Peckham DAZ

Week 2 – 21/01/2025

Coding Basics

Conditional statements

Loops

Functions

Conditional statements

Conditionals – If

- Allow us to make decisions about what code is run using variables and/or comparisons

```
if a > b:
  print("a is greater than b")
```

Conditionals – If

```
if condition:
   #code


#code
```

- If the condition is True, the code within the statement will run
- The condition must be a True or False statement, eg:
  - a is greater than b
  - It is before 6pm
  - The user clicked the up-arrow key

Note: Indentation

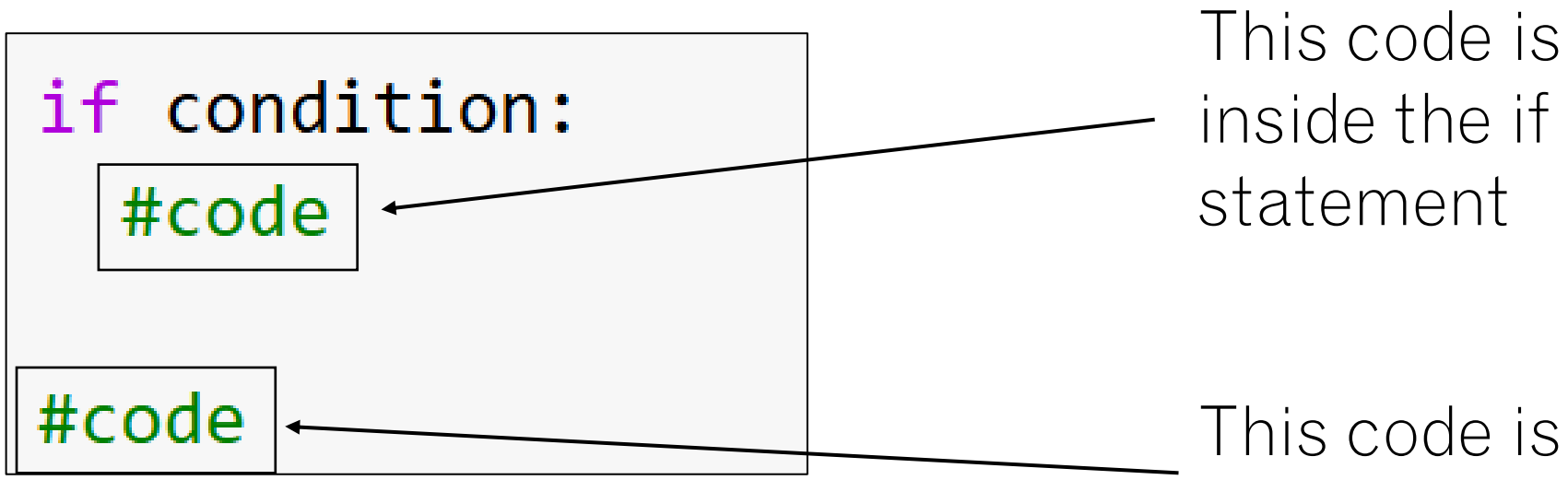
- In python, indentation is VERY important
- Other languages, such as Javascript, use brackets to separate statements, eg:

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

A Javascript if statement

- Python just uses indentation, so you have to pay close attention

Note: Indentation



Note: Indentation

It can get complicated!

```
num = int(input("Enter a number: "))

if num >= 0:
    if num < 10:
        print("The number is a single-digit positive number.")
    else:
        if num < 100:
            print("The number is a two-digit positive number.")
        else:
            print("The number is a positive number with more than two digits
else:
    print("The number is a negative number.")
```

Conditionals – If… else

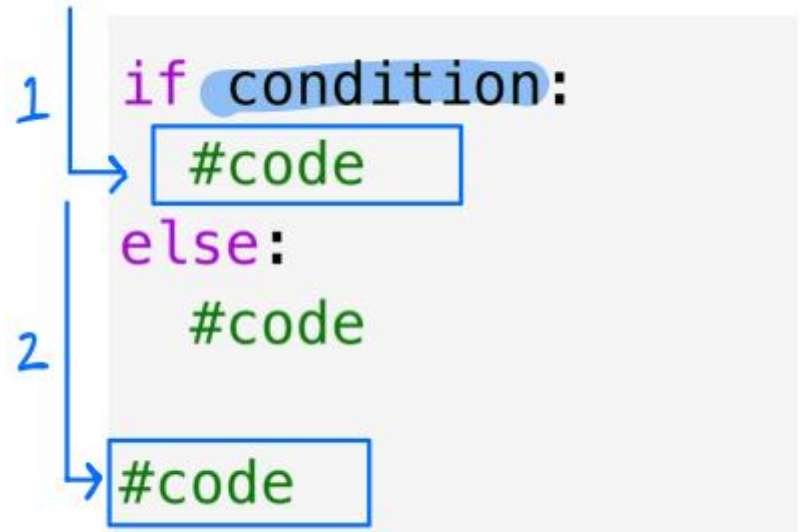
```
if condition:
  #code
else:
  #code

#code
```

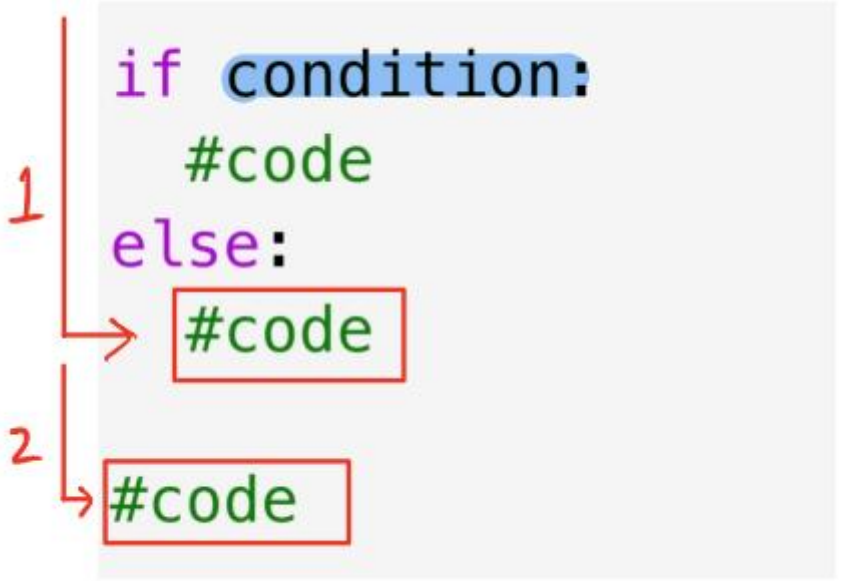
- If the condition is False, the code inside the else will be run
- You don't have to have an else for every if
  - E.g. the user clicking a button on a webpage

Conditionals – If... else

Condition is True



Condition is False



Conditionals – If… else

```
if a > b:
  print("a is greater than b")
else:
  print("b is greater than a")
```

However, what if a is equal to b?

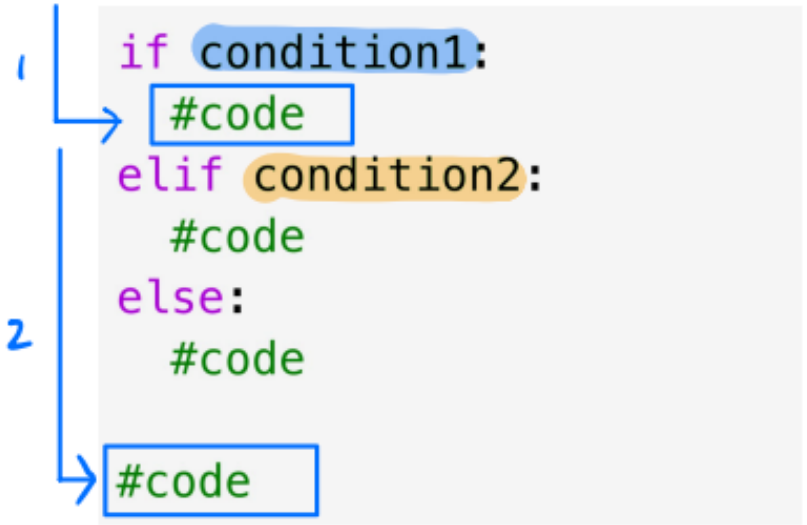
Conditionals – If… elif… else

```
if condition1:
  #code
elif condition2:
  #code
else:
  #code
```

- Comes from the combination of the words ‘else if’
- If condition 1 is False, it will then move on to condition 2
- Only one block of code within the whole statement can run

Conditionals – If... elif... else

Condition 1 is True



Condition 1 is False
AND
Condition 2 is True

Conditionals – If… elif… else

```
if false_condition:
  #code1
elif true_condition:
  #code2
elif false_condition:
  #code3
elif false_condition:
  #code4
elif false_condition:
  #code5
elif false_condition:
  #code6
else:
  #code7
```

Which one of these code blocks will run?

Code 2

Conditionals – If… elif… else

```
if false_condition:
  #code1
elif false_condition:
  #code2
elif false_condition:
  #code3
elif false_condition:
  #code4
elif true_condition:
  #code5
elif true_condition:
  #code6
else:
  #code7
```

Which one of these code blocks will run?

Code 5

Conditionals – If… elif… else

```
if false_condition:
  #code1
elif false_condition:
  #code2
elif false_condition:
  #code3
elif false_condition:
  #code4
```

Which one of these code blocks will run?

NONE!

Conditionals – If... elif... else

```
if a > b:
  print("a is greater than b")
elif a == b:
  print("a is equal to b")
else:
  print("b is greater than a")
```

== is used for comparison

e.g. `a == b`

= is used for assignment

e.g. `a = 5`

Logical operators

and:

```
if a > b and a > c:
  print("a is greater than b and c")
```

or:

```
if a > b or a > c:
  print("a is greater than at least one of b or c")
```

not:

```
if not a > b:
  print("a is NOT greater than b")
```

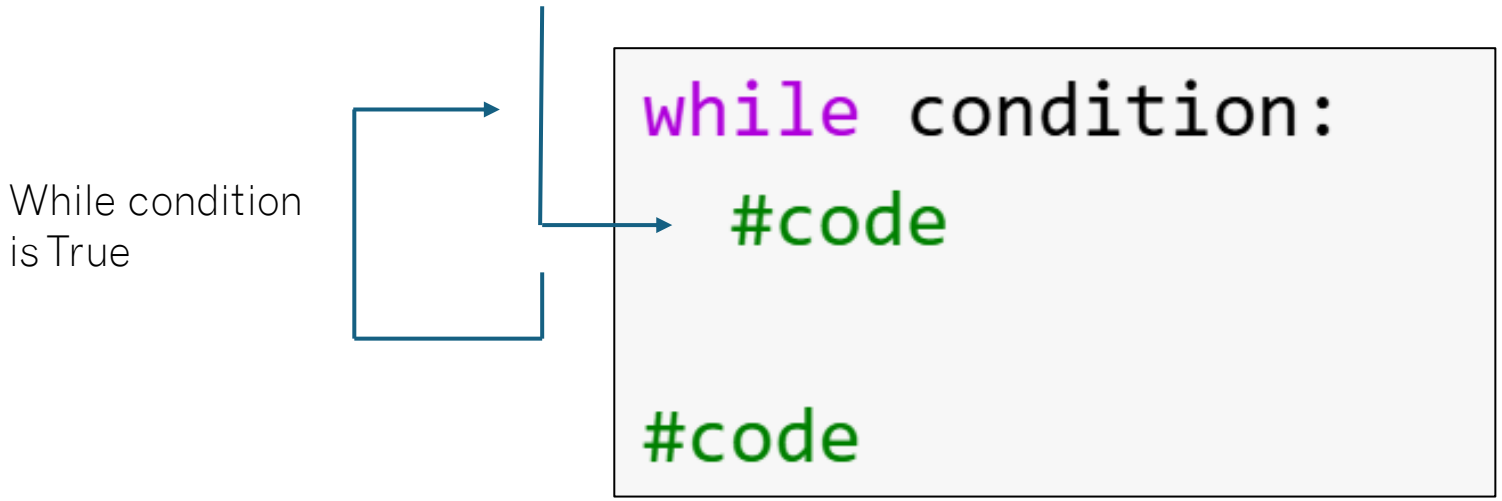
Loops

While loops

For loops

While loops

- The code within the loop will run whilst the condition is true



While loops

```
a = 0

while a < 6:
    a += 1
    print(a)
```

```
1
2
3
4
5
6
```

Equivalent to writing `a = a + 1`

While loops

- Beware: it is easy to get stuck in while loops!

```
a = 0

while a < 6:
    a -= 1
```

- This loop will run forever, or until something crashes…

While loops

- You can use the break statement to exit a while loop even if the condition is still true

```
a = 0

while a < 6:
  a += 1
  if a == 3:
    break
```

For loops

- Used for iterating over a sequence of objects (list, dictionary, etc)

OR

- Running a block of code a specified number of times

For loops – iterating over a sequence

```
for variable in sequence:
  #code
```

- Running the block of code once for each item in the list

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday', 'Saturday', 'Sunday']

for x in days:
  print(x)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

For loops – iterating over a sequence

```
for variable in sequence:
  #code
```

- Running the block of code once for each item in the list

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday', 'Saturday', 'Sunday']

for day in days:
  print(day)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

For loops – specified range

- To loop through a block of code a specific number of times, we can use the range() function

Convention to use `i` as the variable name as it will be an integer

```
for i in range(range_value):
  #code
```

```
for i in range(6):
  print(i)
```

```
0
1
2
3
4
5
```

For loops – specified range

- As a default, the loop starts at 0 but this can be changed

```
for i in range(3,6):
  print(i)
```

```
3
4
5
```

- The loop will START WITH the first number but END BEFORE the last

```
for i in range(start_with, end_before):
  #code
```

For loops - specified range

- You can also specify the step size (default is 1 and it can be negative):

```
for i in range(start_with, end_before, step_size):
  #code
```

```
for i in range(2,10,2):
  print(i)
```

```
2
4
6
8
```

Start at 2

End before 10

Increase by 2 each time

For loops – specified range RECAP

- 1 argument:
  - Starts at 0
  - Step size is 1

```
for i in range(end_before):
  #code
```

- 2 arguments:
  - Step size is 1

```
for i in range(start_with, end_before):
  #code
```

- 3 arguments:

```
for i in range(start_with, end_before, step_size):
  #code
```

For loops - RECAP

1. Used for iterating over a sequence of objects (list, dictionary, etc)

```
for variable in sequence:
  #code
```

- Variable will be an object (e.g. the items in a list)

OR

2. Running a block of code a specified number of times

```
for i in range(start_with, end_before):
  #code
```

- Variable will be an integer
- Uses the range() function

Functions

Functions

- A block of code that will only run when it is **called**
- Makes code more efficient as it reduces repeats
  - If you have to write a block of code more than once, it should be a function!

- Information can be passed into a function
- The function can then return information at the end

Functions – definition and calling

- A function is **defined** using the def keyword:

```
def function_name():
  #code
```

```
def my_function():
  print("this is my function")
```

- It can then be **called** using the function name followed by brackets:

```
my_function()
```

- If a function is not called, it will not run! Just defining a function will not run it, it has to be called.

Functions – parameters and arguments

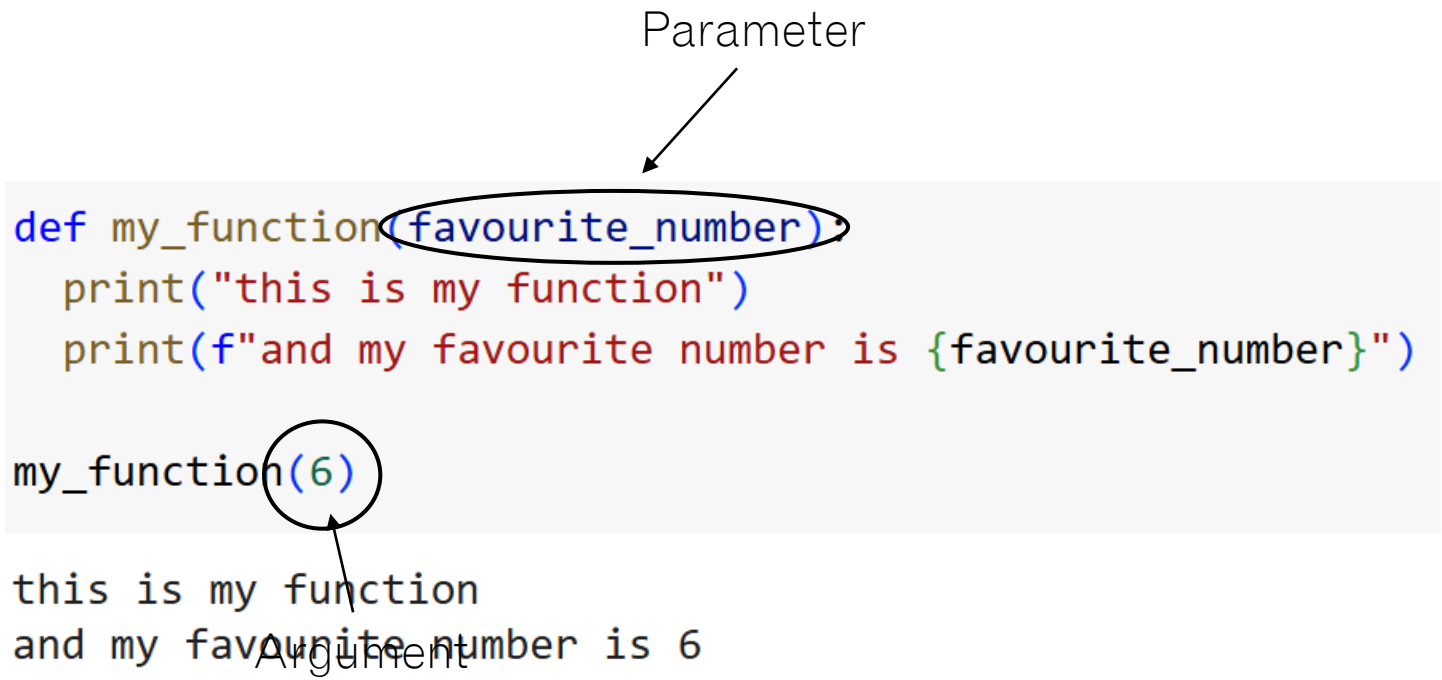
- These are used to pass information into the function
- A *parameter* is defined within the brackets after the function name

```
def function_name(parameter):
  #code
```

- When the function is called, you specify the value of this parameter, this is called the *argument*

```
function_name(argument)
```

Functions – parameters and arguments



Functions – number of arguments

- A function can have multiple parameters, but it MUST be called with the correct number of arguments

```
def my_function(first_favourite_number, second_favourite_number):
  print("this is my function")
  print(f"and my first favourite number is {first_favourite_number}")
  print(f"and my second favourite number is {second_favourite_number}")

my_function(6)
```

Functions – number of arguments

- A function MUST be called with the correct number of arguments

```
def my_function(first_favourite_number, second_favourite_number):
  print("this is my function")
  print(f"and my first favourite number is {first_favourite_number}")
  print(f"and my second favourite number is {second_favourite_number}")

my_function(6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-028c16b1ebd8> in <cell line: 0>()
      4   print(f"and my second favourite number is {second_favourite_number}")
      5
----> 6 my_function(6)

TypeError: my_function() missing 1 required positional argument: 'second_favourite_number'
```

Functions – return values

- The function can return information back to you after running using the return keyword

```
def function_name(parameter1, parameter2):
  #code
  return return_value
```

```
def my_function(first_favourite_number, second_favourite_number):
  return first_favourite_number + second_favourite_number

print(my_function(6,7))

favourite_combo = my_function(6,7)
```

Functions – return values

- Note: If the return keyword isn't used, the function won't return anything

```
def my_function(first_favourite_number, second_favourite_number):
  print("this is my function")
  print(f"and my first favourite number is {first_favourite_number}")
  print(f"and my second favourite number is {second_favourite_number}")


empty_return = my_function(6,7)
print(empty_return)
```

```
this is my function
and my first favourite number is 6
and my second favourite number is 7
None
```

Web Scraping

(some of the following slides are credited to James Gibbons-Macgregor, Terence Broad, and Rebecca Fiebrink @ UAL CCI)

Jupyter Notebooks and Libraries

Jupyter notebooks

- Jupyter notebooks is a web-based interactive computational environment for creating notebook documents
- File contains input/output cells which can contain code and be executed individually
- Allows you to edit and rerun operations on datasets without having to download, clean, etc every time
- File extension .ipynb

Jupyter notebooks

- Jupyter notebooks is a web-based interactive computational environment for creating notebook documents
- File contains input/output cells which can contain code and be executed individually
- Allows you to edit and rerun operations on datasets without having to download, clean, etc every time
- File extension .ipynb

Libraries

- A collection of precompiled code that can be used later in a program for some specific well-defined operations.
- It makes programming simpler and more convenient as you don't need to write the same code again and again for different programs
- You must have installed the library then imported it into your sketch

```
import numpy as np

print(np.add(3,4))
```

```
7
```

Useful Python Libraries for Data Science

Numpy

- Stands for ‘numerical python
- Used for mathematical and numerical calculations from trigonometry and rounding to large matrices and multi-dimensional data

Pandas

- Data analysis library
- Used for analysis, manipulation, data cleaning, etc

Beautiful Soup

- Web scraping library
- Pulls data out of HTML files
- Used for easily scraping data from webpages

Web Scraping and Crawling

Web scraping

- Web scraping is the process of collecting unstructured and structured data in an automated manner. Then working with that data to find patterns, correlations, trends etc.
- Some of the main use cases of web scraping include price monitoring, price intelligence, news monitoring and market research among many others.
- In general, it is used by people and businesses who want to make use of publicly available web data to generate valuable insights and make smarter decisions.

Web crawling

- Web crawling is the practice of building software that automatically searches for webpages by navigating through hyperlinks — to scrape data from a whole website (or lots of websites)
- A web crawler can collect vast amounts of data by exhaustively clicking on every hyperlink on a website
- You can set rules for what links to click through and what subdomains you want to retrieve data from

Scraping vs crawling

- Web scraping
  - Identifies and locates specific target data from web pages, we know the exact data set identifier, e.g., an heading, image or table

- Web crawling
  - Used to index the information on the page using bots, also known as crawlers.
  - The crawler goes through every page and every link, scraping ALL the data from each one
  - Similar to what search engines do, it's all about viewing a page as a whole and indexing it.

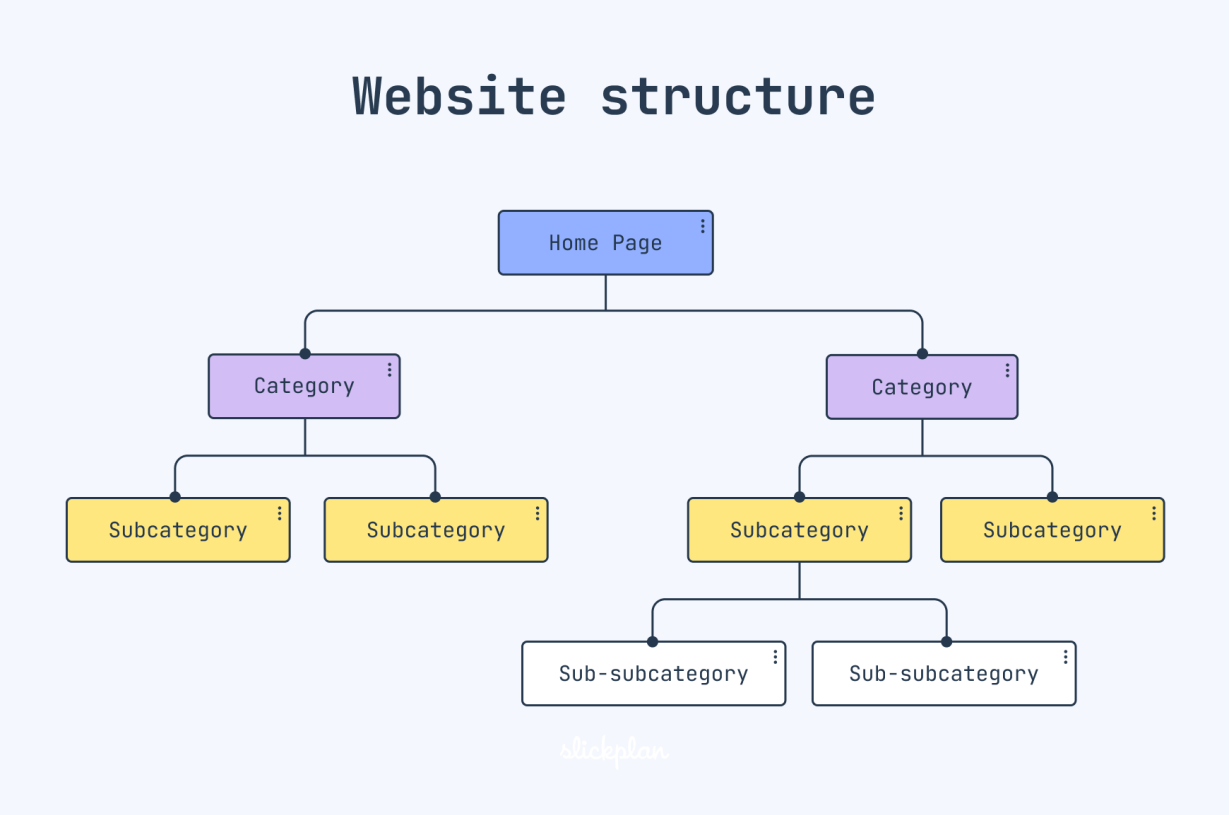
Technical and ethical issues

- Many websites try to stop web crawling by:
  - Not putting content in HTML, or making the format difficult to parse
  - Blocking IP address that make lots of requests very quickly
  - Adding captchas or other interactive elements to prevent software scripts accessing the page
  - Putting data behind a paywall

- Ethical Issues:
  - It is often against a website or platforms terms of service to scrape all of their data
  - The dataset may be copyrighted by the website or other party.
  - The data may contain private or personally identifying information. (especially on social media)
  - The data may contain offensive or malicious content
  - The data is full of factually inaccurate content or material

How are websites structured?

- A website is structured through a hierarchical organisation, with a homepage followed by various interconnected web-pages
- A website has a domain name (e.g wikipedia.org) and pages all have separate URLs that stem from the domain name (e.g. en.wikipedia.org/wiki/Camberwell)
- We can connect web pages together with hyperlinks

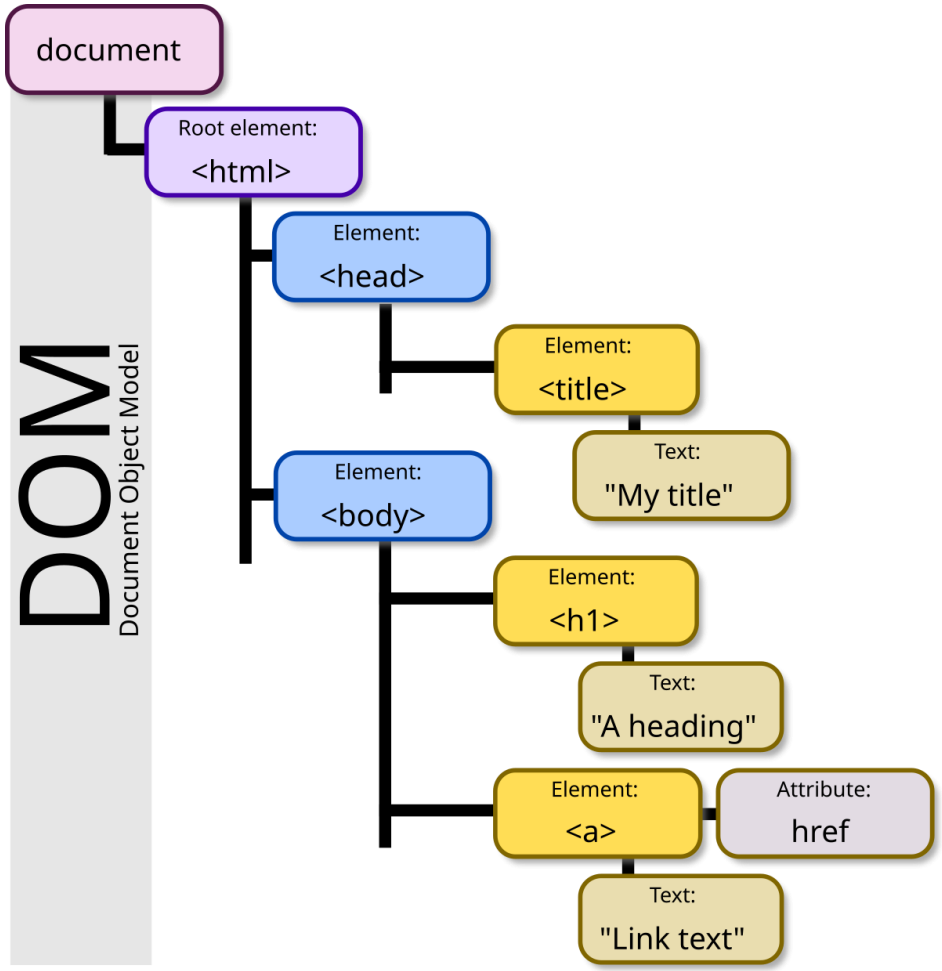


How are websites structured?

- Web pages are made primarily using three different scripting or markup languages:
    - HTML: Hypertext Markup Language — Where the content of the web page is stored
    - CSS: Cascading Stylesheets — Where the instructions on the layout and visual characteristics are stored
    - JS: Javascript — Code that allows for interactive elements (and other things) on a web page

The document object model (DOM)

- The DOM is how a web page gets represented by the computer and is built from the content in the HTML (or XML) file
- It gives a hierarchy and organisation to how the elements in the web page are structured
- Using the DOM specific elements can be manipulated using software scripts
- It is always a ‘tree’ structure (a network that only branches)



HTML overview

- When we are performing web scraping, we are mostly only interested in parsing HTML – as this is where text data is usually stored

- The HTML defines things like:
  - Document titles
  - Headings
  - Paragraphs
  - Images
  - Hyperlinks

HTML overview

- In HTML, elements are enclosed in tags
  - Opening tag: <tag>
  - Closing tag: </tag>

- We can then put the content between the tags

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Welcome to HTML</title>
    </head>
  <body>
    <h1>SIMPLE HTML CODE</h1>
    <p>HTML stands for Hyper Text Markup
Language.</p>
  </body>
  </html>
```

Parsing

- Parsing is taking a set of data and extracting the meaningful information from it
- With HTML parsing, you're looking to read some HTML and return with a structured set text/images/etc

Parsing HTML with Beautiful Soup

- Luckily, we don't have to parse HTML ourselves, the Python library **beautifulsoup** can do it for us!
- We just need to run:

```
import requests
from bs4 import BeautifulSoup

url = 'https://en.wikipedia.org/wiki/Giraffe'
response = requests.get(url)
bs_html = BeautifulSoup(response.text, features="html.parser")
```

And we get a Python object that we can search and extract information from.

How to find the content you want to scrape

- We can use tags to search for specific data you want to collect
  - <p> text
  - <h> headings
  - <a> links
  - <img> images

- We can use the beautifulsoup function find_all() to do this

```
bs_html.find_all('a')
```

Cleaning and sorting data after scraping

- Data collected from web scraping will generally be messy and disorganised
- You will normally need to:
  - Extract and reformat data from HTML into a structured format (.txt, .csv, etc)
  - Remove data from unwanted pages
  - Remove junk/bad data, which could be:
    - Empty cells
    - Data in the wrong format
    - Incorrect data
    - Duplicates
- This can all be done with pandas

Reading and saving a .csv file with Pandas

- A simple way to store big data sets is to use .csv (comma-separated values) files
  - Pandas .read_csv() function reads the .csv file and puts the data in a pandas DataFrame object which we can then
  - After cleaning/sorting etc, this can then be saved as a new .csv file using the .to_csv() function

```
import pandas as pd

data = pd.read_csv('input_data_file.csv')

#code to clean/sort data

data.to_csv('output_data_file.csv')
```