I wrote ``*slash*``), and if there are two asterisks, it is written out `**creek**` (because I wrote ``**creek**``). A lines are written out with minus signs and are written out

- so.

- as

- should be,

if we want to number them,

1. pa

2. by

3. number them.

Prefix headings with one, two, three, four or as many ``#`` characters as necessary (the more, the less important the heading).

Links are written ``[by putting the text in square brackets](https://ucilnica.fri.uni-lj.si)``, and in brackets tell where the link leads.

Anything put in backquotes (`*backtick*`) is printed as "code"; I used this above to make the text between the asterisks or in the link appear with asterisks and square brackets, not to turn it into italics and bold or into a link.

Anyone who wants to know more will look at [plonkec](https://www.markdownguide.org/cheat-sheet/), but there is not much more there either, as there is not much more.

## HTML

Web pages are written in HTML, which is similar to XML, which we learned about last time.

XML is more general. XML is used to record and convey any kind of data - from records of routes travelled or cycled in Strava using the standardised GPX format, to vector images in SVG format. Each of these uses its own specific set of tags, which are written in ``<...>``. Each element must start and end (for example ``<track>`` for the beginning of a trail description and ``</track>`` for the end) and must be properly nested. If there is a ``<track>`` element inside a ``<trkpoint>`` element, the latter must be closed before the former.

Just as GPX and SVG are concrete XML formats for writing paths and vector images, HTML is an XML format for describing web pages. The essential - and unfortunately important - difference is that HTML readers are more forgiving about closing elements. If, for example, an element is closed prematurely before the inner element has been closed, the reader will automatically close the inner element. If we close an element that we haven't even opened, the reader will just ignore it. XML must have a single root element that contains all the others. HTML only has one if it is well behaved. If it's not, the reader just shrugs and reads the document normally anyway. They did this because in the old days of the web, web pages were often typed by hand, like HTML, and they were typed by people who weren't even computer people, so we (computer people) took pity on them. It was still possible to show a web page, and if it looked strange because of errors in the use of tags, the author would just fix it.

As a result, HTML is not so easy to read with a program. Actually, there are two reasons.

- One is the aforementioned: HTML is often incorrectly formatted. This used to be a pain, but nowadays we have libraries like Beautiful Soup that save us the drudgery and instead of ignoring the errors, read the HTML into a tree structure.
- The second problem is that HTML is not fundamentally about writing data, it's about writing text. If that text contains data, it may not be formatted in a way that makes it easy to extract.

## HTML basics

To be able to read anything from HTML, we need to have at least a rough understanding of its structure.

### Tags

Here are the basic tags (basic in the sense that we will need them when parsing data, not when building pages).

- A ``<p>`` is a paragraph. HTML ignores empty space, including blank lines, so paragraphs should be marked with P.
- ``<div>`` and ``<span>`` are units within the text. What they mean and what the difference between them is is irrelevant to us, we mention them because what we are looking for will often be within DIVs with a particular tag.
- ``<h1>``, ``<h2>``, ``<h3>`` ... are the title, subtitle, sub-subtitle and so on.
- The ``<a>`` is a hyperlink. This is probably the one we will be most interested in.
- The ``<img>`` is the image.
- The ``<ol>`` and ``<ul>`` are ordered and unordered lists. Inside them will be indented ``<li>``.

## Attributes

Elements have attributes, these will be important when searching for data because a page may have lots of DIV or A tags, but we are interested in those that are tagged with a particular attribute. Here are the most important

- The `class` defines the formatting of the elements. How it does this is not important. What is important is this: if we have a web page with movie titles, all the titles are likely to be formatted the same way - same font, same format, and so on. The titles might be in `<div>` elements, and they will all have a specific one, for example `<div class="movie-title" ...>` or `<div class="title-of-the-movie" ...>` or `<div class="ipc-metadata-list-summary-item__t">` ... until we look, we can't know. So if we want to get to movie titles, we pick up all the `<div>`'s, they have a certain desired class. The movie poster images we see next to the links are maybe `img` with class `ipc-image` (but only on IMDB). The classes are like some kind of variables that the page builders come up with themselves, and we can look at them.

- `id` is a bit similar, but completely different. :) Ids are unique: there can only be one element with such an id on each page. Ids can also define the design, but often they also serve to allow a program running inside the page to read or modify the ids. An id may be used to denote a page title (which may contain the information we want to read). For example - if we are just on IMDB, the `imdbHeader` is used to denote the line at the top of the page.

- The `href` attribute is an attribute of the `a` tag (and a few others we're not interested in). The `href` attribute contains the URL to which the link points. This will obviously come in handy if we want to write a program that reads all the links to certain subpages and reads what those links point to.

- The `name` is also found next to the `a` tag. This does not link to a web page, but indicates a location within that web page so that other pages can refer to it. More on this below.

- The `src` attribute is an attribute of the `img` element. It contains a link to the image. If we use `urlopen` in Python to open (and then use `read` to read) a url written in `href` we get bytes (hopefully we still remember - this is what a string is like, but it's not made up of characters but numbers) which we can, say, store in a file with the appropriate extension, and we have an image.

There are hundreds and hundreds of attributes and tags. We have learned enough about them here to cover a few basic examples and exercises.

## URLs

There is one more thing to say about URLs. These, like file paths, can be absolute or relative and worse.

- If a URL starts with `http://` or `https://`, it leads to a completely different server. (Or to the same one, just with a URL that is too long.) It will be followed, of course, by the address of the server, for example `ucilnica.fri.uni-lj.si`.

- If the URL starts with `ftp://` or `mailto://` or `somethingelse://`, then it doesn't lead to a web page, but to some other server or somewhere else entirely. For example, a `mailto://` link causes a mail sending program to open.

- Absolute paths: if a URL starts with `/`, it is, as with files, an absolute path, of course within an existing server. If the URL `/something/other.html` is encountered on the `http://primer.com/primer/uporaba.html` page, it will lead to the

`http://primer.com/nekaj/drugega.html` page. The server, therefore, remains the same, but the path is replaced by a new one.

- Relative paths: if the URL does not start with `/` (or even `http://`), it is a relative path. If `http://primer.com/primer/uponaba.html` encounters the URL `/something/other.html`, it \*probably\* leads to `http://primer.com/primer/nekaj/drugega.html`. The link is therefore \*probably\* relative to the path the current page is on. Why \*most likely\*? Because a web page can specify otherwise by adding a `

- Headings within the page. If the web page `http://primer.com/primer/uponaba.html` contains `

As with everything in these notes, the same is true for URLs: web developers are among the highest paid programmers for a reason. Here we are just roughly (and in places wrongly) scratching the surface of something that requires years of training.

## Reading HTML

To read HTML, we will use the Beautiful Soup library. This does not come with Python, but must be installed as an extra.

In Jupiter, the easiest way to do this is to type

```
...
```

```
%pip install bs4
```

```
...
```

and then restart Python (Kernel / Restart) if necessary.

We will now read from the Project Gutenberg page the names of all authors whose surname starts with the letter B. They can be found at `https://www.gutenberg.org/browse/authors/b`.

```
from urllib.request import urlopen

html = urlopen("https://www.gutenberg.org/browse/authors/b").read()
```

As we hopefully remember, we got bytes - let's look at the beginning.

```
html[:100]
```

```
b'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n
```

It looks quite normal, like ASCII, but later on it contains all sorts of things, from Portuguese a's to Chinese a's, so we have to decode it as UTF. (Incidentally, he also says, in the above, that his "charset" is the same as "UTF-8".

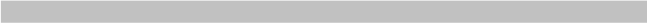
```
html = html.decode("utf-8")
```

Python

```
html[:100]
```

Python

```
'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n <meta char
```



**\*\*Warning:\*\*** Project Gutenberg changed its pages just a day or two after these notes were written. We will therefore read here the page as it was at the time of the notes and as we have it on file.

```
html = open("authors-b.html").read()
```

And now we read the HTML into the tag tree, thanks BeautifulSoup.

```
from bs4 import BeautifulSoup
```

```
soup = BeautifulSoup(html)
```

And now? Now we really need to look at the website, at the code, and see what tags the authors' names are written in.

Go to the page and press Ctrl-Alt-I on Windows or Cmd-Alt-I on macOS; this works in Chrome, Firefox and Safari, but anyone using something I haven't tried should try it themselves. The HTML structure tree appears in the bottom (or right, depending on your settings) of the window, viewed in the top (or left) pane.

**Bååth, A. U. (Albert Ulrik), 1853-1912 ¶**

- [Nordmanna-Mystik: Bilder Från Nordens Forntid](#) (Swedish) (as Author)

**Babbage, Charles, 1791-1871 ¶**

- [en.wikipedia](#)
- [The calculating engine](#) (English) (as Author)
- [On the Economy of Machinery and Manufactures](#) (English) (as Author)
- [Passages from the Life of a Philosopher](#) (English) (as Author)
- [Reflections on the Decline of Science in England, and on Some of Its Causes](#) (English) (as Author)

**Babbitt, Ellen C. ¶**

- [Jataka tales](#) (English) (as Author)
- [More Jataka Tales](#) (English) (as Author)

**Babbitt, George Franklin, 1848-1926 ¶**

**Developer Tool:**

**Elements:**

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
  <head>...</head>
  <body cz-shortcut-listen="true">
    <div class="container">
      <!-- start body -->
      <nav>...</nav>
      <div class="page_content">
        <!-- start content -->
        <div class="pgdbnavbar">...</div>
        <div class="pgdbbbyauthor">
          <h2>...</h2>
          <ul>...</ul>
          <h2>...</h2>
          <ul>...</ul>
          <h2>...</h2>
          <ul>...</ul>
          <h2>...</h2>
          <ul>...</ul>
          <p>...</p>
          <h2>...</h2>
          <ul>...</ul>
          <h2>...</h2>
          <ul>...</ul>
        </div>
      </div>
    </body>
  </html>
```

**Styles:**

```
element.style {
}

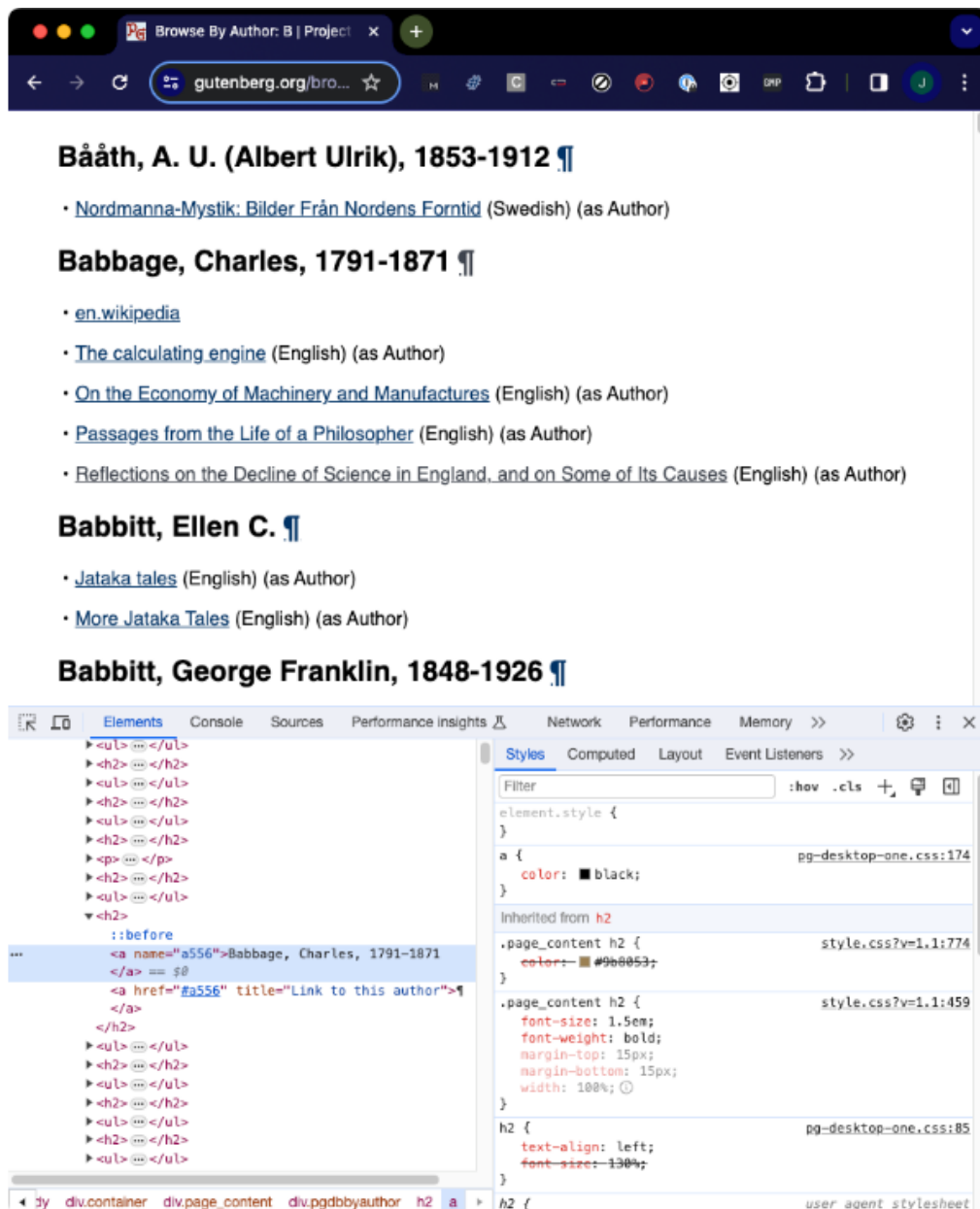
ul {
  list-style-type: "\2022";
  margin: 0;
  padding-left: 10px;
}

ul {
  display: block;
  list-style-type: disc;
  margin-block-start: 1em;
  margin-block-end: 1em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
  padding-inline-start: 40px;
}

Inherited from body
html, body {
  color: black;
  background-color: white;
}

html, body {
}
```

Now, if we move the mouse over the elements of the tree in the lower part, it highlights the corresponding elements in the upper part. And vice versa: if we select the square with the arrow that is in the row between the top and bottom of the window, we can click on the elements in the top part and the corresponding tree element in the bottom part will be shown. For example, if I click on Charles Babbage, this happens.



Apparently the authors' names are inside H2, i.e. as subtitles. They contain two `a` tags. The first one sets the link within the page, `name`, to some code - for Babbage this is a556. This means that a link to the page `https://www.gutenberg.org/browse/authors/b#a556` (the URL of that page with an additional `#a556`) puts us at Babbagea instead of the start of the page. Within this link is his name and the years of his birth and death. This is followed by another `a` element containing the text "¶", and the link points to <https://www.gutenberg.org/browse/authors/b#a556>. Clicking on this ¶ does not change the page, but it does change the address in the browser so that it can be copied or saved. (The first `a` also makes the name look like a link, but does nothing. I really like those.)

Now we probably know where to find the authors' names: we'll try just about any `a` element that contains `name`. Three things can happen

- we get some other charm from the page besides the authors' names,
- we don't get all the authors' names
- we get all the authors' names and nothing but all the authors' names.

Of course, there is also a variant 4, where we don't get all the authors' names, but we do get an extra charm.

If the former happens, we will consider how to be more specific. We might say that this `a` must be within `h2`. If the second happens, we will have to look at the missing authors and think about what is specific to them, how to find them and whether we could find others in the same way.

Let's listen to the Latins, who knew how to say that *\*audentes fortuna iuvat\**. `soup` has a method `find\_all` which takes as argument the name of an element and returns all elements with that name.

```
soup.find_all("a")
```

```
[<a class="no-hover" href="/" id="main_logo">
  
</a>,
<a href="/about/">About
  <span class="drop-icon">▼</span>
</a>,
<a href="/about/">About Project Gutenberg</a>,
<a href="/policy/collection_development.html">Collection Development</a>,
<a href="/about/contact_information.html">Contact Us</a>,
<a href="/about/background/">History & Philosophy</a>,
<a href="/policy/permission.html">Permissions & License</a>,
<a href="/policy/privacy_policy.html">Privacy Policy</a>,
<a href="/policy/terms_of_use.html">Terms of Use</a>,
<a href="/ebooks/">Search and Browse
  <span class="drop-icon">▼</span>
</a>,
<a href="/ebooks/">Book Search</a>,
<a href="/ebooks/bookshelf/">Bookshelves</a>,
<a href="/browse/scores/top">Frequently Downloaded</a>,
<a href="/ebooks/offline_catalogs.html">Offline Catalogs</a>,
<a href="/help/">Help
  <span class="drop-icon">▼</span>
</a>,
<a href="/help/">All help topics →</a>,
<a href="/help/copyright.html">Copyright How-To</a>,
...
<a href="/ebooks/53877">The Sanitary Condition of the Poor in Relation to Disease, Poverty, and Crime
<br/>With an appendix on the control and prevention of infectious diseases</a>,
<a name="a25535">Baker, B. Granville (Bernard Granville), 1870-1957</a>,
<a href="#a25535" title="Link to this author">¶</a>,
...]
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

A word of warning: the bulb is behaving strangely. If we get stuck on a method name, it returns `None` instead of `AttributeError`.

```
print(soup.tralala)
```

None

It does this because `soup.tralala` or `soup.whatever` returns the (first) `tralala` or `whatever` tag in the document. If it doesn't exist (more precisely: because it doesn't exist), it just returns `None` instead of an error. This is a pain if, say, we forget `\_` in `find\_all`.



```
soup.findall("a")
× raises-exception + Tag

-----
TypeError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 soup.findall("a")

TypeError: 'NoneType' object is not callable
```

`soup.findall` is `None`, so it cannot be called with `"a"` or any other arguments. Or at all.

Well, if we've already learned how to get to the `a` element cheaply, let's just grab the first one that comes along.

```
link = soup.a
```

How do we know that the author's name is not hidden in it? As we said: elements with authors' names. All the elements in the county have `attrs` attributes, which contains all the attributes. The part of the HTML that contains our `link` is

```
link

<a class="no-hover" href="/" id="main_logo">

</a>
```

The attributes are `class`, `href` and `id`. It is true:

```
link.attrs

{'id': 'main_logo', 'href': '/', 'class': ['no-hover']}
```

(The `class` attribute is a list, because an element can belong to more than one class. Incidentally, here we see another example of the use of id: it denotes main\_logo. Probably just for formatting reasons: in the page design rules, they specified how an element with id `main\_logo` should be formatted, instead of element `**i**` in the class `main\_logo`; they did this for hygiene reasons, since things that are unique are not "a class unto themselves".)

Now we know how to look up items with authors: they must have a `name` in the `attrs` dictionary. Let's find the first one so we can play with it and figure out how to extract the author name.

```
for link in soup.find_all("a"):
    if "name" in link.attrs:
        break
```

link

```
<a name="a32541">Baader, Bernhard</a>
```

Yes, you see, we have caught a man. Probably the first alphabetically among all of them, on `a`. :)

Elements have `string` and `strings` attributes.

link.string

```
'Baader, Bernhard'
```

link.strings

```
<generator object Tag._all_strings at 0x11412e540>
```

The second one looks less attractive. :)

Let's make a dictionary, with the authors as the keys, and their internal links on the page as the values.

```

authors = {}
for link in soup.find_all("a"):
    if "name" in link.attrs:
        authors[link.string] = link.attrs["name"]

authors

```

```

{'Baader, Bernhard': 'a32541',
 'Baadsgaard, Anna, 1865-1954': 'a52090',
 'Baarslag, C.': 'a26759',
 'Bååth, A. U. (Albert Ulrik), 1853-1912': 'a47041',
 'Babbage, Charles, 1791-1871': 'a556',
 'Babbitt, Ellen C.': 'a2498',
 'Babbitt, George Franklin, 1848-1926': 'a56365',
 'Babbitt, Harold E. (Harold Eaton), 1888-1970': 'a51605',
 'Babbitt, Irving, 1865-1933': 'a45766',
 'Babbitt, Katharine': 'a42830',
 'Babcock, Bernie, 1868-1962': 'a9252',
 'Babcock, Charles Almanzo, 1847-1922': 'a9811',
 'Babcock, Edwina Stanton': 'a44576',
 'Babcock, Elizabeth Jones, 1887-1963': 'a35898',
 'Babcock, Kendric Charles, 1864-1932': 'a42489',
 'Babcock, Retta': 'a7693',
 'Babcock, Sidney, 1797?-1884': 'a49911',
 'Babcock & Wilcox Company': 'a25476',
 'Babcock, William Henry, 1849-1922': 'a53184',
 'Babelon, Ernest, 1854-1924': 'a46000',
 'Bab, 'Ali Muhammad Shirazi, 1819-1850': 'a8279',
 'Babillotte, Arthur, 1887-1916': 'a42588',
 'Babinet, M. (Jacques), 1794-1872': 'a33233',
 'Babington, B. G. (Benjamin Guy), 1794-1866': 'a710',
 'Babits, Mihály, 1883-1941': 'a40766',
 ...
 'Bell, Henry, Captain': 'a3166',
 'Bell, Henry Glassford, 1803-1874': 'a38621',
 'Bellinger, Martha Idell Fletcher, 1870-1960': 'a7425',
 'Bellings, Richard, -1677': 'a55958',
 ...}

```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

Great: we've only got the authors and, on a quick glance, they're probably all there.

Since it would be nice to do something with this dictionary now, let's store it in Markdown. :) This will contain the indented entries themselves: each will be the name of an author and will act as a link to the authors page on B, specifically to that particular person within the page.

```

f = open("avtorji-na-b.md", "w")
url = "https://www.gutenberg.org/browse/authors/b"
for auth, aname in authors.items():
    f.write(f"- [{auth}]({url}#{aname})\n")
f.close()

```

Where to go with the file? Let's use it somewhere that accepts Markdown. We can, say, copy the first few lines here, into a notebook cell.

- [Baader, Bernhard](#)
- [Baadsgaard, Anna, 1865-1954](#)
- [Baarslag, C.](#)
- [Bååth, A. U. \(Albert Ulrik\), 1853-1912](#)
- [Babbage, Charles, 1791-1871](#)
- [Babbitt, Ellen C.](#)
- [Babbitt, George Franklin, 1848-1926](#)

And then we click the link to see if it really works.

### Making search a little easier

The code we used to search for `a` elements was unnecessarily complex. The `find\_all` accepts more arguments; among other things, we can require that an element has a certain attribute.

This gives all `a`s that have the specified `id`:

```
soup.find_all("a", id=True)
```

```
[<a class="no-hover" href="/" id="main_logo">
  
</a>]
```

There just happens to be one. This would give `a` whose `id` is the same as `main\_logo`.

```
soup.find_all("a", id="main_logo")
```

```
[<a class="no-hover" href="/" id="main_logo">
  
</a>]
```

We don't really care about `id`. We want to find those who have a certain `name`.

```
soup.find_all("a", name=True)
```

× raises-exception + Tag

```
-----
TypeError                                Traceback (most recent call last)
Cell In[21], line 1
----> 1 soup.find_all("a", name=True)
```

```
TypeError: Tag.find_all() got multiple values for argument 'name'
```

That is why I used the not very meaningful `id` as an example. The problem is that `name` is just the name of the first argument, the one we gave the value `a`. Python therefore gets angry that the value of the `name` argument should not be set twice, once to `a` and then to `True`.

Similarly with `class`.

```
Cell In[22], line 1
    soup.find_all("a", class="no-hover")
    ^
```

SyntaxError: invalid syntax

The word ``class`` is reserved and cannot be the name of a variable or argument. When we want to query ``name`` or ``class``, we have to specify them with a dictionary.

```
soup.find_all("a", {"name": True})
```

```
[<a name="a32541">Baader, Bernhard</a>,<br><a name="a52090">Baadsgaard, Anna, 1865-1954</a>,<br><a name="a26759">Baarslag, C.</a>,<br><a name="a47041">Bååth, A. U. (Albert Ulrik), 1853-1912</a>,<br><a name="a556">Babbage, Charles, 1791-1871</a>,<br><a name="a2498">Babbitt, Ellen C.</a>,<br><a name="a56365">Babbitt, George Franklin, 1848-1926</a>,<br><a name="a51605">Babbitt, Harold E. (Harold Eaton), 1888-1970</a>,<br><a name="a45766">Babbitt, Irving, 1865-1933</a>,<br><a name="a42830">Babbitt, Katharine</a>,<br><a name="a9252">Babcock, Bernie, 1868-1962</a>,<br><a name="a9811">Babcock, Charles Almanzo, 1847-1922</a>,<br><a name="a44576">Babcock, Edwina Stanton</a>,<br><a name="a35898">Babcock, Elizabeth Jones, 1887-1963</a>,<br><a name="a42489">Babcock, Kendrick Charles, 1864-1932</a>,<br><a name="a7693">Babcock, Retta</a>,<br><a name="a49911">Babcock, Sidney, 1797?-1884</a>,<br><a name="a25476">Babcock & Wilcox Company</a>,<br><a name="a53184">Babcock, William Henry, 1849-1922</a>,<br><a name="a46000">Babelon, Ernest, 1854-1924</a>,<br><a name="a8279">Bab, 'Ali Muhammad Shirazi, 1819-1850</a>,<br><a name="a42588">Babillotte, Arthur, 1887-1916</a>,<br><a name="a33233">Babinet, M. (Jacques), 1794-1872</a>,<br><a name="a710">Babington, B. G. (Benjamin Guy), 1794-1866</a>,<br><a name="a40766">Babits, Mihály, 1883-1941</a>,<br>...<br><a name="a3166">Bell, Henry, Captain</a>,<br><a name="a38621">Bell, Henry Glassford, 1803-1874</a>,<br><a name="a7425">Bellinger, Martha Idell Fletcher, 1870-1960</a>,<br><a name="a55958">Bellings, Richard, -1677</a>,<br>...]
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

We can get rid of the conditional clause when building a dictionary.

```

● authors = {}
  for link in soup.find_all("a", {"name": True}):
      authors[link.string] = link.attrs["name"]

```

Or we can just write

```
authors = {link.string: link.attrs["name"] for link in soup.find_all("a", {"name": True})}
```

Getting information from websites can be easy if we know how and if they are friendly...

## A little more reading

Now let's set ourselves a more difficult task. Let's get the author or, to make it easier, his internal code - for Babbage, let's say a556. The task is to find all his works published on Gutenberg.

If we see them on the website, they are obviously in the document, we just need to know how to get to them. Go back to the browser, open the items under `h2` at the bottom.

**Bååth, A. U. (Albert Ulrik), 1853-1912 ¶**

- [Nordmanna-Mystik: Bilder Från Nordens Forn tid](#) (Swedish) (as Author)

**Babbage, Charles, 1791-1871 ¶**

- [en.wikipedia](#)
- [The calculating engine](#) (English) (as Author)
- [On the Economy of Machinery and Manufactures](#) (English) (as Author)
- [Passages from the Life of a Philosopher](#) (English) (as Author)
- [Reflections on the Decline of Science in England, and on Some of Its Causes](#) (English) (as Author)

**Babbitt, Ellen C. ¶**

- [Jataka tales](#) (English) (as Author)
- [More Jataka Tales](#) (English) (as Author)

**Babbitt, George Franklin, 1848-1926 ¶**

**Developer Tool HTML Structure:**

```
<h2>Babbage, Charles, 1791-1871</h2>
<ul>
  <li>
    <a href="#">en.wikipedia</a>
  </li>
  <li>
    <a href="#">The calculating engine</a>
    " (English) (as Author)"
  </li>
  <li>
    <a href="#">On the Economy of Machinery and Manufactures</a>
    " (English) (as Author)"
  </li>
  <li>
    <a href="#">Passages from the Life of a Philosopher</a>
    " (English) (as Author)"
  </li>
  <li>
    <a href="#">Reflections on the Decline of Science in England, and on Some of Its Causes</a>
    " (English) (as Author)"
  </li>
</ul>
```

**Developer Tool CSS Styles:**

```
ul {
  list-style-type: none;
  padding-left: 10px;
}
li {
  display: list-item;
  text-align: -webkit-match-parent;
}
ul {
  list-style-type: none;
  margin: 0;
  padding-left: 10px;
}
ul {
  list-style-type: none;
}
```

Do we see structure? We start with `a`. Its father is `h2`. The brother of `h2` is `ul`. `ul`'s children contain singletons.

So first, let's grab Babbage. Since we know that there is only one `a` on the page whose `name` is equal to `a556`, instead of calling `find\_all` (which we would have to loop over), we just call `find`, which returns the first - and in this case only - element that matches the given arguments.

```
link = soup.find("a", {"name": "a556"})
```

```
link
```

```
<a name="a556">Babbage, Charles, 1791-1871</a>
```

His father is saved in `parent`.

```
link.parent
```

```
<h2><a name="a556">Babbage, Charles, 1791-1871</a> <a href="#a556" title="Link to this author">¶</a></h2>
```

His brother is saved in `next\_sibling`.

```
link.parent.next_sibling
```

```
'\n'
```

Eh. Since in HTML there's a blank line character in between (we've seen the same in XML, haven't we?), my brother is the blank line. I suppose we should take the next brother?

```
link.parent.next_sibling.next_sibling
```

```
<ul>
<li class="pgdbxlink"><a href="https://en.wikipedia.org/wiki/Charles_Babbage">en.wikipedia</a></li>
<li class="pgdbetext"><a href="/ebooks/71292">The calculating engine</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/4238">On the Economy of Machinery and Manufactures</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/57532">Passages from the Life of a Philosopher</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/1216">Reflections on the Decline of Science in England, and on Some of Its Causes</a> (English) (as Author)</li>
</ul>
```

Yes. But we can be cautious and move so many brothers until we reach `ul`.

```
ul = link.parent.next_sibling
while ul.name != "ul":
    ul = ul.next_sibling
```

So, by the way, we have learned about another attribute: `name`. Each element has a `name`, which contains the name of the tag. A `link`, for example, is `a`.

```
link.name
```

```
'a'
```

(If this were real, your experienced professor, who has seen a lot of bad things from website builders, would be a bit more careful and, for a start, check that `ul.name` is not already `h2`, which would mean that we are already at the next author and that the one we are investigating has no published works.)

Otherwise, finding the nearest sibling with a given tag is common, so the above loop can be replaced by a simple call to `find\_next\_sibling`:

```
ul = link.parent.find_next_sibling("ul")
ul

<ul>
<li class="pgdbxlink"><a href="https://en.wikipedia.org/wiki/Charles_Babbage">en.wikipedia</a></li>
<li class="pgdbetext"><a href="/ebooks/71292">The calculating engine</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/4238">On the Economy of Machinery and Manufactures</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/57532">Passages from the Life of a Philosopher</a> (English) (as Author)</li>
<li class="pgdbetext"><a href="/ebooks/1216">Reflections on the Decline of Science in England, and on Some of Its Causes</a> (English) (as Author)</li>
</ul>
```

To search backwards, through previous siblings, call `find\_prev\_sibling`.

So, by one way or another, we arrive at `ul` with Babbage's works.

The works are inside the `li` elements. `li` contains `a` and the strings inside it are works by that author.

```
for li in ul.find_all("li"):
    print(li.find("a").string)
```

```
en.wikipedia
The calculating engine
On the Economy of Machinery and Manufactures
Passages from the Life of a Philosopher
Reflections on the Decline of Science in England, and on Some of Its Causes
```

## HTML in reality

This was, of course, a school example. Gutenberg's pages are kindly simple. In reality, we can have a lot more fun reading them. But we've seen the idea.

## Regular expressions

The authors' names include the year of birth and death. But not all of them. How do we extract the year so that we have it stored and can do something useful with it? Or, say, leave it out when you print it out.

We could probably do something with `split`, but here we will show a different, more powerful approach. A year has a certain shape, a pattern, which is easy to describe: it contains a few digits, then a minus and a few more digits. The "something" can also be 0 if the year is unknown. To be precise, we would add that the digits are a maximum of four, but this restriction will probably not be necessary.

To make the case suitably challenging, the digits can be followed by a question mark if we are not sure of the year and the man (or woman) may have been married or died at an earlier or later date.

The pattern of this form is described by the regular expression `d\*\??-d\*\??`.



True. Thus, from `"Tannenbaum, Samuel A. (Samuel Aaron), 1874?-1948"` we extract when Tannenbaum was born and died.

```
import re

re.search(r"\d*\??-\d*\??", "Tannenbaum, Samuel A. (Samuel Aaron), 1874?-1948").group()

'1874?-1948'
```

If necessary, we can also get each number separately.

```
rojstvo, smrt = re.search(r"(\d*)\??-(\d*)\??", "Tannenbaum, Samuel A. (Samuel Aaron), 1874?-1948").groups()

rojstvo

'1874'

smrt

'1948'
```

Was the *\*teaser\** OK? Are we interested? So first we need to learn the syntax of regular expressions, and then we need to learn the Python functions to work with them.

## The syntax of regular expressions

What we'll learn here applies in general, not just in Python. Regular expressions are a pattern description language familiar to all general-purpose programming languages, and regular expressions can also be used to search through text in text editors.

Regular expressions consist of the characters we want to search for (in the example above, it was just ``-``) and characters with a special meaning. Here is a (non-exhaustive) list:

- ``.``: any character
- ``\w``: any letter or digit
- ``\d``: any digit
- ``\s``: white space (space, tab, newline)
- ``^``: start of string
- ``$``: end of string.

Characters also have negations: ``\W`` is anything that is not a letter or a digit, ``\D`` is anything that is not a digit, and ``\S`` is anything that is not a blank space.

Let's look at some simple examples

- ``l.pa`` can be ``lipa`` or ``lopa``, but it can also be ``lrpa``, or it can also be ``l1pa``, ``l)pa`` or ``l pa``. The dot can be anything.

- ``l\wpa`` is similar, but eliminates non-alphanumeric characters - among the above examples, it forbids ``l)pa`` or ``l pa``.

- ``^l\wpa`` will catch the same thing, but additionally requires that the substring matching the search pattern, ``l\wpa``, is at the beginning of the string being searched.

- The ``l.p.`` is ``lipa``, ``lipe``, ``lipi``, ``lipo``, as well as ``lopa``, ``lope``, ``lopi``, ``lopo``, and, of course, also ``lipr``, ``lrpr``, and even ``l(p)``, ``l)p^``, and ``l-p``.

- ``l...`` is anything beginning with ``l``.

- ``....`` is any sequence of four characters.

Obviously both ``.`` and ``w`` are too liberal: it would be good if we could list the characters we allow. For ``l.pa`` we could allow e, i, o and u. When we want to list the allowed characters, we enclose them in square brackets:

- ``l[eiou]pa`` could be ``lepa``, ``lipa``, ``lop`` or ``lupa``. And nothing else.

- But ``l[eiou]p[aeio]`` is all of the above in the first four syllables.

If you put a ``^`` immediately after ``[``, the regular expression would catch all **except** the characters in the parentheses. So letters and digits that are not vowels.

What about ``lip``? Can we tell that a character may or may not be present?

- The ``*`` allows the thing before the asterisk to be repeated any number of times.

- ``+`` does the same, but wants it to occur at least once.

- ``?`` says it can appear at most once.

A couple of examples:

- ``le+pa`` is ``beautiful``, ``beautiful``, ``leeepa``, ``leeeepa``, or even more beautiful.

- ``le*pa`` can be all of the above, plus ``lpa`` next to it.

- But ``beautiful`` can be ``beautiful`` or ``beautiful``.

All three, ``*``, ``+`` and ``?`` do not necessarily refer to the letter.

- ``l[eiou]+pa`` can also be ``leieoieoieoieoieiiiiioiepa``

- ``l.+pa`` can be ``l8umv9 jv &BT G#)GUJl pa``. The important thing is that it starts with ``l`` and ends with ``pa``, and it can be anything in between - but there must be at least one thing.

Round brackets can be used to group parts of a regular expression.

- An ``l(ep)*a`` is ``la``, ``lepa``, ``lepepa``, ``lepepepa`` ...

- ``l([eiou]p)*a`` can be, say, ``lepapapopopopupa``.

What about when we want the expression to contain a full stop? So really a full stop, not just anything. How do we tell that we mean a full stop? Or with a question mark? By bracketing it with parentheses? Before we put a backslash.

- An `l.p.` is a `blind.`, `lop.`, `lp.` ... The full stop at the end is really a full stop.

What if we want a backslash? In this case, we write two slashes. We've got that down - it's the same as when some stubborn Windows user wanted to use backslashes in path descriptions at all costs.

We can now return to the regular expression for years: “

- `\d` wants a digit

- `\d*` says that there can be more digits. There may be more than one.

- `\d*?` says it has to be followed by a question mark (a real, literal question mark, since we put a ``` in front of it).

- `\d*??` The question mark is a question mark: there may be a question mark, or there may not be one.

- `\d*??-`: as Freud said, sometimes a cigar is just a cigar; a minus here is just a minus.

- `\d*??-\d*??`: adds a pattern for the year of death.

## Regular expressions in Python

`findall`

Let's start with a string.

```
s = "Pred lipo je stala lepa lopa. Z lupo bi lahko iskal lepotne napake, a še tak lopov jih zlepa ne
```

"In front of the lime tree was a beautiful shed. With a magnifying glass I could have looked for beauty defects, but a rogue would never have found them."

All sheds could be searched with

```
import re

re.findall("lop.", s)
```

```
['lopa', 'lopo']
```

You could also search for `lepa` (`lepo`, `lepi` ...).

```
re.findall("lep.", s)
```

Python

```
['lepa', 'lepo', 'lepa']
```

`re.findall` preprosto vrne vse podnize, ki ustrezajo regularnemu izrazu. Ta del se nam je posrečil.

Če želimo dobiti le cele besede, bomo zapovedali, da spredaj in zadaj ne sme biti ne-črk.

raw

```
re.findall("\Wlop.\W", s)
```

Python

```
['lopa.']
```

```
re.findall("\Wlep.\W", s)
```

Python

```
['lepa ']
```

We got rid of the villain, but we got some spaces and dots. Dots? But isn't it meant that a full stop is not a full stop, but ... Yeah, no worries. That dot in the pattern was matched with an a. The `\W` is the dot in the string. Just like a space. Since we said that the pattern had to contain something that wasn't a letter, we just got that too.

Now we have a bunch of details to work out. First: **\*\*beware of backslashes\*\***. Backslashes have their own meaning. We know `\n` is the sign for a new line. Fortunately for us, `\W` doesn't have any special meaning, otherwise the above wouldn't work.

Every time we use backslashes in a string in Python, and we only want to get backslashes, we have to double them. So not `\\Wlep.\\W` but `\\Wlep.\\W`. In regular expressions this gets tricky because sometimes we have to write double slashes in an expression (see above) and when we double each one we get quadruple slashes. We will therefore prefer to write regular expressions with r-nices. If we put an `r` before the quotation mark, the backslashes will just be backslashes. (The `r` here does not stand for *\*regular expression\**, but for *\*raw string\**; such strings can be used in other places than regular expressions.) So the correct way to use it is

```
re.findall(r"\Wlep.\W", s)
```

```
['lepa ']
```

There is no difference here, but in general there will be.

But what are we going to do about the spaces? Before we tackle them, let's point out something else:

```
s2 = "Lepa lopa je stala pod lipo."
```

```
re.findall(r"\Wlep.\W", s2)
```

```
[]
```

He didn't catch the word "Beautiful" at the beginning, first of all because it is capitalized. We can add [flags to change how it works](https://docs.python.org/3/library/re.html#flags) to the `re.findall` function, and one of them is `re.IGNORECASE`.

```
re.findall(r"^\\Wlep\\.\\W", s2, re.IGNORECASE)
```

```
[]
```

She is still not Beautiful. Now, obviously, because there is nothing in front of the word that is not a letter - simply because there is nothing in front of it at all. The pattern will have to be reformulated to allow the word to be preceded by either a non-letter or the beginning of a string.

To avoid doctoring the patterns, we simply paste a space at the beginning and end of the string.

```
re.findall(r"\\Wlep\\.\\W", " " + s2 + " ", re.IGNORECASE)
```

```
[' Lepa ']
```

**Now let's get rid of the spaces in the found string: if the pattern contains a group, `findall` will not return the whole substring but only what is in the group.**

```
re.findall(r"\\W(lep\\.\\W)", s)
```

```
['lepa']
```

Finally, we half everything that makes a nice magnifying glass or loupe.

```
re.findall(r"\\W(l[eiou]p[aieo])\\W", s)
```

```
['lipo', 'lepa', 'lupo']
```

Now let's try this: we'd like all words that start with lep-, lip-, lop- or lup-.

```
re.findall(r"\\W(l[eiou]p\\W*)", s)
```

```
['lipo', 'lepa', 'lopa', 'lupo', 'lepotne', 'lopov']
```

The pattern is now that it must start with something that is not a letter (but that is not part of the group), followed by `l`, then `e`, `i`, `o` or `u`, then `p` and then any number (0 is possible) of letters.

What about words that contain `l[eiou]p`, but can have any other letters in front and behind?

```
re.findall(r"\\W*l[eiou]p\\W*", s)
```

```
['lipo', 'lepa', 'lopa', 'lupo', 'lepotne', 'lopov', 'zlepa']
```

## `Operators in distress`

A word of caution: what does `l.\*pa` return? Unexpectedly, this:

TELL for `\*`?

```
re.findall(r"l.*pa", s)
```

```
['lipo je stala lepa lopa. Z lupo bi lahko iskal lepotne napake, a še tak lopov jih zlepa']
```

Python is right. It is a string starting with `l` and ending with `pa`. I suppose we were expecting a shorter one, weren't we?

The `\*` and `+` operators are greedy: they take as much as possible - but of course in such a way that it's still possible to add `pa` (or whatever) to the end. We often imagine that they'll take the \*shortest\* possible string. This is achieved by adding a question mark after `\*` or `+`. So `\*?` and `+?` are the unmissable versions of asterisk and plus.

```
re.findall(r"l.*?pa", s)
```

```
['lipo je stala lepa',  
'lopa',  
'lupo bi lahko iskal lepotne napa',  
'lopov jih zlepa']
```

This is, of course, again not in line with expectations, but we see the difference: he starts with `l` and then adds as much as necessary to get to the next `pa`. The result is undesirable, but at least it's fun. :)

The above would of course be avoided if `.\*?` were replaced by `\w\*`:

```
re.findall(r"l\w*pa", s)
```

```
['lepa', 'lopa', 'lepa']
```

By the way, let's show how to extract all the words from the text (the St Nicholas Letters homework required something like this, but in a more complicated way).

```
print(re.findall(r"\w+", s))
```

Python

```
['Pred', 'lipo', 'je', 'stala', 'lepa', 'lopa', 'Z', 'lupo', 'bi', 'lahko', 'iskal', 'lepotne', 'napake',
```

Regular expressions have many more options and tricks, complications and solutions. We cannot list them all in this short introduction; it would require several lectures in theory, and a lot of experience in practice.

## `search` in `match` (in `finditer`)

So far we have only been concerned with patterns and finding subsets that match them. There is much more you can do with regular expressions in Python. Patterns will often consist of several parts, groups, and we will usually be interested in their contents.

Recall the expression we used to look for birth and death years, ``d*\??-`d*\??``. This obviously has two parts, the year of birth and the year of death. They can also be formally referred to as groups.

```
author = "Tannenbaum, Samuel A. (Samuel Aaron), 1874?-1948"

re.findall(r"(\d*)(\??)-(\d*)(\??)", author)

[('1874', '?', '1948', '')]
```

If the pattern contains multiple groups, ``findall`` returns a list of targets rather than a list of strings. This would be enough to be able to tell the year of birth and death, and - because we have cunningly enclosed a question mark in its own group instead of appending it to the digits - we have a separate indicator that tells whether the year is reliable or not.

What if we wanted to change the output - say, remove the years from the author's name? Unfortunately, ``findall`` only tells us *what* it found, not *where*. On the other hand, ``findall`` returns a list, even though we know here that this list will contain at most one element.

For such cases, the ``search`` and ``match`` methods are more useful.

```
re.search(r"(\d*)(\??)-(\d*)(\??)", author)

<re.Match object; span=(38, 48), match='1874?-1948'>
```

```
re.match(r"(\d*)(\??)-(\d*)(\??)", author)
```

This one, obviously, is more ``search``. :) The difference is that ``match`` requires the pattern to appear at the beginning of the string, while ``search`` does not.

The result returned by ``search`` is ``re.Match``: an object containing more information about what the pattern captured.

```
mo = re.search(r"(\d*)(\??)-(\d*)(\??)", author)
```

First: ``re.search`` may return ``None`` if it finds nothing. The function call will therefore typically be followed by ``if``, which will take care of this scenario.

The ``group`` method returns the contents of each group.

```
mo.group(1)
```

```
'1874'
```

```
mo.group(3)
```

```
'1948'
```

Or several groups.

```
mo.group(1, 3)
```

```
('1874', '1948')
```

We start counting groups with `1` because `0` represents the whole captured subset.

```
mo.group(0)
```

```
'1874?-1948'
```

Since this is exactly what we usually need, `0` is also the default value. The complete string is therefore obtained by

```
mo.group()
```

```
'1874?-1948'
```

All subgroups - what would be returned by `findall` - are given by `groups`.

```
mo.groups()
```

```
('1874', '?', '1948', '')
```

As there can be many groups and it is difficult to count them, we can also name them. This complicates the expression, but simplifies working with it. Naming groups is done by replacing `(…)` with `(?P<name>…)`.

```
mo = re.search(r"(?P<born>d*)(?P<bcertain>??)-(?P<died>d*)(?P<dcertain>??)", author)
```

Now we can address groups with names, not just indexes.



```
mo.group("born")
```

```
'1874'
```

```
mo.group("died")
```

```
'1948'
```

Or we can just get a dictionary of groups.

```
mo.groupdict()
```

```
{'born': '1874', 'bcertain': '?', 'died': '1948', 'dcertain': ''}
```

For a group, we can find out where in the original string it starts or ends. Or both.

```
mo.start()
```

```
38
```

```
mo.end()
```

```
48
```

```
mo.span()
```

```
(38, 48)
```

You can add an index or a group name.

```
mo.span("born")
```

```
(38, 42)
```

Now let's take our Tannenbaum, store his birth and death years in a separate variable and remove them from the string, leaving only the author's name.

```
author = "Tannenbaum, Samuel A. (Samuel Aaron), 1874?-1948"

mo = re.search(r" (?P<born>\d*)(?P<bcertain>??)-(?P<died>\d*)(?P<dcertain>??)", author)

if mo:
    born, died = mo.group("born", "died")
    author = author[:mo.start()]
else:
    born = died = ""

author
```

```
'Tannenbaum, Samuel A. (Samuel Aaron)'
```

```
born
```

```
'1874'
```

```
died
```

```
'1948'
```

## `sub` and `split`

Arrays have a `replace` method to replace a given substring with another. Regular expressions have a `sub` method that does the same thing, except that the string to be replaced is given by a regular expression.

```
s
```

```
'Pred lipo je stala lepa lopa. Z lupo bi lahko iskal lepotne napake, a še tak lopov jih zlepa ne bi našel.'
```

```
re.sub("l.p.", "hišo", s)
```

```
'Pred hišo je stala hišo hišo. Z hišo bi lahko iskal hišotne napake, a še tak hišov jih zhišo ne bi našel.'
```

To je seveda narobe. Lipo zamenjajmo s hišo, lepa pa bi bilo treba očitno zamenjati s heša. Storimo tako.

```
re.sub("l(.)p(.)", r"h\g<1>š\g<2>", s)
```

```
'Pred hišo je stala heša hoša. Z hušo bi lahko iskal hešotne napake, a še tak hošov jih zheša ne bi našel.'
```

This is of course wrong. Let us replace lime with house, and obviously lepa should be replaced by hasha. Let us do so.

```
def increase1(mo):  
    return str(int(mo.group()) + 1)
```

```
re.sub(r"\d+", increase1, "Vzamemo 5 jajc in 6 litrov vode, pomešamo in dodamo 200 g sladkorja.")
```

```
'Vzamemo 6 jajc in 7 litrov vode, pomešamo in dodamo 201 g sladkorja.'
```

And finally, a `split`. A string has a `split` method that receives a delimiter by which to split the string. The separator is always just the string. Regular expressions have a `split` method that receives a regular expression. Let's split the word according to the vowels.

```
re.split("[aeiou]", "samoglasniki")
```

```
['s', 'm', 'gl', 'sn', 'k', '']
```

```
re.split("(?:[aeiou])", "samoglasniki")
```

```
['s', 'm', 'gl', 'sn', 'k', '']
```

(Interlude: if we want to get syllables - in the sense of combinations of letters ending in a vowel, then it's not `split` but `findall`:

```
re.findall("[^aeiou]*[aeiou]", "samoglasniki")
```

```
1]
```

```
['sa', 'mo', 'gla', 'sni', 'ki']
```

End of interlude.)

## Capturing data from texts

In this lecture, we touched briefly on a topic that requires a lot of knowledge and experience, and even more patience: extracting data from documents that were not intended to be extracted (automatically) in the first place. HTML is basically meant to be read, and free text is not meant to be read at all. Since HTML pages are usually formatted, and since they are often assembled automatically (especially if the data comes from a database), we can hope that the formatting will be sufficiently transparent and consistent to allow the tags to be used to identify the parts that contain the data we are looking for. Within them, we will often be able to use regular expressions.

The page we have seen here is simple. In practice, it will not always be so. And, worse, in practice, pages change and if we struggle to develop a program to capture data from a page, there is no guarantee that next week everything will be different and we will have to start all over again. When it is possible to get the data in a readable format by some other means, this is always a better option. Otherwise - good luck. :)