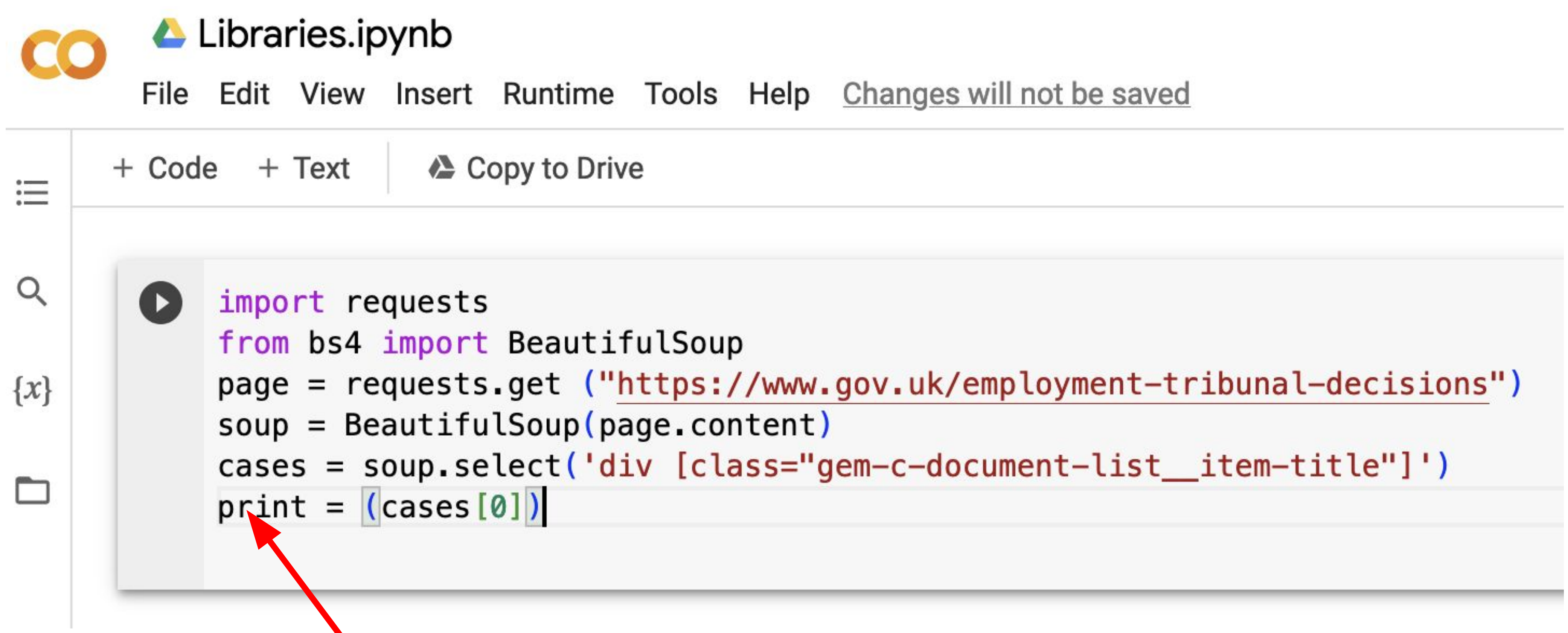

Libraries.ipynb
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive

```python
import requests
from bs4 import BeautifulSoup
page = requests.get ("https://www.gov.uk/employment-tribunal-decisions")
soup = BeautifulSoup(page.content)
cases = soup.select('div [class="gem-c-document-list__item-title"]')
print = (cases[0])
```

[George's code] Instead of using the print() command, this line assigned a value to print, making it a variable. That's why it didn't print, but also why trying to use the print() command afterwards produced odd results.
To fix, change the code to print(cases[0])
But also disconnect to wipe the print variable, and run the code again

```python
import requests
from bs4 import BeautifulSoup

def get_headlines(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    headlines = soup.find_all('h3', class_='title-container')
    return [headline.text for headline in headlines]

def main():
    url = 'https://www.eltiempo.com/'
    headlines = get_headlines(url)
    for headline in headlines:
        if 'desempleados' in headline:
            print(headline)

if __name__ == '__main__':
    main()
```

[Paige's code] This code defines two functions. The second uses the first. Last 2 lines run the second function.

What do the lines do:
if __name__ == '__main__':
  main()

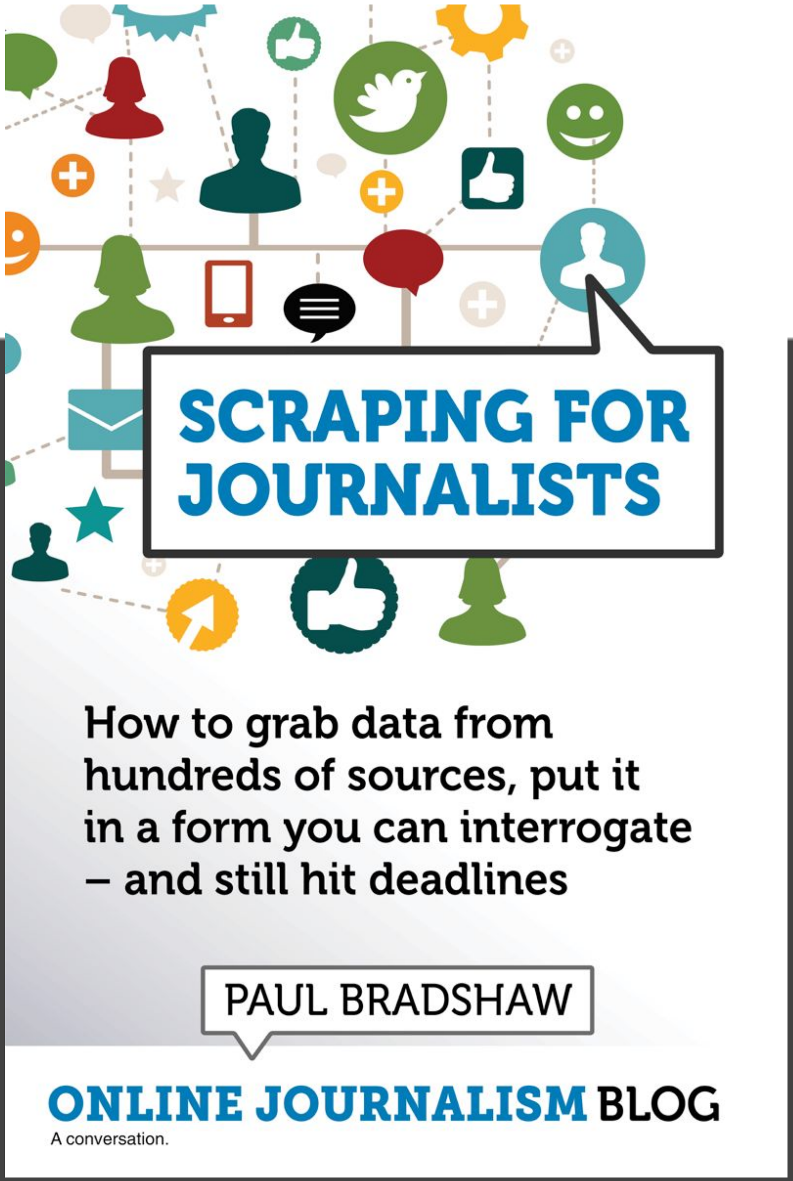
The lines `if __name__ == '__main__':` `main()` are used to execute the code in the `main()` function only when the file is being run as a script, not when it is being imported as a module.

The `__name__` variable is a special variable in Python that contains the name of the module or script that is currently being executed. When a file is being run as a script, the `__name__` variable is set to the name of the file. However, when a file is being imported as a module, the `__name__` variable is set to the name of the module.

The `if __name__ == '__main__':` statement checks if the `__name__` variable is equal to the string `__main__`. If it is, then the code within the `if` statement is executed. Otherwise, the code within the `if` statement is skipped.

In the code you provided, the `main()` function is defined to print the headlines that contain the word `desempleados`. The `if __name__ == '__main__':` statement ensures that the `main()` function is only executed when the file is being run as a script. This is useful because it allows you to define functions and variables in a file that can be used by other files, without having to worry about those functions and variables being executed when the file is imported.

Creating functions for scrapers


SCRAPING FOR JOURNALISTS

How to grab data from hundreds of sources, put it in a form you can interrogate – and still hit deadlines

PAUL BRADSHAW

ONLINE JOURNALISM BLOG

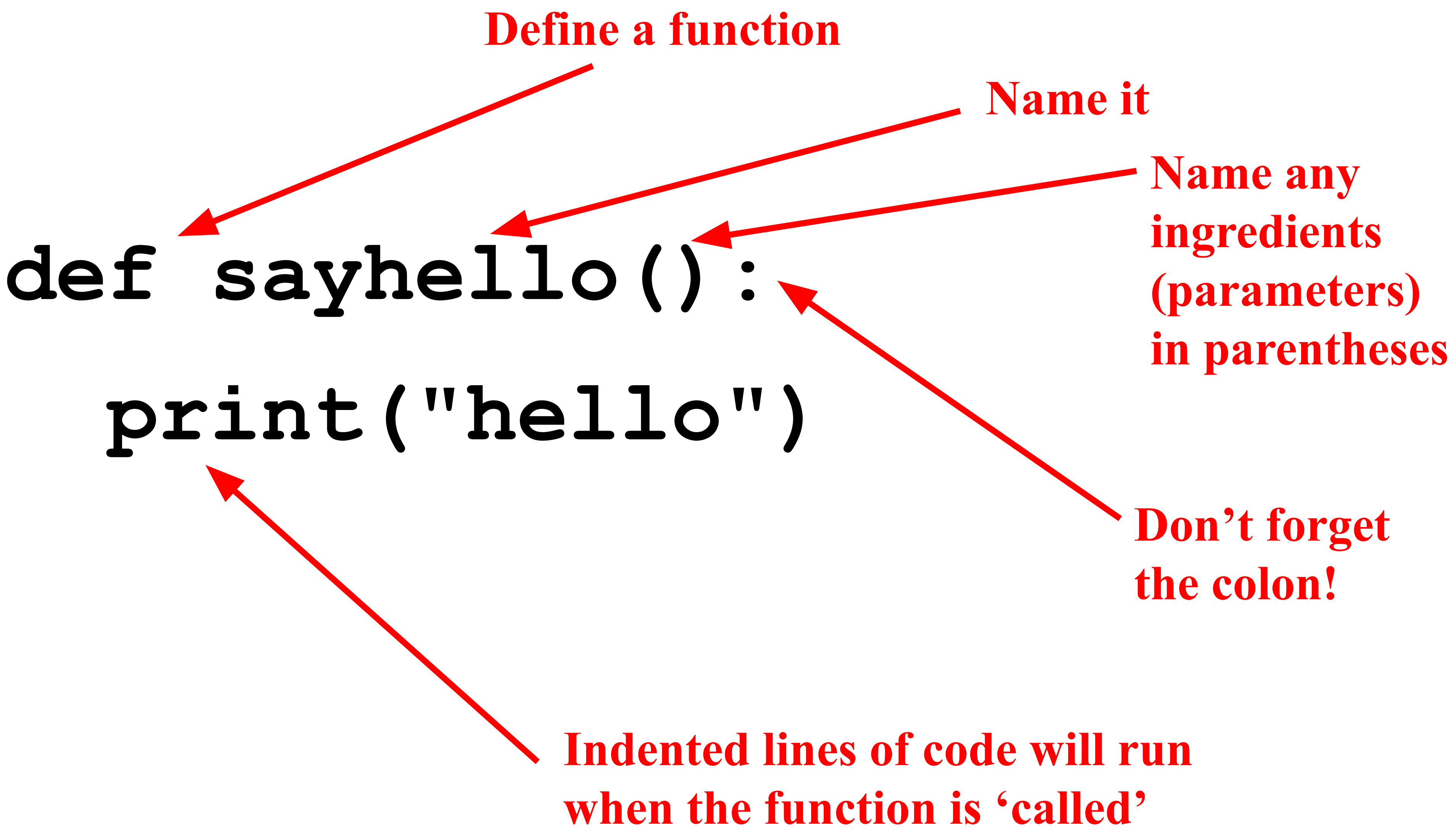
A conversation.

Paul Bradshaw

Leanpub.com/scrapingforjournalists

What we’ll cover

- How to create your own functions
- Scraping multiple pages



```
def sayhello():
    print("hello")
```

Define a function
Name it
Name any ingredients (parameters) in parentheses
Don’t forget the colon!
Indented lines of code will run when the function is ‘called’

Those ingredients

- Start with **`def`**
- Then name the function (arbitrary)
- Then brackets
- Inside those: name the ingredients
- Then colon
- Then indented lines which represent the ‘recipe’ you are storing in the function (this will likely use the ingredients you named)

```
def print_this_word(thisword):
  print(thisword)
```

‘Calling’ the function

...is like using any other function:

- Type the name of the function
- Then brackets
- Inside those: specify the ingredient(s) (‘arguments’)
- Run it!

```
print_this_word("pumpkin")
```

When this function is called
it needs one ingredient.
We ‘pass’ that inside the parentheses

If a function has multiple ingredients they are separated by commas

```
def addtwonumbers(numone, numtwo):
   #add the two ingredients
   total = numone+numtwo
   #return that value
   return(total)
```

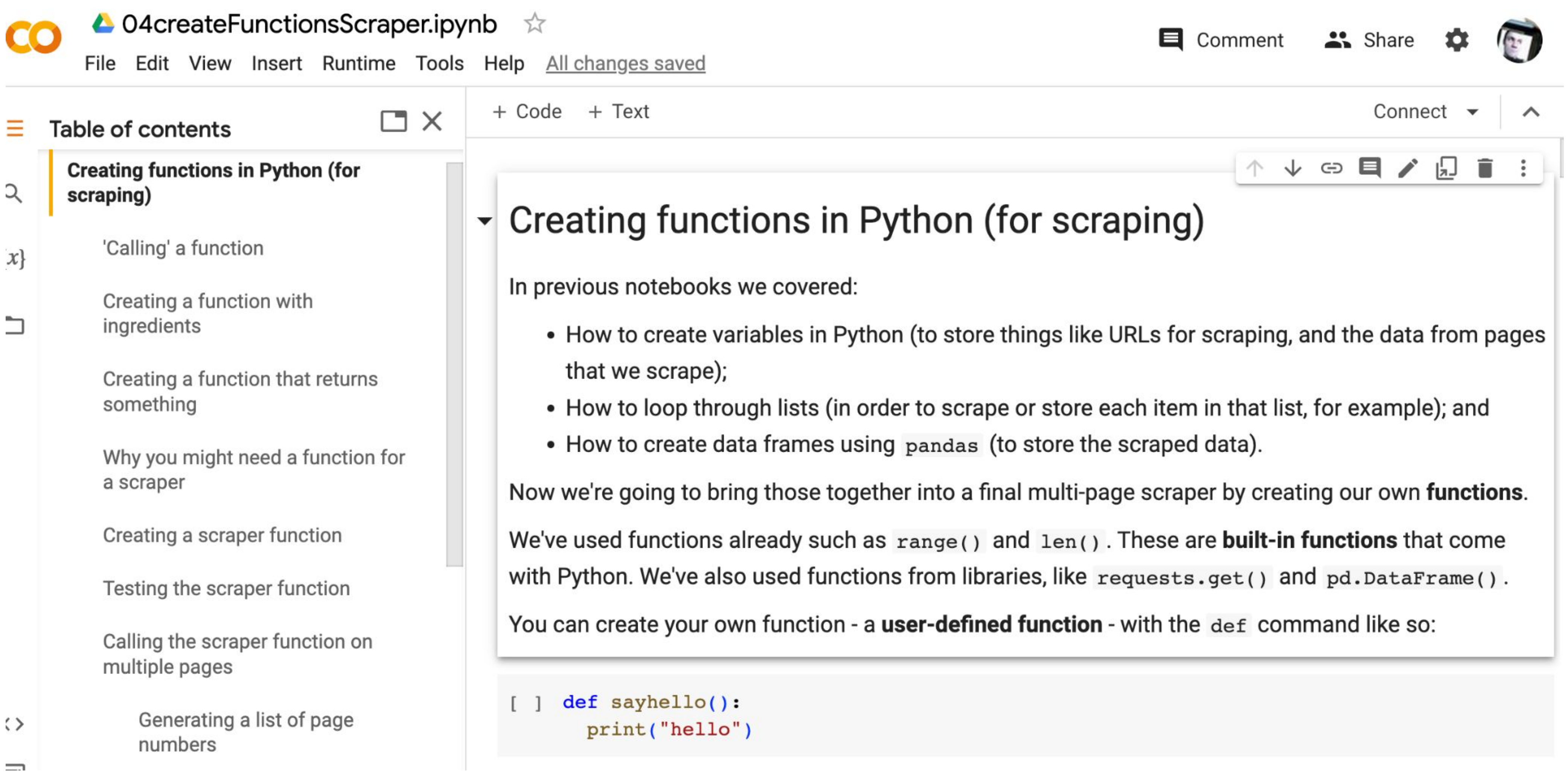
The `return` command is often used to return information to whatever 'called' the function

```
#call the function and
#store result in a variable
whatisit = addtwonumbers(3,8)
print(whatisit)
```

The results ‘returned’ by the function are stored in a new variable

This function needs two ingredients, so we ‘pass’ those with commas between

```
#define a function - it takes one ingredient and calls it 'theurl'
def scrapepage(theurl):
  #fetch URL from that 'theurl'
  page = requests.get(theurl)
  #command beautiful soup to parse the page
  soup = BeautifulSoup(page.content,'html.parser')
  ...
  #return that dataframe to whatever called the function
  return(casedataframe)
```


04createFunctionsScraper.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
Connect

Creating functions in Python (for scraping)

In previous notebooks we covered:

- How to create variables in Python (to store things like URLs for scraping, and the data from pages that we scrape);
- How to loop through lists (in order to scrape or store each item in that list, for example); and
- How to create data frames using `pandas` (to store the scraped data).

Now we're going to bring those together into a final multi-page scraper by creating our own **functions**.

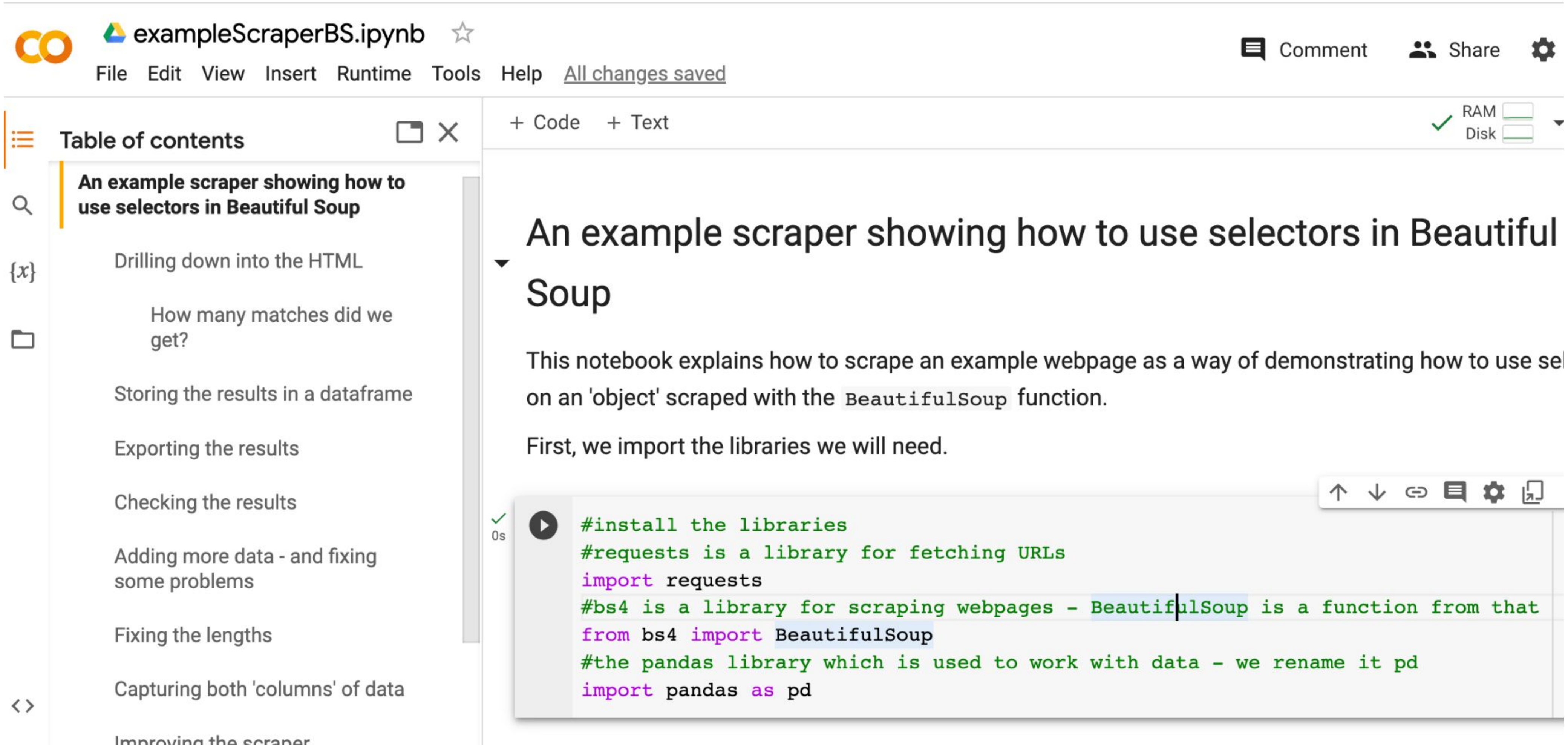
We've used functions already such as `range()` and `len()`. These are **built-in functions** that come with Python. We've also used functions from libraries, like `requests.get()` and `pd.DataFrame()`.

You can create your own function - a **user-defined function** - with the `def` command like so:

```
[ ] def sayhello():
      print("hello")
```

https://colab.research.google.com/drive/13L-09cYMqDOBXxIwHNdZnnW6QAvmNI3H?usp=sharing

You’ve already written the code!



An example scraper showing how to use selectors in Beautiful Soup

This notebook explains how to scrape an example webpage as a way of demonstrating how to use sel
on an 'object' scraped with the `BeautifulSoup` function.

First, we import the libraries we will need.

```
#install the libraries
#requests is a library for fetching URLs
import requests
#bs4 is a library for scraping webpages - BeautifulSoup is a function from that
from bs4 import BeautifulSoup
#the pandas library which is used to work with data - we rename it pd
import pandas as pd
```

https://colab.research.google.com/drive/1UuFhIQYB7K6cjONPNOaGfQeQbqTv-FkE?usp=sharing

Before:

```
#fetch URL
page =
requests.get("https://www.gov.uk/employment-tribunal-decisions")

#command beautiful soup to parse the page
soup = BeautifulSoup(page.content,'html.parser')

#grab all the <div> tags with class="gem-c-document-list__item-title"
divswewant =
soup.select('div[class="gem-c-document-list__item-title"]')
```

After:

Old code is 'wrapped' in a function. You give a name to the variable that will change (the URL)

```
#define a function - it takes one ingredient and calls it
'theurl'
def scrapepage(theurl):
    #fetch URL from that 'theurl'
    page = requests.get(theurl)
    #command beautiful soup to parse the page
    soup = BeautifulSoup(page.content,'html.parser')
    #grab all the <div> tags with
    class="gem-c-document-list__item-title"
    divswewant =
    soup.select('div[class="gem-c-document-list__item-titl
    e"]')
    ...LOOP THROUGH THE MATCHES AND CREATE A DATAFRAME...
    #return that dataframe to whatever called the function
    return(casedataframe)
```

Your previous code is now indented, with the specific URL replaced with the variable taken by the function

At the end the function returns some results

Adapting your code

- Instead of a specific URL string, you use a **variable** to represent ‘any url’
- There may be code to handle **variation** between URLs (e.g. different numbers of items) or contents
- Add a line to **‘return’** the results once the scraper function is finished

```python
#define a function - it takes one ingredient and calls it 'theurl'
def scrapepage(theurl):
     #fetch URL from that 'theurl'
     page = requests.get(theurl)
     #command beautiful soup to parse the page
     soup = BeautifulSoup(page.content,'html.parser')
     #grab all the <div> tags with class="gem-c-document-list__item-title"
     divswewant = soup.select('div[class="gem-c-document-list__item-title"]')
     #this grabs the <time> tags
     times = soup.select('time')
     #create an empty list
     casetitles = []
     #loop through the divswewant list
     for i in divswewant:
          #extract the text
          casename = i.get_text()
          #add the text and link to the previously empty lists
          casetitles.append(casename)
          #create an empty list
     datelist = []
     #loop through the divswewant list
     for i in times:
          #extract the text
          timetext = i.get_text()
          #add the text and link to the previously empty lists
          datelist.append(timetext)
     #create a new dataframe which uses those two lists as its two columns
     casedataframe = pd.DataFrame({"case name" : casetitles,
     "date" : datelist})
     #return that dataframe to whatever called the function
     return(casedataframe)
```

Expand the code inside the function if you want it to do more. Extra lines here fetch all the <time> tags and extract all the contents.

As before, we create a dataframe from the two lists that are generated

Running a function on multiple URLs (lists again!)

Create a range of numbers to loop through - they'll need to be converted to a string to be part of a URL

```
#loop through the numbers 1 to 2
for i in range(1,3):
    #convert to a string and add it to the end of a URL
    fullurl =
    "https://www.gov.uk/employment-tribunal-decisions?page="+str(i)
    #and print it
    print(fullurl)
    #run the scraper function, and store what's returned
    theseresults = scrapepage(fullurl)
    #print what was returned
    print(theseresults)
```

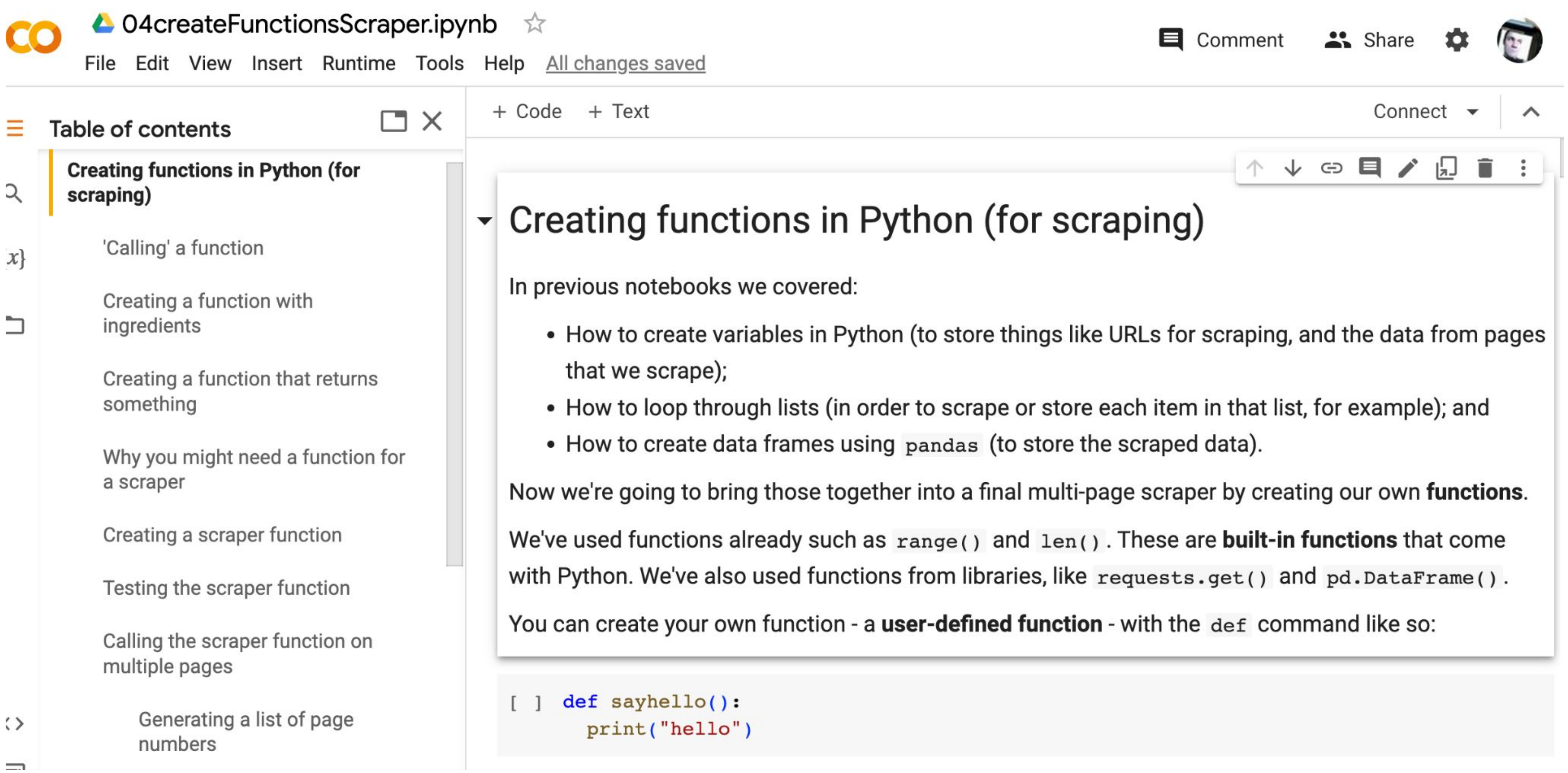
The generated URL is 'passed' to the function as its main ingredient. What the function returns is stored in a variable.

```
#create an empty dataframe
fillme = pd.DataFrame()

#loop through the numbers 1 to 2
for i in range(1,3):
    #add it to the end of a URL,
    fullurl =
    "https://www.gov.uk/employment-tribunal-decisions?page="+str(i)
    #and print it
    print(fullurl)
    #run the scraper function, and store what's returned
    theseresults = scrapepage(fullurl)
    #print what was returned
    #print(theseresults)
    fillme = pd.concat([fillme,theseresults])

fillme
```

pd.concat joins multiple dataframes - a [list of dataframes] needs to be provided in square brackets

What’s happening

- We create an empty data frame for the results of the scraper
- We loop through the URLs we want to scrape, and run the scraper function on each one
- Each time it stores the data frame ‘returned’ by the function in a variable
- We then update the empty data frame by concatenating the empty data frame with the new data frame
- After 1 loop it has 50 items, after 2 it has 100 (50+50 more) and so on


04createFunctionsScraper.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
Table of contents
+ Code + Text
Connect

Creating functions in Python (for scraping)

In previous notebooks we covered:

- How to create variables in Python (to store things like URLs for scraping, and the data from pages that we scrape);
- How to loop through lists (in order to scrape or store each item in that list, for example); and
- How to create data frames using `pandas` (to store the scraped data).

Now we're going to bring those together into a final multi-page scraper by creating our own **functions**.

We've used functions already such as `range()` and `len()`. These are **built-in functions** that come with Python. We've also used functions from libraries, like `requests.get()` and `pd.DataFrame()`.

You can create your own function - a **user-defined function** - with the `def` command like so:

```
[ ] def sayhello():
      print("hello")
```

https://colab.research.google.com/drive/13L-09cYMqDOBXxIwHNdZnnW6QAvmNI3H?usp=sharing

Recap

- Want it done more than once? Create a user-defined function

```
def iamlazy(ingredient1, ingredient2):
   storesomething = dosomething(ingredient1)
   return(storesomething)
```

- The function turns your previous code into a recipe that can be run on multiple URLs
- Trial and error: later pages may not be quite the same - adapt code to handle errors

Try it now:

- Create a notebook and put the code you've already written for one page into a function
- Test it on the same page - does it work?
- Test it on a couple pages
- Test it on the last page